Rodolfo Bonnin

Foreword by:
Claudio Delrieux
Full Professor at Electric and Computer Engineering Department - Universidad Nacional del Sur
Fellow - National Research and Technology Council of Argentina
Chair - Imaging Sciences Laboratory

Machine Learning for Developers

Uplift your regular applications with the power of statistics, analytics, and machine learning

Packt

Contents

Chapter 1. Introduction - Machine Learning and Statistical Science

Machine learning has definitely been one of the most talked about fields in recent years, and for good reason. Every day new applications and models are discovered, and researchers around the world announce impressive advances in the quality of results on a daily basis.

Each day, many new practitioners decide to take courses and search for introductory materials so they can employ these newly available techniques that will improve their applications. But in many cases, the whole corpus of machine learning, as normally explained in the literature, requires a good understanding of mathematical concepts as a prerequisite, thus imposing a high bar for programmers who typically have good algorithmic skills but are less familiar with higher mathematical concepts.

This first chapter will be a general introduction to the field, covering the main study areas of machine learning, and will offer an overview of the basic statistics, probability, and calculus, accompanied by source code examples in a way that allows you to experiment with the provided formulas and parameters.

In this first chapter, you will learn the following topics:

- What is machine learning?
- Machine learning areas
- Elements of statistics and probability
- Elements of calculus

The world around us provides huge amounts of data. At a basic level, we are continually acquiring and learning from text, image, sound, and other types of information surrounding us. The availability of data, then, is the first step in the process of acquiring the skills to perform a task.

A myriad of computing devices around the world collect and store an overwhelming amount of information that is image-, video-, and text-based. So, the raw material for learning is clearly abundant, and it's available in a format that a computer can deal with.

That's the starting point for the rise of the discipline discussed in this book: the study of techniques and methods allowing computers to learn from data without being explicitly programmed.

A more formal definition of machine learning, from *Tom Mitchell*, is as follows:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

This definition is complete, and reinstates the elements that play a role in every machine learning project: the task to perform, the successive experiments, and a clear and appropriate performance measure. In simpler words, we have a program that improves how it performs a task based on experience and guided by a certain criterion.

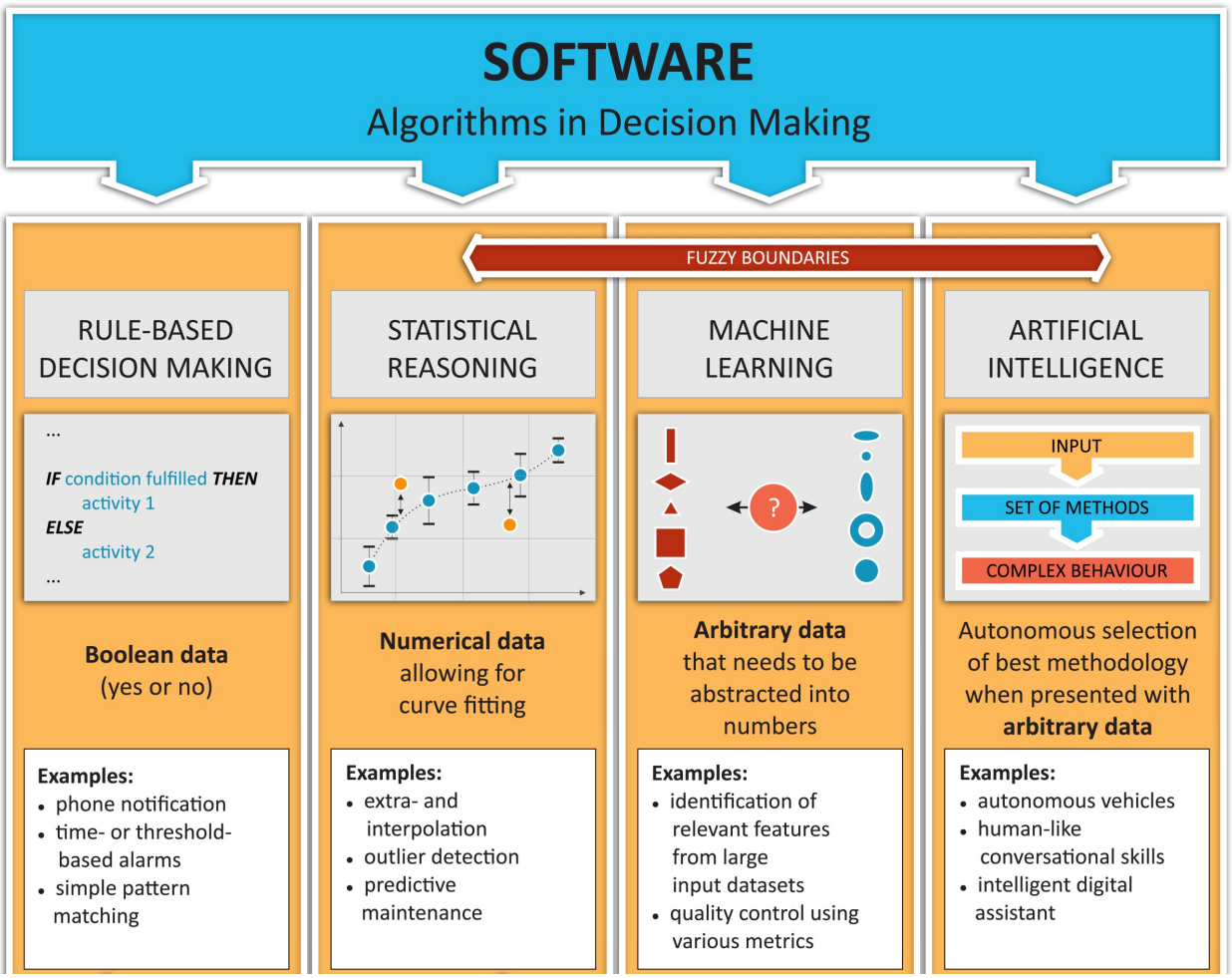
Machine learning in the bigger picture

Machine learning as a discipline is not an isolated field—it is framed inside a wider domain, **Artificial Intelligence** (**AI**). But as you can guess, machine learning didn't appear from the void. As a discipline it has its predecessors, and it has been evolving in stages of increasing complexity in the following four clearly differentiated steps:

1. The first model of machine learning involved rule-based decisions and a simple level of data-based algorithms that includes in itself, and as a prerequisite, all the possible ramifications and decision rules, implying that all the possible options will be hardcoded into the model beforehand by an expert in the field. This structure was implemented in the majority of applications developed since the first programming languages appeared in 1950. The main data type and function being handled by this kind of algorithm is the Boolean, as it exclusively dealt with yes or no decisions.

2. During the second developmental stage of statistical reasoning, we started to let the probabilistic characteristics of the data have a say, in addition to the previous choices set up in advance. This better reflects the fuzzy nature of real-world problems, where outliers are common and where it is more important to take into account the nondeterministic tendencies of the data than the rigid approach of fixed questions. This discipline adds to the mix of mathematical tools elements of **Bayesian probability theory**. Methods pertaining to this category include curve fitting (usually of linear or polynomial), which has the common property of working with numerical data.
3. The machine learning stage is the realm in which we are going to be working throughout this book, and it involves more complex tasks than the simplest Bayesian elements of the previous stage. The most outstanding feature of machine learning algorithms is that they can generalize models from data but the models are capable of generating their own feature selectors, which aren't limited by a rigid target function, as they are generated and defined as the training process evolves. Another differentiator of this kind of model is that they can take a large variety of data types as input, such as speech, images, video, text, and other data susceptible to being represented as vectors.
4. AI is the last step in the scale of abstraction capabilities that, in a way, include all previous algorithm types, but with one key difference: AI algorithms are able to apply the learned knowledge to solve tasks that had never been considered during training. The types of data with which this algorithm works are even more generic than the types of data supported by machine learning, and they should be able, by definition, to transfer problem-solving capabilities from one data type to another, without a complete retraining of the model. In this way, we could develop an algorithm for object detection in black and white images and the model could abstract the knowledge to apply the model to color images.

In the following diagram, we represent these four stages of development towards real AI applications:

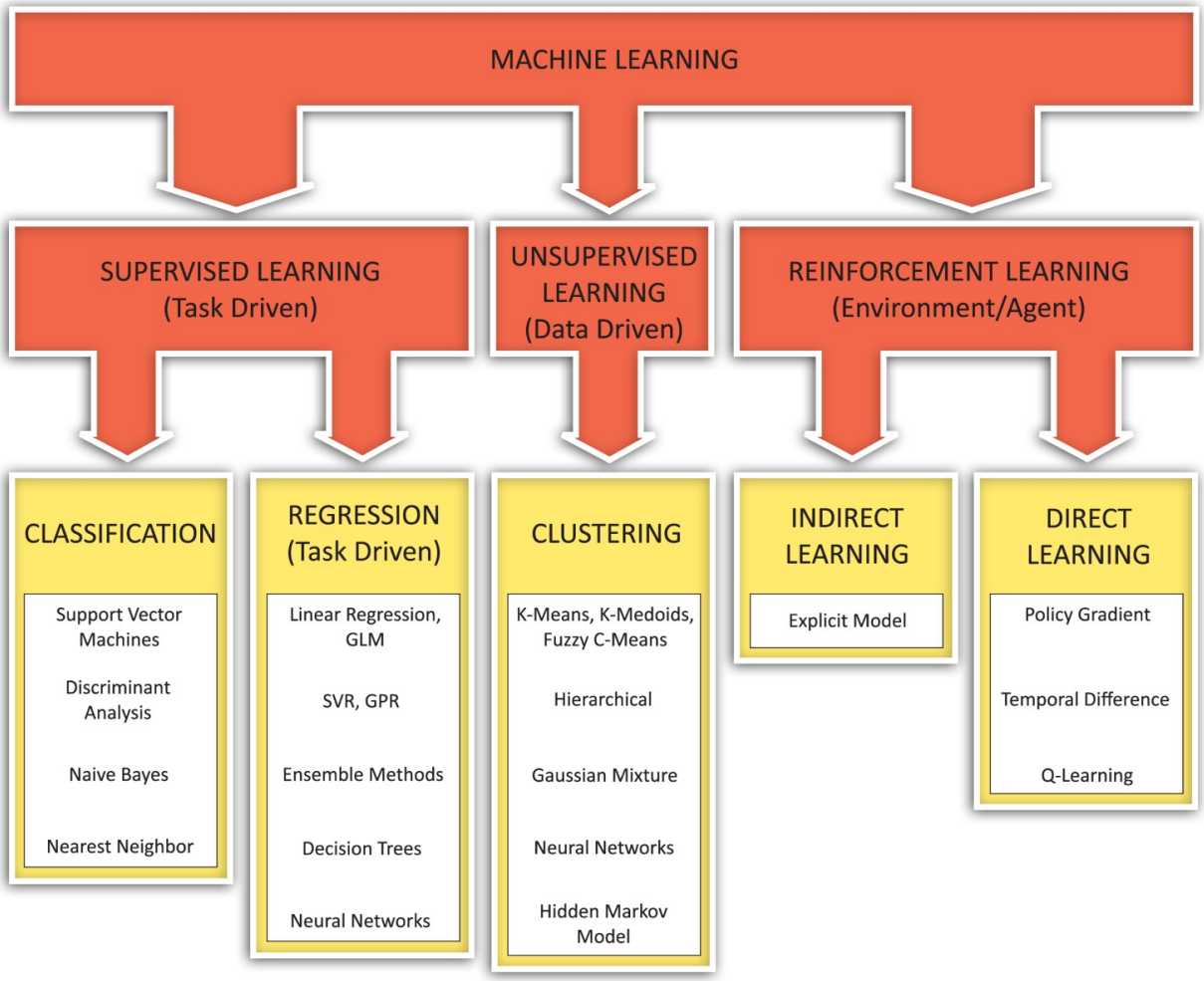


Types of machine learning

Let's try to dissect the different types of machine learning project, starting from the grade of previous knowledge from the point of view of the implementer. The project can be of the following types:

- **Supervised learning**: In this type of learning, we are given a sample set of real data, accompanied by the result the model should give us after applying it. In statistical terms, we have the outcome of all the training set experiments.
- **Unsupervised learning:** This type of learning provides only the sample data from the problem domain, but the task of grouping similar data and applying a category has no previous information from which it can be inferred.
- **Reinforcement learning:** This type of learning doesn't have a labeled sample set and has a different number of participating elements, which include an agent, an environment, and learning an optimum policy or set of steps, maximizing a goal-oriented approach by using rewards or penalties (the result of each attempt).

Take a look at the following diagram:



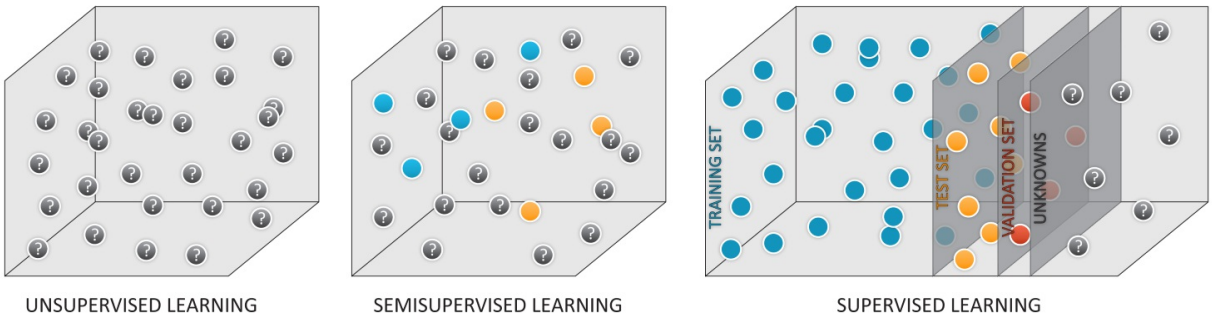
Main areas of Machine Learning

Grades of supervision

The learning process supports gradual steps in the realm of supervision:

- Unsupervised Learning doesn't have previous knowledge of the class or value of any sample, it should infer it automatically.
- Semi-Supervised Learning, needs a seed of known samples, and the model infers the remaining samples class or value from that seed.
- Supervised Learning: This approach normally includes a set of known samples, called training set, another set used to validate the model's generalization, and a third one, called test set, which is used after the training process to have an independent number of samples outside of the training set, and warranty independence of testing.

In the following diagram, depicts the mentioned approaches:



Graphical depiction of the training techniques for Unsupervised, Semi-Supervised and Supervised Learning

Supervised learning strategies - regression versus classification

This type of learning has the following two main types of problem to solve:

- **Regression problem**: This type of problem accepts samples from the problem domain and, after training the model, minimizes the error by comparing the output with the real answers, which allows the prediction of the right answer when given a new unknown sample
- **Classification problem**: This type of problem uses samples from the domain to assign a label or group to new unknown samples

Unsupervised problem solving–clustering

The vast majority of unsupervised problem solving consist of grouping items by looking at similarities or the value of shared features of the observed items, because there is no certain information about the *apriori* classes. This type of technique is called clustering.

Outside of these main problem types, there is a mix of both, which is called semi-supervised problem solving, in which we can train a labeled set of elements and also use inference to assign information to unlabeled data during training time. To assign data to unknown entities, three main criteria are used—smoothness (points close to each other are of the same class), cluster (data tends to form clusters, a special case of smoothness), and manifold (data pertains to a manifold of much lower dimensionality than the original domain).

Tools of the trade–programming language and libraries

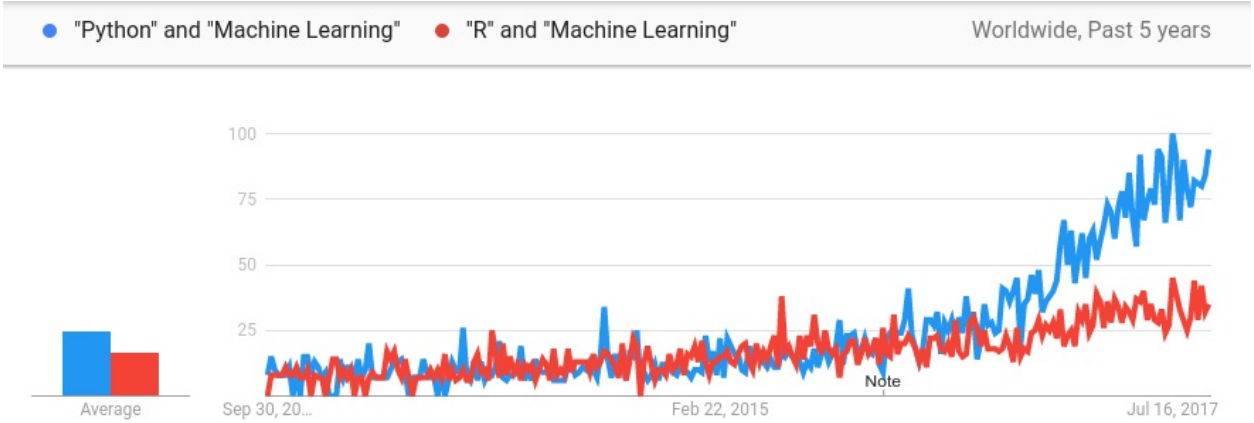
As this book is aimed at developers, we think that the approach of explaining the mathematical concepts using real code comes naturally.

When choosing the programming language for the code examples, the first approach was to use multiple technologies, including some cutting-edge libraries. After consulting the community, it was clear that a simple language would be preferable when explaining the concepts.

Among the options, the ideal candidate would be a language that is simple to understand, with real-world machine learning adoption, and that is also relevant.

The clearest candidate for this task was Python, which fulfils all these conditions, and especially in the last few years has become the go-to language for machine learning, both for newcomers and professional practitioners.

In the following graph, we compare the previous star in the machine learning programming language field, R, and we can clearly conclude the huge, favorable tendency towards using Python. This means that the skills you acquire in this book will be relevant now and in the foreseeable future:



Interest graph for R and Python in the Machine Learning realm.

In addition to Python code, we will have the help of a number of the most well-known numerical, statistical, and graphical libraries in the Python ecosystem, namely pandas, NumPy, and matplotlib. For the **deep neural network** examples, we will use the Keras library, with TensorFlow as the backend.

The Python language

Python is a general-purpose scripting language, created by the Dutch programmer Guido Van Rossum in 1989. It possesses a very simple syntax with great extensibility, thanks to its

numerous extension libraries, making it a very suitable language for prototyping and general coding. Because of its native C bindings, it can also be a candidate for production deployment.

The language is actually used in a variety of areas, ranging from web development to scientific computing, in addition to its use as a general scripting tool.

The NumPy library

If we had to choose a definitive must-use library for use in this book, and a non-trivial mathematical application written in Python, it would have to be NumPy. This library will help us implement applications using statistics and linear algebra routines with the following components:

- A versatile and performant N-dimensional array object
- Many mathematical functions that can be applied to these arrays in a seamless manner
- Linear algebra primitives
- Random number distributions and a powerful statistics package
- Compatibility with all the major machine learning packages

> ## Note
>
> The NumPy library will be used extensively throughout this book, using many of its primitives to simplify the concept explanations with code.

The matplotlib library

Data plotting is an integral part of data science and is normally the first step an analyst performs to get a sense of what's going on in the provided set of data.

For this reason, we need a very powerful library to be able to graph the input data, and also to represent the resulting output. In this book, we will use Python's matplotlib library to describe concepts and the results from our models.

What's matplotlib?

Matplotlib is an extensively used plotting library, especially designed for 2D graphs. From this library, we will focus on using the `pyplot` module, which is a part of the API of matplotlib and has MATLAB-like methods, with direct NumPy support. For those of you not familiar with MATLAB, it has been the default mathematical notebook environment for the scientific and engineering fields for decades.

The method described will be used to illustrate a large proportion of the concepts involved, and in fact, the reader will be able to generate many of the examples in this book with just these two libraries, and using the provided code.

Pandas

Pandas complements the previously mentioned libraries with a special structure, called `DataFrame`, and also adds many statistical and data mangling methods, such as I/O, for many

different formats, such as slicing, subsetting, handling missing data, merging, and reshaping, among others.

The `DataFrame` object is one of the most useful features of the whole library, providing a special 2D data structure with columns that can be of different data types. Its structure is very similar to a database table, but immersed in a flexible programming runtime and ecosystem, such as SciPy. These data structures are also compatible with NumPy matrices, so we can also apply high-performance operations to the data with minimal effort.

SciPy

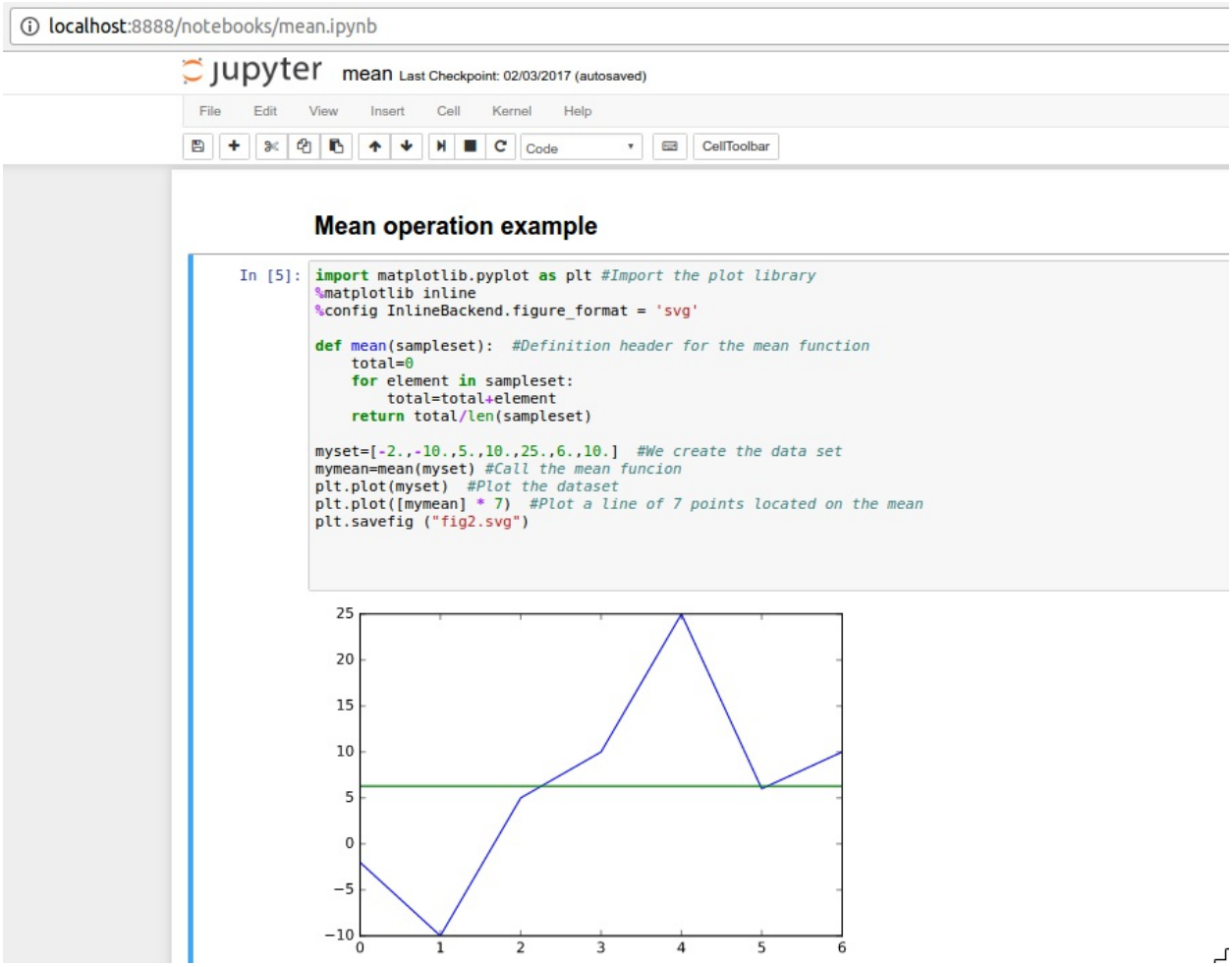
SciPy is a stack of very useful scientific Python libraries, including NumPy, pandas, matplotlib, and others, but it also the core library of the ecosystem, with which we can also perform many additional fundamental mathematical operations, such as integration, optimization, interpolation, signal processing, linear algebra, statistics, and file I/O.

Jupyter notebook

Jupyter is a clear example of a successful Python-based project, and it's also one of the most powerful devices we will employ to explore and understand data through code.

Jupyter notebooks are documents consisting of intertwined cells of code, graphics, or formatted text, resulting in a very versatile and powerful research environment. All these elements are wrapped in a convenient web interface that interacts with the **IPython** interactive interpreter.

Once a Jupyter notebook is loaded, the whole environment and all the variables are in memory and can be changed and redefined, allowing research and experimentation, as shown in the following screenshot:



Jupyter notebook

This tool will be an important part of this book's teaching process, because most of the Python examples will be provided in this format. In the last chapter of the book, you will find the full installation instructions.

Note

After installing, you can cd into the directory where your notebooks reside, and then call Jupyter by typing `jupyter notebook`

Basic mathematical concepts

As we saw in the previous sections, this main target audience of the book is developers who want to understand machine learning algorithms. But in order to really grasp the motivations and reason behind them, it's necessary to review and build all the fundamental reasoning, which includes statistics, probability, and calculus.

We will first start with some of the fundamentals of statistics.

Statistics - the basic pillar of modeling uncertainty

Statistics can be defined as a discipline that uses data samples to extract and support conclusions about larger samples of data. Given that machine learning comprises a big part of the study of the properties of data and the assignment of values to data, we will use many statistical concepts to define and justify the different methods.

Descriptive statistics - main operations

In the following sections, we will start defining the fundamental operations and measures of the discipline of statistics in order to be able to advance from the fundamental concepts.

Mean

This is one of the most intuitive and most frequently used concepts in statistics. Given a set of numbers, the mean of that set is the sum of all the elements divided by the number of elements in the set.

The formula that represents the mean is as follows:

$$\mu = \frac{1}{n}\sum_i x_i$$

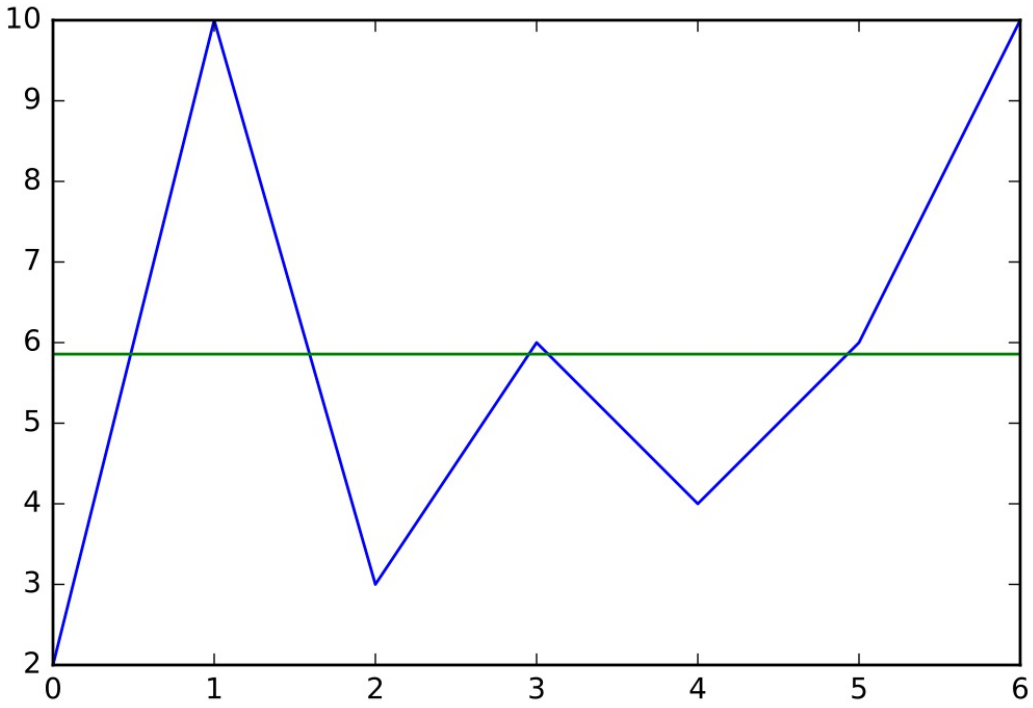
Although this is a very simple concept, we will write a Python code sample in which we will create a sample set, represent it as a line plot, and mark the mean of the whole set as a line, which should be at the weighted center of the samples. It will serve as an introduction to Python syntax, and also as a way of experimenting with Jupyter notebooks:

```
import matplotlib.pyplot as plt #Import the plot library

def mean(sampleset):  #Definition header for the mean function
    total=0
    for element in sampleset:
        total=total+element
    return total/len(sampleset)

myset=[2.,10.,3.,6.,4.,6.,10.]  #We create the data set
mymean=mean(myset) #Call the mean funcion
```

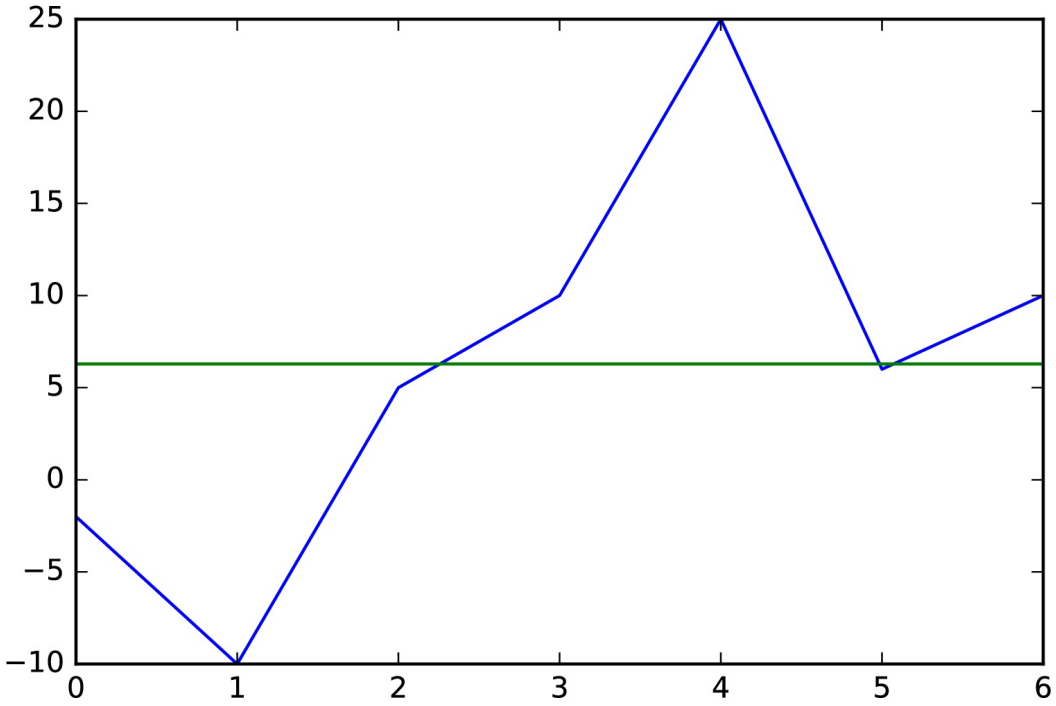
```
plt.plot(myset)  #Plot the dataset
plt.plot([mymean] * 7)  #Plot a line of 7 points located on the mean
```

This program will output a time series of the dataset elements, and will then draw a line at the mean height.

As the following graph shows, the mean is a succinct (one value) way of describing the tendency of a sample set:



In this first example, we worked with a very homogeneous sample set, so the mean is very informative regarding its values. But let's try the same sample with a very dispersed sample set (you are encouraged to play with the values too):



Variance

As we saw in the first example, the mean isn't sufficient to describe non-homogeneous or very dispersed samples.

In order to add a unique value describing how dispersed the sample set's values are, we need to look at the concept of variance, which needs the mean of the sample set as a starting point, and then averages the distances of the samples from the provided mean. The greater the variance, the more scattered the sample set.

The canonical definition of variance is as follows:

$$\sigma^2 = \frac{1}{n}\sum\left(x_i - \mu\right)^2$$

Let's write the following sample code snippet to illustrate this concept, adopting the previously used libraries. For the sake of clarity, we are repeating the declaration of the `mean` function:

```
import math #This library is needed for the power operation
def mean(sampleset):  #Definition header for the mean function
    total=0
    for element in sampleset:
        total=total+element
    return total/len(sampleset)

def variance(sampleset):  #Definition header for the mean function
    total=0
    setmean=mean(sampleset)
    for element in sampleset:
        total=total+(math.pow(element-setmean,2))
    return total/len(sampleset)

myset1=[2.,10.,3.,6.,4.,6.,10.]  #We create the data set
myset2=[1.,-100.,15.,-100.,21.]
print "Variance of first set:" + str(variance(myset1))
print "Variance of second set:" + str(variance(myset2))
```

The preceding code will generate the following output:

```
Variance of first set:8.69387755102
Variance of second set:3070.64
```

As you can see, the variance of the second set was much higher, given the really dispersed values. The fact that we are computing the mean of the squared distance helps to really outline the differences, as it is a quadratic operation.

Standard deviation

Standard deviation is simply a means of regularizing the square nature of the mean square used in the variance, effectively linearizing this term. This measure can be useful for other, more complex operations.

Here is the official form of standard deviation:

$$\sigma = \sqrt{\frac{1}{n}\sum\left(x_i - \mu\right)^2}$$

Probability and random variables

We are now about to study the single most important discipline required for understanding all the concepts of this book.

Probability is a mathematical discipline, and its main occupation is the study of random events. In a more practical definition, probability normally tries to quantify the level of certainty (or conversely, uncertainty) associated with an event, from a universe of possible occurrences.

Events

In order to understand probabilities, we first need to define events. An event is, given an experiment in which we perform a determined action with different possible results, a subset of all the possible outcomes for that experiment.

Examples of events are a particular dice number appearing, and a product defect of particular type appearing on an assembly line.

Probability

Following the previous definitions, probability is the likelihood of the occurrence of an event. Probability is quantified as a real number between *0* and *1*, and the assigned probability *P* increases towards *1* when the likelihood of the event occurring increases.

The mathematical expression for the probability of the occurrence of an event is `P(E)`.

Random variables and distributions

When assigning event probabilities, we could also try to cover the entire sample and assign one probability value to each of the possible outcomes for the sample domain.

This process does indeed have all the characteristics of a function, and thus we will have a random variable that will have a value for each one of the possible event outcomes. We will call this function a random function.

These variables can be of the following two types:

- **Discrete**: If the number of outcomes is finite, or countably infinite
- **Continuous**: If the outcome set belongs to a continuous interval

This probability function is also called **probability distribution**.

Useful probability distributions

Between the multiple possible probability distributions, there are a number of functions that have been studied and analyzed for their special properties, or the popular problems they represent.

We will describe the most common ones that have a special effect on the development of machine learning.

Bernoulli distributions

Let's begin with a simple distribution: one that has a binary outcome, and is very much like tossing a (fair) coin.

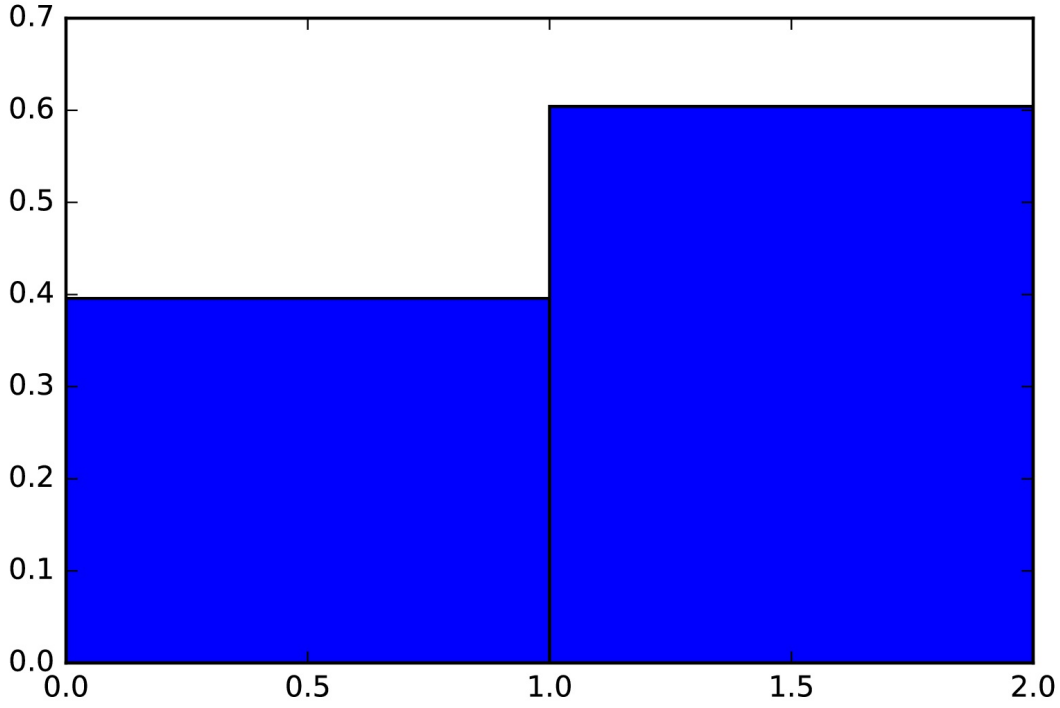
This distribution represents a single event that takes the value *1* (let's call this *heads*) with a probability of *p*, and *0* (lets call this *tails*), with probability *1-p*.

In order to visualize this, let's generate a large number of events of a Bernoulli distribution using

`np` and graph the tendency of this distribution, with the following only two possible outcomes:

```
plt.figure()
distro = np.random.binomial(1, .6, 10000)/0.5
plt.hist(distro, 2 , normed=1)
```

The following graph shows the binomial distribution, through an histogram, showing the complementary nature of the outcomes' probabilities:

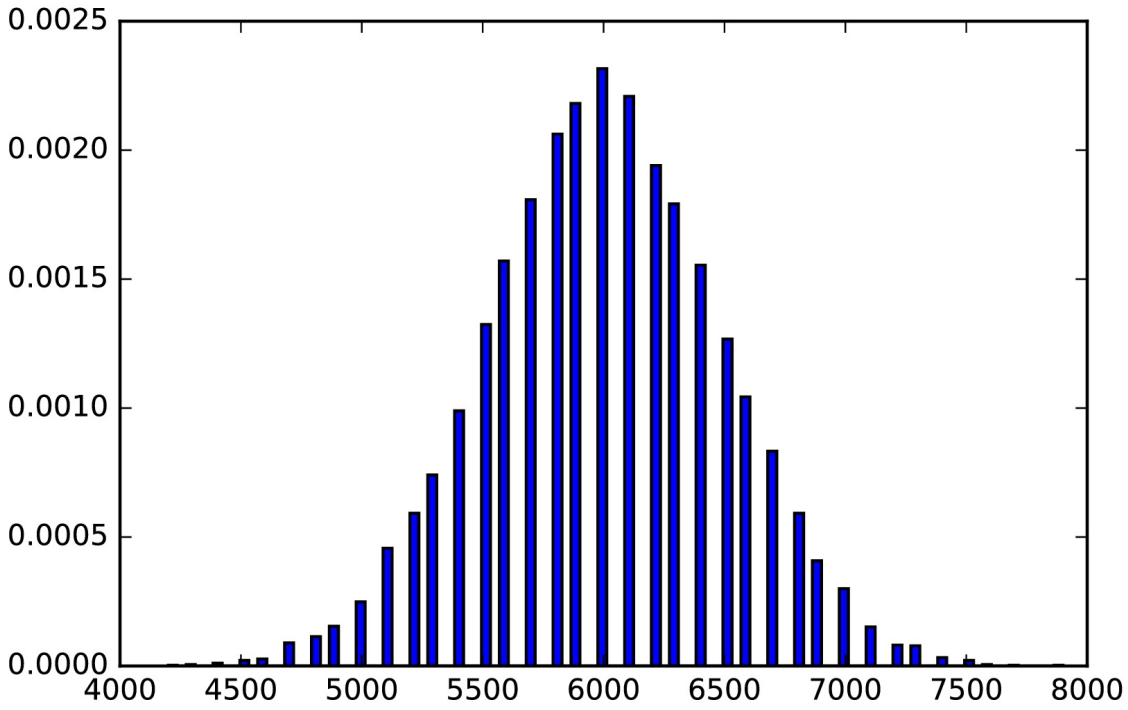


Binomial distribution

So, here we see the very clear tendency of the complementing probabilities of the possible outcomes. Now let's complement the model with a larger number of possible outcomes. When their number is greater than 2, we are talking about a **multinomial distribution**:

```
plt.figure()
distro = np.random.binomial(100, .6, 10000)/0.01
plt.hist(distro, 100 , normed=1)
plt.show()
```

Take a look at the following graph:



Multinomial distribution with 100 possible outcomes

Uniform distribution

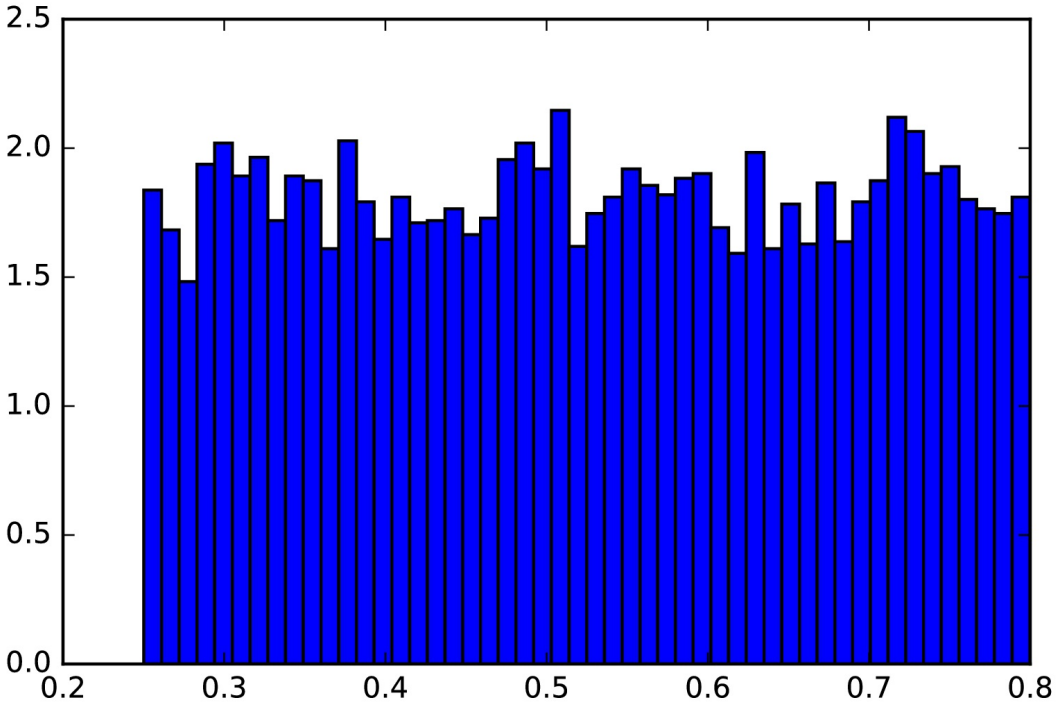
This very common distribution is the first continuous distribution that we will see. As the name implies, it has a constant probability value for any interval of the domain.

In order to integrate to 1, *a* and *b* being the extreme of the function, this probability has the value of *1/(b-a)*.

Let's generate a plot with a sample uniform distribution using a very regular histogram, as generated by the following code:

```
plt.figure()
uniform_low=0.25
uniform_high=0.8

plt.hist(uniform, 50, normed=1)
plt.show()
```

Take look at the following graph:



Uniform distribution

Normal distribution

This very common continuous random function, also called a **Gaussianfunction**, can be defined with the simple metrics of the mean and the variance, although in a somewhat complex form.

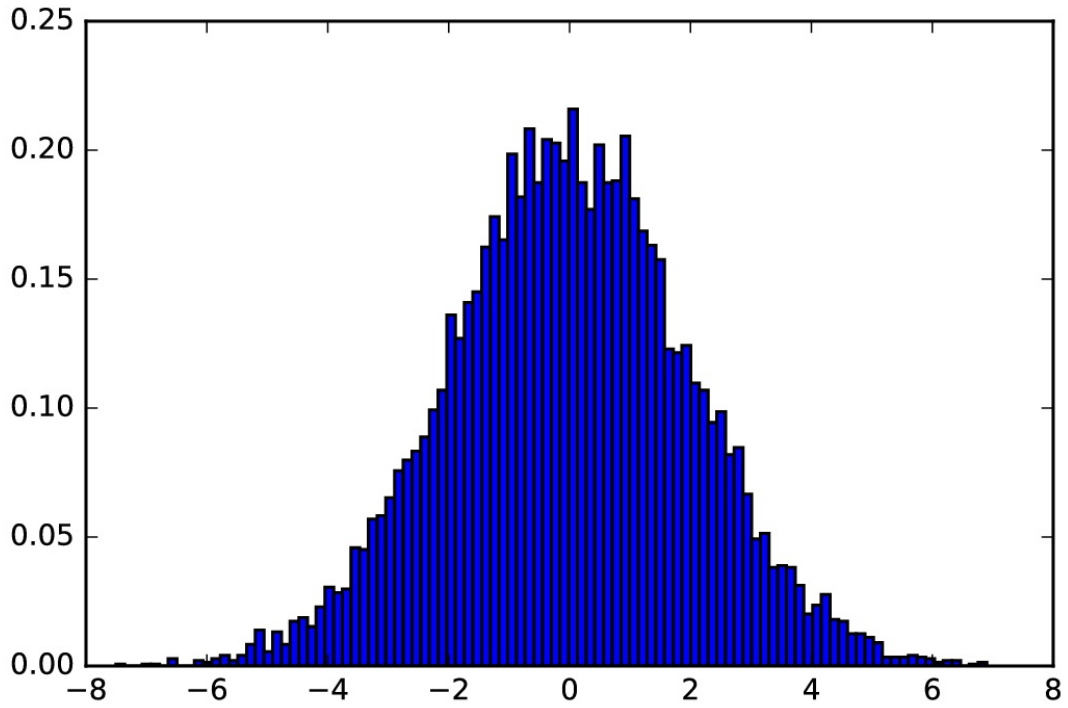
This is the canonical form of the function:

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Take a look at the following code snippet:

```
import matplotlib.pyplot as plt #Import the plot library
import numpy as np
mu=0.
sigma=2.
distro = np.random.normal(mu, sigma, 10000)
plt.hist(distro, 100, normed=True)
plt.show()
```

The following graph shows the generated distribution's histogram:



Normal distribution

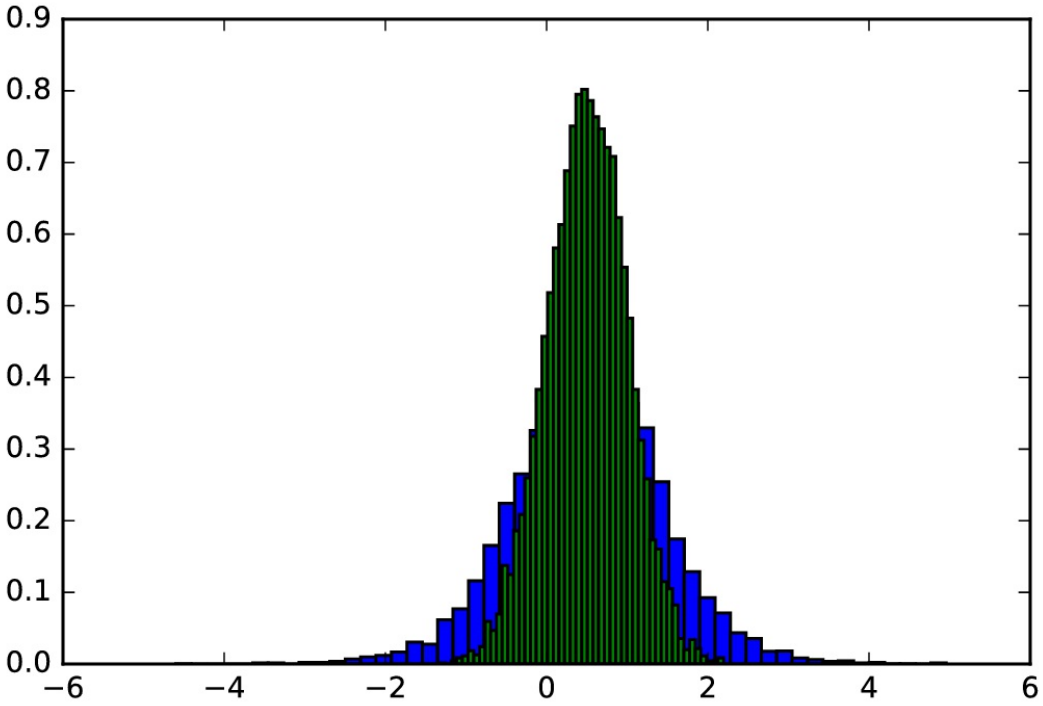
Logistic distribution

This distribution is similar to the normal distribution, but with the morphological difference of having a more elongated tail. The main importance of this distribution lies in its **cumulative distribution function** (**CDF**), which we will be using in the following chapters, and will certainly look familiar.

Let's first represent the base distribution by using the following code snippet:

```
import matplotlib.pyplot as plt #Import the plot library
import numpy as np
mu=0.5
sigma=0.5
distro2 = np.random.logistic(mu, sigma, 10000)
plt.hist(distro2, 50, normed=True)
distro = np.random.normal(mu, sigma, 10000)
plt.hist(distro, 50, normed=True)
plt.show()
```

Take a look at the following graph:

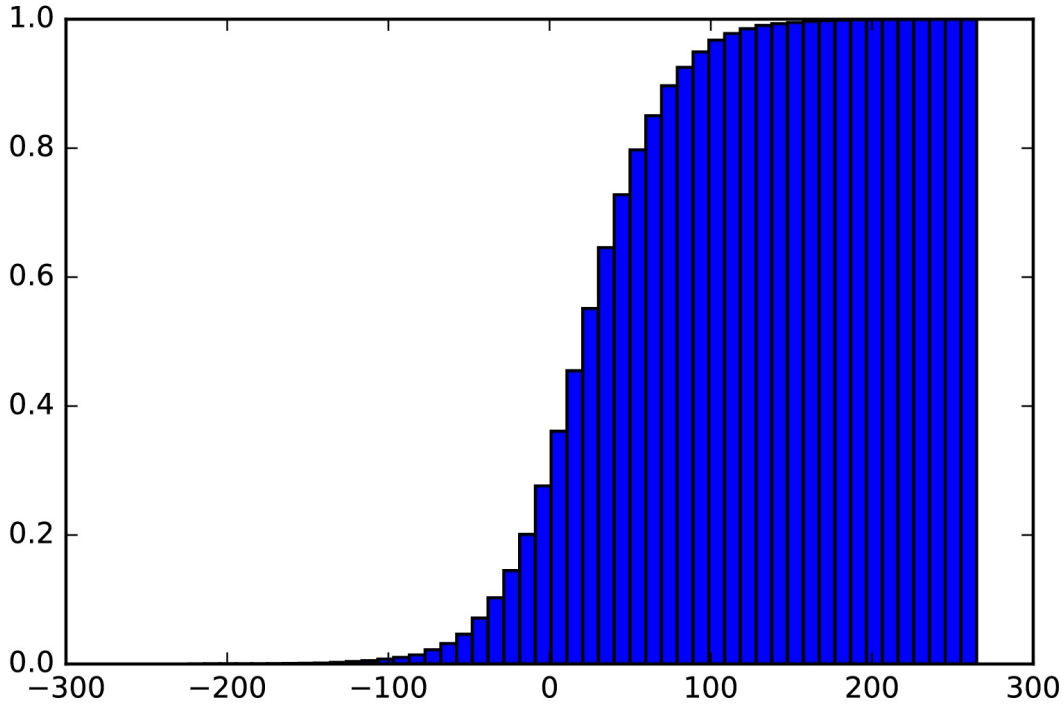


Logistic (red) vs Normal (blue) distribution

Then, as mentioned before, let's compute the CDF of the logistic distribution so that you will see a very familiar figure, the **sigmoid** curve, which we will see again when we review neural network activation functions:

```
plt.figure()
logistic_cumulative = np.random.logistic(mu, sigma, 10000)/0.02
plt.hist(logistic_cumulative, 50, normed=1, cumulative=True)
plt.show()
```

Take a look at the following graph:



Inverse of the logistic distribution

Statistical measures for probability functions

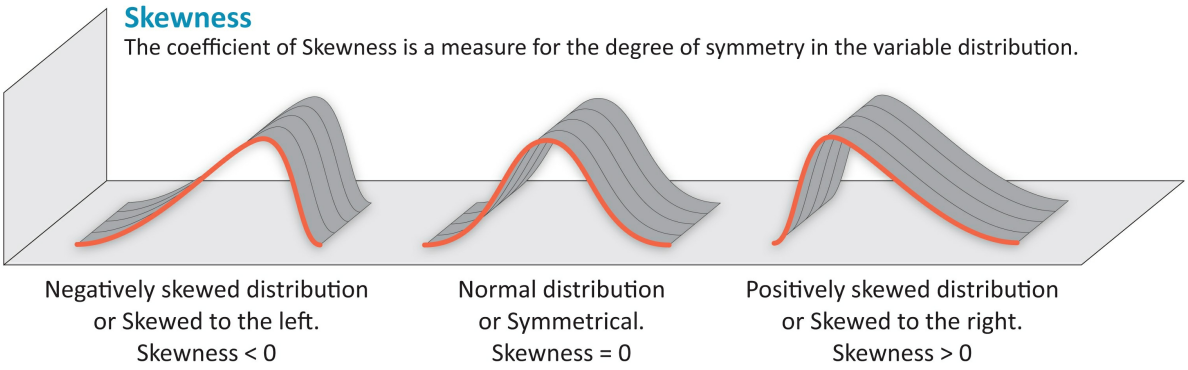
In this section, we will see the most common statistical measures that can be applied to probabilities. The first measures are the mean and variance, which do not differ from the definitions we saw in the introduction to statistics.

Skewness

This measure represents the lateral deviation, or in general terms, the deviation from the center, or the symmetry (or lack thereof) of a probability distribution. In general, if skewness is negative, it implies a deviation to the right, and if it is positive, it implies a deviation to the left:

$$\left(S_k\right) = \frac{1}{n} \frac{\sum_{i=1}^{n} \left(X_i - \overline{X}\right)^3}{s^3}$$

Take a look at the following diagram, which depicts the skewness statistical distribution:



Depiction of the how the distribution shape influences Skewness.

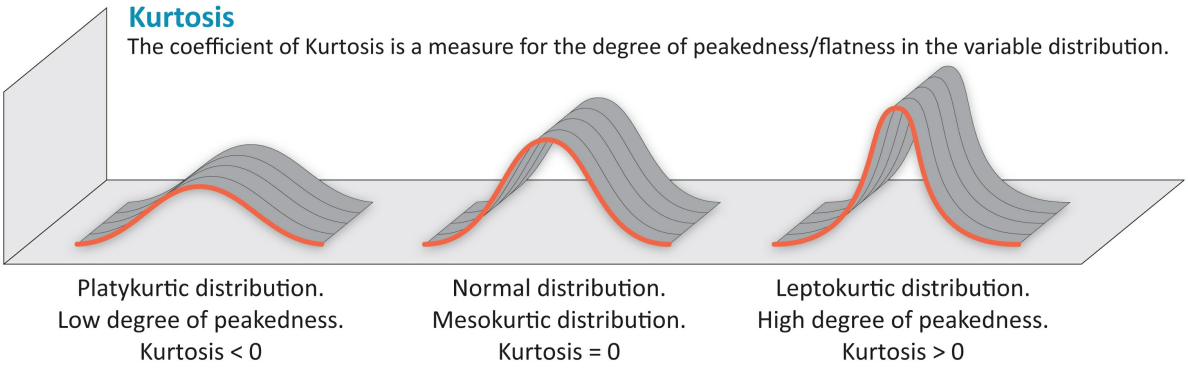
Kurtosis

Kurtosis gives us an idea of the central concentration of a distribution, defining how acute the central area is, or the reverse—how distributed the function's tail is.

The formula for kurtosis is as follows:

$$Kurtosis = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{SD(x)}\right)^4$$

In the following diagram, we can clearly see how the new metrics that we are learning can be intuitively understood:



Depiction of the how the distribution shape influences Kurtosis

Differential calculus elements

To cover the minimum basic knowledge of machine learning, especially the learning algorithms such as gradient descent, we will introduce you to the concepts involved in differential calculus.

Preliminary knowledge

Covering the calculus terminology necessary to get to gradient descent theory would take many chapters, so we will assume you have an understanding of the concepts of the properties of the most well-known continuous functions, such as **linear**, **quadratic**, **logarithmic**, and **exponential**, and the concept of **limit**.

For the sake of clarity, we will develop the concept of the functions of one variable, and then expand briefly to cover multivariate functions.

In search of changes–derivatives

We established the concept of functions in the previous section. With the exception of constant functions defined in the entire domain, all functions have some sort of value dynamics. That means that *f(x1)* is different than *f(x2)* for some determined values of *x*.

The purpose of differential calculus is to measure change. For this specific task, many mathematicians of the 17th century (Leibniz and Newton were the most prominent exponents) worked hard to find a simple model to measure and predict how a symbolically defined function changed over time.

This research guided the field to one wonderful concept—a symbolic result that, under certain conditions, tells you how much and in which direction a function changes at a certain point. This is the concept of a derivative.

Sliding on the slope

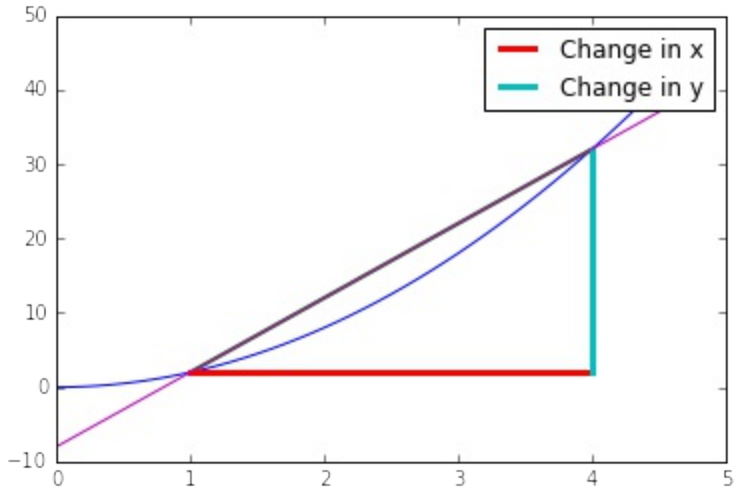
If we want to measure how a function changes over time, the first intuitive step would be to take the value of a function and then measure it at the subsequent point. Subtracting the second value from the first would give us an idea of how much the function changes over time:

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

def quadratic(var):
    return 2* pow(var,2)
x=np.arange(0,.5,.1)
plt.plot(x,quadratic(x))
plt.plot([1,4], [quadratic(1), quadratic(4)],  linewidth=2.0)
plt.plot([1,4], [quadratic(1), quadratic(1)],  linewidth=3.0,
label="Change in x")
plt.plot([4,4], [quadratic(1), quadratic(4)],  linewidth=3.0,
label="Change in y")
plt.legend()
plt.plot (x, 10*x -8 )
plt.plot()
```

In the preceding code example, we first defined a sample quadratic equation (`2*x²`*)* and then defined the part of the domain in which we will work with the `arange` function (from `0` to `0.5`, in

`0.1` steps).

Then, we define an interval for which we measure the change of *y* over *x*, and draw lines indicating this measurement, as shown in the following graph:



Initial depiction of a starting setup for implementing differentiation

In this case, we measure the function at *x=1* and *x=4*, and define the rate of change for this interval as follows:

$$\text{diff} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Applying the formula, the result for the sample is *(36-0)/3= 12*.

This initial approach can serve as a way of approximately measuring this dynamic, but it's too dependent on the points at which we take the measurement, and it has to be taken at every interval we need.

To have a better idea of the dynamics of a function, we need to be able to define and measure the instantaneous change rate at every point in the function's domain.

This idea of instantaneous change brings to us the need to reduce the distance between the domain's *x* values, taken at a point where there are very short distances between them. We will formulate this approach with an initial value *x*, and the subsequent value, $x + \Delta x$:

$$\text{diff} = \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

In the following code, we approximate the difference, reducing *Δx* progressively:

```
initial_delta = .1
x1 = 1
for power in range (1,6):
    delta = pow (initial_delta, power)
    derivative_aprox= (quadratic(x1+delta) - quadratic (x1) )/
    ((x1+delta) - x1 )
    print "del    ta: " + str(delta) + ", estimated derivative: " +
    str(derivative_aprox)
```

In the preceding code, we first defined an initial delta, which brought an initial approximation. Then, we apply the difference function, with diminishing values of delta, thanks us to powering `0.1` with incremental powers. The results we get are as follows:

```
delta: 0.1, estimated derivative: 4.2
delta: 0.01, estimated derivative: 4.02
delta: 0.001, estimated derivative: 4.002
delta: 0.0001, estimated derivative: 4.0002
delta: 1e-05, estimated derivative: 4.00002
```

As the separation diminishes, it becomes clear that the change rate will hover around `4`. But when does this process stop? In fact, we could say that this process can be followed ad infinitum, at least in a numeric sense.

This is when the concept of limit intuitively appears. We will then define this process, of making Δ indefinitely smaller, and will call it the derivative of *f(x)* or *f'(x)*:

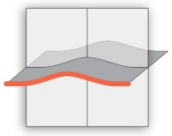
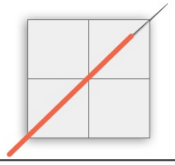
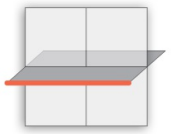
$$\frac{dy}{dx} = f'(x) = \lim_{\Delta \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x - x}$$

This is the formal definition of the derivative.

But mathematicians didn't stop with these tedious calculations, making a large number of numerical operations (which were mostly done manually of the 17th century), and wanted to further simplify these operations.

What if we perform another step that can symbolically define the derivative of a function?

That would require building a function that gives us the derivative of the corresponding function, just by replacing the `x` variable value. That huge step was also reached in the 17th century, for different function families, starting with the parabolas *($y=x^2+b$)*, and following with more complex functions:

NAME	FUNCTION $y = f(x)$	DERIVATIVE $\partial y / \partial x$
Logistic	$\frac{1}{1+e^{-x}}$ 	$y(1-y)$ 
Tanh	$Tanh(x)$ 	$1-y^2$ 
Gaussian	$e^{-x^2/2}$ 	$-xe^{-x^2/2}$ 
Linear	x 	1 

Chain rule

One very important result of the symbolic determination of a function's derivative is the chain rule. This formula, first mentioned in a paper by Leibniz in 1676, made it possible to solve the derivatives of composite functions in a very simple and elegant manner, simplifying the solution for very complex functions.

In order to define the chain rule, if we suppose a function *f*, which is defined as a function of another function *g*, *f(g(x))* of *F*, the derivative can be defined as follows:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}.$$

The formula of the chain rule allows us to differentiate formulas whose input values depend on another function. This is the same as searching the rate of change of a function that is linked to a previous one. The chain rule is one of the main theoretical concepts employed in the training phase of neural networks, because in those layered structures, the output of the first neuron layers will be the inputs of the following, giving, as a result, a composite function that, most of the time, is of more than one nesting level.

Partial derivatives

Until now we've been working with univariate functions, but the type of function we will mostly work with from now on will be multivariate, as the dataset will contain much more than one column and each one of them will represent a different variable.

In many cases, we will need to know how the function changes in a relationship with only one dimension, which will involve looking at how one column of the dataset contributes to the total number of function changes.

The calculation of partial derivatives consists of applying the already known derivation rules to the multivariate function, considering the variables are not being derived as constant.

Take a look at the following power rule:

$f(x,y) = 2x^3y$

When differentiating this function with respect to *x*, considering *y* a constant, we can rewrite it as $3 . 2 y x^2$, and applying the derivative to the variable *x* allows us to obtain the following derivative:

$d/dx (f(x,y)) = 6y*x^2$

Using these techniques, we can proceed with the more complex multivariate functions, which will be part of our feature set, normally consisting of much more than two variables.

Summary

In this chapter, we worked through many different conceptual elements, including an overview of some basic mathematical concepts, which serve as a base for the machine learning concepts.

These concepts will be useful when we formally explain the mechanisms of the different modeling methods, and we encourage you to improve your understanding of them as much as possible, before and while reading the chapters, to better grasp how the algorithm works.

In the next chapter, we will have a quick overview of the the the complete workflow of a machine learning project, which will help us to understand the various elements involved, from data gathering to result evaluation.

Chapter 2. The Learning Process

In the first chapter, we saw a general overview of the mathematical concepts, history, and areas of the field of machine learning.

As this book intends to provide a practical but formally correct way of learning, now it's time to explore the general thought process for any machine learning process. These concepts will be pervasive throughout the chapters and will help us to define a common framework of the best practices of the field.

The topics we will cover in this chapter are as follows:

- Understanding the problem and definitions
- Dataset retrieval, preprocessing, and feature engineering
- Model definition, training, and evaluation
- Understanding results and metrics

Every machine learning problem tends to have its own particularities. Nevertheless, as the discipline advances through time, there are emerging patterns of what kind of steps a machine learning process should include, and the best practices for them. The following sections will be a list of these steps, including code examples for the cases that apply.

Understanding the problem

When solving machine learning problems, it's important to take time to analyze both the data and the possible amount of work beforehand. This preliminary step is flexible and less formal than all the subsequent ones on this list.

From the definition of machine learning, we know that our final goal is to make the computer learn or generalize a certain behavior or model from a sample set of data. So, the first thing we should do is understand the new capabilities we want to learn.

In the enterprise field, this is the time to have more practical discussions and brainstorms. The main questions we could ask ourselves during this phase could be as follows:

- What is the real problem we are trying to solve?
- What is the current information pipeline?
- How can I streamline data acquisition?
- Is the incoming data complete, or does it have gaps?
- What additional data sources could we merge in order to have more variables to hand?
- Is the data release periodical, or can it be acquired in real time?
- What should be the minimal representative unit of time for this particular problem?
- Does the behavior I try to characterize change in nature, or are its fundamentals more or less stable through time?

Understanding the problem involves getting on the business knowledge side and looking at all the valuable sources of information that could influence the model. Once identified, the following task will generate an organized and structured set of values, which will be the input to our model.

Let's proceed to see an example of an initial problem definition, and the thought process of the initial analysis.

Let's say firm A is a retail chain that wants to be able to predict a certain product's demand on certain dates. This could be a challenging task because it involves human behavior, which has some non-deterministic components.

What kind of data input would be needed to build such a model? Of course, we would want the transaction listings for that kind of item. But what if the item is a commodity? If the item depends on the price of soybean or flour, the current and past harvest quantities could enrich the model. If the product is a medium-class item, current inflation and salary changes could also correlate with the current earnings.

Understanding the problem involves some business knowledge and looking to gather all the valuable sources of information that could influence the model. In some sense, it is more of an art form, and this doesn't change its importance a little bit.

Let's then assume that the basics of the problem have been analyzed, and the behavior and characteristics of the incoming data and desired output are clearer. The following task will generate an organized and structured set of values that will be the input to our model. This group

of data, after a process of cleaning and adapting, will be called our dataset.

Dataset definition and retrieval

Once we have identified the data sources, the next task is to gather all the tuples or records as a homogeneous set. The format can be a tabular arrangement, a series of real values (such as audio or weather variables), and N-dimensional matrices (a set of images or cloud points), among other types.

The ETL process

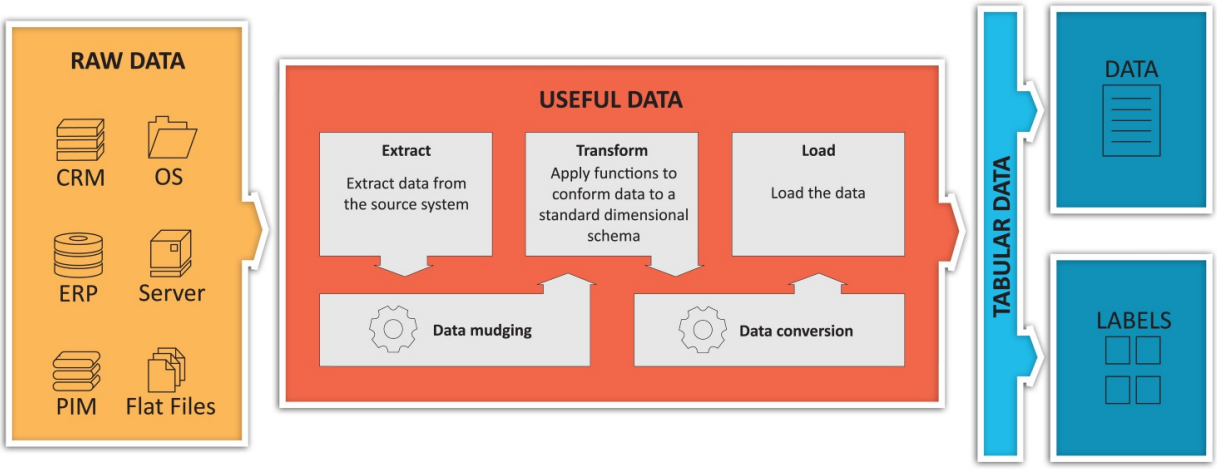
The previous stages in the big data processing field evolved over several decades under the name of data mining, and then adopted the popular name of **big data**.

One of the best outcomes of these disciplines is the specification of the **Extraction**, **Transform**, **Load** (**ETL**) process.

This process starts with a mix of many data sources from business systems, then moves to a system that transforms the data into a readable state, and then finishes by generating a data mart with very structured and documented data types.

For the sake of applying this concept, we will mix the elements of this process with the final outcome of a structured dataset, which includes in its final form an additional label column (in the case of supervised learning problems).

This process is depicted in the following diagram:



Depiction of the ETL process, from raw data to a useful dataset

The diagram illustrates the first stages of the data pipeline, starting with all the organization's data, whether it is commercial transactions, IoT device raw values, or other valuable data sources' information elements, which are commonly in very different types and compositions. The ETL process is in charge of gathering the raw information from them using different software filters, applying the necessary transforms to arrange the data in a useful manner, and finally, presenting the data in tabular format (we can think of this as a single database table with

a last feature or result column, or a big CSV file with consolidated data). The final result can be conveniently used by the following processes without practically thinking of the many quirks of data formatting, because they have been standardized into a very clear table structure.

Loading datasets and doing exploratory analysis with SciPy and pandas

In order to get a practical overview of some types of dataset formats, we will usethepreviously presented Python libraries (SciPy and pandas) for this example, given their almost universal use.

Let's begin by importing and performing a simple statistical analysis of several dataset input formats.

Note

The sample data files will be in the data directory inside each chapter's code directory.

Working interactively with IPython

In this section, we will introduce **Python interactive console**, or **IPython**, a command-line shell that allows us to explore concepts and methods in an interactive way.

To run IPython, you call it from the command line:

```
$ ipython
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

Here we see IPython executing, and then the initial quick help. The most interesting part is the last line - it will allow you to import libraries and execute commands and will show the resulting objects.An additional and convenient feature of IPython is that you can redefine variables on the fly to see how the results differ with different inputs.

In the current examples, we are using the standard Python version for the most supported Linux distribution at the time of writing (Ubuntu 16.04). The examples should be equivalent for Python 3.

First of all, let's import pandas and load a sample `.csv` file (a very common format with one row per line, and registers). It contains a very famous dataset for classification problems with the dimensions of the attributes of 150 instances of iris plants, with a numerical column indicating the class (1, 2, or 3):

```
In [1]: import pandas as pd #Import the pandas library with pd alias
```

In this line, we import pandas in the usual way, making its method available for use with the `import` statement. The `as`modifier allows us to use a succinct name for all objects and methods in the library:

```
In [2]: df = pd.read_csv ("data/iris.csv") #import iris data as dataframe
```

In this line, we use the `read_csv` method, allowing pandas to guess the possible item separator for the `.csv` file, and storing it in a `dataframe` object.

Let's perform some simple exploration of the dataset:

```
In [3]: df.columns
Out[3]:
Index([u'Sepal.Length', u'Sepal.Width', u'Petal.Length', u'Petal.Width',
u'Species'],
dtype='object')

In [4]: df.head(3)
Out[4]:
5.1 3.5 1.4 0.2 setosa
0 4.9 3.0 1.4 0.2 setosa
1 4.7 3.2 1.3 0.2 setosa
2 4.6 3.1 1.5 0.2 setosa
```

We are now able to see the column names of the dataset and explore the first *n* instances of it. Looking at the first registers, you can see the varying measures for the `setosa` iris class.

Now, let's access a particular subset of columns and display the first three elements:

```
In [19]: df[u'Sepal.Length'].head(3)
Out[19]:
0 5.1
1 4.9
2 4.7
Name: Sepal.Length, dtype: float64
```

Note

Pandas includes many related methods for importing tabulated data formats, such as HDF5 (`read_hdf`), JSON (`read_json`), and Excel (`read_excel`). For a complete list of formats, visit http://pandas.pydata.org/pandas-docs/stable/io.html.

In addition to these simple exploration methods, we will now use pandas to get all the descriptive statistics concepts we've seen in order to characterize the distribution of the `Sepal.Length` column:

```
#Describe the sepal length column
print "Mean: " + str (df[u'Sepal.Length'].mean())
print "Standard deviation: " + str(df[u'Sepal.Length'].std())
print "Kurtosis: " + str(df[u'Sepal.Length'].kurtosis())
print "Skewness: " + str(df[u'Sepal.Length'].skew())
```

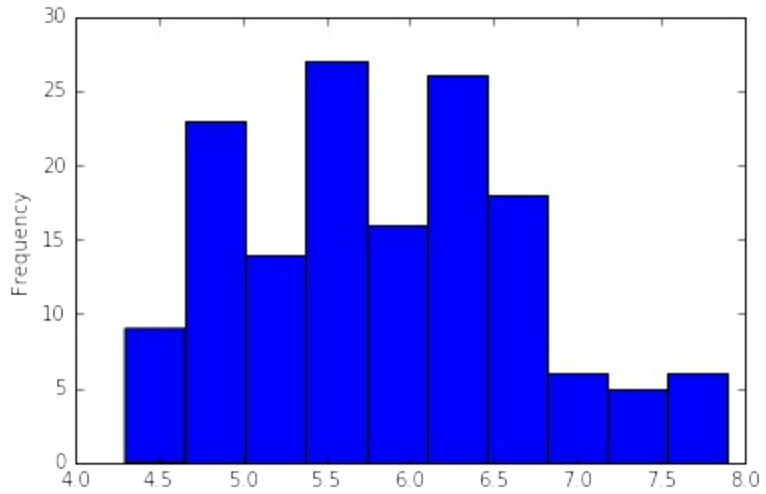
And here are the main metrics of this distribution:

```
Mean: 5.84333333333
Standard deviation: 0.828066127978
Kurtosis: -0.552064041316
Skewness: 0.314910956637
```

Now we will graphically evaluate the accuracy of these metrics by looking at the histogram of

this distribution, this time using the built-in `plot.hist` method:

```
#Plot the data histogram to illustrate the measures
import matplotlib.pyplot as plt
%matplotlib inline
df[u'Sepal.Length'].plot.hist()
```



Histogram of the Iris Sepal Length

As the metrics show, the distribution is right skewed, because the skewness is positive, and it is of the plainly distributed type (has a spread much greater than 1), as the kurtosis metrics indicate.

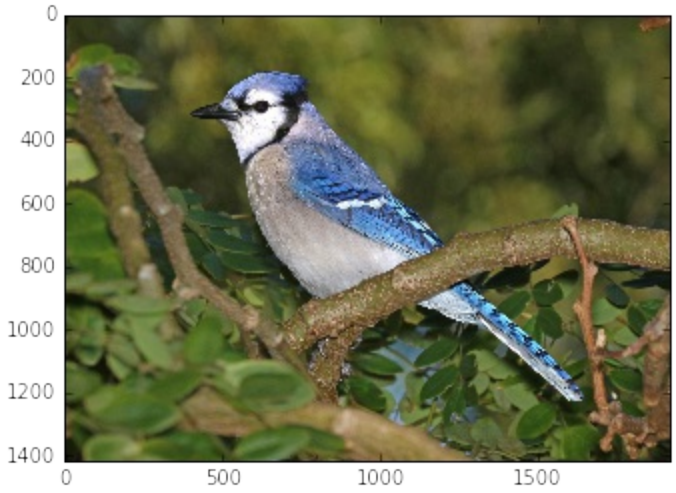
Working on 2D data

Let's stop here for tabular data, and go for 2D data structures. As images are the most commonly used type of data in popular machine learning problems, wewillshow you some useful methods included in the SciPy stack.

The following code is optimized to run on the Jupyter notebook with inline graphics. You will find the source code in the source file, `Dataset_IO.pynb`:

```
import scipy.misc
from matplotlib import pyplot as plt
%matplotlib inline
testimg = scipy.misc.imread("data/blue_jay.jpg")
plt.imshow( testimg)
```

Importing a single image basically consists of importing the corresponding modules, using the `imread` method to read the indicated image into a matrix, and showing it using matplotlib. The `%` starting line corresponds to a parameter modification and indicates that the following`matplotlib`graphics should be shown inline on the notebook, with the following results (the axes correspond to pixel numbers):



Initial RGB image loaded

The testing variable will contain a height * width * channel number array, with all the red, green, and blue values for each image pixel. Let's get this information:

```
testimg.shape
```

The interpreter will display the following:

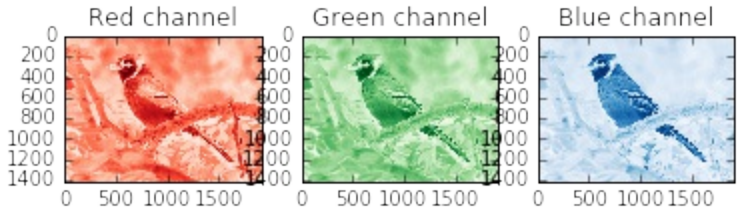
```
(1416, 1920, 3)
```

We could also try to separate the channels and represent them separately, with red, green, and blue scales, to get an idea of the color patterns in the image:

```
plt.subplot(131)
plt.imshow( testimg[:,:,0], cmap="Reds")
plt.title("Red channel")
plt.subplot(132)
plt.imshow( testimg[:,:,1], cmap="Greens")
plt.title("Green channel")
plt.subplot(133)
plt.imshow( testimg[:,:,2], cmap="Blues")
plt.title("Blue channel")
```

In the previous example, we create three subplots indicating the structure and position with a three-digit code. The first indicates the row number, the second indicates the column number, and the last, the plot position on that structure. The `cmap` parameter indicates the colormap assigned to each graphic.

The output will be as follows:



Depiction of the separated channels of the sample image.

Note

Note that red and green channels share a similar pattern, while the blue tones are predominant in this bird figure. This channel separation could be an extremely rudimentary preliminary way to detect this kind of bird in its habitat.

This section is a simplified introduction to the different methods of loading datasets. In the following chapters, we will see different advanced ways to get the datasets, including loading and training the different batches of sample sets.

Feature engineering

Feature engineering is in some ways one of the most underrated parts of the machine learning process, even though it is considered the cornerstone of the learning process by many prominent figures of the community.

What's the purpose of this process? In short, it takes the raw data from databases, sensors, archives, and so on, and transforms it in a way that makes it easy for the model to generalize. This discipline takes criteria from many sources, including common sense. It's indeed more like an art than a rigid science. It is a manual process, even when some parts of it can be automatized via a group of techniques grouped in the feature extraction field.

As part of this process we also have many powerful mathematical tools and dimensionality reduction techniques, such as **Principal Component Analysis** (**PCA**) and **Autoencoders**, that allow data scientists to skip features that don't enrich the representation of the data in useful ways.

Imputation of missing data

When dealing with not-so-perfect or incomplete datasets, a missing register may not add value to the model in itself, but all the other elements of the row could be useful to the model. This is especially true when the model has a high percentage of incomplete values, so no row can be discarded.

The main question in this process is *"how do you interpret a missing value?"* There are many ways, and they usually depend on the problem itself.

A very naive approach could be set the value to zero, supposing that the mean of the data distribution is 0. An improved step could be to relate the missing data with the surrounding content, assigning the average of the whole column, or an interval of *n* elements of the same columns. Another option is to use the column's median or most frequent value.

Additionally, there are more advanced techniques, such as robust methods and even k-nearest neighbors, that we won't cover in this book.

One hot encoding

Numerical or categorical information can easily be normally represented by integers, one for each option or discrete result. But there are situations where `bins` indicating the current option are preferred. This form of data representation is called **one hot encoding**. This encoding simply transforms a certain input into a binary array containing only zeros, except for the value indicated by the value of a variable, which will be one.

In the simple case of an integer, this will be the representation of the list [1, 3, 2, 4] in one hot encoding:

```
[[0 1 0 0 0]
[0 0 0 1 0]
```

```
[0 0 1 0 0]
[0 0 0 0 1]]
```

Let's perform a simple implementation of a one hot integer encoder for integer arrays, in order to better understand the concept:

```
import numpy as np
def get_one_hot(input_vector):
result=[]
for i in input_vector:
 newval=np.zeros(max(input_vector))
 newval.itemset(i-1,1)
 result.append(newval)
 return result
```

In this example, we first define the `get_one_hot` function, which takes an array as input and returns an array.

What we do is take the elements of the arrays one by one, and for each element in it, we generate a zero array with length equal to the maximum value of the array, in order to have space for all possible values. Then we insert `1` on the index position indicated by the current value (we subtract `1` because we go from 1-based indexes to 0-based indexes).

Let's try the function we just wrote:

```
get_one_hot([1,5,2,4,3])

#Out:
[array([ 1., 0., 0., 0., 0.]),
array([ 0., 0., 0., 0., 1.]),
array([ 0., 1., 0., 0., 0.]),
array([ 0., 0., 0., 1., 0.]),
array([ 0., 0., 1., 0., 0.])]
```

Dataset preprocessing

When we first dive into data science, a common mistake is expecting all the data to be very polished and with good characteristics from the very beginning. Alas, that is not the case for a very considerable percentage of cases, for many reasons such as null data, sensor errors that cause outliers and NAN, faulty registers, instrument-induced bias, and all kinds of defects that lead to poor model fitting and that must be eradicated.

The two key processes in this stage are data normalization and feature scaling. This process consists of applying simple transformations called **affine** that map the current unbalanced data into a more manageable shape, maintaining its integrity but providing better stochastic properties and improving the future applied model. The common goal of the standardization techniques is to bring the data distribution closer to a normal distribution, with the following techniques:

Normalization and feature scaling

One very important step in dataset preprocessing is normalization and feature scaling. Data normalization allows our optimization techniques, specially the iterative ones, to converge better, and makes the data more manageable.

Normalization or standardization

This technique aims to give the dataset the properties of a normal distribution, that is, a mean of 0 and a standard deviation of 1.

The way to obtain these properties is by calculating the so-called *z* scores, based on the dataset samples, with the following formula:

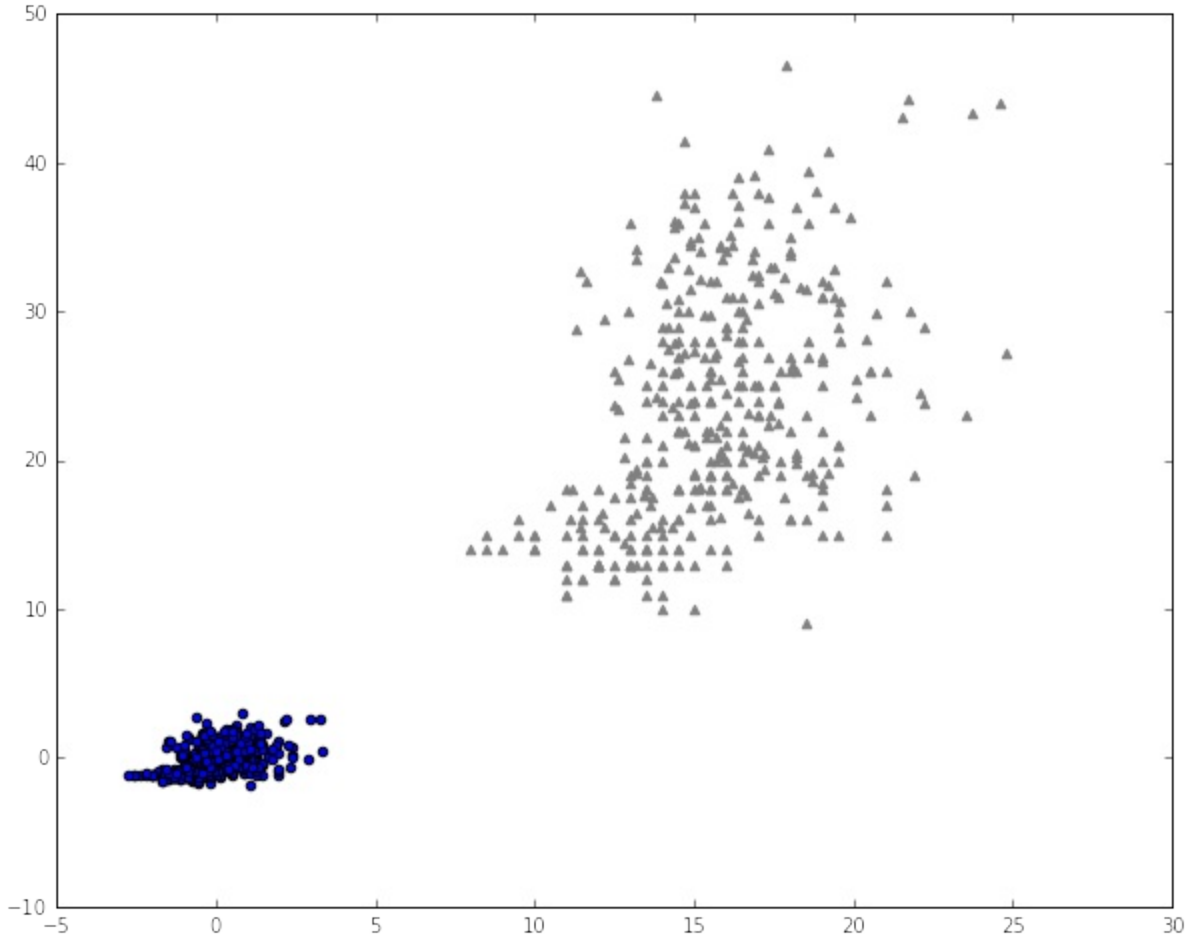
$$z = \frac{x - \mu}{\sigma}$$

Let's visualize and practice this new concept with the help of scikit-learn, reading a file from the `MPG` dataset, which contains city-cycle fuel consumption in miles per gallon, based on the following features: `mpg`, `cylinders`, `displacement`, `horsepower`, `weight`, `acceleration`, `model year`, `origin`, and `car name`.

```
from sklearn import preprocessing
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

df=pd.read_csv("data/mpg.csv")
plt.figure(figsize=(10,8))
print df.columns
partialcolumns = df[['acceleration', 'mpg']]
std_scale = preprocessing.StandardScaler().fit(partialcolumns)
df_std = std_scale.transform(partialcolumns)
```

```
plt.scatter(partialcolumns['acceleration'], partialcolumns['mpg'], color="grey", marker='^')
plt.scatter(df_std[:,0], df_std[:,1])
```

The following picture allows us to compare the non normalized and normalized data representations:



Depiction of the original dataset, and its normalized counterpart.

Note

It's very important to have an account of the denormalizing of the resulting data at the time of evaluation so that you do not lose the representative of the data, especially if the model is applied to regression, when the regressed data won't be useful if not scaled.

Model definition

If we wanted to summarize the machine learning process using just one word, it would certainly be models. This is because what we build with machine learning are abstractions or models representing and simplifying reality, allowing us to solve real-life problems based on a model that we have trained on.

The task of choosing which model to use is becoming increasingly difficult, given the increasing number of models appearing almost every day, but you can make general approximations by grouping methods by the type of task you want to perform and also the type of input data, so that the problem is simplified to a smaller set of options.

Asking ourselves the right questions

At the risk of generalizing too much, let's try to summarize a sample decision problem for a model:

- Are we trying to characterize data by simply grouping information based on its characteristics, without any or a few previous hints? This is the domain of clustering techniques.
- The first and most basic question: are we trying to predict the instant outcome of a variable, or to tag or classify data into groups? If the former, we are tackling a regression problem. If the latter, this is the realm of classification problems.
- Having the former questions resolved, and opting for any of the options of point 2, we should ask ourselves: is the data sequential, or rather, should we take the sequence in account? Recurrent neural networks should be one of the first options.
- Continuing with non-clustering techniques: is the data or pattern to discover spatially located? Convolutional neural networks are a common starting point for this kind of problem.
- In the most common cases (data without a particular arrangement), if the function can be represented by a single univariate or multivariate function, we can apply linear, polynomial, or logistic regression, and if we want to upgrade the model, a multilayer neural network will provide support for more complex non-linear solutions.
- How many dimensions and variables are we working on? Do we just want to extract the most useful features (and thus data dimensions), excluding the less interesting ones? This is the realm of dimensionality reduction techniques.
- Do we want to learn a set of strategies with a finite set of steps aiming to reach a goal? This belongs to the field of reinforcement learning. If none of these classical methods are fit for our research, a very high number of niche techniques appear and should be subject to additional analysis.

Note

In the following chapters, you will find additional information about how to base your decision on stronger criteria, and finally apply a model to your data. Also, if you see your answers don't relate well with the simple criteria explained in this

section, you can check *Chapter 8*, *Recent Models and Developments*, for more advanced models.

Loss function definition

This machine learning process step is also very important because it provides a distinctive measure of the quality of your model, and if wrongly chosen, it could either ruin the accuracy of the model or its efficiency in the speed of convergence.

Expressed in a simple way, the loss function is a function that measures the distance from the model's estimated value to the real expected value.

An important fact that we have to take into account is that the objective of almost all of the models is to minimize the error function, and for this, we need it to be differentiable, and the derivative of the error function should be as simple as possible.

Another fact is that when the model gets increasingly complex, the derivative of the error will also get more complex, so we will need to approximate solutions for the derivatives with iterative methods, one of them being the well-known gradient descent.

Model fitting and evaluation

In this part of the machine learning process, we have the model and data ready, and we proceed to train and validate our model.

Dataset partitioning

At the time of training the models, we usually partition all the provided data into three sets: the training set, which will actually be used to adjust the parameters of the models; the validation set, which will be used to compare alternative models applied to that data (it can be ignored if we have just one model and architecture in mind); and the test set, which will be used to measure the accuracy of the chosen model. The proportions of these partitions are normally 70/20/10.

Common training terms – iteration, batch, and epoch

When training the model, there are some common terms that indicate the different parts of the iterative optimization:

- An **iteration** defines one instance of calculating the error gradient and adjusting the model parameters. When the data is fed into groups of samples, each one of these groups is called a **batch**.
- Batches can include the whole dataset (traditional batching), or include just a tiny subset until the whole dataset is fed forward, called mini-batching. The number of samples per batch is called the **batch size**.
- Each pass of the whole dataset is called an **epoch**.

Types of training – online and batch processing

The training process provides many ways of iterating over the datasets and adjusting the parameters of the models according to the input data and error minimization results.

Of course, the dataset can and will be evaluated many times and in a variety of ways during the training phase.

Parameter initialization

In order to assure a good fitting start, the model weights have to be initialized to the most effective values. Neural networks, which normally have a *tanh* activation function, are mainly sensitive to the range [-1,1], or [0,1]; for this reason, it's important to have the data normalized, and the parameters should also be within that range.

The model parameters should have useful initial values for the model to converge. One important decision at the start of training is the initialization values for the model parameters (commonly called **weights**). A canonical initial rule is not initializing variables at 0 because it prevents the models from optimizing, as they do not have a suitable function slope multiplier to adjust. A common sensible standard is to use a normal random distribution for all the values.

Using NumPy, you would normally initialize a coefficient vector with the following code:

```
mu,sigma=0, 1
dist =np.random.normal(mu,sigma,1000)
>>> dist = np.random.normal(mu, sigma, 10)
>>> print dist
[ 0.32416595 1.48067723 0.23039378 -0.59140674 1.65827372 -0.8241832
 0.86016434 -0.05996878 2.2855467 -0.19759244]
```

Note

One particular source of problems at this stage is setting all of the model's parameters to zero. As many optimization techniques normally multiply the weights by a determinate coefficient to the approximate minimum, multiplying by zero will prevent any change in the model, excepting the bias terms.

Model implementation and results interpretation

No model is practical if it can't be used outside the training and test sets. This is when the model is deployed into production.

In this stage, we normally load all the model's operation and trained weights, wait for new unknown data, and when it arrives, we feed it through all the chained functions of the model, informing the outcomes of the output layer or operation via a web service, printing to standard output, and so on.

Then, we will have a final task - to interpret the results of the model in the real world to constantly check whether it works in the current conditions. In the case of generative models, the suitability of the predictions is easier to understand because the goal is normally the representation of a previously known entity.

Regression metrics

For regression metrics, a number of indicators are calculated to give a succinct idea of the fitness of the regressed model. Here is a list of the main metrics.

Mean absolute error

The `mean_absolute_error` function computes mean absolute error, a risk metric corresponding to the expected value of the absolute error loss, or *l1-norm* loss. If $\hat{y}_i$ is the predicted value of the *i*th sample, and y_i is the corresponding true value, then the **mean absolute error** (**MAE**) estimated over *n* samples is defined as follows:

$$MAE\left(y, \hat{y}\right) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} \left|y_i - \hat{y}_i\right|$$

Median absolute error

The median absolute error is particularly interesting because it is robust to outliers. The loss is calculated by taking the median of all absolute differences between the target and the prediction. If $\hat{y}$ is the predicted value of the *i*th sample and y_i is the corresponding true value, then the median absolute error estimated over *n* samples is defined as follows:

$$MedAE\left(y, \hat{y}\right) = median\left(\left|y_1 - \hat{y}_1\right|, \ldots, \left|y_n - \hat{y}_n\right|\right)$$

Mean squared error

The **mean squared error** (**MSE**) is a risk metric equal to the expected value of the squared (quadratic) error loss.

If $\hat{y}_i$ is the predicted value of the *i*th sample and y_i is the corresponding true value, then the MSE estimated over *n* samples is defined as follows:

$$MSE(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} (y_i - \hat{y}_i)^2$$

Classification metrics

The task of classification implies different rules for the estimation of the error. The advantage we have is that the number of outputs is discrete, so it can be determined exactly whether a prediction has failed or not in a binary way. That leads us to the main indicators.

Accuracy

The accuracy calculates either the fraction or the count of correct predictions for a model. In multi-label classification, the function returns the subset's accuracy.

If the entire set of predicted labels for a sample strictly matches the true set of labels, then the subset's accuracy is *1.0*; otherwise, it is *0.0*. If $\hat{y}_i$ is the predicted value of the *i*th sample and y_i is the corresponding true value, then the fraction of correct predictions over *n* samples is defined as follows:

$$accuracy(y, \hat{y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} 1(\hat{y}_i = y_i)$$

Precision score, recall, and F-measure

Precision score is as follows:

$$precision = \frac{tp}{tp + fp}$$

Here, t_p is the number of true positives and f_p is the number of false positives. The precision is the ability of the classifier to not label as positive a sample that is negative. The best value is 1 and the worst value is *0*.

Recall is as follows:

$$recall = \frac{tp}{tp + fn}$$

Here, t_p is the number of true positives and f_n is the number of false negatives. The recall can be described as the ability of the classifier to find all the positive samples. Its values range from 1 (optimum) to zero.

F measure (F_β and F_1 measures) can be interpreted as a special kind of mean (weighted harmonic mean) of the precision and recall. A F_β measure's best value is 1 and its worst score is 0. With $\beta = 1$, F_β and F_1 are equivalent, and the recall and the precision are equally important:

$$F_\beta = \left(1 + \beta^2\right) \frac{precision \cdot recall}{\beta^2 precision + recall}$$

Confusion matrix

Every classification task aims to predict a label or tag for new unknown data. A very efficient way of showing the classification's accuracy is through a confusion matrix, where we show [classified sample, ground truth] pairs and a detailed view of how the predictions are doing.

The expected output should be the main diagonal of the matrix with a 1.0 score; that is, all the expected values should match the real ones.

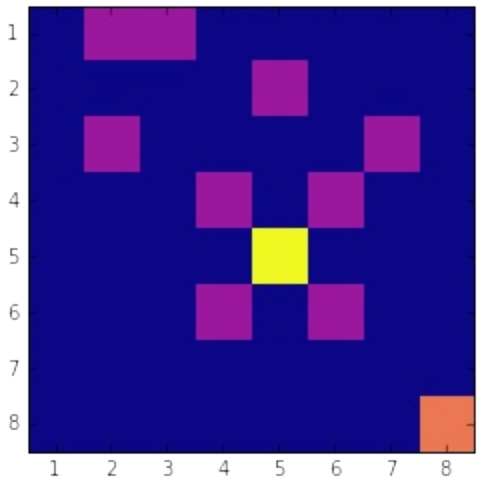
In the following code example, we will do a synthetic sample of predictions and real values, and generate a confusion matrix of the final data:

```
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import numpy as np
y_true = [8,5,6,8,5,3,1,6,4,2,5,3,1,4]
y_pred = [8,5,6,8,5,2,3,4,4,5,5,7,2,6]
y = confusion_matrix(y_true, y_pred)
print y
plt.imshow(confusion_matrix(y_true, y_pred), interpolation='nearest', cmap='plasma')
plt.xticks(np.arange(0,8), np.arange(1,9))
plt.yticks(np.arange(0,8), np.arange(1,9))
plt.show()
```

The result will be the following:

```
[[0 1 1 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 1 0 0 0 0 1 0]
 [0 0 0 1 0 1 0 0]
 [0 0 0 0 3 0 0 0]
 [0 0 0 1 0 1 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 2]]
```

And the final confusion matrix graphic representation for these values will be as follows:



Confusion matrix

In the image, we see the high accuracy value in the (*5*,*5*) diagonal value with three correct predictions, and the (*8*,*8*) value with two. As we can see, the distribution of the accuracy by value can be intuitively extracted just by analyzing the graph.

Clustering quality measurements

Unsupervised learning techniques, understood as the labeling of data without a ground truth, makes it a bit difficult to implement significant metrics for the models. Nevertheless, there are a number of measures implemented for this kind of technique. In this section, we have a list of the most well-known ones.

Silhouette coefficient

The **silhouettecoefficient** is a metric that doesn't need to know the labeling of the dataset. It gives an idea of the separation between clusters.

It is composed of two different elements:

- The mean distance between a sample and all other points in the same class (*a*)
- The mean distance between a sample and all other points in the nearest cluster (*b*)

The formula for this coefficient *s* is defined as follows:

$$s = \frac{b - a}{\max(a, b)}$$

Note

The silhouette coefficient is only defined if the number of classes is at least two, and the coefficient for a whole sample set is the mean of the coefficient for all samples.

Homogeneity, completeness, and V-measure

Homogeneity, completeness, and V-measure are three key related indicators of the quality of a clustering operation. In the following formulas, we will use *K* for the number of clusters, *C* for the number of classes, *N* for the total number of samples, and a_{ck} for the number of elements of class *c* in cluster *k*.

Homogeneity is a measure of the ratio of samples of a single class pertaining to a single cluster. The fewer different classes included in one cluster, the better. The lower bound should be 0.0 and the upper bound should be 1.0 (higher is better), and the formulation for it is expressed as follows:

$$H(C) = -\sum_{c=1}^{|C|} \frac{\sum_{k=1}^{|K|} a_{ck}}{N} \log \frac{\sum_{k=1}^{|K|} a_{ck}}{N}$$

Completeness measures the ratio of the member of a given class that is assigned to the same cluster:

$$H(K) = -\sum_{k=1}^{|K|} \frac{\sum_{c=1}^{|C|} a_{ck}}{N} \log \frac{\sum_{c=1}^{|C|} a_{ck}}{N}$$

V-measure is the harmonic mean of homogeneity and completeness, expressed by the following formula:

$$V_\beta = (1+\beta) \frac{h \cdot c}{\beta \cdot h + c}$$

Summary

In this chapter, we reviewed all the main steps involved in a machine learning process. We will be, indirectly, using them throughout the book, and we hope they help you structure your future work too.

In the next chapter, we will review the programming languages and frameworks that we will be using to solve all our machine learning problems and become proficient with them before starting with the projects.

References

- Lichman, M. (2013). UCI Machine Learning Repository (http://archive.ics.uci.edu/ml). Irvine, CA: University of California, School of Information and Computer Science.
- Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In *Proceedings on the Tenth International Conference of Machine Learning*, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
- Townsend, James T. *Theoretical analysis of an alphabetic confusion matrix.* Attention, Perception, & Psychophysics 9.1 (1971): 40-50.
- Peter J. Rousseeuw (1987). *Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis*. Computational and Applied Mathematics 20: 53-65.
- Kent, Allen, et al, Machine literature searching VIII. *Operational criteria for designing information retrieval systems.* Journal of the Association for Information Science and Technology 6.2 (1955): 93-101.
- Rosenberg, Andrew, and Julia Hirschberg, V-Measure: *A Conditional Entropy-Based External Cluster Evaluation Measure*. EMNLP-CoNLL. Vol. 7. 2007.

Chapter 3. Clustering

Congratulations! You have finished this book's introductory section, in which you have explored a great number of topics, and if you were able to follow it, you are prepared to start the journey of understanding the inner workings of many machine learning models.

In this chapter, we will explore some effective and simple approaches for automatically finding interesting data conglomerates, and so begin to research the reasons for natural groupings in data.

This chapter will covers the following topics:

- A line-by-line implementation of an example of the K-means algorithm, with explanations of the data structures and routines
- A thorough explanation of the **k-nearest neighbors (K-NN)** algorithm, using a code example to explain the whole process
- Additional methods of determining the optimal number of groups representing a set of samples

Grouping as a human activity

Humans typically tend to agglomerate everyday elements into groups of similar features. This feature of the human mind can also be replicated by an algorithm. Conversely, one of the simplest operations that can be initially applied to any unlabeled dataset is to group elements around common features.

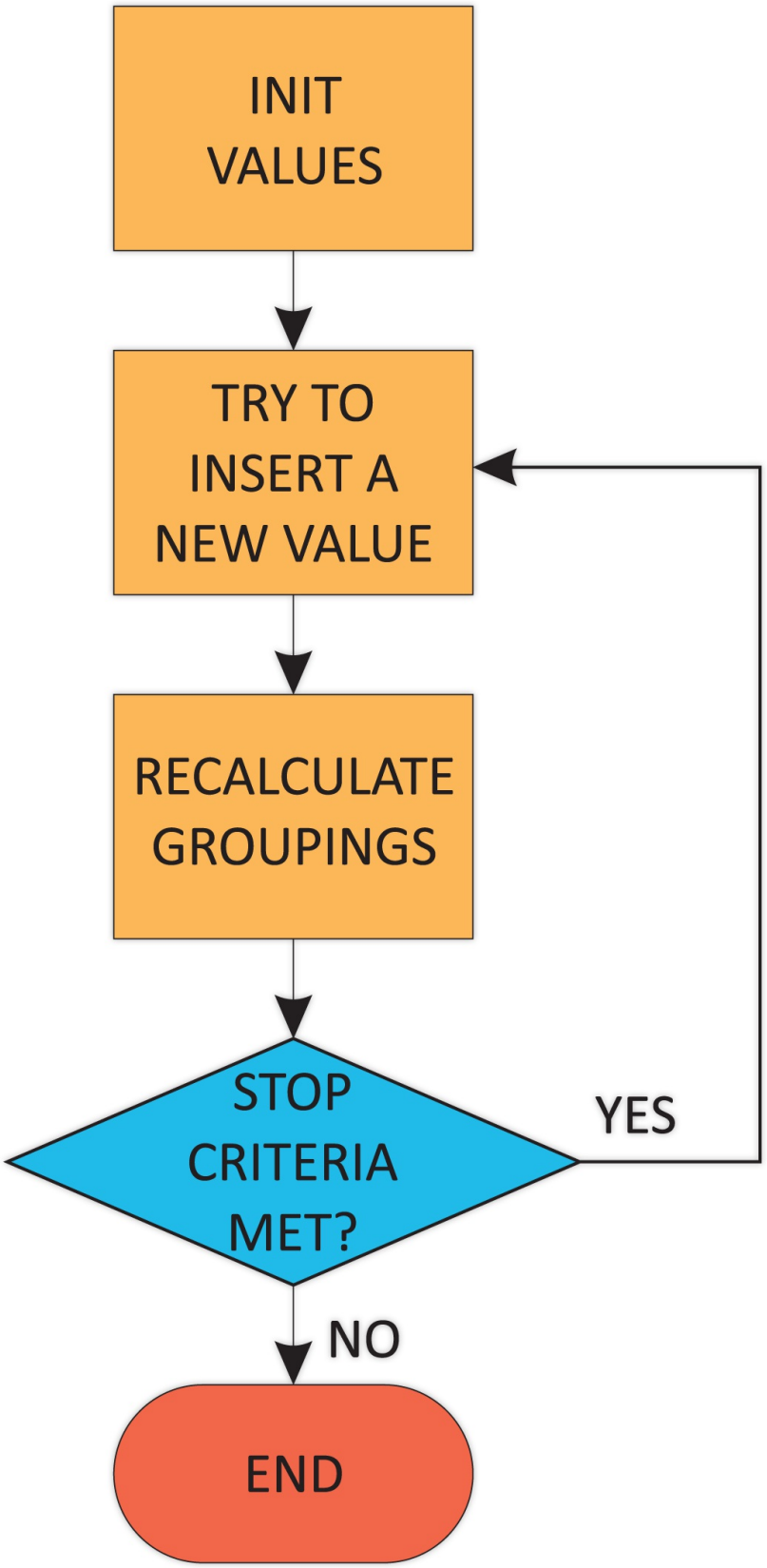
As we have described, in this stage of the development of the discipline, clustering is taught as an introductory theme that's applied to the simplest categories of element sets.

But as an author, I recommend researching this domain, because the community is hinting that the current model's performance will all reach a plateau, before aiming for the full generalization of tasks in AI. And what kinds of method are the main candidates for the next stages of crossing the frontier towards AI? Unsupervised methods, in the form of very sophisticated variations of the methods explained here.

But let's not digress for now, and let's begin with the simplest of grouping criteria, the distance to a common center, which is called **K-means**.

Automating the clustering process

The grouping of information for clustering follows a common pattern for all techniques. Basically, we have an initialization stage, followed by the iterative insertion of new elements, after which the new group relationships are updated. This process continues until the stop criteria is met, where the group characterization is finished. The following flow diagram illustrates this process:


INIT VALUES
TRY TO INSERT A NEW VALUE
RECALCULATE GROUPINGS
STOP CRITERIA MET?
YES
NO
END

General scheme for a clustering algorithm

After we get a clear sense of the overall process, let's start working with several cases where this scheme is applied, starting with **K-means**.

Finding a common center - K-means

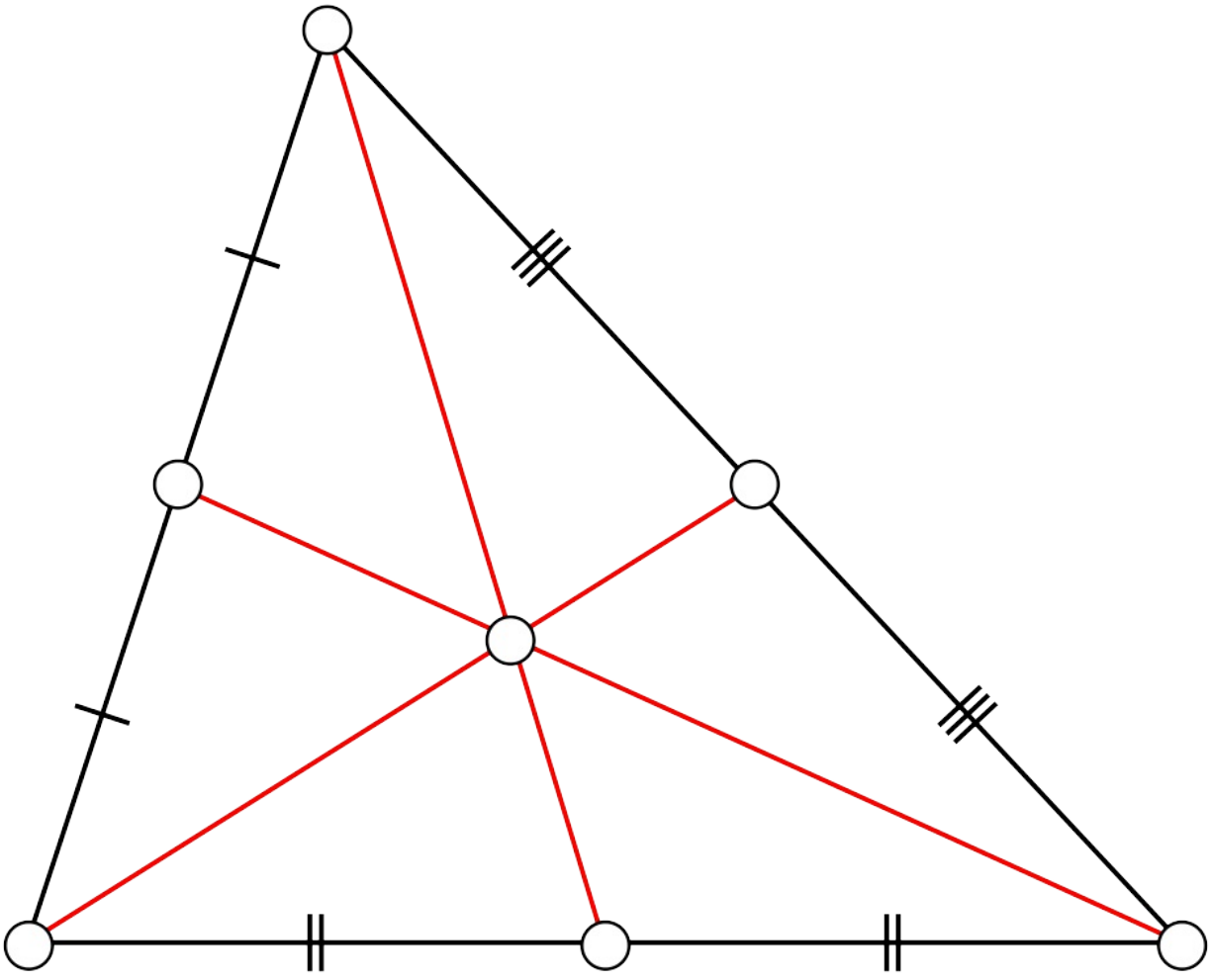
Here we go! After some necessary preparation review, we will finally start to learn from data; in this case, we are looking to label data we observe in real life.

In this case, we have the following elements:

- A set of N-dimensional elements of numeric type
- A predetermined number of groups (this is tricky because we have to make an educated guess)
- A set of common representative points for each group (called **centroids**)

The main objective of this method is to split the dataset into an arbitrary number of clusters, each of which can be represented by the mentioned centroids.

The word centroid comes from the mathematics world, and has been translated to calculus and physics. Here we find a classical representation of the analytical calculation of a triangle's centroid:



Graphical depiction of the centroid finding scheme for a triangle

The centroid of a finite set of k points, $x_1, x_2, ..., x_k$ in R^n, is as follows:

$$\mathbf{C} = \frac{\mathbf{x}_1 + \mathbf{x}_2 + \cdots + \mathbf{x}_k}{k}$$

Analytic definition of centroid

So, now that we have defined this central metric, let's ask the question, "*What does it have to do with the grouping of data elements?*"

To answer this, we must first understand the concept of **distance to a centroid.** Distance has many definitions, which could be linear, quadratic, and other forms.

So, let's review some of the main distance types, and then we will mention which one is normally used:

> ## Note
>
> In this review, we will work with 2D variables when defining the measure types, as a mean of simplification.

Let's take a look at the following distance types:

- **Euclidean distance:** This distance metric calculates the distance in the form of a straight line between two points, or has the following formula:

$$\sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2}$$

- **Chebyshev distance**: This distance is equivalent to the maximum distance, along any of the axes. It's also called the **chess** distance, because it gives the minimum quantity of moves a king needs to get from the initial point to the final point. Its defined by the following formula:

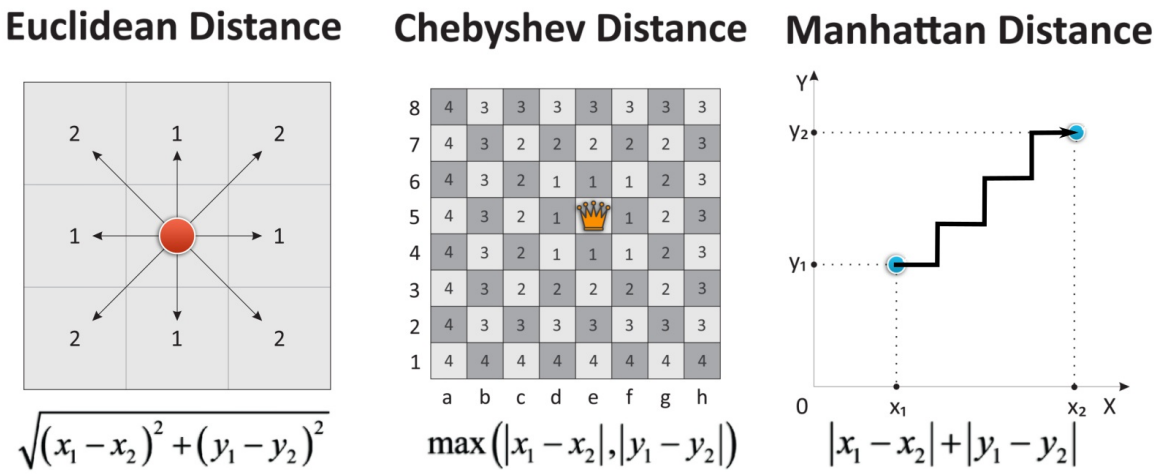
$$\max\left(\left|x_1 - x_2\right|, \left|y_1 - y_2\right|\right)$$

- **Manhattan distance**: This distance is the equivalent to going from one point to another in

a city, with unit squares. This L1-type distance sums the number of horizontal units advanced, and the number of vertical ones. Its formula is as follows:

$$|x_1 - x_2| + |y_1 - y_2|$$

The following diagram further explains the formulas for the different types of distance:



Graphical representation of some of the most well known distance types

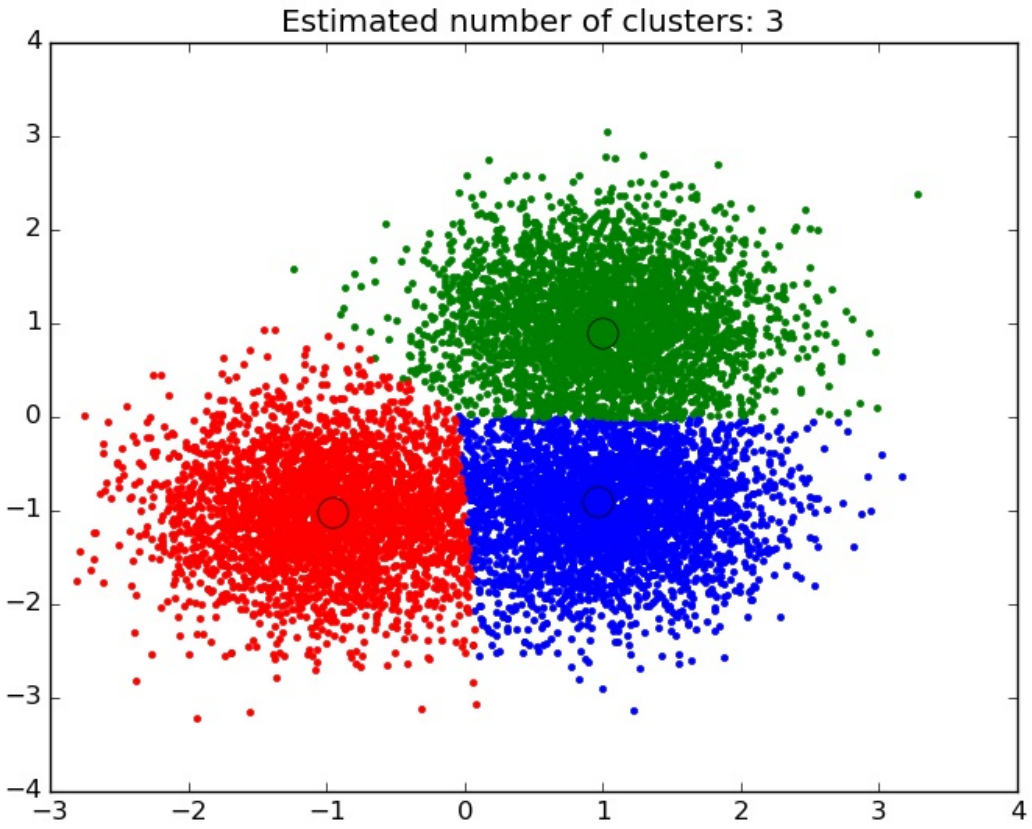
The distance metric chosen for K-means is the Euclidean distance, which is easy to compute and scales well to many dimensions.

Now that we have all the elements, it's time to define the criteria we will use to define which label we will assign to any given sample. Let's summarize the learning rule with the following statement:

"A sample will be assigned to the group represented by the closest centroid."

The goal of this method is to minimize the sum of squared distances from the cluster's members to the actual centroid of all clusters that contain samples. This is also known as **minimization of inertia.**

In the following diagram, we can see the results of a typical K-means algorithm applied to a sample population of blob-like groups, with a preset number of clusters of 3:



Typical result of a clustering process using K-means, with a seed of 3 centroids

K-means is a simple and effective algorithm that can be used to obtain a quick idea of how a dataset is organized. Its main differentiation is that objects belonging to the same class will share a common distance center, which will be incrementally upgraded with the addition of each new sample.

Pros and cons of K-means

The advantages of this method are as follows:

- It scales very well (most of the calculations can be run in parallel)
- It has been used in a very large range of applications

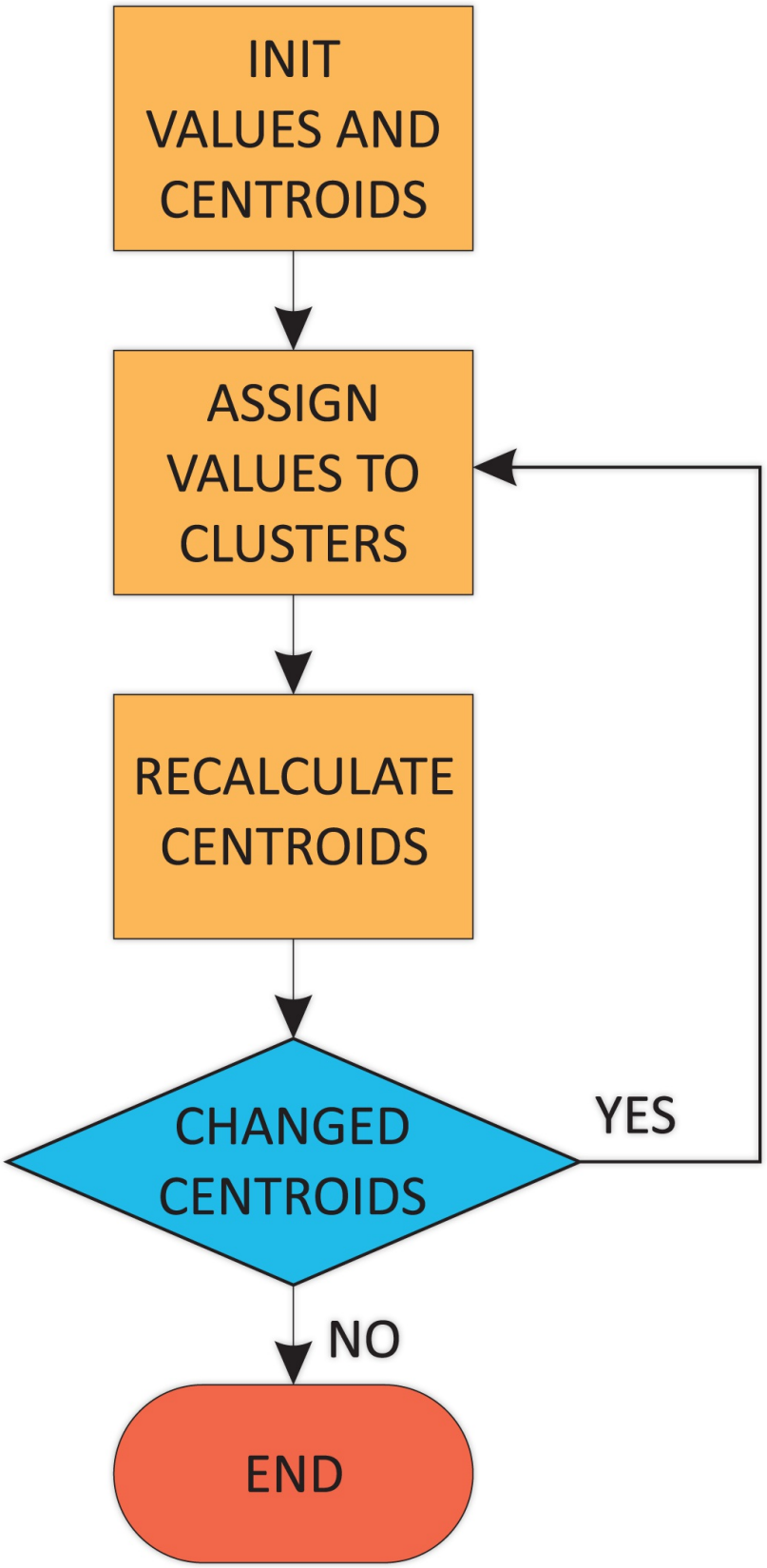
But simplicity also has some costs (the no silver bullet rule applies):

- It requires a priori knowledge (the number of possible clusters should be known beforehand)
- The outlier values can skew the values of the centroids, as they have the same weight as any other sample
- As we assume that the figure is convex and isotropic, it doesn’t work very well with non

blob alike clusters

K-means algorithm breakdown

The mechanism of the K-means algorithm can be summarized with the following flowchart:


INIT
VALUES AND
CENTROIDS
ASSIGN
VALUES TO
CLUSTERS
RECALCULATE
CENTROIDS
CHANGED
CENTROIDS
YES
NO
END

Flowchart of the K-means process

Let's describe the process in more detail:

We start with the unclassified samples and take K elements as the starting centroids. There are also possible simplifications of this algorithm that take the first element in the element list for the sake of brevity.

We then calculate the distances between the samples and the first chosen samples, and so we get the first calculated centroids (or other representative values). You can see in the moving centroids in the illustration a shift toward a more intuitive (and mathematically correct) center location.

After the centroids change, their displacement will cause the individual distances to change, and so the cluster membership may change.

This is the time when we recalculate the centroids and repeat the first steps, in case the stop condition isn’t met.

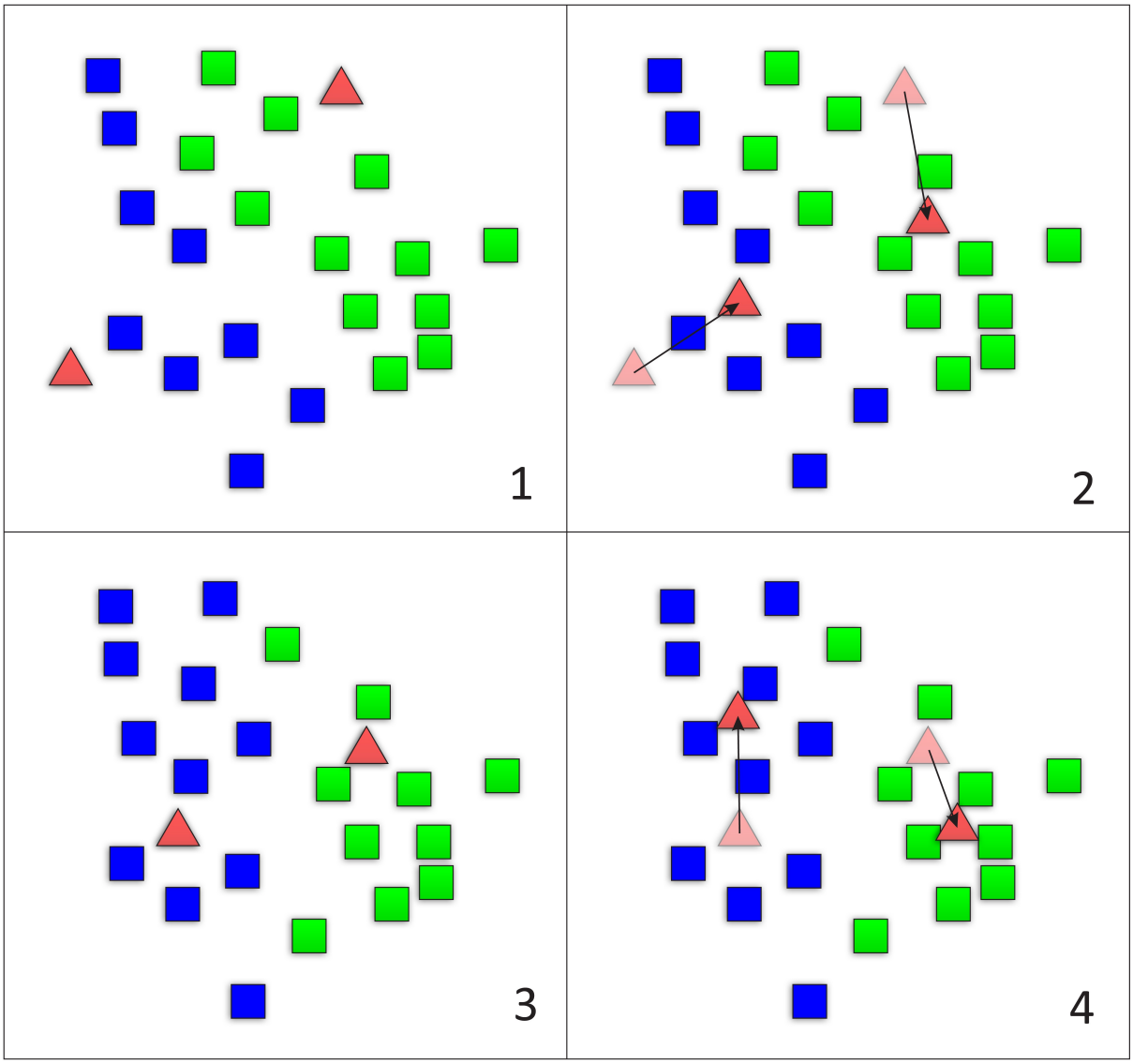
The stopping conditions can be of various types:

- After n-iterations. It could be that either we chose a very large number, and we’ll have unnecessary rounds of computing, or it could converge slowly, and we will have very unconvincing results if the centroid doesn't have a very stable mean.
- A possible better criterion for the convergence of the iterations is to take a look at the changes of the centroids, whether in total displacement or total cluster element switches. The last one is employed normally, so we will stop the process once there are no more elements changing from their current cluster to another one.

Note

The N iterations condition could also be used as a last resort, because it could lead to very long processes where no observable change is observed on a large number of iterations.

Let's try to summarize the process of K-NN clustering visually, going through a few steps and looking at how the clusters evolve through time:



Graphical example of the cluster reconfiguration loop

In subfigure 1, we start seeding the clusters with possible centroids at random places, to which we assign the closest data elements; and then in subfigure 2, we reconfigure the centroids to the center of the new clusters, which in turn reconfigures the clusters again (subfigure 3), until we reach a stationary status. The element aggregation could also be made element by element, which will trigger softer reconfigurations. This will be the implementation strategy for the practical part of this chapter.

K-means implementations

In this section, we will review the concept of K-means with a practical sample, from the very basic concepts.

First, we will import the libraries we need. In order to improve the understanding of the algorithms, we will use the `numpy` library. Then we will use the well-known `matplotlib` library for the graphical representation of the algorithms:

```
import numpy as np

import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline
```

These will be a number of 2D elements, and will then generate the candidate centers, which will be of four 2D elements.

In order to generate a dataset, normally a random number generator is used, but in this case we want to set the samples to predetermined numbers, for convenience, and also to allow you to repeat these procedures manually:

```
samples=np.array([[1,2],[12,2],[0,1],[10,0],[9,1],[8,2],[0,10],[1,8],[2,9],[9,9],[10,8],[8,9]
], dtype=np.float)
centers=np.array([[3,2], [2,6], [9,3], [7,6]], dtype=np.float)
N=len(samples)
```

Let's represent the sample's center. First, we will initialize a new `matplotlib` figure, with the corresponding axes. The `fig` object will allow us to change the parameters of all the figures.

The `plt` and `ax` variable names are a standarized way to refer to the plot in general and one of the plots' axes.

So let’s try to have an idea of what the samples look like. This will be done through the `scatter` drawing type of the `matplotlib` library. It takes as parameters the *x* coordinates, the *y* coordinates, size (in points squared), the marker type, and color.

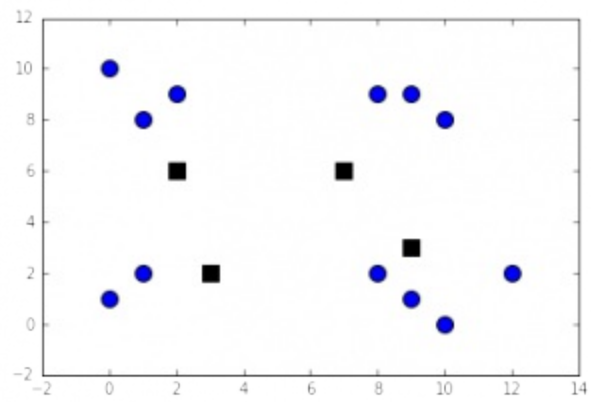
Note

There are a variety of markers to choose from, such as point (`.`), circle (`o`), square (`s`). To see the full list, visit https://matplotlib.org/api/markers_api.html.

Let's take a look at the following code snippet:

```
fig, ax = plt.subplots()
ax.scatter(samples.transpose()[0], samples.transpose()[1], marker =
'o', s = 100 )
ax.scatter(centers.transpose()[0], centers.transpose()[1], marker =
's', s = 100, color='black')
plt.plot()
```

Let's now take a look at the following graph:



Initial clusters status, Centers as black squares

Let’s define a function that, given a new sample, will return a list with the distances to all the current centroids in order to assign this new sample to one of them, and afterward, recalculate the centroids:

```
def distance (sample, centroids):
    distances=np.zeros(len(centroids))
    for i in range(0,len(centroids)):
        dist=np.sqrt(sum(pow(np.subtract(sample,centroids[i]),2)))
        distances[i]=dist
    return distances
```

Then we need a function that will build, one by one, the step-by-step graphic of our application.

It expects a maximum of 12 subplots, and the `plotnumber` parameter will determine the position on the 6 x 2 matrix (`620` will be the upper-left subplot, 621 the following to the right, and so on writing order). After that, for each picture we will do a scatterplot of the clustered samples, and then of the current centroid position:

```
def showcurrentstatus (samples, centers, clusters, plotnumber):
    plt.subplot(620+plotnumber)
    plt.scatter(samples.transpose()[0], samples.transpose()[1], marker =
    'o', s = 150 , c=clusters)
    plt.scatter(centers.transpose()[0], centers.transpose()[1], marker =
    's', s = 100, color='black')
    plt.plot()
```

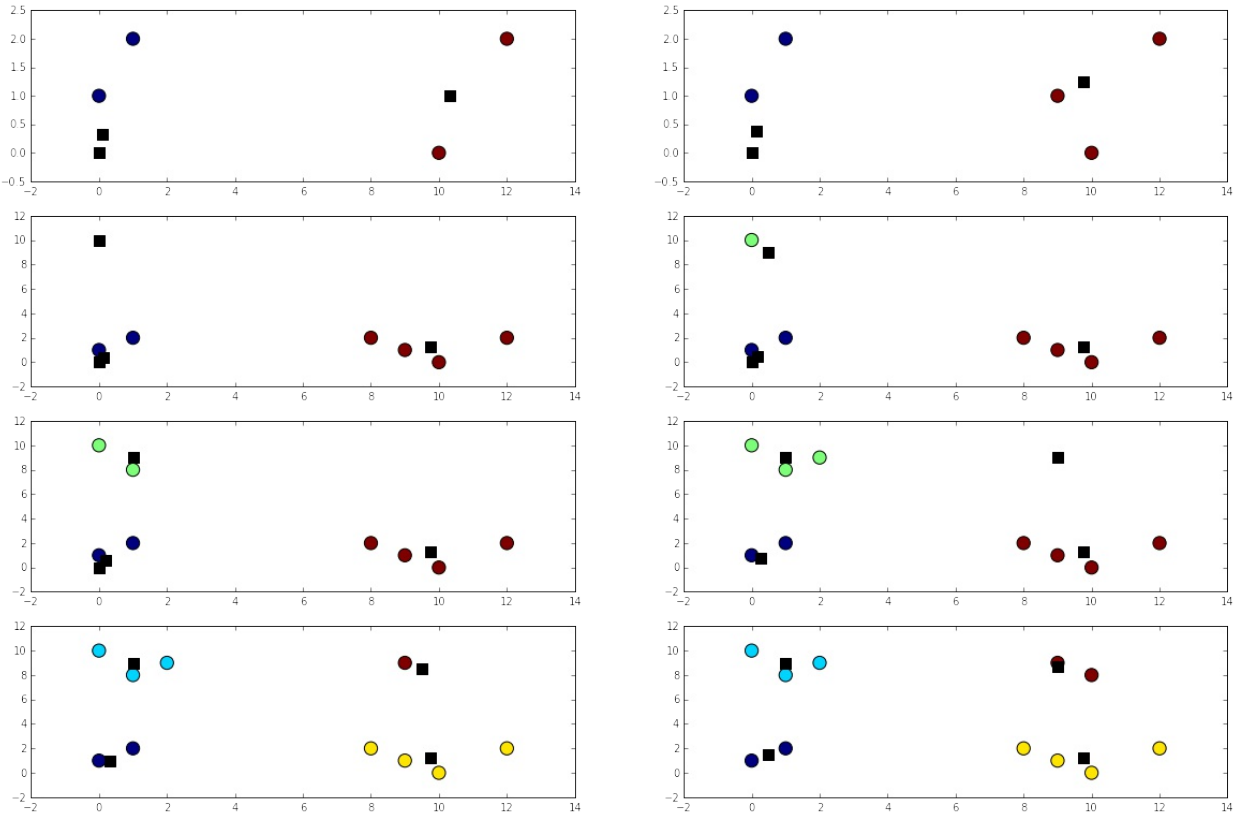
The following function, called `kmeans`, will use the previous distance function to store the centroid that the samples are assigned to (it will be a number from `1` to `K`). The main loop will go from sample `0` to `N`, and for each one, it will look for the closest centroid, assign the centroid number to index `n` of the clusters array, and sum the samples' coordinates to its currently assigned centroid. Then, to get the sample, we use the `bincount` method to count the number of samples for each centroid, and by building a `divisor` array, we divide the sum of a class elements by the previous `divisor` array, and there we have the new centroids:

```
def kmeans(centroids, samples, K, plotresults):
    plt.figure(figsize=(20,20))
    distances=np.zeros((N,K))
    new_centroids=np.zeros((K, 2))
    final_centroids=np.zeros((K, 2))
    clusters=np.zeros(len(samples), np.int)

    for i in range(0,len(samples)):
        distances[i] = distance(samples[i], centroids)
        clusters[i] = np.argmin(distances[i])
        new_centroids[clusters[i]] += samples[i]
        divisor = np.bincount(clusters).astype(np.float)
        divisor.resize([K])
        for j in range(0,K):
        final_centroids[j] = np.nan_to_num(np.divide(new_centroids[j] ,
        divisor[j]))
        if (i>3 and plotresults==True):
            showcurrentstatus(samples[:i], final_centroids,
            clusters[:i], i-3)
        return final_centroids
```

Now it’s time to kickstart the K-means process, using the initial samples and centers we set up at first. The current algorithm will show how the clusters are evolving, starting from a few elements, into their final state:

```
finalcenters=kmeans (centers, samples, 4, True)
```

Let's take a look at the following screenshot:



Depiction of the clustering process, with the centroids represented as black squares

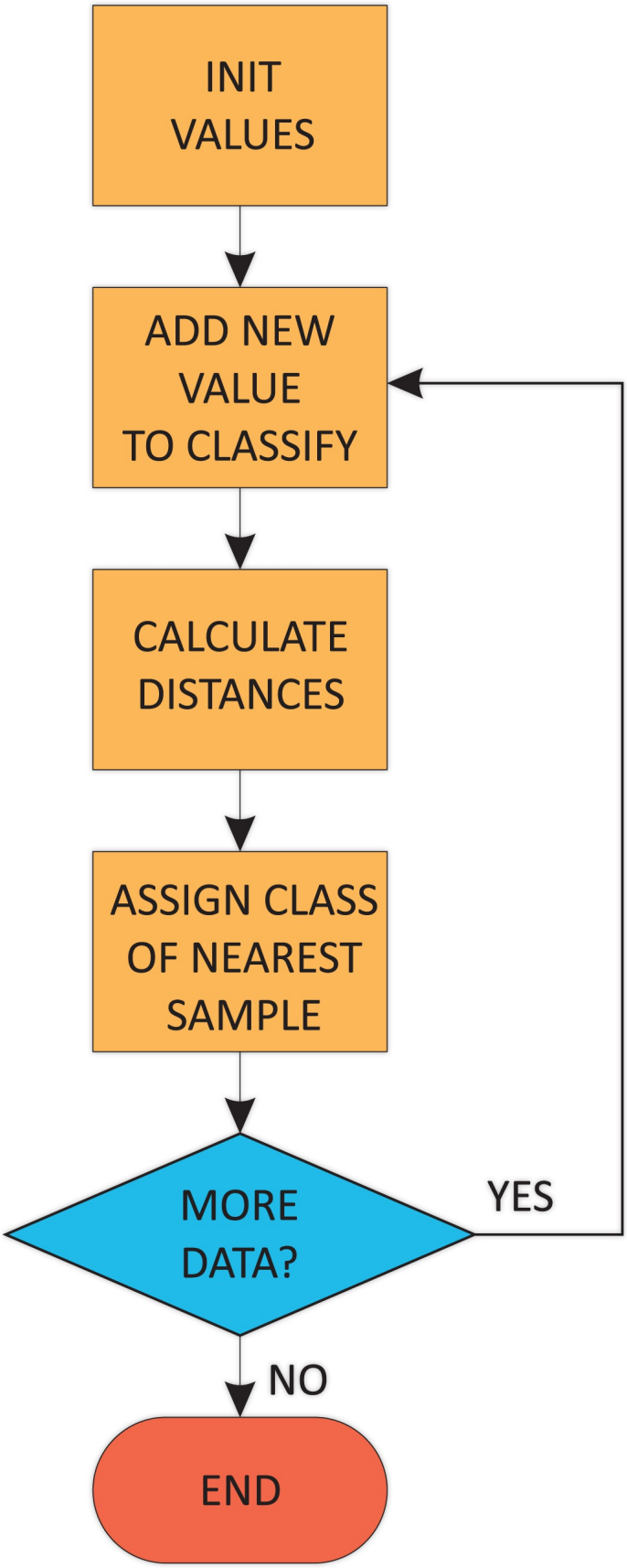
Nearest neighbors

K-NN is another classical method of clustering. It builds groups of samples, supposing that each new sample will have the same class as its neighbors, without looking for a global representative central sample. Instead, it looks at the environment, looking for the most frequent class on each new sample's environment.

Mechanics of K-NN

K-NN can be implemented in many configurations, but in this chapter we will use the semi-supervised approach, starting from a certain number of already assigned samples, and later guessing the cluster membership using the main criteria.

In the following diagram, we have a breakdown of the algorithm. It can be summarized with the following steps:

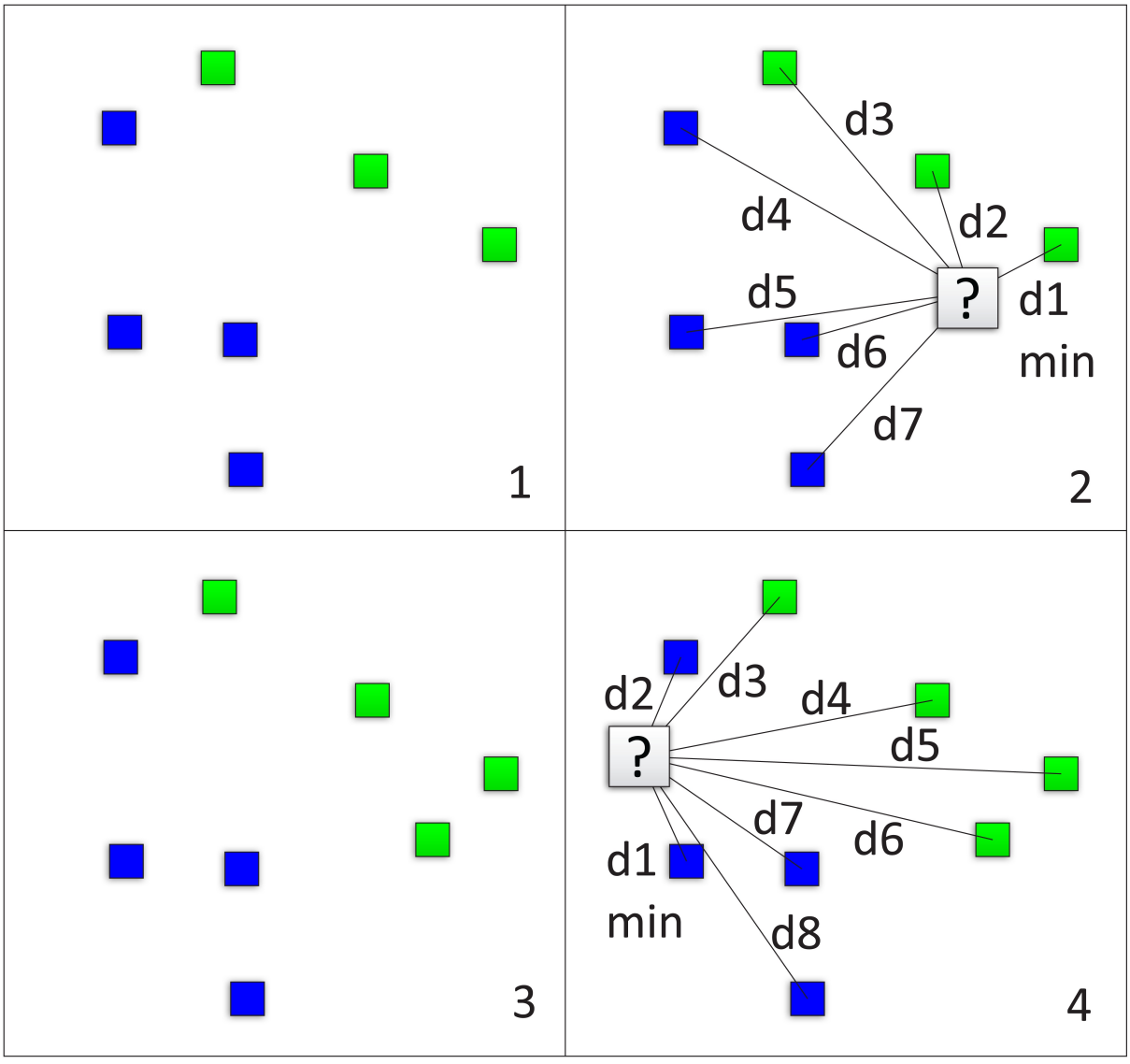

INIT
VALUES
ADD NEW
VALUE
TO CLASSIFY
CALCULATE
DISTANCES
ASSIGN CLASS
OF NEAREST
SAMPLE
MORE
DATA?
YES
NO
END

Flowchart for the K-NN clustering process

Let's go over all the following involved steps, in a simplified form:

1. We place the previously known samples on the data structures.
2. We then read the next sample to be classified, and calculate the Euclidean distance from the new sample to every sample in the training set.
3. We decide the class of the new element by selecting the class of the nearest sample, by Euclidean distance. The K-NN method requires the vote of the K closest samples.
4. We repeat the procedure until there are no more remaining samples.

This picture will give us a idea of how the new samples are being added. In this case, we use a κ of 1, for simplicity:



Sample application of a K-NN loop

K-NN can be implemented in more than one of the configurations that we have learned, but in

this chapter, we will use the semi-supervised approach; we will start from a certain number of already assigned samples, and we will later guess the cluster membership based on the characteristics of the training set.

Pros and cons of K-NN

The advantages of this method are as follows:

- **Simplicity**: There's no need to tune parameters
- **No formal training needed**: We just need more training examples to improve the model

The disadvantage is as follows:

It is computationally expensive - in a naive approach, all distances between points and every new sample have to be calculated, except when caching is implemented.

K-NN sample implementation

For this simple implementation of the K-NN method, we will use the NumPy and Matplotlib libraries. Also, as we will be generating a synthetic dataset for better comprehension, we will use the `make_blobs` method from scikit-learn, which will generate well-defined and separated groups of information so we have a sure reference for our implementation.

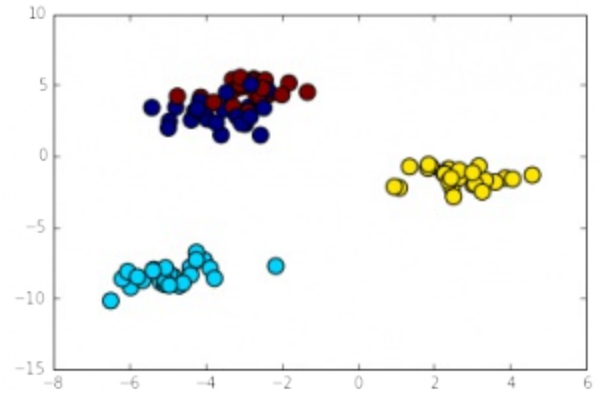
Importing the required libraries:

```
    importnumpyasnp

    importmatplotlib
    importmatplotlib.pyplotasplt

    fromsklearn.datasets.samples_generatorimportmake_blobs
    %matplotlib inline
```

So, it's time to generate the data samples for this example. The parameters of `make_blobs` are the number of samples, the number of features or dimensions, the quantity of centers or groups, whether the samples have to be shuffled, and the standard deviation of the cluster, to control how dispersed the group samples are:

```
    data,features=make_blobs(n_samples=100,n_features=2,centers=4,
    shuffle=True, cluster_std=0.8)
    fig,ax=plt.subplots()
    ax.scatter(data.transpose()[0],data.transpose()[1],c=features,marker=
'o',s=100)
    pl
```

Here is a representation of the generated sample blobs:



Firstly, let's define our `distance` function, which will be necessary to find the neighbors of all the new elements. We basically provide one sample, and return the distance between the provided new element and all their counterparts:

```
    defdistance(sample,data):
distances=np.zeros(len(data))
foriinrange(0,len(data)):
dist=np.sqrt(sum(pow(np.subtract(sample,data[i]),2)))
distances[i]=dist
returndistances
```

The `add_sample` function will receive a new 2D sample, the current dataset, and an array marking the group of the corresponding sample (from `0` to `3` in this case). In this case, we use `argpartition` to get the indices of the three nearest neighbors of the new sample, and then we use them to extract a subset of the `features` array. Then, `bincount` will return the count of any of the different classes on that three-element subset, and then with `argmax`, we will choose the index (in this case, the class number) of the group with the most elements in that two-element set:

```
    defadd_sample(newsample,data,features):
distances=np.zeros((len(data),len(data[0])))
#calculate the distance of the new sample and the current data
distances=distance(newsample,data)
closestneighbors=np.argpartition(distances,3)[:3]
closestgroups=features[closestneighbors]
returnnp.argmax(np.bincount(closestgroups))
```

Then we define our main `knn` function, which takes the new data to be added and uses the original classified data, represented by the `data` and `features` parameters, to decide the classes of the new elements:

```
    defknn(newdata,data,features):
foriinnewdata:
test=add_sample(i,data,features);
features=np.append(features,[test],axis=0)
data=np.append(data,[i],axis=0)
returndata,features
```

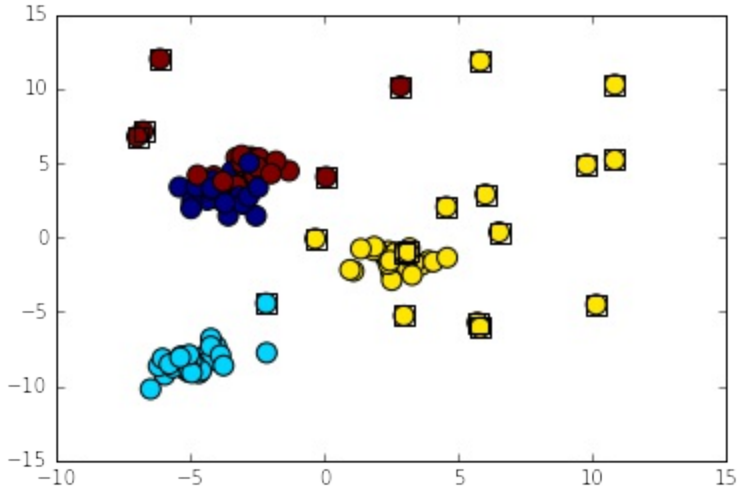
Finally, it's time to kickstart the process. For this reason, we define a set of new samples in the range of -10, 10 on both the `x` and `y` dimensions, and we will call our `knn` routine with it:

```
        newsamples=np.random.rand(20,2)*20-8.
    >    finaldata,finalfeatures=knn(newsamples,data,features)
```

Now it's time to represent the final results. First, we will represent the initial samples, which are much more well-formed than our random values, and then our final values, represented by an empty square (`c='none'`), so that they will serve as a marker of those samples:

```
    fig,ax=plt.subplots()
    ax.scatter(finaldata.transpose()[0],finaldata.transpose()[1],
    c=finalfeatures,marker='o',s=100)
    ax.scatter(newsamples.transpose()[0],newsamples.transpose()
    [1],c='none',marker=
    's',s=100)
    plt.plot()
```

Let's take a look at the following graph:



Final clustering status (new classified items are marked with a square)

In the preceding graph, we can see how our simple model of three neighbors works well to qualify and reform the grouping as the process advances. As the graph shows, the new groupings aren't necessarily of a circular form; they change according to the way the incoming data progresses.

Going beyond the basics

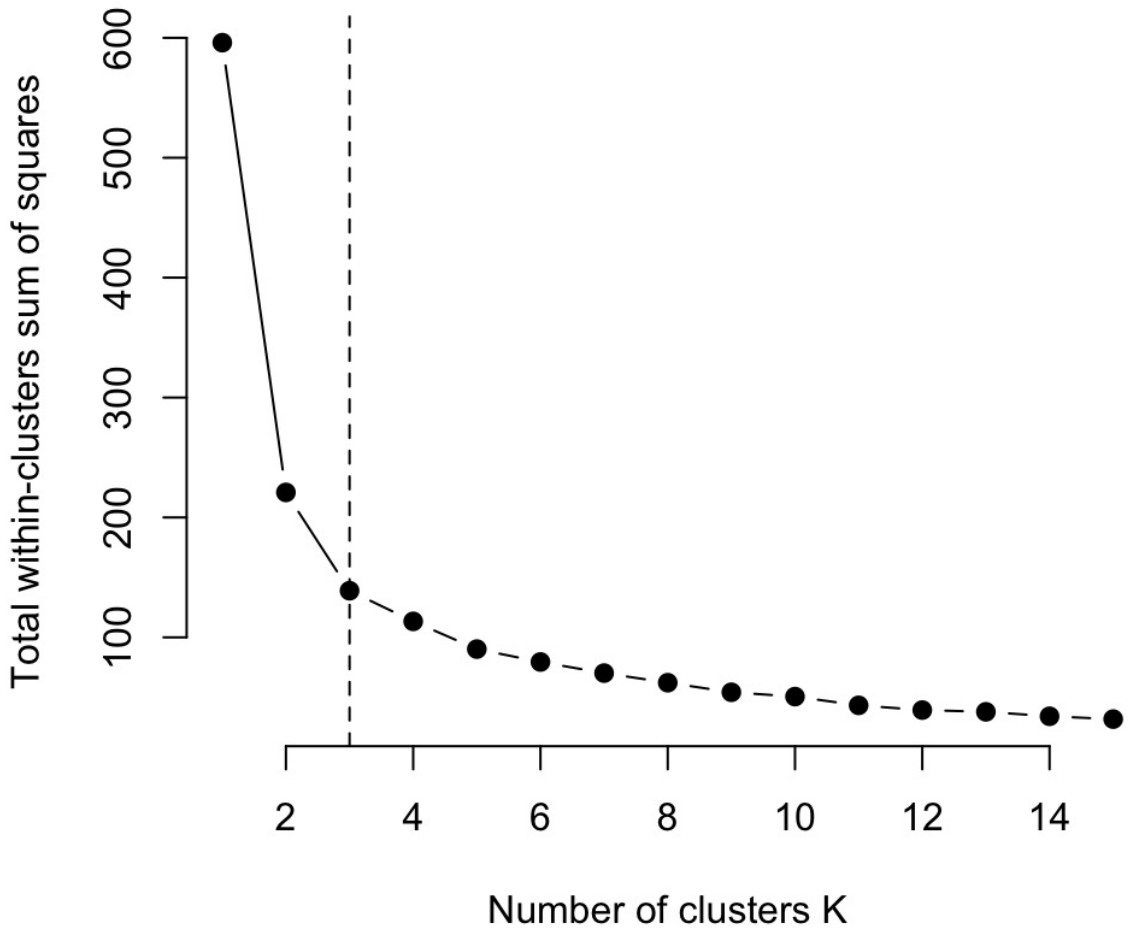
Now that we are done reviewing illustrative cases of the two main clustering techniques, let's explore some more advanced metrics and techniques so we can have them in our toolbox.

The Elbow method

One of the questions that may have arisen when implementing K-means could have been "how do I know that the target number of clusters is the best or most representative for the dataset?"

For this task, we have the **Elbow** method. It consists of a unique statistical measure of the total group dispersion in a grouping. It works by repeating the K-means procedure, using an increasing number of initial clusters, and calculating the total intra-cluster internal distance for all the groups.

Normally, the method will start with a very high value (except if we start with the right number of centroids), and then we will observe that the total intra-cluster distance drops quite quickly, until we reach a point where it doesn't change significantly. Congratulations, we have found the Elbow point, so-called for being an inflection on the following graph:



Graphical depiction of the error evolution, as the number of clusters increases, and the inflection point.

Regarding the accuracy of this indicator, the Elbow method is, as you can see, a heuristic and not mathematically determined, but can be of use if you want to make a quick estimate of the right number of clusters, especially when the curve changes very abruptly at some point.

Summary

In this chapter, we have covered the simplest but still very practical machine learning models in an eminently practical way to get us started on the complexity scale.

In the following chapter, where we will cover several regression techniques, it will be time to go and solve a new type of problem that we have not worked on, even if it's possible to solve the problem with clustering methods (regression), using new mathematical tools for approximating unknown values. In it, we will model past data using mathematical functions, and try to model new output based on those modeling functions.

References

- Thorndike, Robert L, *Who belongs in the family?,* Psychometrika18.4 (1953): 267-276.
- Steinhaus, H, *Sur la division des corp materiels en parties.* Bull. Acad. Polon. Sci 1 (1956): 801–804.
- MacQueen, James, *Some methods for classification and analysis of multivariate observations.* Proceedings of the fifth Berkeley symposium on mathematical statistics and probability. Vol. 1. No. 14. 1967.
- *Cover, Thomas, and Peter Hart, Nearest neighbor pattern classification.* IEEE transactions on information theory 13.1 (1967): 21-27.

Chapter 4. Linear and Logistic Regression

After the insights we gained by grouping similar information using common features, it's time to get a bit more mathematical and start to search for a way to describe the data by using a distinct function that will condense a large amount of information, and will allow us to predict future outcomes, assuming that the data samples maintain their previous properties.

In this chapter, we will cover the following topics:

- Linear regression with a step-by-step implementation
- Polynomial regression
- Logistic regression and its implementation
- Softmax regression

Regression analysis

This chapter will begin with an explanation of the general principles. So, let's ask the fundamental question: w*hat's regression?*

Before all considerations, regression is basically a **statistical process.** As we saw in the introductory section, regression will involve a set of data that has some particular probability distribution. In summary, we have a population of data that we need to characterize.

And what elements are we looking for in particular, in the case of regression? We want to determine the relationship between an independent variable and a dependent variable that optimally adjusts to the provided data. When we find such a function between the described variables, it will be called the **regression function**.

There are a large number of function types available to help us model our current data, the most common example being the linear, polynomial, and exponential.

These techniques will aim to determine an objective function, which in our case will output a finite number of unknown optimum parameters of the function, called **parametric regression** techniques.

Applications of regression

Regression is normally applied in order to predict future variable values, and it's a very commonly employed technique for initial data modeling in data analysis projects, but it also can be used to optimize processes, finding common ground between related but dispersed data.

Here we will list a number of the possible application of regression analysis:

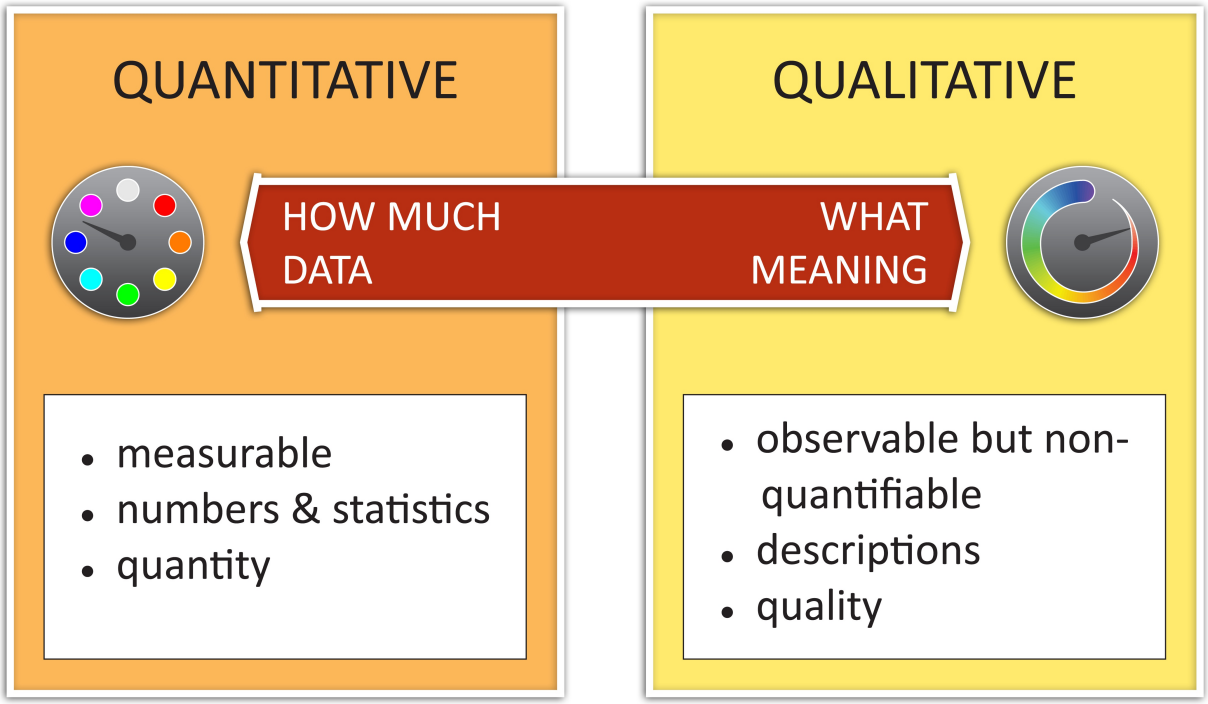
- In social sciences, to predict the future values of all kinds of metrics, such as unemployment and population
- In economics, to predict future inflation rates, interest rates, and similar indicators
- In earth sciences, to predict future phenomena, such as the thickness of the ozone layer
- To help with all elements of the normal corporate dashboard, adding probabilistic estimations for production throughput, earnings, spending, and so on
- Proving the dependencies and relevancies between two phenomena
- Finding the optimum mix of components in reaction experiments
- Minimizing the risk portfolio
- Understanding how sensitive a corporation’s sales are to changes in advertising expenditure
- Seeing how a stock's price is affected by changes to the interest rate

Quantitative versus qualitative variables

In day-to-day work with data, not all the elements we encounter are the same, and thus they require special treatment, depending on their properties. A very important distinction we can make to recognize how appropriate the problem variables are is by dividing the data types into

quantitative and qualitative data variables using the following criteria:

- **Quantitative variables**: In the realm of physical variables or measurements, we normally work with real numbers, or qualitative variables, because what matters the most is the quantity we are measuring. In this group, we have ordinal variables, that is, when we work with orders and rankings in an activity. Both of these variable types fall within the quantitative variables category.
- **Qualitative variables:** On the other hand, we have measurements that show which class a sample belongs to. This can't be expressed by a number, in the sense of a quantity; it is usually assigned a label, tag, or categorical value representing the group to which the sample belongs. We call these variables qualitative variables.



Reference table addressing the differences between quantitative and qualitative analysis

Now let's address the question of which types of variable are suitable to apply to regression problems.

The clear answer is quantitative variables, because the modeling of the data distribution can only be done through functions we use to detect a regular correspondence between those variables, and not on classes or types of elements. Regression requires one continuous output variable, which is only the case with quantitative metrics.

In the case of qualitative variables, we assign the data to classification problems because the very definition of it is to search for non-numeric labels or tags to assign to a sample. This is the mission of classification, which we will see in the following chapter.

Linear regression

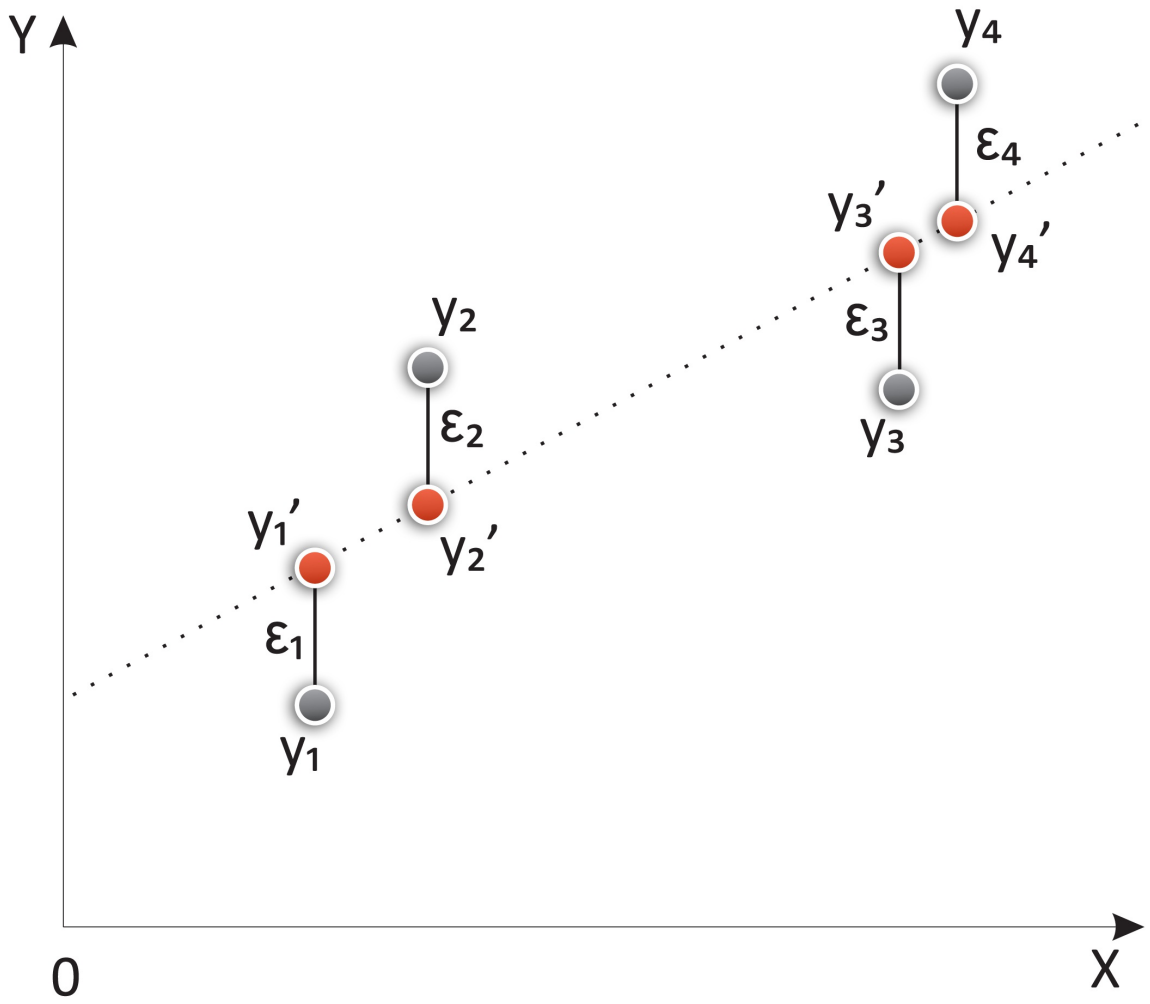
So, it's time to start with the simplest yet still very useful abstraction for our data–a linear regression function.

In linear regression, we try to find a linear equation that minimizes the distance between the data points and the modeled line. The model function takes the following form:

$y_i = ßx_i + α + ε_i$

Here, *α* is the intercept and *ß* is the slope of the modeled line. The variable *x* is normally called the independent variable, and *y* the dependent one, but it can also be called the regressor and the response variables.

The $ε_i$ variable is a very interesting element, and it's the error or distance from the sample *i* to the regressed line.



Depiction of the components of a regression line, including the original elements, the estimated ones (in red), and the error (ε)

The set of all those distances, calculated in the form of a function called the *cost* function, will give us, as a result of the solution process, the values of the unknown parameters that minimize the cost. Let's get to work on it.

Determination of the cost function

As with all machine learning techniques, the process of learning depends on a minimized loss function, which shows us how right or wrong we are when predicting an outcome, depending on the stage of learning we are in.

> ### Note
>
> The most commonly used cost function for linear regression is called *least squares*.

Let's define this cost function taking, for simplification a 2D regression, where we have a list of number tuples (x_0, y_0), (x_1, y_1) ... (x_n, y_n) and the values to find, which are β_0 and β_1. The least squares cost function in this case can be defined as follows:

$$J(\beta_0,\beta_1) = \sum_{i=0}^{n} (y_i - \beta_0 - \beta_1 x_i)^2$$

Least squares function for a linear equation, using the standard variables β_0 and β_1 that will be used in the next sections

The summation for each of the elements gives a unique global number, which gives a global idea of the total differences between all the values (y_i), and the corresponding point in our ideal regressing line ($\beta_0 + \beta 1 x_i$).

The rationale for this operation is pretty clear:

- The summation gives us a unique global number
- The difference model-real point gives us the distance or L1 error
- Squaring this gives us a positive number, which also penalizes distances in an non-linear way, passing the one-error limit, so the more errors we commit, the more willing we will be to increase our penalization rate

Another way of phrasing this is that this process minimizes the sum of squared residuals, the residual being the difference between the value we get from the dataset and the expected value

computed by the model, for the same input value.

The many ways of minimizing errors

The least squares error function has several ways of getting a solution:

- The analytical way
- Using the covariance and correlation values
- The most familiar way to the machine learning family of methods — the gradient descent way

Analytical approach

The analytical approach employs several linear algebra techniques in order to get an exact solution.

Note

We are presenting this technique in a very succinct way because it's not directly related to the machine learning techniques we are reviewing in this book. We are presenting it for completeness.

First of all, we represent the error in a function in a matrix form:

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T (X\theta - y)$$

Canonical form of the linear regression equation in the matrix form

Here, *J* is the cost function and has an analytical solution of:

$$\theta = \left(X^T X\right)^{-1} X^T y$$

Analytical solution of the matrix form of linear regression

Pros and cons of the analytical approach

The approach of using linear algebra techniques to calculate the minimum error solution is an easier one, given the fact that we can give a really simple representation that is deterministic, so there is no additional guessing involved after applying the operations.

But there are some possible problems with this approach:

- First, matrix inversion and multiplication are very computationally intensive operations. They typically have a lower bound of approximately $O(n^2)$ to $O(n^3)$, so when the number of samples increases, the problem can become intractable.
- Additionally, and depending on the implementation, this direct approach could also have limited accuracy, because we can normally use the limits of the floating point capacity of the current hardware.

Covariance/correlation method

Now it's time to introduce a new way of estimating the coefficient of our regressing line, and in the process, we will learn additional statistical measures, such as covariance and correlation, which will also help us when analyzing a dataset for the first time and drawing our first conclusions.

Covariance

Covariance is a statistical term, and can be canonically defined as follows:

> *A measure of the systematic relationship between a pair of random variables wherein a change in one variable is reciprocated by an equivalent change in another variable.*

Covariance can take any value between -∞ to +∞, wherein a negative value is an indicator of a negative relationship, whereas a positive value represents a positive relationship. It also ascertains a linear relationship between the variables.

Therefore, when the value is zero, it indicates no direct linear relationship, and the values tend to form a blob-like distribution.

Covariance is not affected by the unit of measure, that is, there is no change in the strength of the relationship between two variables when changing units. Nevertheless, the value of covariance will change. It has the following formula, which needs the mean of each axis as a prerequisite:

$$\operatorname{cov}(x_i, y) = \frac{1}{n} \sum (x_i - \bar{x}_i)(y - \bar{y})$$

Correlation

Remember when we described the process of normalization of a variable? We centered the variable by subtracting the mean and scaling it with the standard deviation of the dataset with the

following formula:

$$x = \frac{x - \bar{x}}{\sigma}$$

Analytical form of the data normalization operation

This will be the starting point of our analysis, and we will be extending it towards each axis with the correlation value.

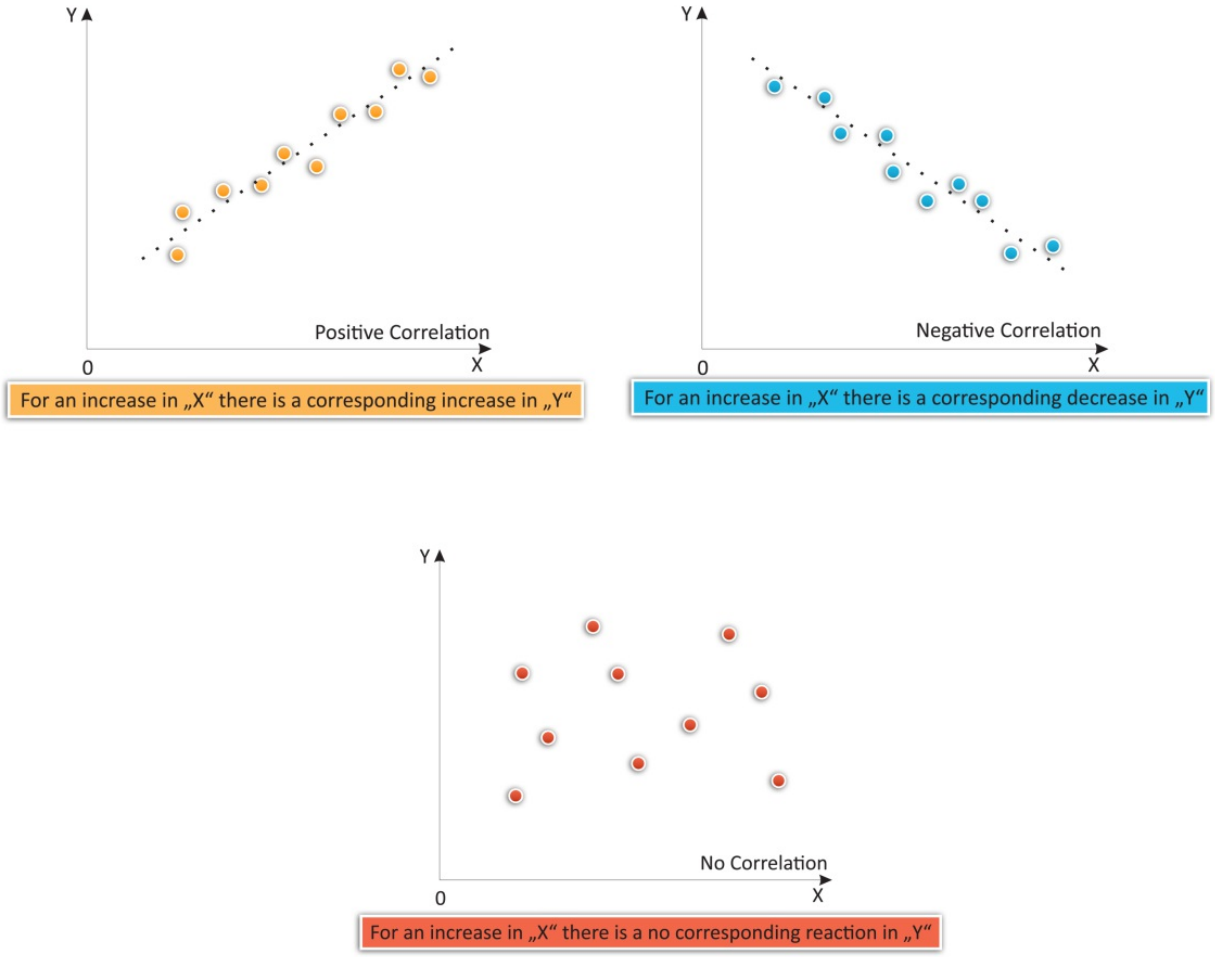
The correlation value determines the degree to which two or more random variables move in tandem. During the study of two variables, if it has been observed that the movement of one variable is concordant with an equivalent movement in another variable, then the variables are said to be correlated, with the exact correlation value given by the following formula:

$$r = \frac{1}{n} \cdot \frac{\sum \left(x_i - \bar{x}_i \right) \left(y - \bar{y} \right)}{\sigma_{x_i} \sigma_y}$$

Canonical definition of correlation

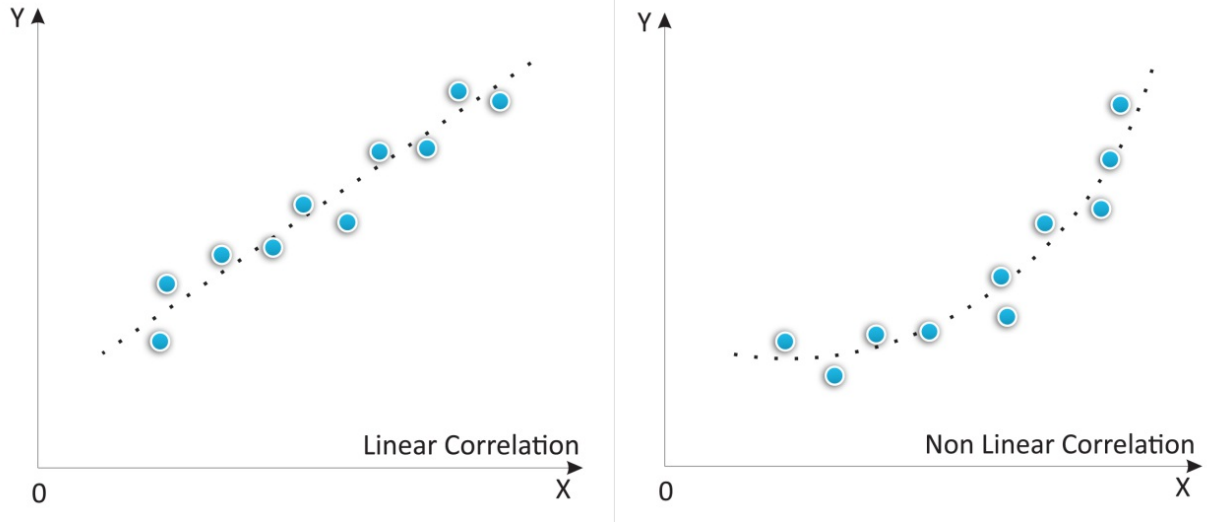
As a real value-based metric, it can be of two types, positive or negative. The variables are positively or directly correlated when the two variables move in the same direction. When the two variables move in opposite direction, the correlation is negative or inverse.

The value of correlation lies between *-1* to *+1*, wherein values close to *+1* represent a strong positive correlation and values close to *-1* are an indicator of a strong negative correlation:



Graphic depiction of how the distribution of the samples affects the correlation values

There are other ways of measuring correlation. In this book, we will be talking mainly about linear correlation. There are other methods of studying non-linear correlation, which won't be covered in this book.



Depiction of the difference between linear correlation and non-linear correlation measures

Note

Within the practical exercises of this chapter, you will find an implementation of both linear covariance and correlation.

Searching for the slope and intercept with covariance and correlation

As we have known from the beginning, what we need is to find the equation of a line, representing the underlying data, in the following form:

$$\hat{y} = \hat{\beta} x + \hat{\alpha}$$

Approximate definition of a linear equation

As we know that this line passes through the average of all the points, we can estimate the intercept, with the only unknown being the estimated slope:

$$\hat{\alpha} = \overline{y} - \hat{\beta}\overline{x},$$

Derived definition of intercept

The slope represents the change in the dependent variable divided by the change in the independent variable. In this case, we are dealing with variation in data rather than absolute differences between coordinates.

As the data is of non-uniform nature, we are defining the slope as the proportion of the variance in the independent variable that covaries with the dependent variable:

$$\hat{\beta} = \frac{Cov(x, y)}{Var(x)}$$

Estimated slope coefficient

As it happens, if our data actually looks like a circular cloud when we plot it, our slope will become *zero,* suggesting no causal relationship between the variation in *x and y,* expressed in the following form:

$$\hat{\beta} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

Expanded form of the estimated slope coefficient

Applying the formulas we have shown previously, we can finally simplify the expression of the slope of the estimated regression line to the following expression:

$$= r_{xy} \frac{S_y}{S_x},$$

Final form of the slope coefficient

Here, S_y is the standard deviation in y, and S_x is the standard deviation in x.

With the help of the remaining elements in the equation, we can simply derive the intercept based on the knowledge that the line will reach the mean dataset point:

$$a = \bar{Y} - b\bar{X}$$

Final form of the approximated intercept coefficient

So, we are done with a very summarized expression of two preliminary forms of regression, which have also left many analysis elements for us to use. Now it's time to introduce the star of the current machine learning techniques, one that you will surely use in many projects as a practitioner, called **gradient descent**.

Gradient descent

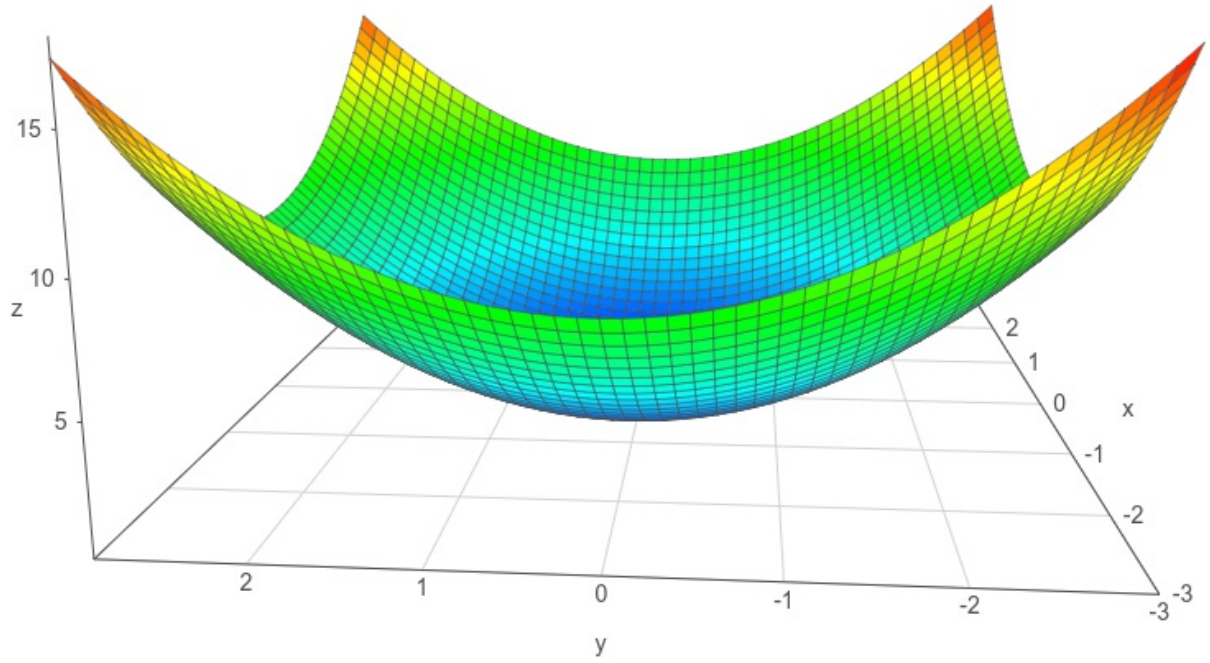
It's time to talk about the method that will take us to the core of modern machine learning. The method explained here will be used with many of the more sophisticated models in a similar fashion, with increased difficulty but with the same principles.

Some intuitive background

To introduce gradient descent, we will first take a look at our objective—the fitting of a line function to a set of provided data. And what do we have as elements?

- A model function
- An error function

Another element that we could get is a representation of all the possible errors for any of the combinations of the parameters. That would be cool, right? But look at what such a function looks like just for a problem with a simple line as the solution. This curve represents $z = x^2 + y^2$, which follows the form of the least squares error function:



Least squares error surface, in the case of two variables. In the case of linear regression, they are the slope and the intersect

As you can see, calculating all the possible outcomes per line parameter would consume too much CPU time. But we have an advantage: we know that the surface of such a curve is **convex** (a discussion beyond the scope of this book), so it roughly looks like a bowl, and it has a unique minimum value (as seen in the previous folder). This will spare us the problem of locating local points that look like a minimum, but in fact are just bumps on the surface.

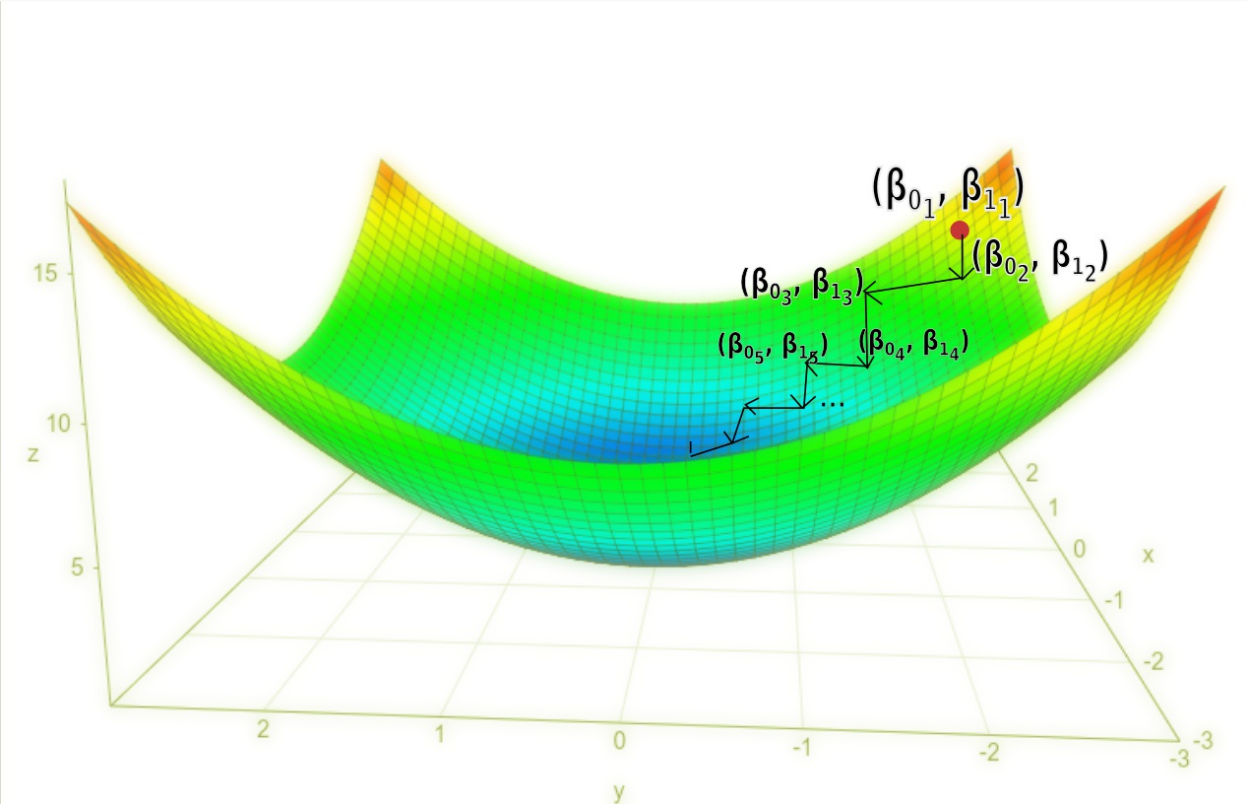
The gradient descent loop

So it's time to look for a method to converge to the minimum of a function, knowing only where I am on the surface, and possibly the gradient at the point on the surface I am standing on:

- Start at a random position (remember, we don't know anything about the surface yet)
- Look for the direction of maximum change (as the function is convex, we know it will guide us to the minimum)
- Advance over the error surface in that direction, proportionally to the error amount
- Adjust the starting point of the next step to the new point on the surface where we landed and repeat the process

This method allows us to discover the path for the minimization of our values in an iterative way and in a limited time, when compared to a brute force method.

The process for two parameters and the least squares function goes like this:



Depiction of the gradient descent algorithm, Beginning from a starting high error point, and descending in the direction of maximum change

This gives us an idea of how the process functions work in a normal setting, when we use and choose good and appropriate initial parameters.

Note

In the upcoming chapters, we will cover the process of gradient descent in more detail, including how choosing different elements (that we will call hyper-parameters) changes the behavior of the process.

Formalizing our concepts

Now let's go over the mathematical side of our process, so we have a reference of all the parts involved, before using them in practice.

The elements or our equations are as follows:

- The linear function variables, β_0 and β_1
- The number of samples in the sampleset, m
- The different elements in the sampleset, $x^{(i)}$ and $y^{(i)}$

Let's start with our error function, J. It was defined in previous sections under the name of the least squares function. We will add, for practicality, the term 1/2m at the start of the equation, as

follows:

$$J(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^{m} ((\beta_0 + \beta_1 x^{(i)}) - y^{(i)})^2$$

Least squares error function

Let's introduce a new operator, which is the basis of all the following work, gradient.

To build the concept of it, we have the following:

- A function of one or more independent variables
- The partial derivative of the function for all independent variables

As we already know how partial derivatives work at the moment, it's enough to say that the gradient is a vector containing all of the already mentioned partial derivatives; in our case, it will be as follows:

$$\nabla J(\beta_0, \beta_1) = \begin{bmatrix} \left(\frac{\partial J}{\partial \beta_0} \right) \\ \left(\frac{\partial J}{\partial \beta_1} \right) \end{bmatrix}$$

Gradient of the error function

What's the purpose of such an operator? If we are able to calculate it, it will give us the direction of change of the whole function at a single point.

First, we calculate the partial derivative. You can try to derive it; basically, it uses the chain rule of deriving a squared expression and then multiplies it by the original expression.

In the second part of the equation, we simplify the linear function by the name of the model function, h_a:

$$\frac{\partial J}{\partial \beta_0} = \frac{1}{m} \sum_{i=1}^{m} ((\beta_0 + \beta_1 x^{(i)}) - y^{(i)}) \ = \frac{1}{m} \sum_{i=1}^{m} (h_a(x^{(i)}) - y^{(i)})$$

Partial derivative of the error function for the β_1 variable

In the case of β_1 we get the additional factor of the $x^{(i)}$ element because the derivative of $\beta_1 x^{(i)}$ is $x^{(i)}$:

$$\frac{\partial J}{\partial \beta_1} = \frac{1}{m}\sum_{i=1}^{m}((\beta_0 + \beta_1 x^i) - y^{(i)})x^{(i)} = \frac{1}{m}\sum_{i=1}^{m}(h_a(x^{(i)}) - y^{(i)})x^{(i)}$$

Partial derivative of the error function for the β_1 variable

Now we introduce the recursion expression, which will provide (when iterating and if conditions are met) a combination of parameters that decrease the total error in a convergent way.

Here, we introduce a really important element: the step size, with the name α. What's the purpose of it? It will allow us to scale how much we will advance in one step. We will discover that not choosing the right amount of power can lead to disastrous consequences, including the divergence of the error to the infinite.

Note that the second formula only has the tiny difference of being multiplied by the current *x* value:

$$(\beta_0)_{k+1} \leftarrow (\beta_0)_k - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_a(x^{(i)}) - y^{(i)})$$

$$(\beta_1)_{k+1} \leftarrow (\beta_1)_k - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_a(x^{(i)}) - y^{(i)})x^{(i)}$$

Recursion equation for the model functions

So, we are ready to go! Now we will just add a bit of mathematical spice in order to produce a more compact representation of the algorithm. Let's now express the unknowns in vector form, so all the expressions will be expressed as a whole:

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

Expression of β in vector form

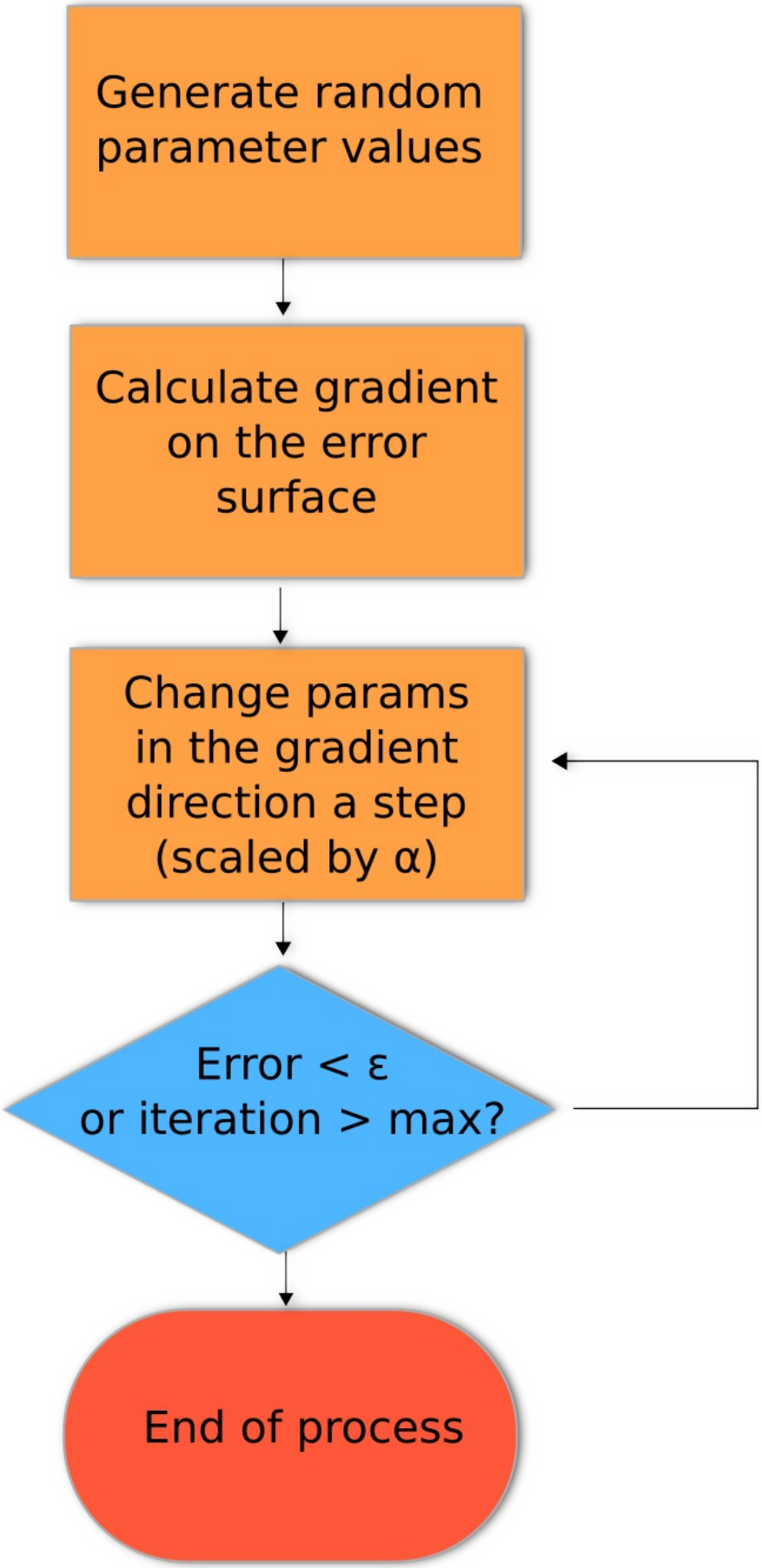
With this new expression, our recursion steps can be expressed in this simple and easy-to-remember expression:

$$\beta_{k+1} \leftarrow \beta_k - \alpha \nabla J(\beta_k)$$

Expression of the gradient descent recursion in vector form

Expressing recursion as a process

The whole method of finding the minimum error can alternatively be expressed in a flowchart, so that we can have all the elements in the same place and understand how easy it looks if we, for a moment don't take into account the somewhat complex analytic mechanisms:


Generate random parameter values
Calculate gradient on the error surface
Change params in the gradient direction a step (scaled by α)
Error < ε or iteration > max?
End of process

Flow diagram of the gradient descent method. Note the simple building blocks, without taking into account the subtle mathematics it involves

So, with this last procedural vision of the gradient descent process, we are ready to go on to the more practical parts of this chapter. We hope you enjoyed this journey towards finding an answer to the question: What's the best way to represent our data in a simple way? And rest assured we will use much more powerful tools in the following sections.

Going practical – new tools for new methods

In this section, we will introduce a new library that will help us with covariance and correlation, especially in the data visualization realm.

What is **Seaborn**?

Seaborn is a library for making attractive and informative statistical graphics in Python. Moreover, it also provides very useful multivariate analysis primitives, which will help you decide whether or not and how to apply determinate regression analysis to your data.

Some of the features that Seaborn offers are as follows:

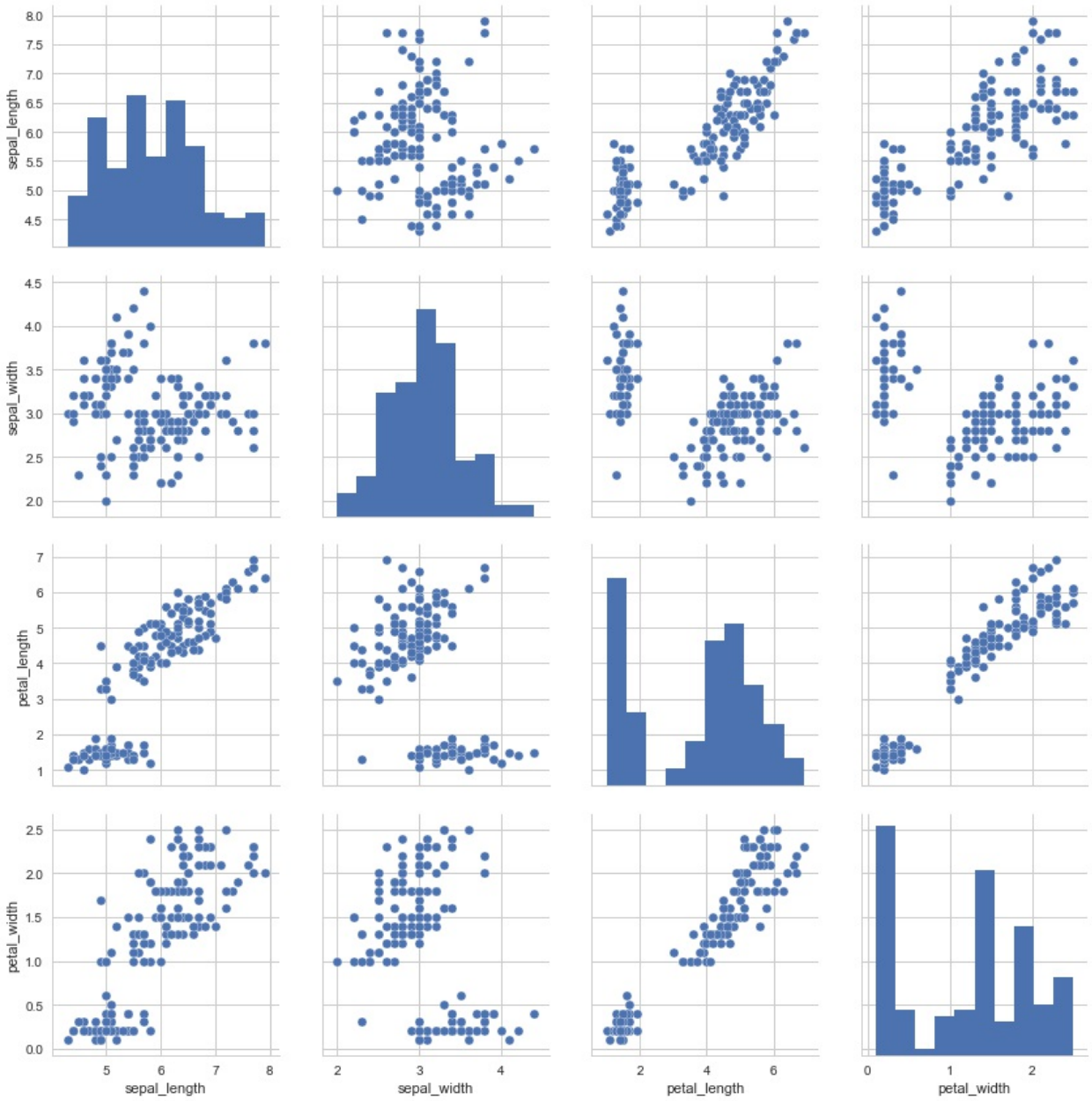
- Several built-in themes of very high quality
- Tools for choosing color palettes to make beautiful plots that reveal patterns in the data
- Very important functions for visualizing univariate and bivariate distributions or for comparing them between subsets of data
- Tools that fit and visualize linear regression models for different kinds of independent and dependent variables
- Plotting functions which try to do something useful when called with a minimal set of arguments; they expose a number of customizable options through additional parameters

An important additional feature is, given that Seaborn uses matplotlib, the graphics can be further tweaked using those tools and rendered with any of the matplotlib backends.

Now let's explore the most useful utilities that Seaborn will bring.

Useful diagrams for variable explorations – pairplot

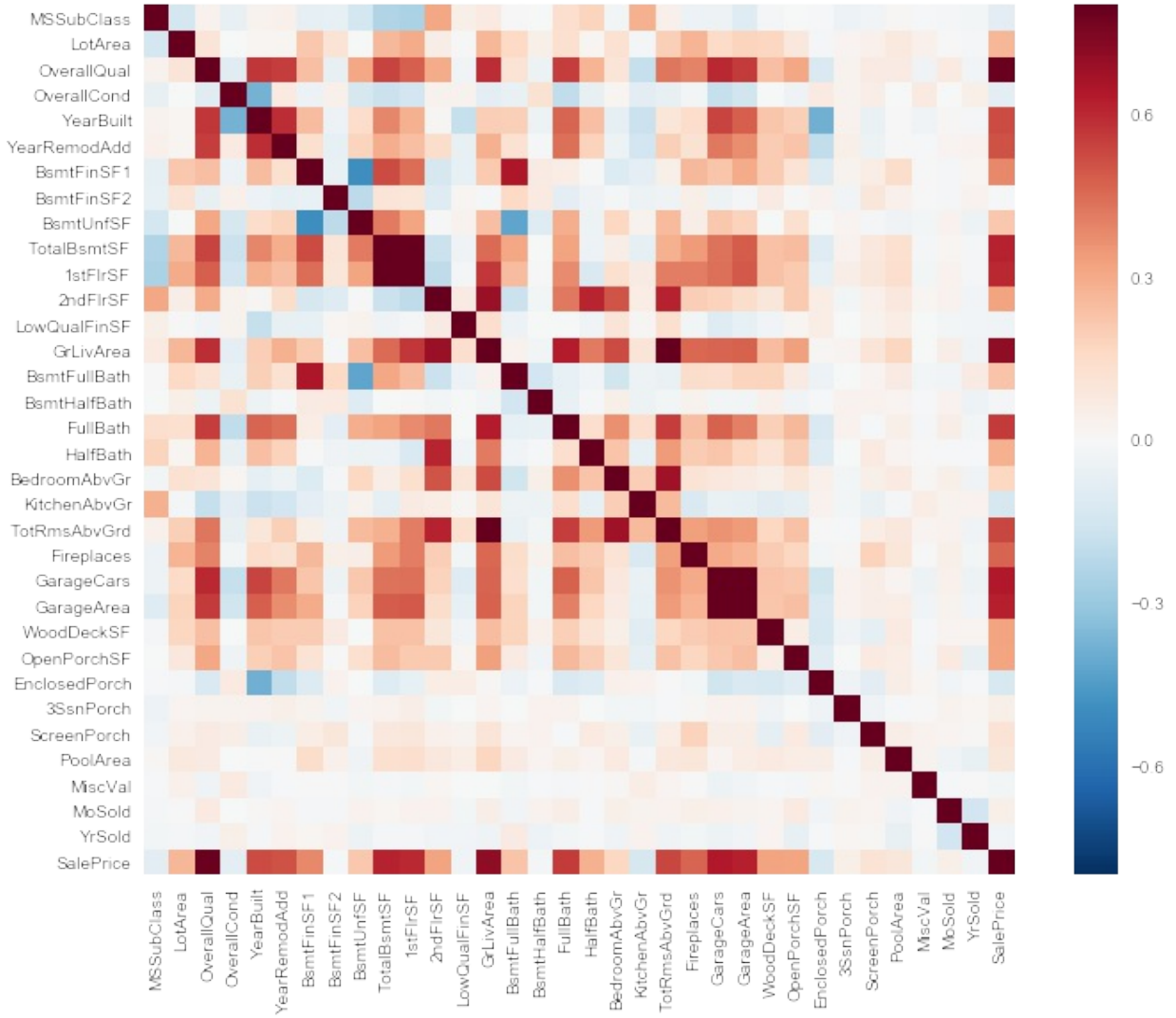
For the stage of data exploration, one of the most useful measures we can have is a graphical depiction of how all the features in the dataset interact, and discover the joint variations in an intuitive manner:



Pairplot for the variables in the Iris dataset

Correlation plot

The correlation plot allows us to summarize the variable dependency in a much more succinct way, because it shows the direct correlation between variable pairs, using a color pallet. The diagonal values are of course 1, because all variables have a maximum correlation with themselves:



Correlation plot of the San Francisco housing dataset.

Data exploration and linear regression in practice

In this section, we will start using one of the most well-known *toy* datasets, explore it, and select one of the dimensions to learn how to build a linear regression model for its values. Let's start by importing all the libraries (`scikit-learn`, `seaborn`, and `matplotlib`); one of the excellent features of Seaborn is its ability to define very professional-looking style settings. In this case, we will use the `whitegrid` style:

```
import numpy as np from sklearn import datasets import seaborn.apionly as sns %matplotlib
inline import matplotlib.pyplot as plt sns.set(style='whitegrid', context='notebook')
```

The Iris dataset

It’s time to load the *Iris* dataset. This is one of the most well-known historical datasets. You will find it in many books and publications. Given the good properties of the data, it is useful for classification and regression examples. The Iris dataset (https://archive.ics.uci.edu/ml/datasets/Iris) contains 50 records for each of the three types of iris, 150 lines in a total over five fields. Each line is a measurement of the following:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm

The final field is the type of flower (*setosa*, *versicolor*, or *virginica*). Let’s use the `load_dataset` method to create a matrix of values from the dataset:

```
iris2 = sns.load_dataset('iris')
```

In order to understand the dependencies between variables, we will implement the covariance operation. It will receive two arrays as parameters and will return the `covariance(x,y)` value:

```
defcovariance(X, Y):
    xhat=np.mean(X)
    yhat=np.mean(Y)
    epsilon=0for x,y in zip (X,Y):
        epsilon=epsilon+(x-xhat)*(y-yhat)
    return epsilon/(len(X)-1)
```

Let's try the implemented function and compare it with the NumPy function. Note that we calculated `cov(a,b)`, and NumPy generated a matrix of all the combinations `cov(a,a)`, `cov(a,b)`, so our result should be equal to the values `(1,0)` and `(0,1)` of that matrix:

```
print (covariance ([1,3,4], [1,0,2]))
print (np.cov([1,3,4], [1,0,2]))

0.5
[[ 2.333333330.5       ]
 [ 0.51.        ]]
```

Having done a minimal amount of testing of the correlation function as defined earlier, receive two arrays, such as `covariance`, and use them to get the final value:

```
defcorrelation(X, Y):return (covariance(X,Y)/(np.std(X,  ddof=1)*np.std(Y,  ddof=1))) ##We
have to indicate ddof=1 the unbiased std
```

Let's test this function with two sample arrays, and compare this with the `(0,1)` and `(1,0)` values of the correlation matrix from NumPy:

```
print (correlation ([1,1,4,3], [1,0,2,2]))
print (np.corrcoef ([1,1,4,3], [1,0,2,2]))
```

```
0.870388279778
[[ 1.0.87038828]
 [ 0.870388281.        ]]
```

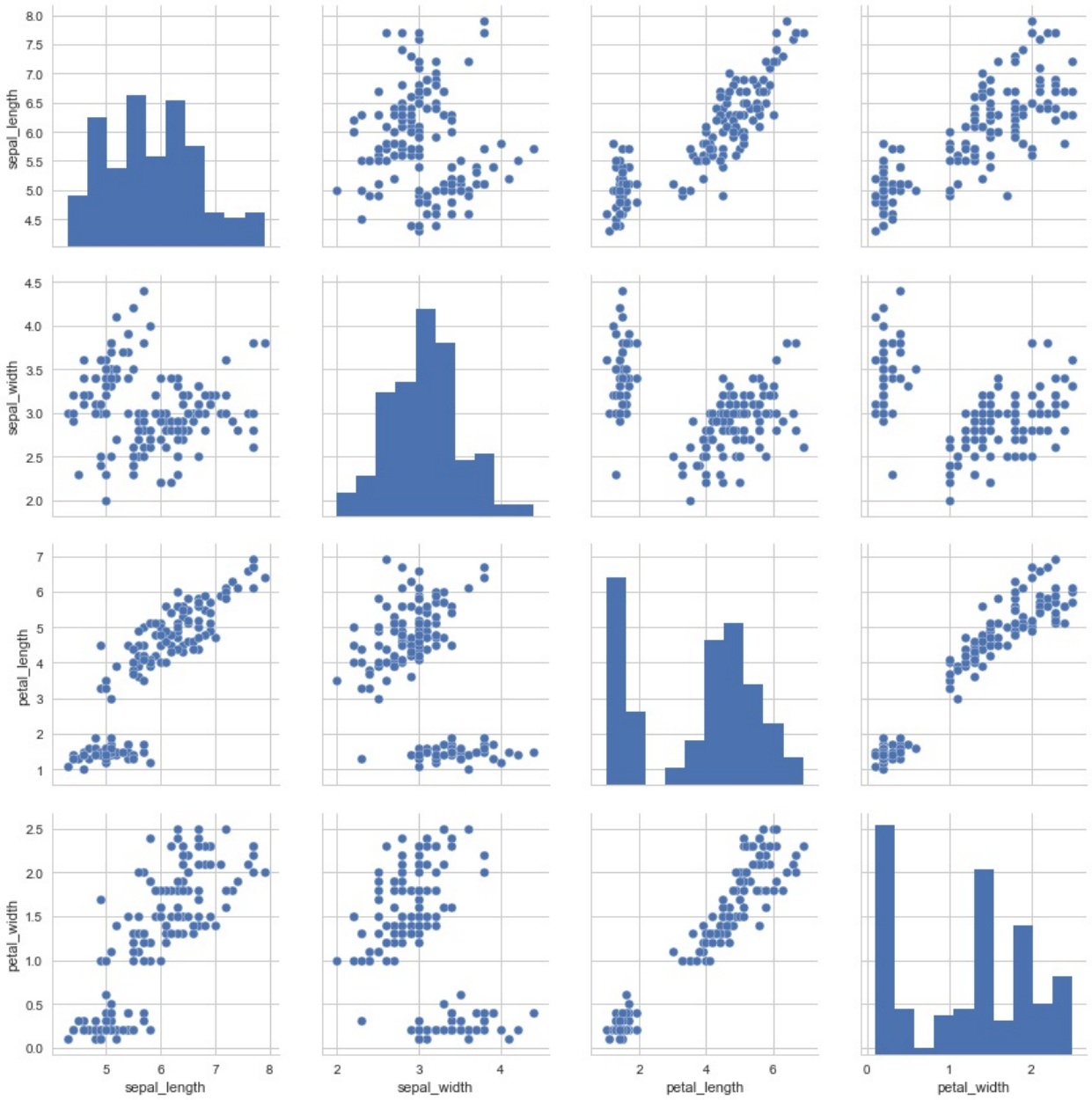
Getting an intuitive idea with Seaborn pairplot

A very good idea when starting worked on a problem is to get a graphical representation of all the possible variable combinations.

Seaborn's `pairplot` function provides a complete graphical summary of all the variable pairs, represented as scatterplots, and a representation of the univariate distribution for the matrix diagonal.

Let's look at how this plot type shows all the variables dependencies, and try to look for a linear relationship as a base to test our regression methods:

```
sns.pairplot(iris2, size=3.0)
<seaborn.axisgrid.PairGrid at 0x7f8a2a30e828>
```



Pairplot of all the variables in the dataset.

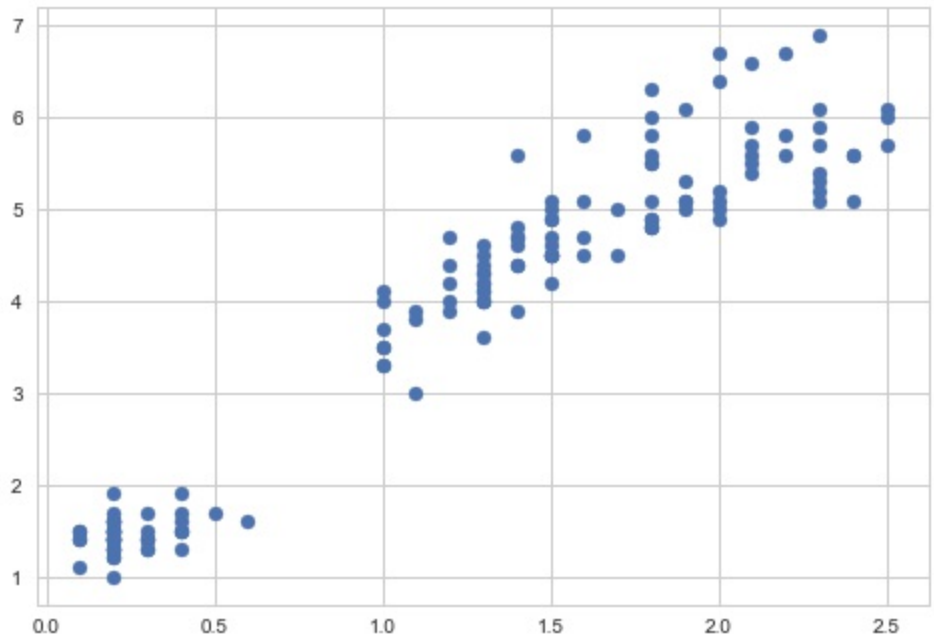
Lets' select two variables that, from our initial analysis, have the property of being linearly dependent. They are `petal_width` and `petal_length`:

```
X=iris2['petal_width']
Y=iris2['petal_length']
```

Let’s now take a look at this variable combination, which shows a clear linear tendency:

```
plt.scatter(X,Y)
```

This is the representation of the chosen variables, in a scatter type graph:



This is the current distribution of data that we will try to model with our linear prediction function.

Creating the prediction function

First, let's define the function that will abstractedly represent the modeled data, in the form of a linear function, with the form *y=beta*x+alpha*:

```
defpredict(alpha, beta, x_i):return beta * x_i + alpha
```

Defining the error function

It’s now time to define the function that will show us the difference between predictions and the expected output during training. As we will explain in depth in the next chapter, we have two main alternatives: measuring the absolute difference between the values (or L1), or measuring a variant of the square of the difference (or L2). Let’s define both versions, including the first formulation inside the second:

```
deferror(alpha, beta, x_i, y_i):#L1return y_i - predict(alpha, beta, x_i)

defsum_sq_e(alpha, beta, x, y):#L2return sum(error(alpha, beta, x_i, y_i) ** 2for x_i, y_i in
zip(x, y))
```

Correlation fit

Now, we will define a function implementing the correlation method to find the parameters for our regression:

```
defcorrelation_fit(x, y):
    beta = correlation(x, y) * np.std(y, ddof=1) / np.std(x,ddof=1)
    alpha = np.mean(y) - beta * np.mean(x)
    return alpha, beta
```

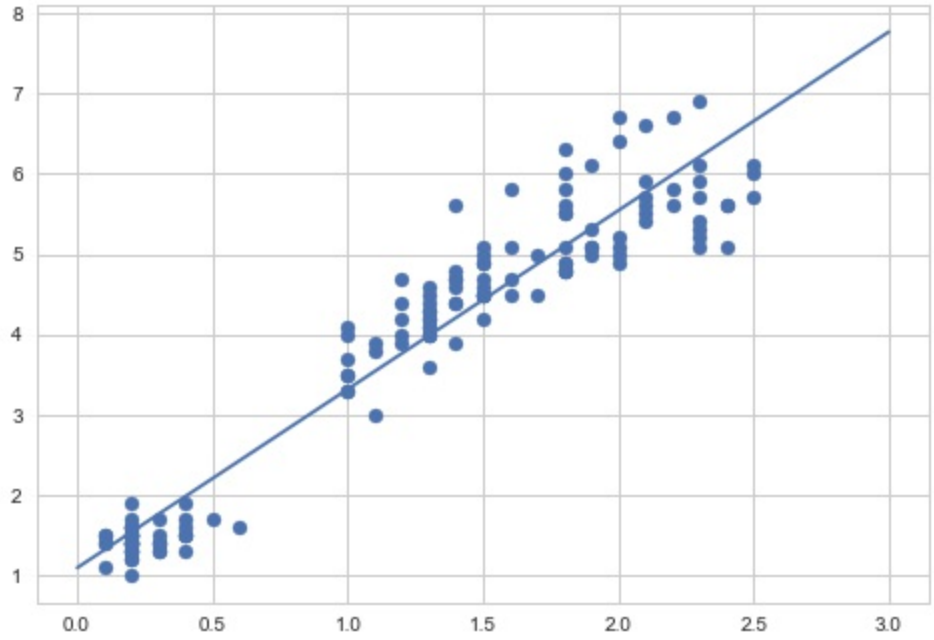
Let’s then run the fitting function and print the guessed parameters:

```
alpha, beta = correlation_fit(X, Y)
print(alpha)
print(beta)

1.08355803285
2.22994049512
```

Let’s now graph the regressed line with the data in order to intuitively show the appropriateness of the solution:

```
plt.scatter(X,Y)
xr=np.arange(0,3.5)
plt.plot(xr,(xr*beta)+alpha)
```

This is the final plot we will get with our recently calculated slope and intercept:



Final regressed line.

Polynomial regression and an introduction to underfitting and overfitting

When looking for a model, one of the main characteristics we look for is the power of generalizing with a simple functional expression. When we increase the complexity of the model, it's possible that we are building a model that is good for the training data, but will be too optimized for that particular subset of data.

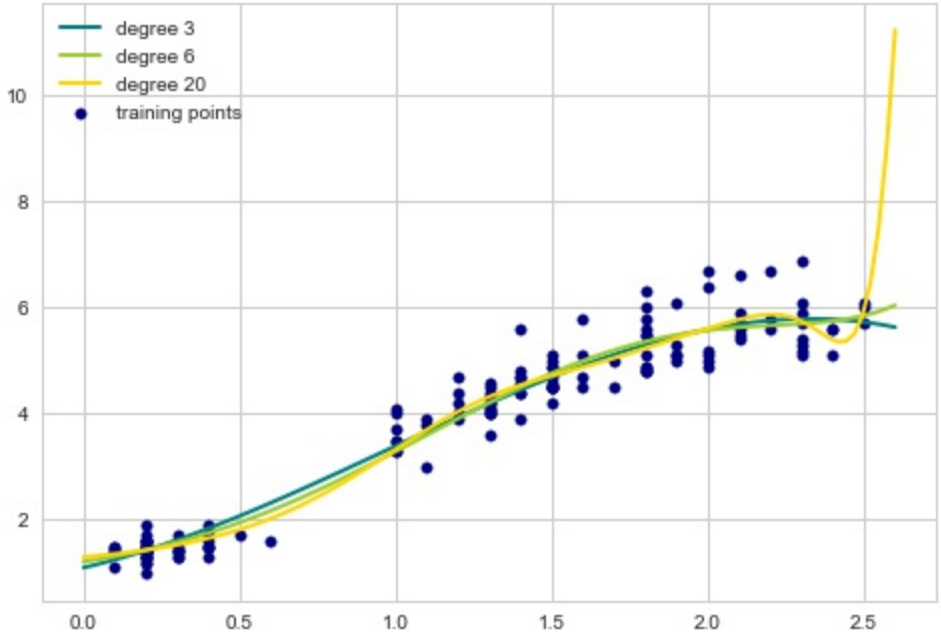
Underfitting, on the other hand, applies to situations where the model is too simple, such as this case, which can be represented fairly well with a simple linear model.

In the following example, we will work on the same problem as before, using the scikit-learn library to search higher-order polynomials to fit the incoming data with increasingly complex degrees.

Going beyond the normal threshold of a quadratic function, we will see how the function looks to fit every wrinkle in the data, but when we extrapolate, the values outside the normal range are clearly out of range:

```
from sklearn.linear_model import Ridge
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

ix=iris2['petal_width']
iy=iris2['petal_length']

# generate points used to represent the fitted function
x_plot = np.linspace(0, 2.6, 100)

# create matrix versions of these arrays
X = ix[:, np.newaxis]
X_plot = x_plot[:, np.newaxis]

plt.scatter(ix, iy, s=30, marker='o', label="training points")

for count, degree inenumerate([3, 6, 20]):
    model = make_pipeline(PolynomialFeatures(degree), Ridge())
    model.fit(X, iy)
    y_plot = model.predict(X_plot)
    plt.plot(x_plot, y_plot, label="degree %d"% degree)

plt.legend(loc='upper left')
plt.show()
```

The combined graph shows how the different polynomials' coefficients describe the data population in different ways. The 20 degree polynomial shows clearly how it adjusts perfectly for the trained dataset, and after the known values, it diverges almost spectacularly, going against the goal of generalizing for future data.



Curve fitting of the initial dataset, with polynomials of increasing values.

Linear regression with gradient descent in practice

So now we are working with gradient descent techniques in practice for the first time! The concepts we are now practicing will serve us well during the rest of the book. Let's start by importing the prerequisite library, as always. We will use NumPy for numeric processing, and Seaborn and matplotlib for representation:

```
import numpy as np
```

```
import seaborn as sns
%matplotlib inline
import matplotlib.pyplot as plt
sns.set(style='whitegrid', context='notebook')
```

The loss function will be the guide for us to know how well we are doing. As we saw in the theoretical section, the least squares method will be used.

Note

You can review the *J* or loss function definition and properties in the previous sections.

So, this `least_squares` function will receive the current regression line parameters, b_0 and b_0, and the data elements to measure how good our representation of reality is:

```
def least_squares(b0, b1, points):
    totalError =0
    N=float(len(points))
    for x,y in points:
        totalError += (y - (b1 * x + b0)) **2return totalError /2.*N
```

Here, we will define each step of the recurrence. As parameters, we will receive the current b_0 and b_1, the points used to train the model, and the learning rate. On line five of the `step_gradient` function, we see the calculation of both gradients, and then we create the `new_b0` and `new_b1` variables, updating their values in the error direction, scaled by the learning rate. On the last line, we return the updated values and the current error level after all points have been used for the gradient:

```
def step_gradient(b0_current, b1_current, points, learningRate):
    b0_gradient =0
    b1_gradient =0
    N =float(len(points))
    for x,y in points:
        b0_gradient += (1/N) * (y - ((b1_current * x) + b0_current))
        b1_gradient += (1/N) * x * (y - ((b1_current * x) + b0_current))
    new_b0 = b0_current + (learningRate * b0_gradient)
    new_b1 = b1_current + (learningRate * b1_gradient)
    return [new_b0, new_b1, least_squares(new_b0, new_b1, points)]
```

Then, we define a function that will run a complete training outside the model so we can check all the combinations of parameters in one place. This function will initialize the parameters and will repeat the gradient step a fixed number of times:

```
def run_gradient_descent(points, starting_b0, starting_b1, learning_rate, num_iterations):
    b0 = starting_b0
    b1 = starting_b1
    slope=[]
    intersect=[]
    error=[]
    for i inrange(num_iterations):
        b0, b1 , e= step_gradient(b0, b1, np.array(points), learning_rate)
        slope.append(b1)
        intersect.append(b0)
        error.append(e)
    return [b0, b1, e, slope, intersect,error]
```

Note

This process could prove inefficient when the convergence rate is high, wasting

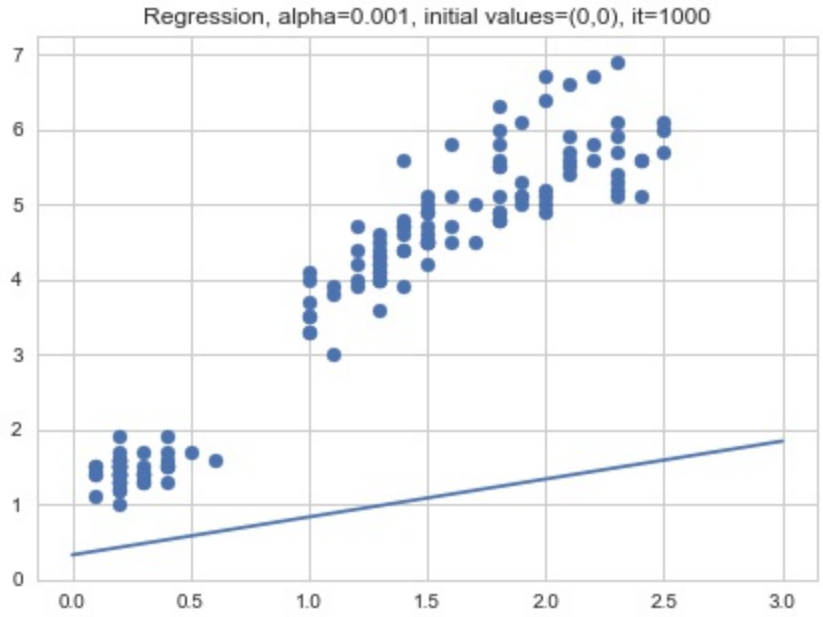
precious CPU iterations. A more clever stop condition would consist of adding an acceptable error value, which would stop the iteration.

Well, time to try our model! Let's start loading the Iris dataset again, for reference, and as a means of checking the correctness of our results. We will use the `petal_width` and `petal_length` parameters, which we have already seen and decided they are good candidates for linear regression. The `dstack` command from NumPy allows us to merge the two columns, which we converted to a list to discard the column headers. The only caveat is that the resulting list has an unused extra dimension, which we discard using the `[0]` index selector:

```
iris = sns.load_dataset('iris')
X=iris['petal_width'].tolist()
Y=iris['petal_length'].tolist()
points=np.dstack((X,Y))[0]
```

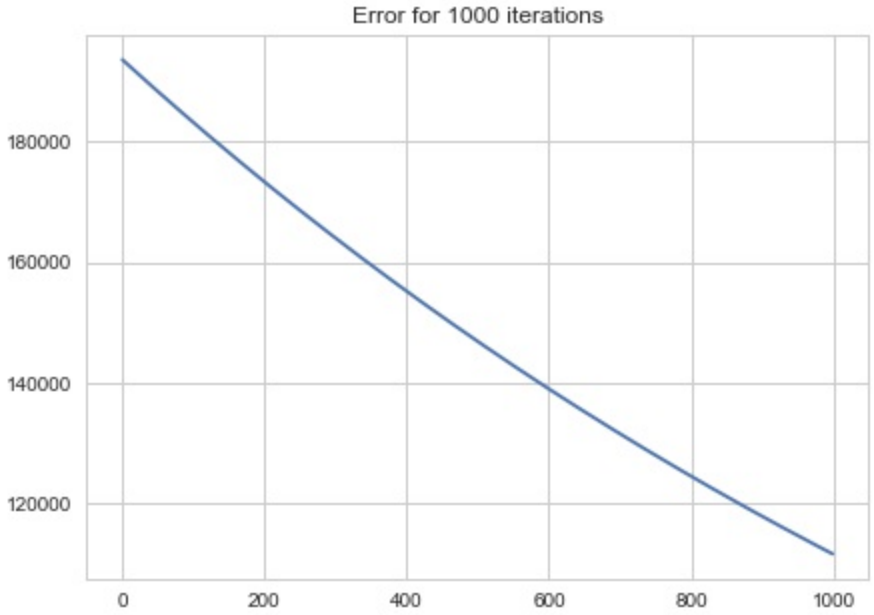
So, let's try our model with what seem to be good initial parameters, a `0.0001` learning rate, initial parameters at `0`, and `1000` iterations; lets see how it behaves:

```
learning_rate =0.0001
initial_b0 =0
initial_b1 =0
num_iterations =1000
[b0, b1, e, slope, intersect, error] = run_gradient_descent(points, initial_b0, initial_b1,
learning_rate, num_iterations)

plt.figure(figsize=(7,5))
plt.scatter(X,Y)
xr=np.arange(0,3.5)
plt.plot(xr,(xr*b1)+b0);
plt.title('Regression, alpha=0.001, initial values=(0,0), it=1000');
```



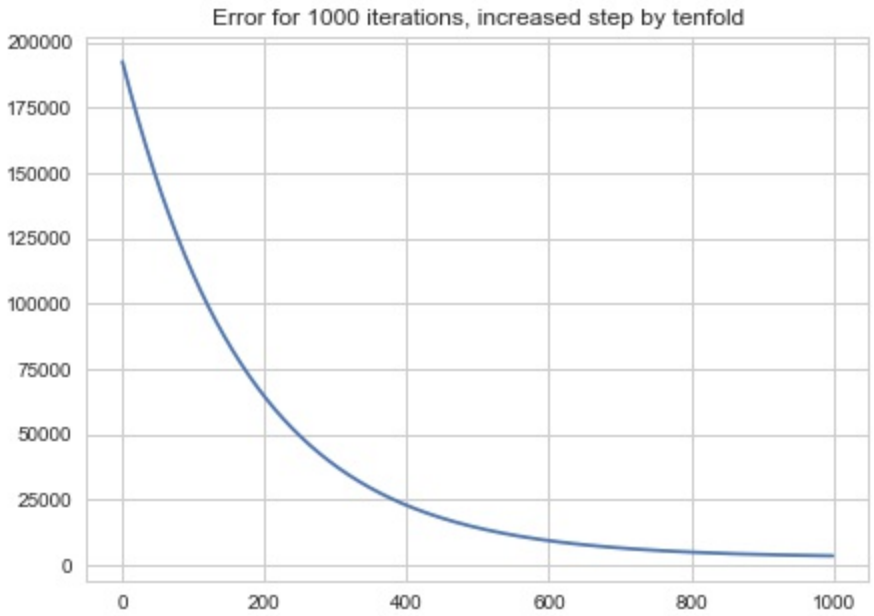
Well, that's bad; clearly, we are not yet there. Let's see what happened with the error during training:

```
plt.figure(figsize=(7,5))
xr=np.arange(0,1000)
plt.plot(xr,np.array(error).transpose());
plt.title('Error for 1000 iterations');
```



The process seems to be working, but it's a bit slow. Maybe we can try to increase the step by a factor of 10 to see if it converges quickly? Let's check:

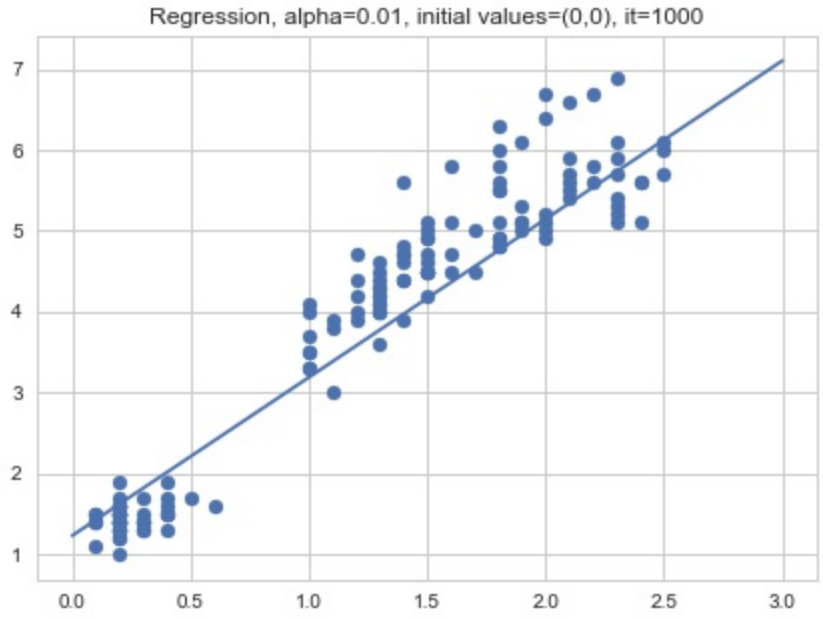
```
learning_rate =0.001#Last one was 0.0001
initial_b0 =0
initial_b1 =0
num_iterations =1000
[b0, b1, e, slope, intersect, error] = run_gradient_descent(points, initial_b0, initial_b1,
learning_rate, num_iterations)
plt.figure(figsize=(7,5))
xr=np.arange(0,1000)
plt.plot(xr,np.array(error).transpose());
plt.title('Error for 1000 iterations, increased step by tenfold');
```



That was better! The process converges much more quickly. Let's check how the regressed line looks now:

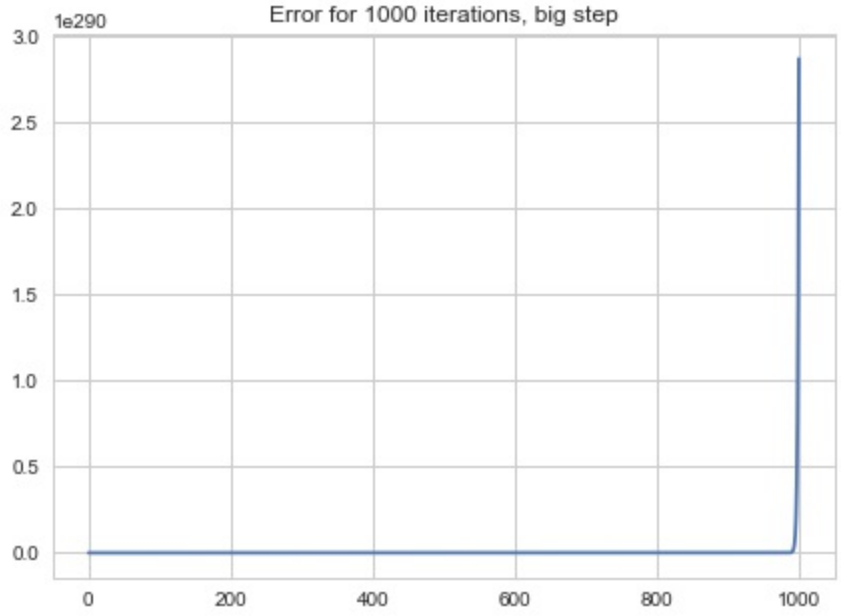
```
plt.figure(figsize=(7,5))
plt.scatter(X,Y)
```

```
xr=np.arange(0,3.5)
plt.plot(xr,(xr*b1)+b0);
plt.title('Regression, alpha=0.01, initial values=(0,0), it=1000');
```



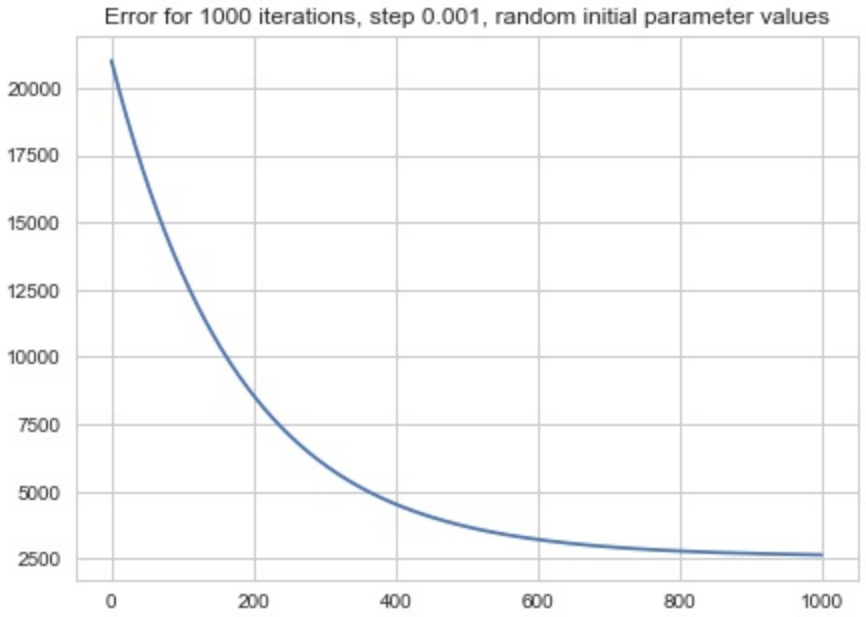
Yes! It looks much better. We could think we are done, but a developer always wants to go faster. Let's see what would occur if we wanted to go faster, with an enormous step of 2, for example:

```
learning_rate =0.85#LAst one was 0.0001
initial_b0 =0
initial_b1 =0
num_iterations =1000
[b0, b1, e, slope, intersect, error] = run_gradient_descent(points, initial_b0, initial_b1,
learning_rate, num_iterations)
plt.figure(figsize=(7,5))
xr=np.arange(0,1000)
plt.plot(xr,np.array(error).transpose());
plt.title('Error for 1000 iterations, big step');
```



This is a bad move; as you can see, the error finally went to infinity! What happens here? Simply, the steps we are taking are so radical that instead of slicing the imaginary bowl we described before, we are just jumping around the surface, and as the iterations advance, we began to escalate the accumulated errors without control. Another measure that could be taken is to improve our seed values, which, as you have seen started, with a value of `0`. This is a very bad idea in general for this technique, especially when you are working with data that is not normalized. There are more reasons for this, which you can find in more advanced literature. So, let's try to initialize the parameter on a pseudo-random location in order to allow the graphics to be the same across the code examples, and see what happens:

```
learning_rate =0.001#Same as last time
initial_b0 =0.8#pseudo random value
initial_b1 =1.5#pseudo random value
num_iterations =1000
[b0, b1, e, slope, intersect, error] = run_gradient_descent(points, initial_b0, initial_b1,
learning_rate, num_iterations)
plt.figure(figsize=(7,5))
xr=np.arange(0,1000)
plt.plot(xr,np.array(error).transpose());
plt.title('Error for 1000 iterations, step 0.001, random initial parameter values');
```

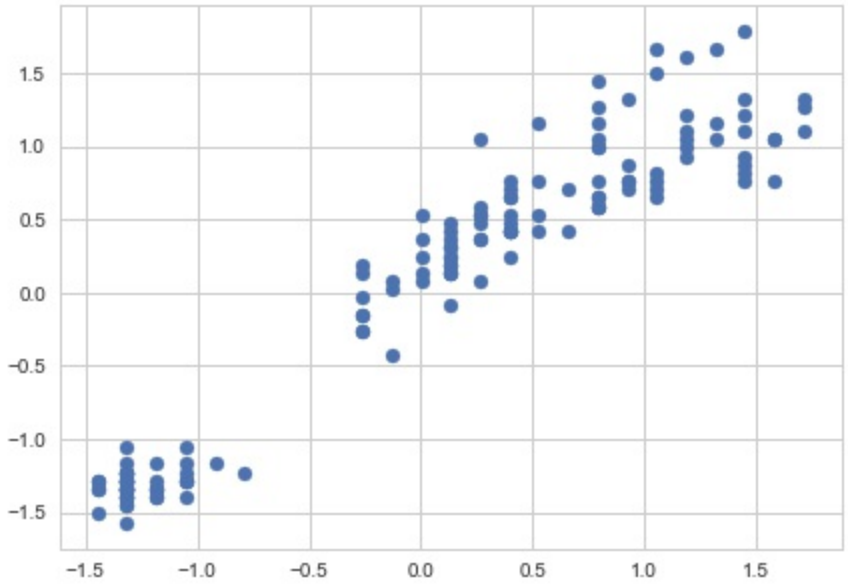


As you can see, even if you have the same sloppy error rate, the initial error value decreases tenfold (from 2e5 to 2e4). Now let's try a final technique to improve the convergence of the parameters based on the normalization of the input values. As you have already studied in Chapter 2, *The Learning Process*, it consists of centering and scaling the data. What's the effect of that operation on the data? Using a graphical image, when data is not normalized, the error surface tends to be shallow and the values oscillate a lot. The normalization transforms that data into a more deep surface, with more definite gradients towards the center:

```
learning_rate =0.001#Same as last time
initial_b0 =0.8#pseudo random value
initial_b1 =1.5#pseudo random value
num_iterations =1000
x_mean =np.mean(points[:,0])
y_mean = np.mean(points[:,1])
x_std = np.std(points[:,0])
y_std = np.std(points[:,1])

X_normalized = (points[:,0] - x_mean)/x_std
Y_normalized = (points[:,1] - y_mean)/y_std
```

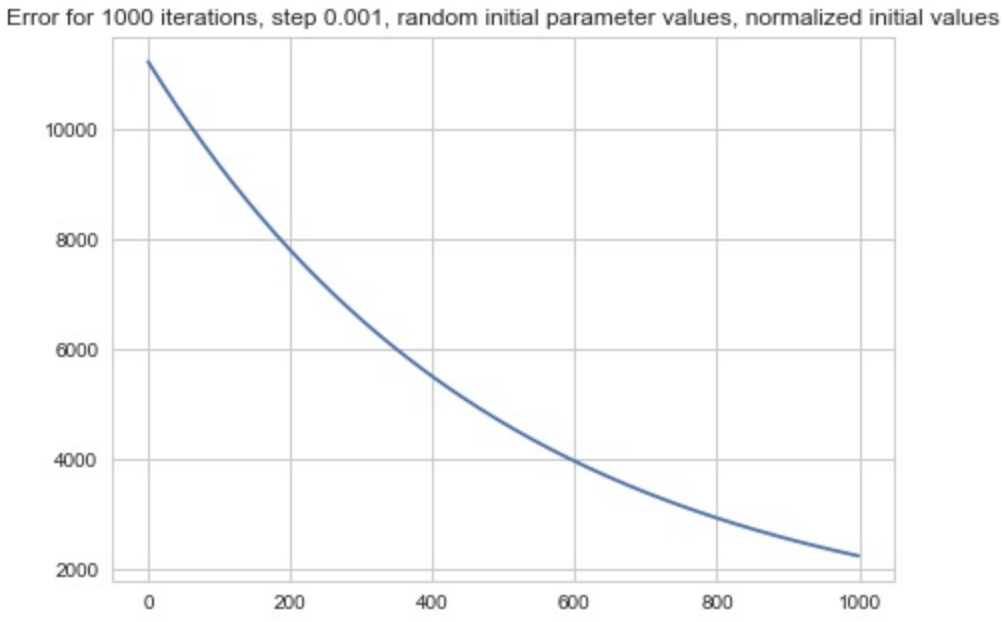
```
plt.figure(figsize=(7,5))
plt.scatter(X_normalized,Y_normalized)
```

```
<matplotlib.collections.PathCollection at 0x7f9cad8f4240>
```



Now that we have this set of clean and tidy data, let's try again with the last slow convergence parameters, and see what happens to the error minimization speed:

```
points=np.dstack((X_normalized,Y_normalized))[0]
learning_rate =0.001#Same as last time
initial_b0 =0.8#pseudo random value
initial_b1 =1.5#pseudo random value
num_iterations =1000
[b0, b1, e, slope, intersect, error] = run_gradient_descent(points, initial_b0, initial_b1,
learning_rate, num_iterations)
plt.figure(figsize=(7,5))
xr=np.arange(0,1000)

plt.plot(xr,np.array(error).transpose());
plt.title('Error for 1000 iterations, step 0.001, random initial parameter values, normalized
initial values');
```



A very good starting point indeed! Just by normalizing the data, we have half the initial error values, and the error went down 20% after 1,000 iterations. The only thing we have to remember is to denormalize after we have the results, in order to have the initial scale and data center. So, that's all for now on gradient descent. We will be revisiting it in the next chapters for new challenges.

Logistic regression

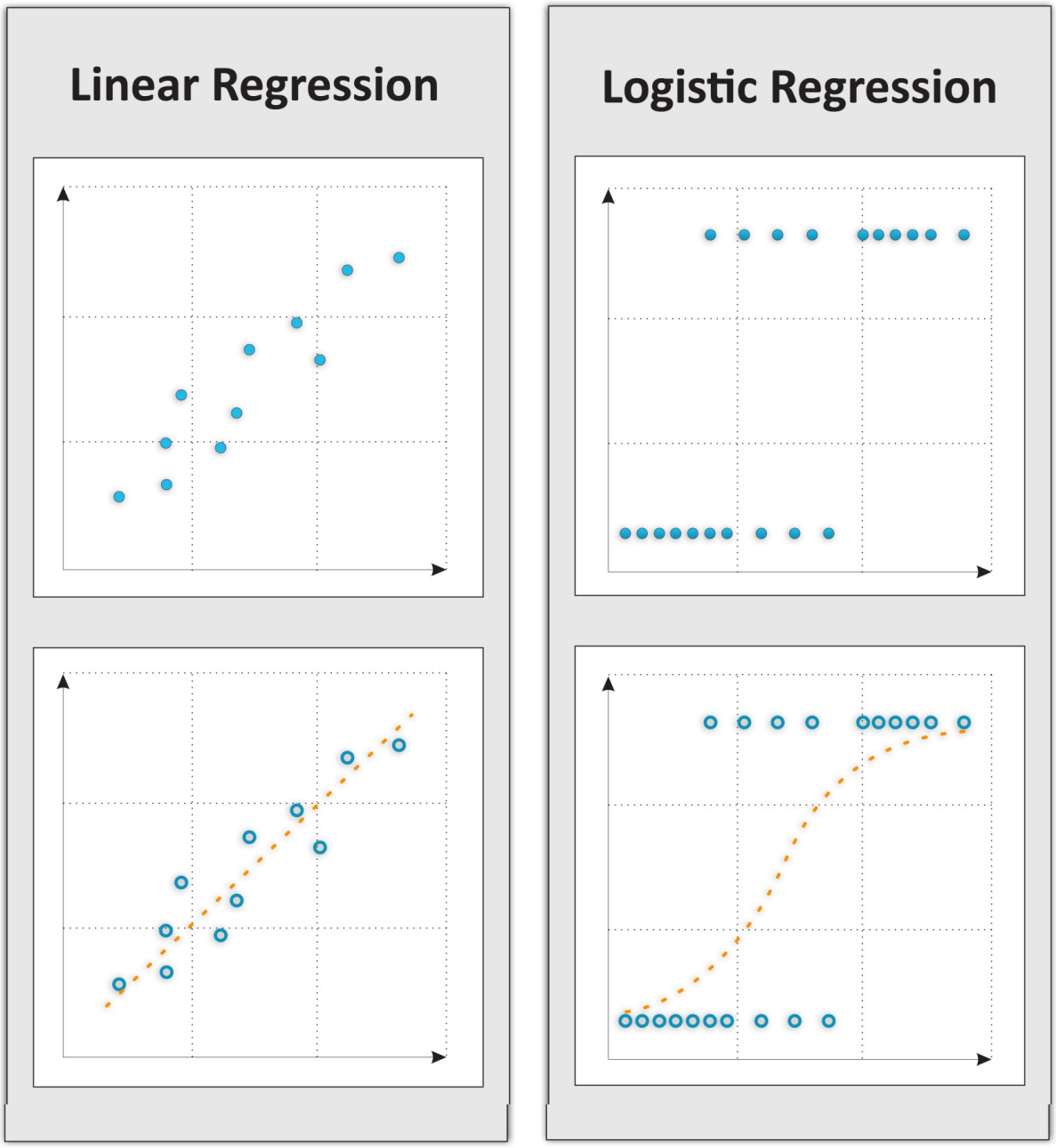
The way of this book is one of generalizations. In the first chapter, we began with simpler representations of the reality, and so simpler criteria for grouping or predicting information structures.

After having reviewed linear regression, which is used mainly to predict a real value following a modeled linear function, we will advance to a generalization of it, which will allow us to separate binary outcomes (indicating that a sample belongs to a class), starting from a previously fitted linear function. So let's get started with this technique, which will be of fundamental use in almost all the following chapters of this book.

Problem domain of linear regression and logistic regression

To intuitively understand the problem domain of the logistic regression, we will employ a graphical representation.

In the first we show the linear fitting function, which is the main objective of the whole model building process, and at the bottom, the target data distribution. As you clearly see, data is now of a binary nature, and a sample belongs to one or another options, nothing in the middle. Additionally, we see that the modelling function is of a new type; we will later name it and study its properties. You may wonder what this has it to do with a linear function? Well, as we will see later, it will be inside of that s-like function, adapting its shape.



Simplified depiction of the common data distributions where Linear or Logistic regression are applied.

Summarizing, Linear regression can be imagined as a continuum of increasingly growing values. The other is a domain where the output can have just two different values based on the *x* value. In the particular case shown in the image, we can see a clear trend towards one of the possible outcomes, as the independent variable increases, and the sigmoid function allows us to transition from two outcomes, which don't have a clear separation in time, which gives us an estimated probability in the overlap zone of the non-occurrence/occurrence zone.

In some ways the terminology is a bit confusing, given that we are doing a regression that is obtaining a continuous value, but in reality, the final objective is building a prediction for a classification problem with discrete variables.

The key here is to understand that we will obtain probabilities of an item pertaining to a class and not a totally discrete value.

Logistic function predecessor – the logit functions

Before we study the logistic function, we will review the original function on which it is based, the logit function, which gives it some of its more general properties.

Essentially, when we talk about the logit function, we are working with the function of a random variable *p*; more specifically, one corresponding with a Bernoulli distribution.

Link function

As we are trying to build a **generalized linear model**, we want to start from a linear function and obtain a mapping to a probability distribution from the dependent variable.

Since the output type of our model is of a binary nature, the normally chosen distribution is the Bernoulli distribution, and the link function, leaning toward the logistic function, is the **logit function**.

Logit function

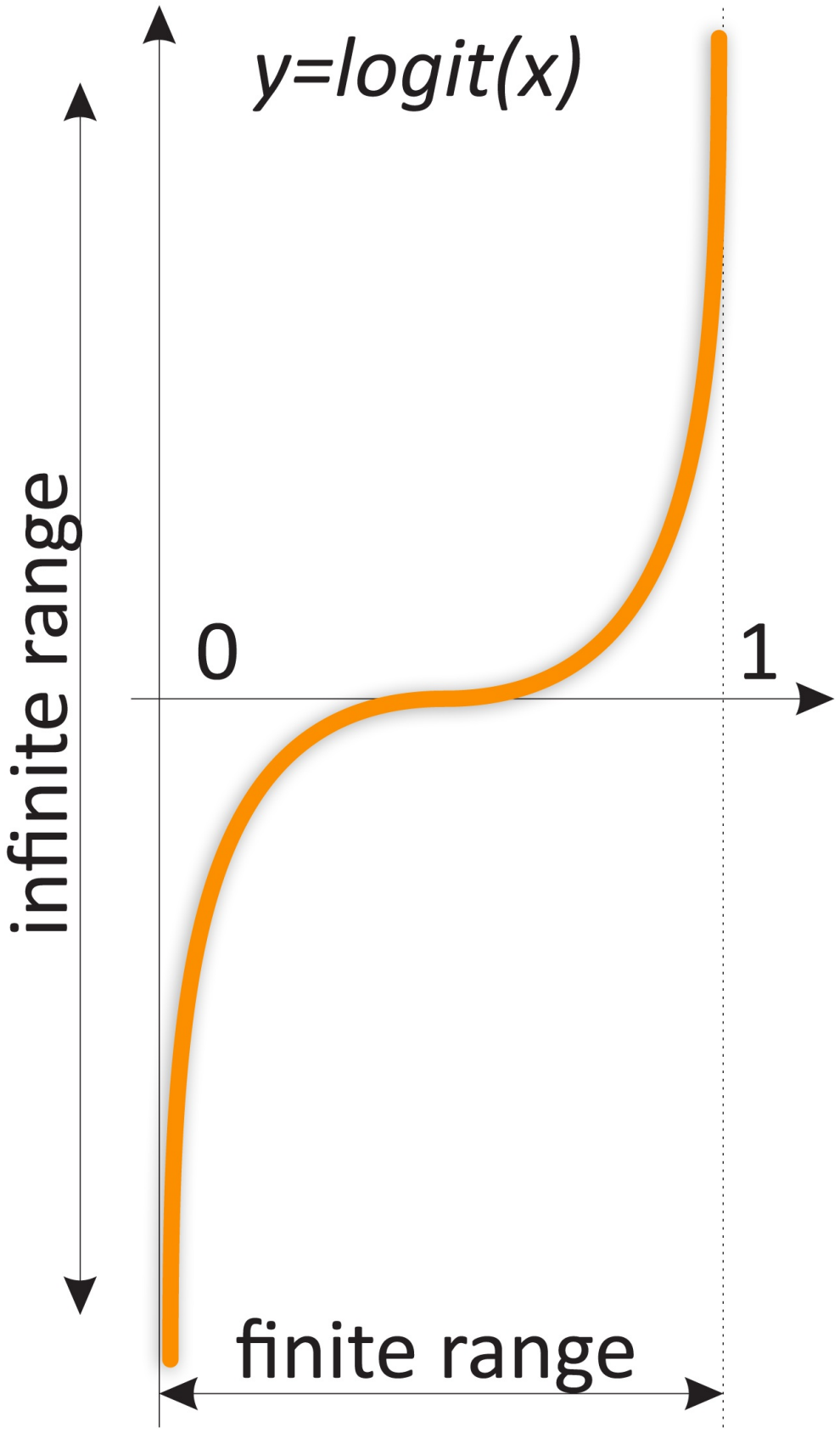
One of the possible variables that we could utilize is the natural logarithm of the odds, that *p* equals one. This function is called the logit function:

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

We can also call the logit function a log-odd function, because we are calculating the log of the odds *(p/1-p)* for a given probability *p*.

Logit function properties

So, as we can visually infer, we are replacing *x* with a combination of the independent variables, no matter their values, and replacing *x* with any occurrence from minus infinity to infinity. We are scaling the response to be between *0* and *1*.


y=logit(x)
infinite range
0
1
finite range

Depiction of the main range characteristics of the logit function

The importance of the logit inverse

Suppose that we calculate the inverse of the logit function. The simple inverse transformation of the logit will give us the following expression:

$$\text{logit}^{-1}(\alpha) = \text{logistic}(\alpha) = \frac{1}{1+\exp(-\alpha)} = \frac{\exp(\alpha)}{\exp(\alpha)+1}$$

Analytical definition of the logit function

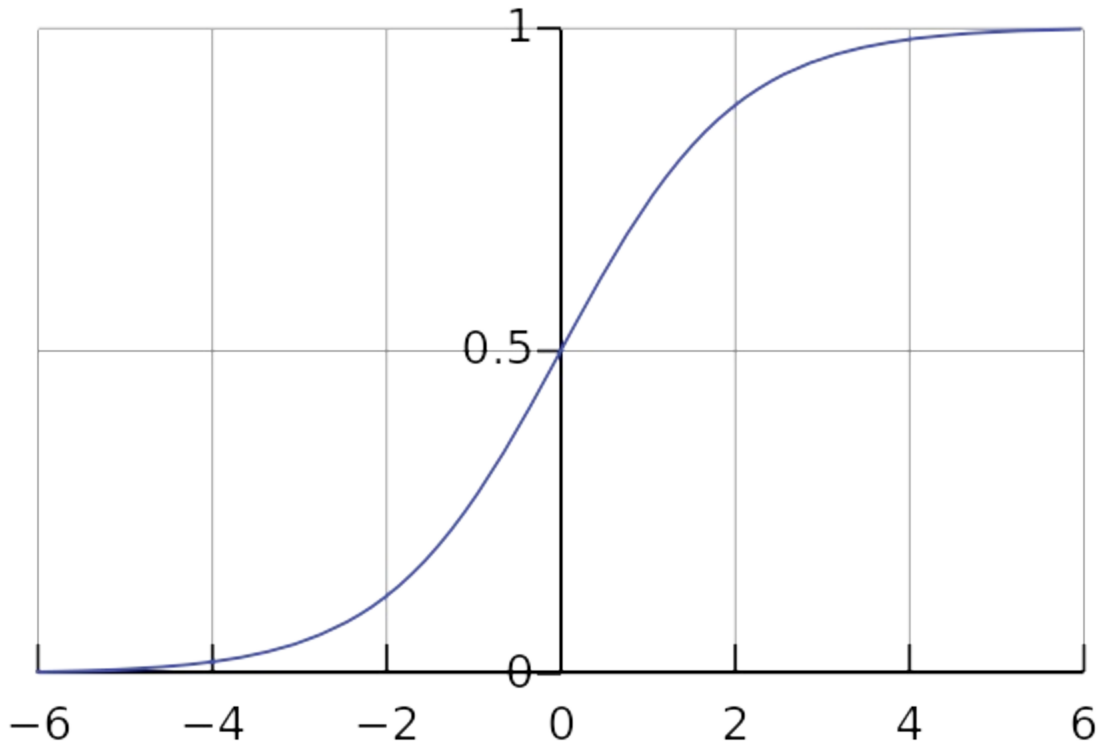
This function is nothing less than a **sigmoid function**.

The sigmoid or logistic function

The logistic function will represent the binary options we are representing in our new regression tasks. The logistic function is defined as follows (changing the independent variable from α to *t* for clarity):

$$\sigma(t) = \frac{e^t}{e^t+1} = \frac{1}{1+e^{-t}}$$

You will find this new figure common in the following sections, because it will be used very frequently as an activation function for neural networks and other applications. In the following figure, you will find the graphical representation of the sigmoid function:



Standard sigmoid

How can we interpret and give this function a meaning for our modeling tasks? The normal interpretation of this equations is that *t* represents a simple independent variable, but we will improve this model, assuming that *t* is a linear function of a single explanatory variable *x* (the case where *t* is a linear combination of multiple explanatory variables is treated similarly), expressing it as follows:

$$t = \beta_0 + \beta_1 x$$

So, we can start again from the original logit equation:

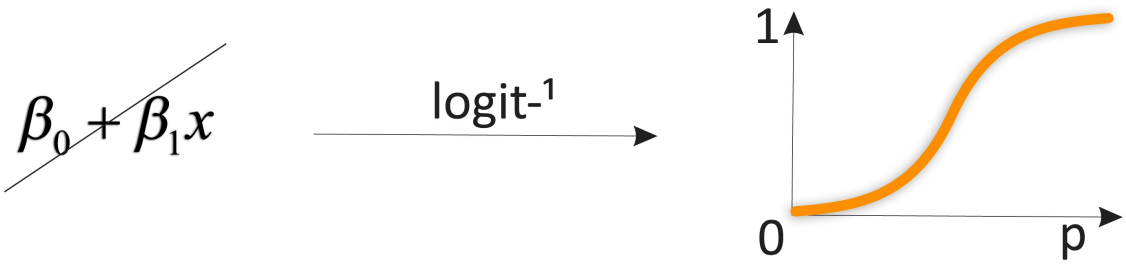
$$\operatorname{logit}(p) = \ln\left(\frac{p}{1-p}\right) = \beta_0 + b_1 x$$

We will reach to the regression equation, which will give us the regressed probability with the following equation:

$$\hat{p} = \frac{e^{\beta_0 + \beta_1 x_1}}{1 + e^{\beta_0 + \beta_1 x_1}}$$

Note that *p* (hat) denotes an estimated probability. What will give us a measure of how approximate we are to the solution? Of course, a carefully chosen loss function!

The following image shows how the mapping from an infinite domain of possible outcomes which will be finally reduced to the *[0,1]* range, with *p* being the probability of the occurrence of the event being represented. This is shown in a simple schema, which is the structure and domain transformation of the logit function (from a linear one to a probability modeled by a Sigmoid):



Function mapping for the logit of a linear equation, resulting in a sigmoid curve

What changes will affect the parameters of the linear function? They are the values that will change the central slope and the displacement from zero of the sigmoid function, allowing it to more exactly reduce the error between the regressed values and the real data points.

Properties of the logistic function

Every curve in the function space can be described by the possible objectives it could be applied to. In the case of the logistic function, they are as follows:

- Model the probability of an event *p*, depending on one or more independent variables. For example, the probability of being awarded a prize, given previous qualifications

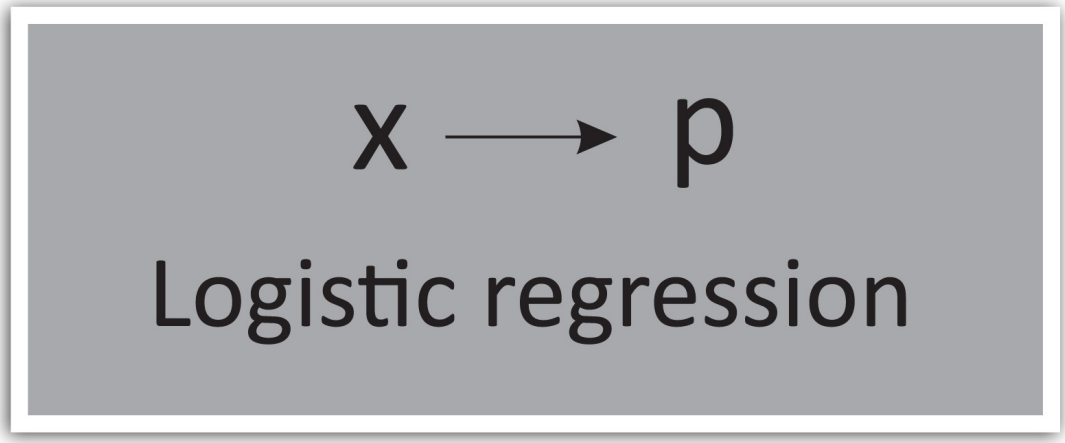
- Estimate (this is the regression part) p for a determined observation, related to the possibility of the event not occurring.
- Predict the effect of the change of independent variables using a binary response.
- Classify observations by calculating the probability of an item being of a determined class.

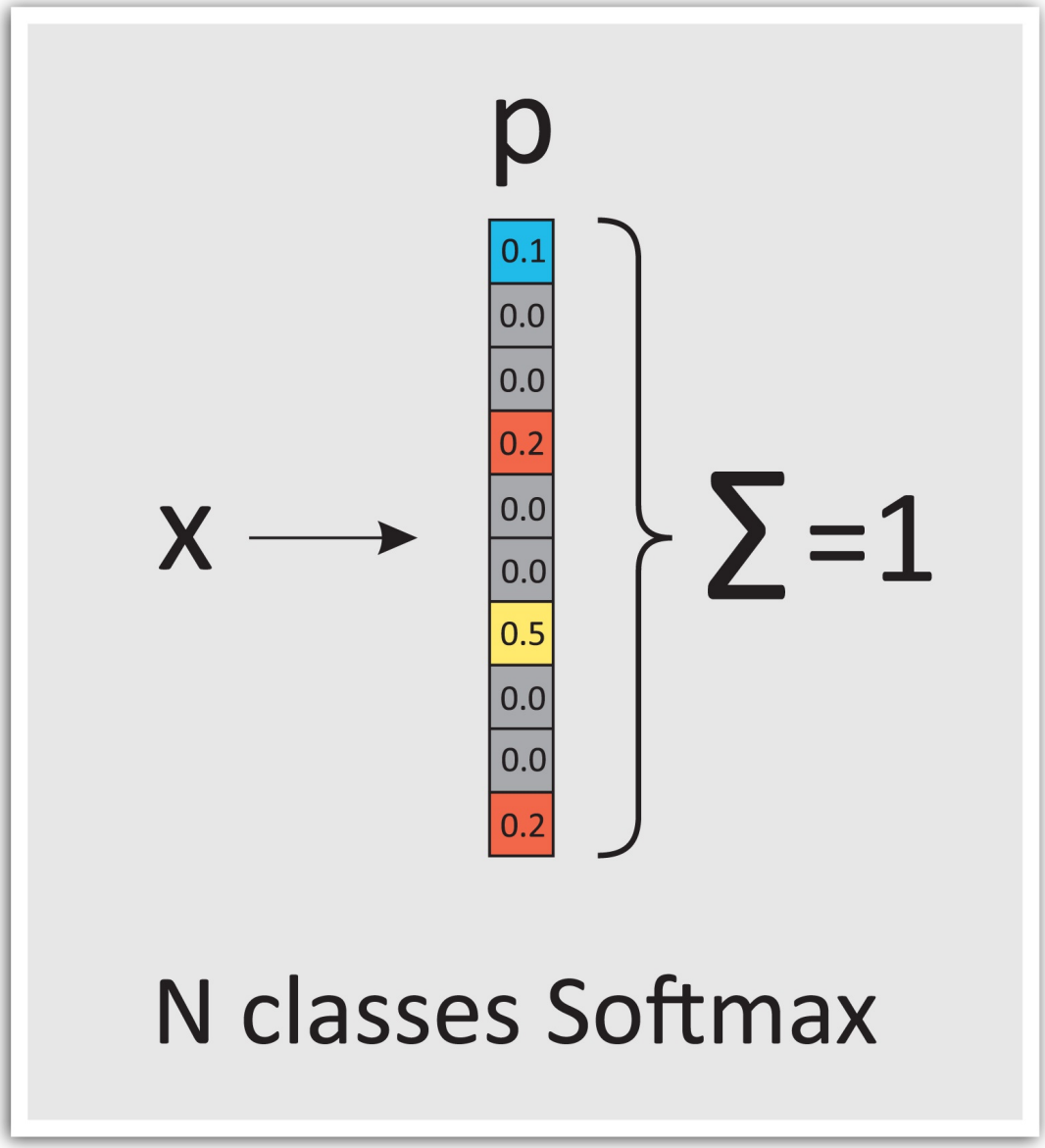
Multiclass application – softmax regression

Up until now, we have been classifying in the case of only having two classes, or in probabilistic language, event occurrence probabilities p. But this logistic regression can also be conveniently generalized to account for many classes.

As we saw before, in logistic regression we assumed that the labels were binary ($y(i) \in \{0,1\}$), but softmax regression allows us to handle $y(i) \in \{1,\ldots,K\}$, where K is the number of classes and the label y can take on K different values, rather than only two.

Given a test input x, we want to estimate the probability that $P(y=k|x)$ for each value of $k=1, \ldots,K$. The softmax regression will make this output a K-dimensional vector (whose elements sum to 1), giving us our K estimated probabilities:


x
p
Logistic regression


p
x
0.1
0.0
0.0
0.2
0.0
0.0
0.5
0.0
0.0
0.2
Σ=1
N classes Softmax

Comparison between the univariate logisic regression outcome and N classes softmax regression

Practical example – cardiac disease modeling with logistic regression

It's time to finally solve a practical example with the help of the very useful logistic regression. In this first exercise, we will work on predicting the probability of having coronary heart disease, based on the age of the population. It's a classic problem, which will be a good start for understanding this kind of regression analysis.

The CHDAGE dataset

For the first simple example, we will use a very simple and often studied dataset, which was published in *Applied Logistic Regression*, from *David W. Hosmer, Jr. Stanley Lemeshow and Rodney X. Sturdivant*. We list the age in years (`AGE`) and the presence or absence of evidence of significant **coronary heart disease** (**CHD**) for 100 subjects in a hypothetical study of risk factors for heart disease. The table also contains an identifier variable (`ID`) and an age group variable (`AGEGRP`).

The outcome variable is `CHD`, which is coded with a value of `0` to indicate that `CHD` is absent, or `1` to indicate that it is present in the individual. In general, any two values could be used, but we have found it most convenient to use zero and one. We refer to this dataset as the `CHDAGE` data.

Dataset format

The `CHDAGE` dataset is a two-column CSV file that we will download from an external repository. In the first chapter, we used native TensorFlow methods to read the dataset. In this chapter, we will alternatively use a complementary and popular library to get the data. The cause for this new addition is that, given that the dataset only has 100 tuples, it is practical to just read it in one line, and also, we can get simple but powerful analysis methods for free from the pandas library.

In the first stage of this project, we will start loading an instance of the `CHDAGE` dataset. Then we will print vital statistics about the data, and then proceed to preprocessing. After doing some plots of the data, we will build a model composed of the activation function, which will be a softmax function, for the special case where it becomes a standard logistic regression, that is, when there are only two classes (existence or not of the illness).

Let's start by importing the required libraries:

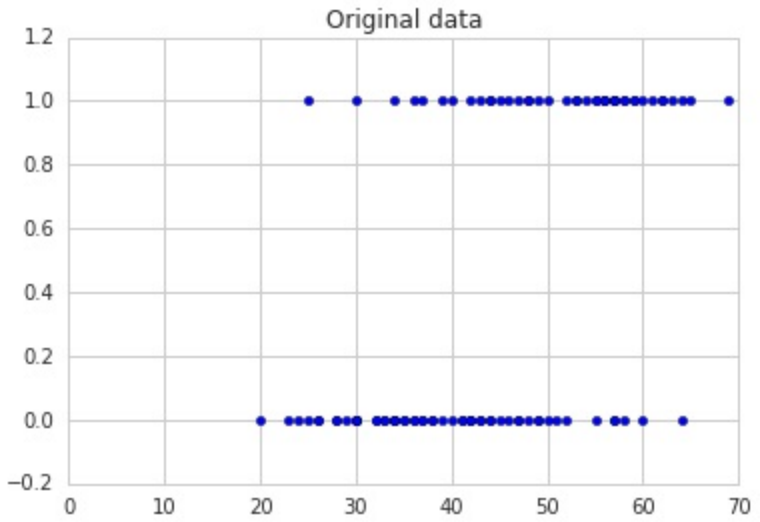
```
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn import linear_model
import seaborn.apionly as sns
%matplotlib inline
import matplotlib.pyplot as plt
sns.set(style='whitegrid', context='notebook')
```

Let's read the dataset from the CSV original file using `read_csv` from pandas, and draw the data distribution using the scatter function of matplotlib. As we can see, there is a definite pattern through the years that correlates to the presence of cardiac disease with increasing age:

```
df = pd.read_csv("data/CHD.csv", header=0)
plt.figure() # Create a new figure
```

```
plt.axis ([0,70,-0.2,1.2])
plt.title('Original data')
plt.scatter(df['age'],df['chd']) #Plot a scatter draw of the random datapoints
```

This is the current plot of the original data:



Now we will create a logistic regression model using the logistic regression object from scikit-learn, and then we will call the `fit` function, which will create a sigmoid optimized to minimize the prediction error for our training data:

```
logistic = linear_model.LogisticRegression(C=1e5)
logistic.fit(df['age'].reshape(100,1),df['chd'].reshape(100,1))
```

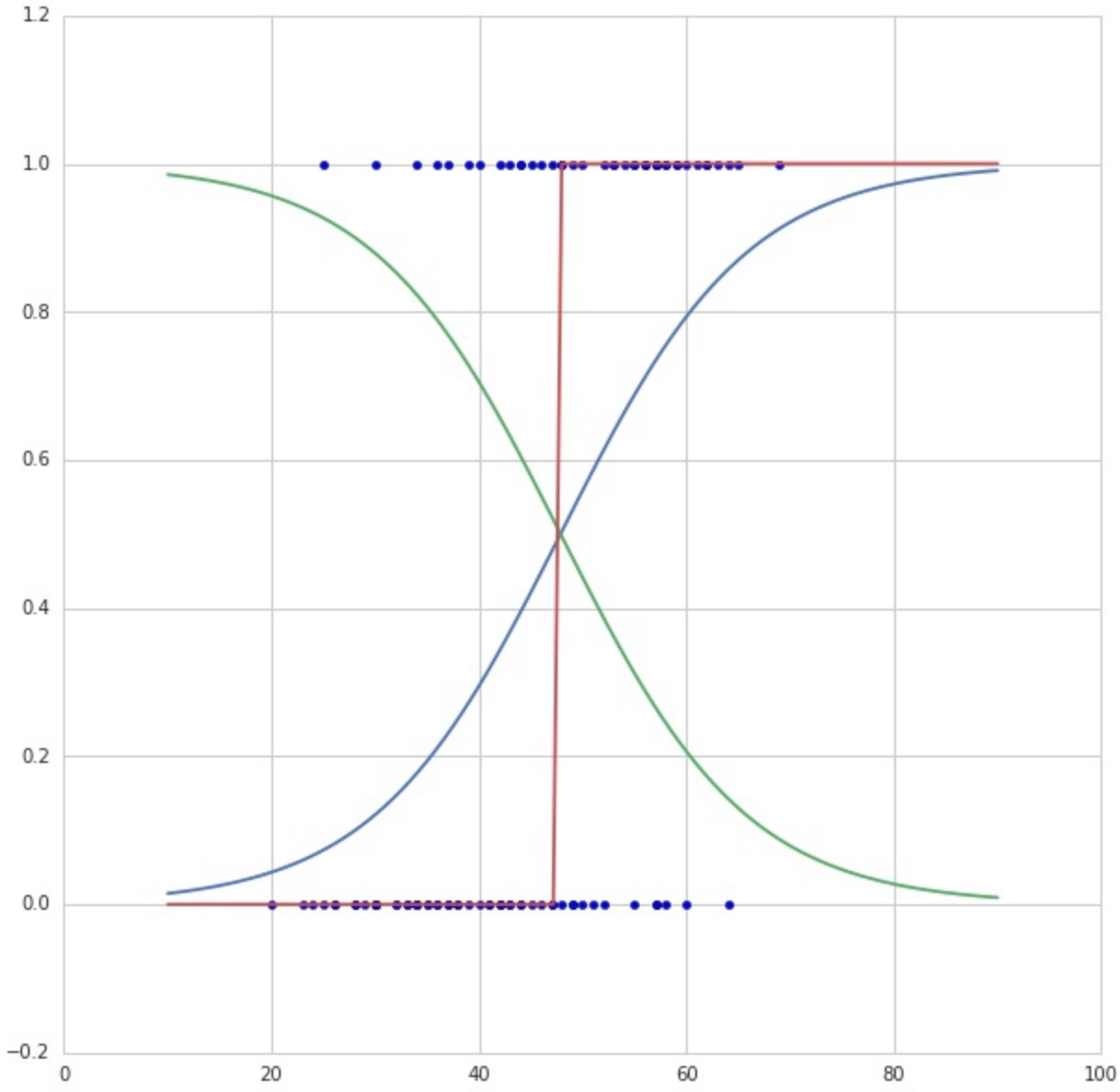
```
LogisticRegression(C=100000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

Now it's time to represent the results. Here, we will generate a linear space from 10 to 90 years with 100 subdivisions.

For each sample of the domain, we will show the probability of occurrence (1) and not occurrence (0, or the inverse of the previous one).

Additionally, we will show the predictions along with the original data points, so we can match all the elements in a single graphic:

```
x_plot = np.linspace(10, 90, 100)
oneprob=[]
zeroprob=[]
predict=[]
plt.figure(figsize=(10,10))
for i in x_plot:
    oneprob.append (logistic.predict_proba(i)[0][1]);
    zeroprob.append (logistic.predict_proba(i)[0][0]);
    predict.append (logistic.predict(i)[0]);

plt.plot(x_plot, oneprob);
plt.plot(x_plot, zeroprob)
plt.plot(x_plot, predict);
plt.scatter(df['age'],df['chd'])
```



Simultaneous plot of the original data distribution, the modeling logistic curve, and its inverse

Summary

In this chapter, we've reviewed the main ways to approach the problem of modeling data using simple and definite functions.

In the next chapter, we will be using more sophisticated models that can reach greater complexity and tackle higher-level abstractions, and can be very useful for the amazingly varied datasets that have emerged recently, starting with simple **feedforward networks**.

References

Galton, Francis, "*Regression towards mediocrity in hereditary stature.*" The Journal of the Anthropological Institute of Great Britain and Ireland 15 (1886): 246-263.

Walker, Strother H., and David B. Duncan, "*Estimation of the probability of an event as a function of several independent variables.*" Biometrika 54.1-2 (1967): 167-179.

Cox, David R, "*The regression analysis of binary sequences.*" Journal of the Royal Statistical Society. Series B (Methodological)(1958): 215-242.

Chapter 5. Neural Networks

As a developer, you have surely gained an interest in machine learning from looking at all the incredibly amazing applications that you see on your regular devices every day—automatic speech translation, picture style transfer, the ability to generate new pictures from sample ones, and so on. Brace yourself... we are heading directly into the technology that has made all these things possible.

Linear and logistic models, such as the ones we've observed, have certain limitations in terms of the complexity of the training dataset they train a model on, even when they are the basis of very articulated and efficient solutions.

How complex does a model have to be to capture the style of writing of an author, the concept of an image of a cat versus an image of a dog, or the classification of a plant based on visual elements? These things require the summation of a huge mix of low-level and high-level details captured, in the case of our brain by specialized sets of neurons, and in computer science by neural models.

Contrary to what's expected, in this book I'll spare you the typical introduction to the nervous system, its capabilities, the number of neurons in the nervous system, its chemical properties, and so on. I find that these topics add an aura of impossibility to problems, but the models we are about to learn are simple math equations with computational counterparts, which we will try to emphasize to allow you, a person with algorithmic interests, to easily understand them.

In this chapter, we will cover the following topics:

- The history of neural models, including perceptron and ADALINE
- Neural networks and the kinds of problems they solve
- The multilayer perceptron
- Implementing a simple neural layer to model a binary function

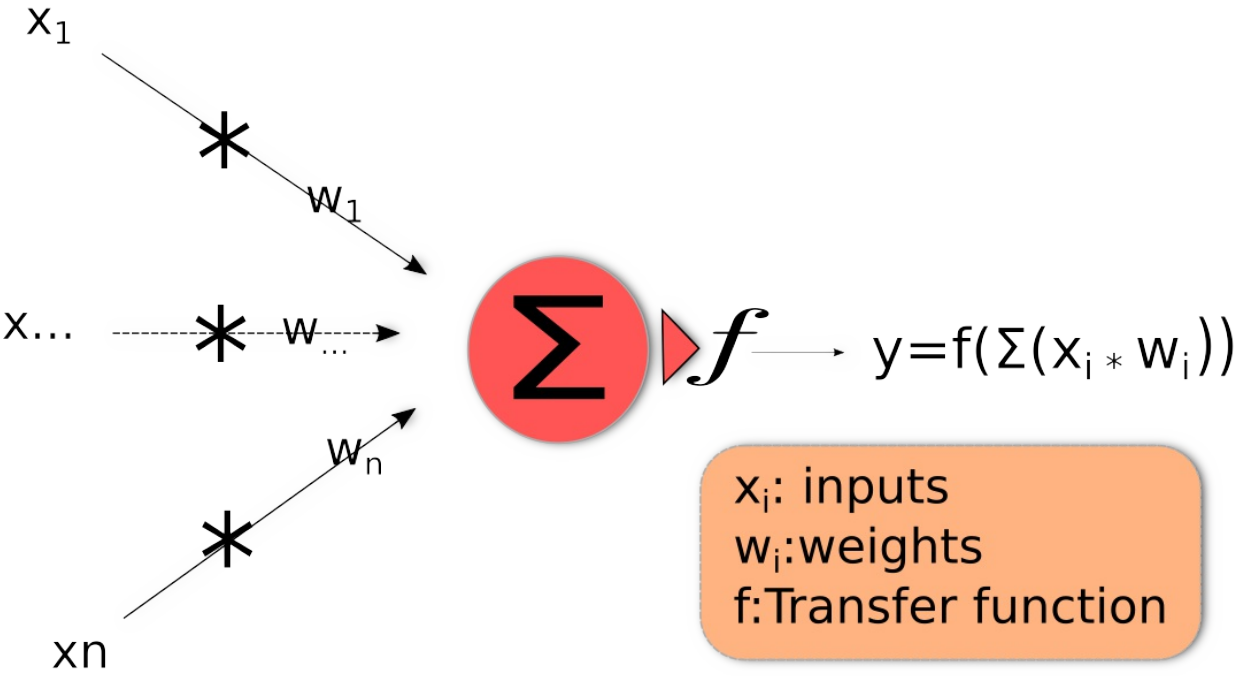
This chapter will allow you to use the building blocks of most of the awesome machine learning applications you see daily. So, let's get started!

History of neural models

Neural models, in the sense of being disciplines that try to build representations of the internal workings of the brain, originated pretty distantly in the computer science timescale. They even date back to the time when the origins of modern computing were being invented, the mid-1940s.

At that time, the fields of neuroscience and computer science began to collaborate by researching ways of emulating the way the brain processes information, starting from its constituent unit—the neuron.

The first mathematical method for representing the learning function of the human brain can be assigned to McCulloch and Pitts, in their 1943 paper *A Logical Calculus of Ideas Immanent in Nervous Activity:*



McCulloch and Pitts model

This simple model was a basic but realistic model of a learning algorithm. You will be surprised by what happens if we use a linear function as a transfer function; this is a simple linear model, as were the ones we saw in the previous chapter.

> ## Note
>
> You may have noted that we are now using the `w` letter to specify the parameters to be tuned by the model. Take this as the new standard from now on. Our old `β` parameters in linear regression models will now be `w`.

But the model hasn't determined a way to tune the parameters. Let's go forward to the 1950s and review the **perceptron** model.

The perceptron model

The perceptron model is one of the simplest ways to implement an artificial neuron. It was initially developed at the end of the 1950s, with the first hardware implementation being carried out in the 1960s. First, it was the name of a machine, and later became the name of an algorithm. Yes, perceptrons are not the strange entities we always thought they were, they're what you as a developer handle each day— algorithms!

Let's take a look at the following steps and learn how it works:

1. Initialize the weights with a random (low value) distribution.
2. Select an input vector and present it to the network.
3. Compute the output *y'* of the network for the input vector specified and the values of the weights.

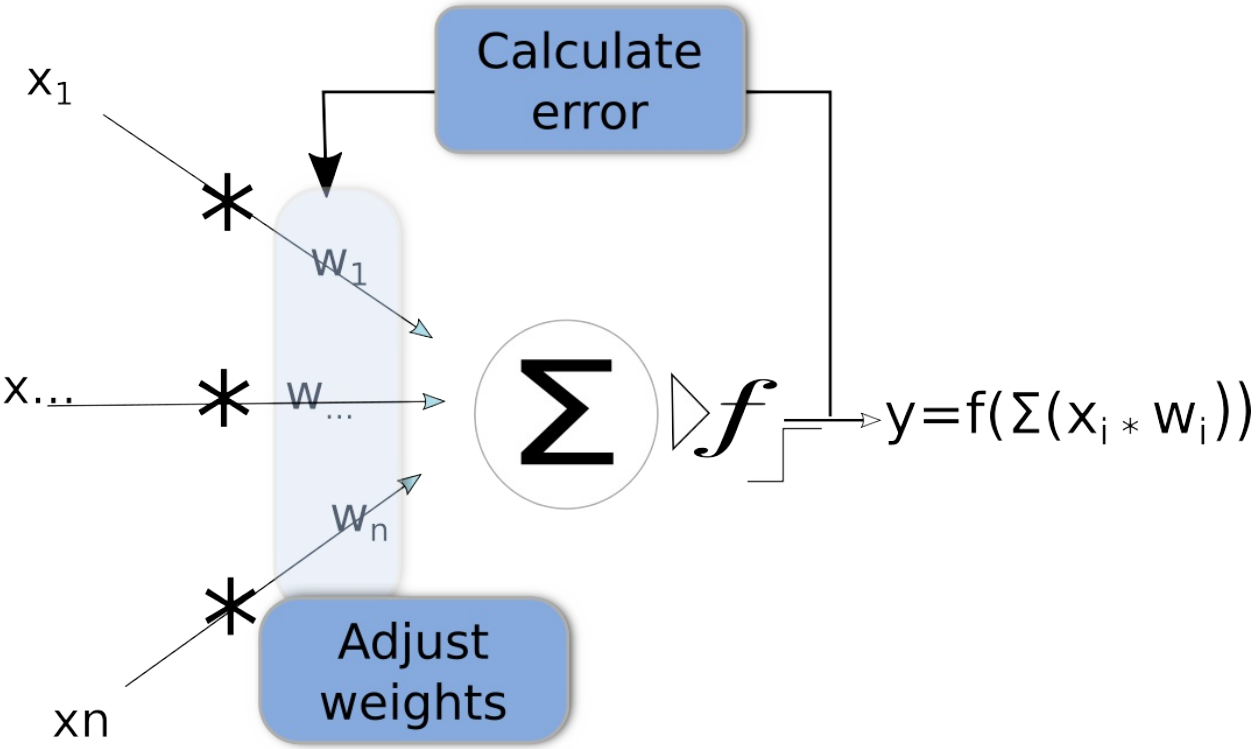
The function for a perceptron is as follows:

$$f(x) = \begin{cases} 1 & if \quad w \cdot x + b > 0 \\ 0 & otherwise \end{cases}$$

4. If *y' ≠ y*, modify all the connections, *wi*, by adding the changes *Δw =yxi*.
5. Return to *step 2*.

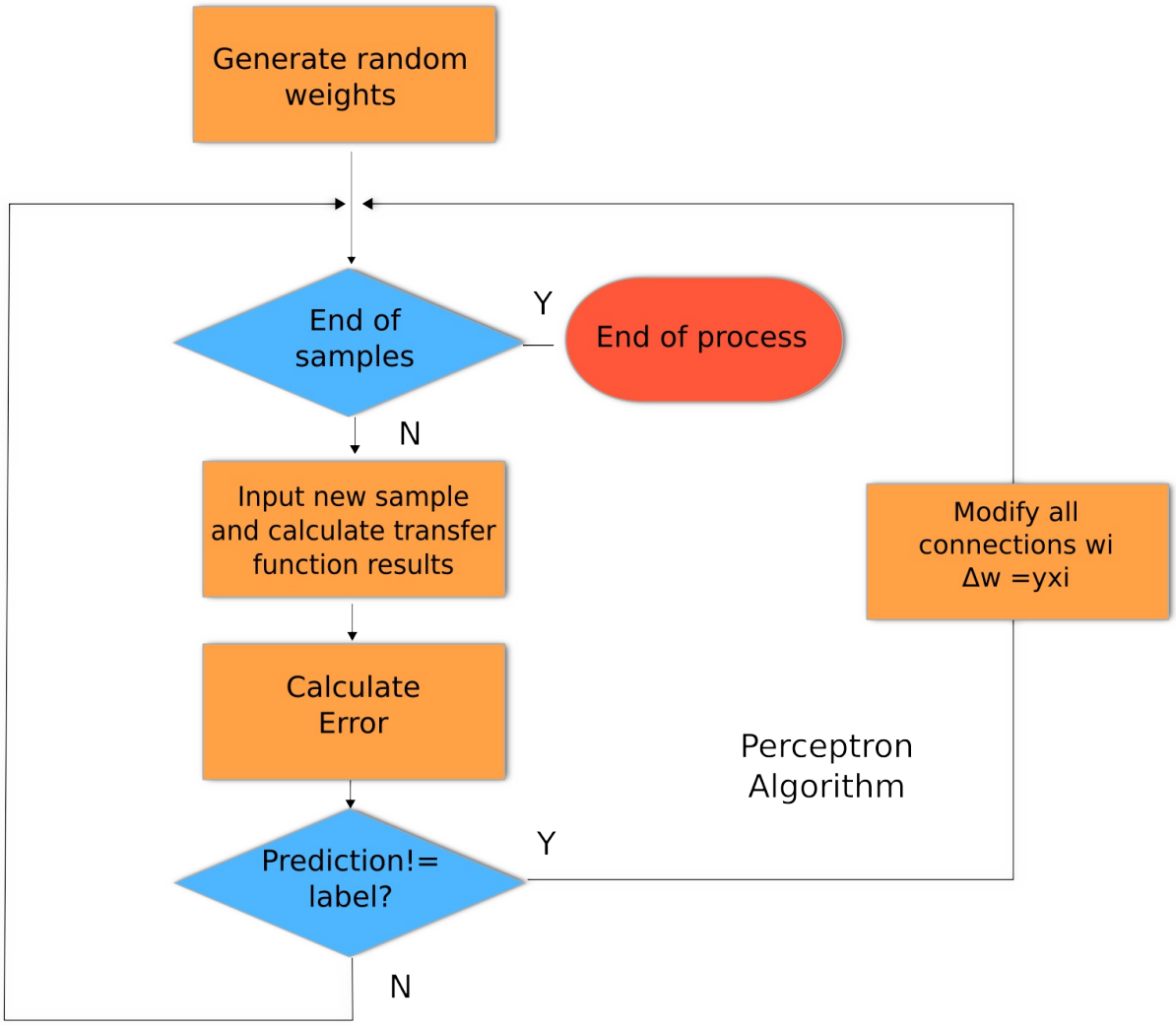
It is basically an algorithm that learns a binary classification function and maps a real function to a single binary function.

Let's depict the new architecture of the perceptron and analyze the new scheme of the algorithm in the following diagram:



Perceptron model (With changes from previous model highlighted)

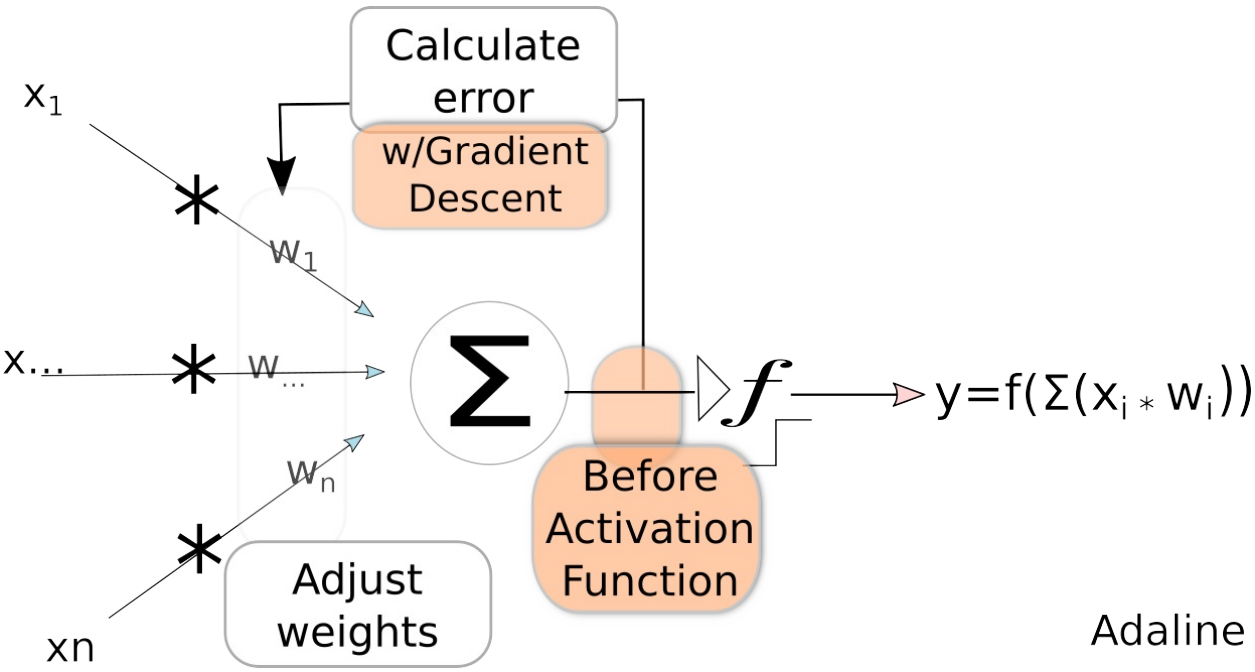
The perceptron builds on the ideas of its predecessors, but the novelty here is that we are adding a proper learning mechanism! In the following diagram, we have highlighted the new properties of the model—the feedback loop, where we calculate the error of our outcomes, and the adjustment of weights—with a predetermined formula:



Perceptron algorithm flowchart

Improving our predictions – the ADALINE algorithm

ADALINE is another algorithm (yes, remember we are talking about algorithms) used to train neural networks. ADALINE is in some ways more advanced than the perceptron because it adds a new training method: gradient descent, which should be known to you by now. Additionally, it changes the point at which the error is measured before the activation output is applied to the summation of the weights:



Adaline model (With additions from Perceptron highlighted)

So, this is the standard way of representing the ADALINE algorithm in a structural way. As an algorithm consists of a series of steps, let's aggregate those in a more detailed way, with some additional mathematical details:

1. Initialize the weights with a random (low value) distribution.
2. Select an input vector and present it to the network.
3. Compute the output *y'* of the network for the input vector specified and the values of the weights.
4. The output value that we will be taking will be the one after the summation:

$y=\Sigma(xi * wi)$

5. Compute the error, comparing the model output with the right label *o*:

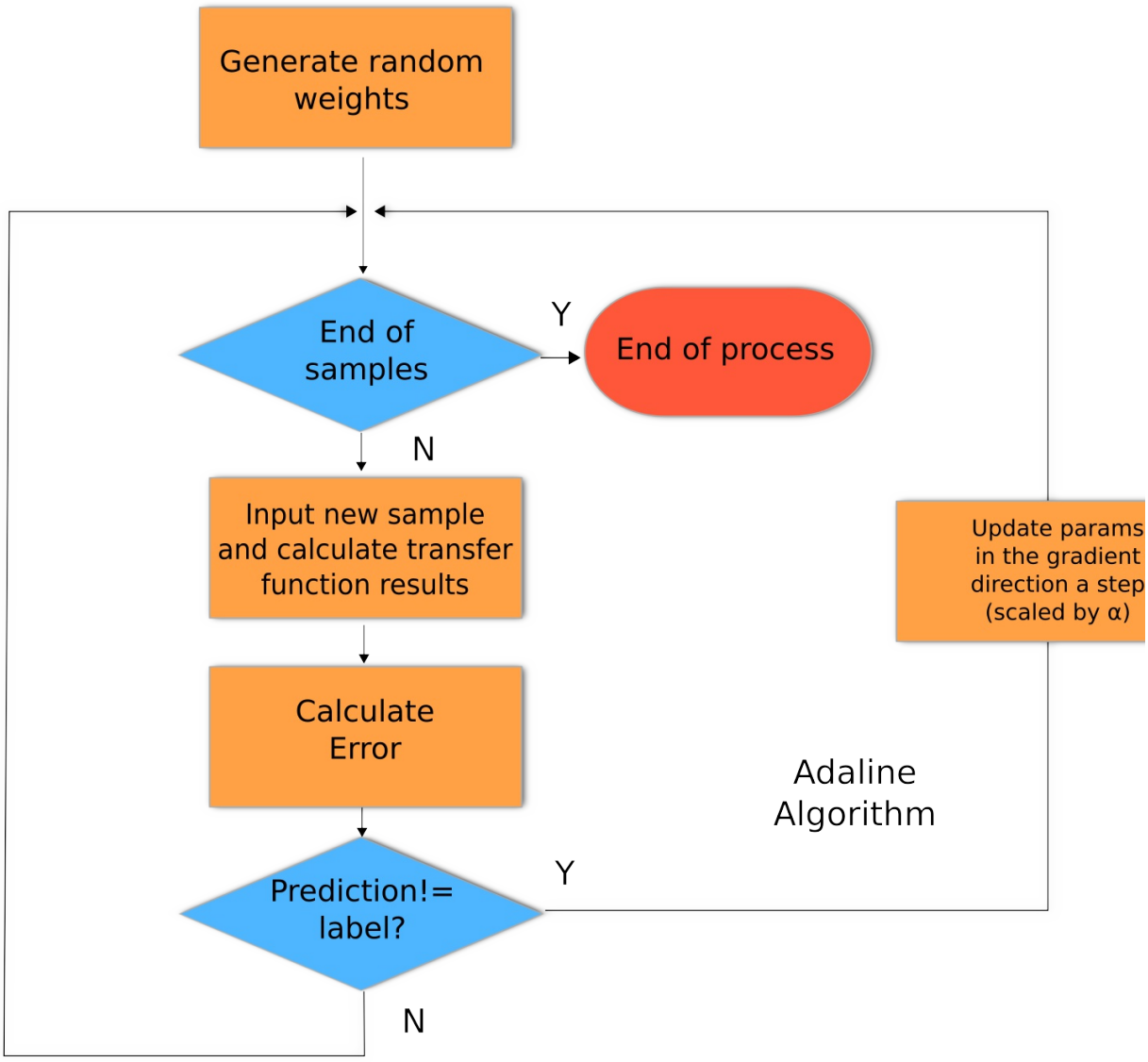
$E=(o-y)^2$

Does it look similar to something we have already seen? Yes! We are basically resolving a regression problem.

6. Adjust the weights with the following gradient descent recursion:

$$w \leftarrow w + \alpha(o - y)x$$

7. Return to *step 2*:



Adaline algorithm flowchart

Similarities and differences between a perceptron and ADALINE

We have covered a simplified explanation of the precursors of modern neural networks. As you can see, the elements of modern models were almost all laid out during the 1950s and the 1960s! Before continuing, let's try to compare the approaches:

- **Similarities**:
  - They are both algorithms (it's important to stress that)
  - They are applied to single-layer neural models
  - They are classifiers for binary classification
  - Both have a linear decision boundary
  - Both can learn iteratively, sample by sample (the perceptron naturally, and ADALINE via stochastic gradient descent)
  - Both use a threshold function

- **Differences**:
  - The perceptron uses the final class decision to train weights
  - ADALINE uses continuous predicted values (from the net input) to learn the model coefficients, and measures subtle changes in error with a continuous float point rank, not a Boolean or integer

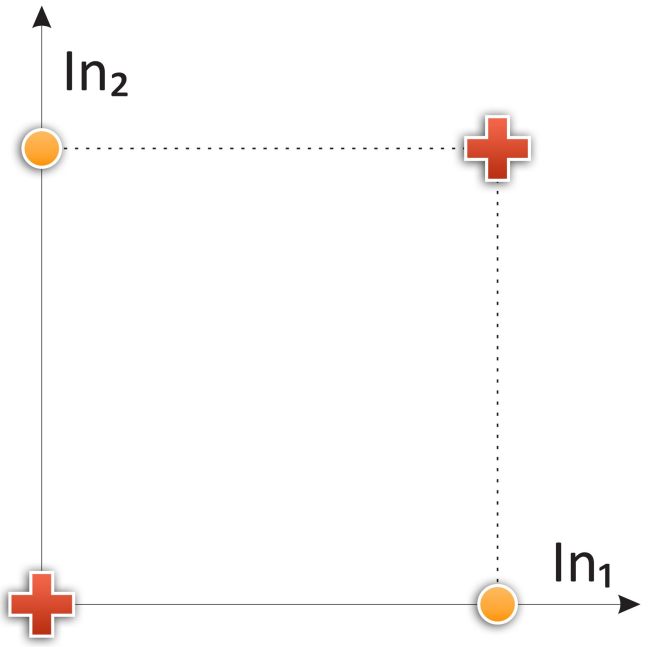
Before we are done with our single-layer architectures and models, we will review some discoveries from the end of the 1960s that provoked quite a stir in the neural models community and were said to generate the first AI winter, or an abrupt collapse in the interest in machine learning research. Happily, some years after that, researchers found ways to overcome the limitations they faced, but this will be covered further on in the chapter.

Limitations of early models

The model itself now has most of the elements of any normal neural model, but it had its own problems. After some years of fast development, the publication of the book *Perceptrons* by Minsky and Papert in 1969 prompted a stir in the field because of its main idea that perceptrons could only work on linearly separable problems, which were only a very tiny part of the problems that the practitioners thought were solvable. In a sense, the book suggested that perceptrons were verging on being useless, with the exception of the simplest classification tasks.

This new-found deficiency could be represented by the inability of the model to represent the XOR function, which is a `Boolean` function with an output of `1` when the inputs are different, and `0` when they are equal:

X1	X2	T
0	0	0
1	0	1
0	1	1
1	1	0



The problem of modelling a XOR function. No single line will correctly separate the 0 and 1 values

As we can see in the preceding diagram, the main problem is that neither class (cross nor point) is linearly separable; that is, we can not separate the two with any linear function on the plane.

This detected problem caused a decrease in activity in the field that lasted for more or less five years, until the development of the backpropagation algorithm, which started in the mid 1970s.

Single and multilayer perceptrons

Now we will discuss more contemporary times, building on the previous concepts to allow more complex elements of reality to be modeled.

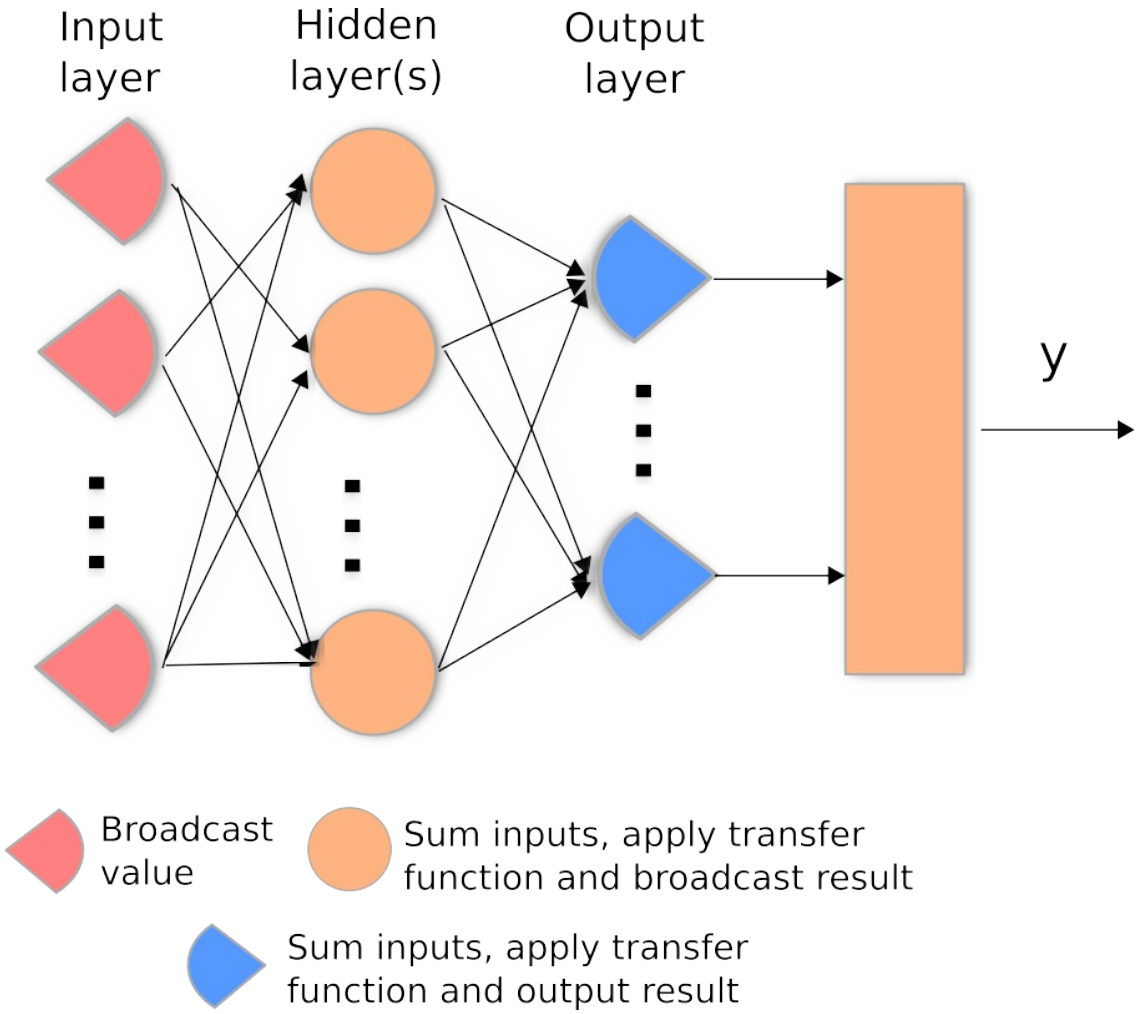
> ### Note
>
> In this section, we will directly study **multilayer perceptrons** (**MLPs**), which are the most common configuration used, and will consider a **single-layer perceptron** as a particular case of the former, highlighting the differences.

Single-layer and multilayer perceptrons were the most commonly used architecture during the 1970s and 1980s and provided huge advances in terms of the capabilities of neural systems. The main innovations they bring to the table are as follows:

- They are feedforward networks because the calculations, starting from the inputs, flow from layer to layer without any cycling (information never returns)
- They use the backpropagation method to adjust their weights
- The use of the `step` function as a transfer function is replaced by non-linear ones such as the sigmoid

MLP origins

After the power of single-unit neural models had been explored, an obvious step was to generate layers or sets of commonly connected units (we define a connection as the act of sending the output of one unit to be part of another unit's summation):



Depiction of a simple multilayer feed forward Neural Network

The feedforward mechanism

In this phase of the operation of the network, the data will be input in the first layer and will flow from each unit to the corresponding units in the following layers. Then it will be summed and passed through in the hidden layers, and finally processed by the output layer. This process is totally unidirectional, so we are avoiding any recursive complications in the data flow.

The feedforward mechanism, has in the MLP its counterpart in the training part of the modeling process, which will be in charge of improving the model's performance. The normally chosen algorithm is called **backpropagation**.

The chosen optimization algorithm – backpropagation

From the perceptron algorithm onward, every neural architecture has had a means to optimize its internal parameters based on the comparison of the ground truth with the model output. The common assumption was to take the derivative of the (then simple) model function and iteratively work towards the minimum value.

For complex multilayer networks, there is an additional overhead, which has to do with the fact that the output layer's output is the result of a long chain of functions compositions, where each

layer's output is wrapped by the next one's transfer function. So, the derivative of the output will involve the derivative of an exceedingly complex function. In this case, the backpropagation method was proposed, with excellent results.

Note

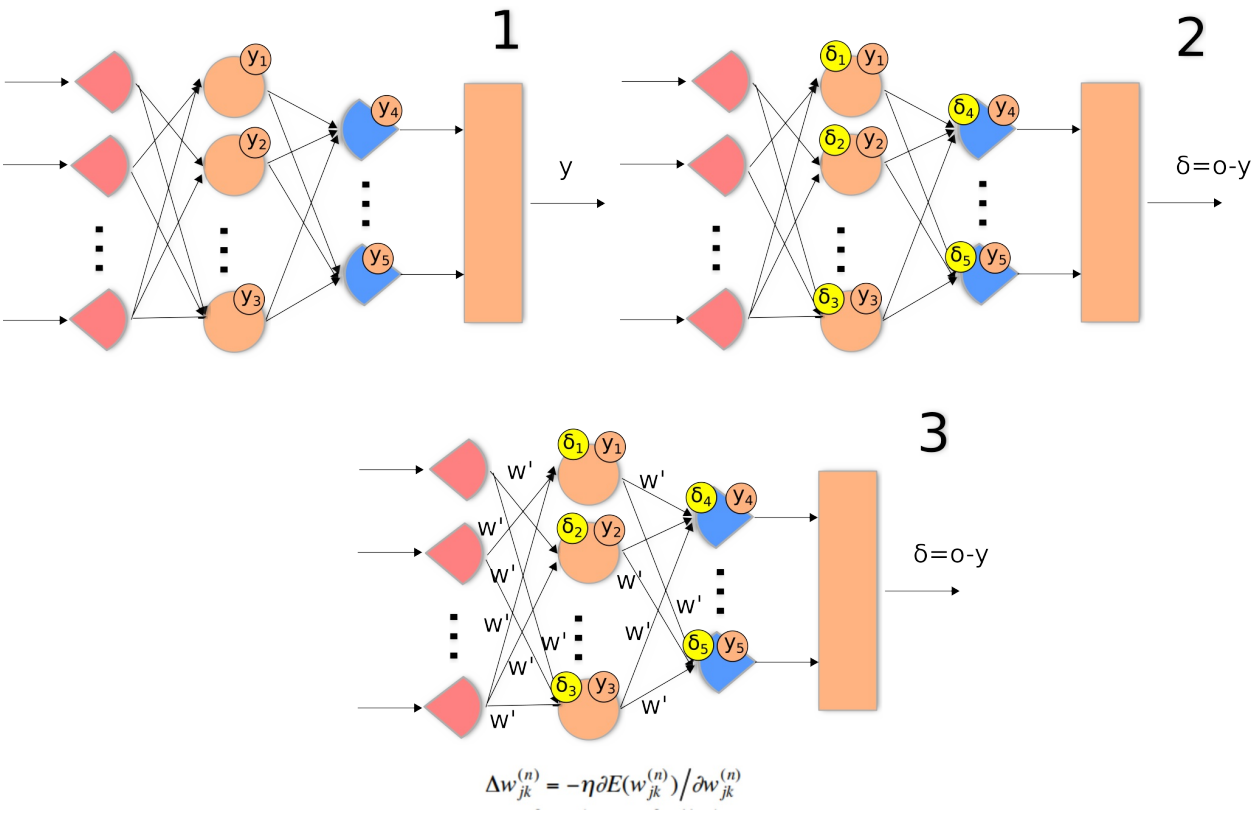
Backpropagation can be summarized as an algorithm used to calculate derivatives. The main attribute is that it is computationally efficient and works with complex functions. It is also a generalization of the least mean squares algorithm in the linear perceptron, which we already knew!

In the backpropagation algorithm, the responsibility of the error will be distributed among all the functions applied to the data in the whole architecture. So, the goal is to minimize the error, the gradient of the loss function, over a set of deeply compounded functions, which will again receive the help of the chain rule.

Now it's time to define our general algorithm for our modern version of a neural network in the following steps:

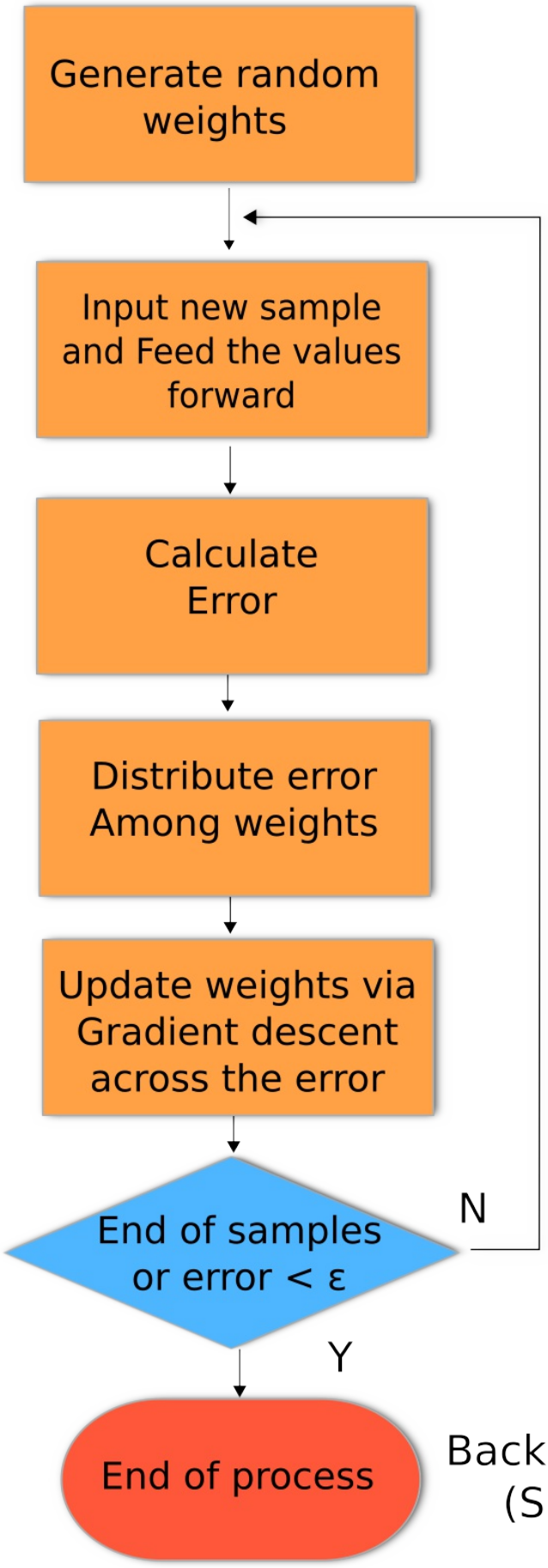
1. Calculate the feedforward signals from the input to the output.
2. Calculate output error E based on the prediction a_k and the target t_k.
3. Backpropagate the error signals by weighting them by the weights in the previous layers and the gradients of the associated activation functions.
4. Calculate the gradients $\delta E/\delta\theta$ for the parameters based on the backpropagated error signal and the feedforward signals from the inputs.
5. Update the parameters using the calculated gradients $\theta \leftarrow \theta - \eta\ \delta E/\delta\theta$.

Lets review this process now, in a graphical way:



Step by step representation of the Feed Forward and Back Propagation training process

In the following diagram, we will represent the whole process in an algorithmic way. You can see the number of coincidences with the previous optimization methods, and the short number of computation blocks involved:



Backpropagation (Simplified)

Flowchart of the feedforward/backpropagation scheme

Types of problem to be tackled

Neural networks can be used for both regression problems and classification problems. The common architectural difference resides in the output layer: in order to be able to bring a real number-based result, no standardization function, such as sigmoid, should be applied. In this manner, we won't be changing the outcome of the variable to one of the many possible class values, getting a continuum of possible outcomes. Let's take a look at the following types of problems to be tackled:

- **Regression/function approximation problems:** This type of problem uses a min squared error function, a linear output activation, and sigmoidal hidden activations. This will give us a real value for output.
- **Classification problems (two classes, one output)**: In this kind of problem we normally have a cross-entropy cost function, a sigmoid output, and hidden activations. The sigmoid function will give us the probability of occurrence or nonoccurrence of one of the classes.
- **Classification problems (multiple-classes, one output per class)**: In this kind of problem, we will have a cross-entropy cost function with softmax outputs and sigmoid hidden activations, in order to have an output of the probabilities for any of the possible classes for a single input.

Implementing a simple function with a single-layer perceptron

Take a look at the following code snippet to implement a single function with a single-layer perceptron:

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
from pprint import pprint
%matplotlib inline
from sklearn import datasets
import matplotlib.pyplot as plt
```

Defining and graphing transfer function types

The learning properties of a neural network would not be very good with just the help of a univariate linear classifier. Even some mildly complex problems in machine learning involve multiple non-linear variables, so many variants were developed as replacements for the transfer functions of the perceptron.

In order to represent non-linear models, a number of different non-linear functions can be used in the activation function. This implies changes in the way the neurons will react to changes in the input variables. In the following sections, we will define the main different transfer functions and define and represent them via code.

In this section, we will start using some **object-oriented programming** (**OOP**) techniques from Python to represent entities of the problem domain. This will allow us to represent concepts in a much clearer way in the examples.

Let's start by creating a `TransferFunction` class, which will contain the following two methods:

- `getTransferFunction(x)`: This method will return an activation function determined by the class type
- `getTransferFunctionDerivative(x)`: This method will clearly return its derivative

For both functions, the input will be a NumPy array and the function will be applied element by element, as follows:

```
>classTransferFunction:defgetTransferFunction(x):raise NotImplementedError
    defgetTransferFunctionDerivative(x):raise NotImplementedError
```

Representing and understanding the transfer functions

Let's take a look at the following code snippet to see how the transfer function works:

```
defgraphTransferFunction(function):
    x = np.arange(-2.0, 2.0, 0.01)
    plt.figure(figsize=(18,8))
    ax=plt.subplot(121)
    ax.set_title(function.__name__)
```

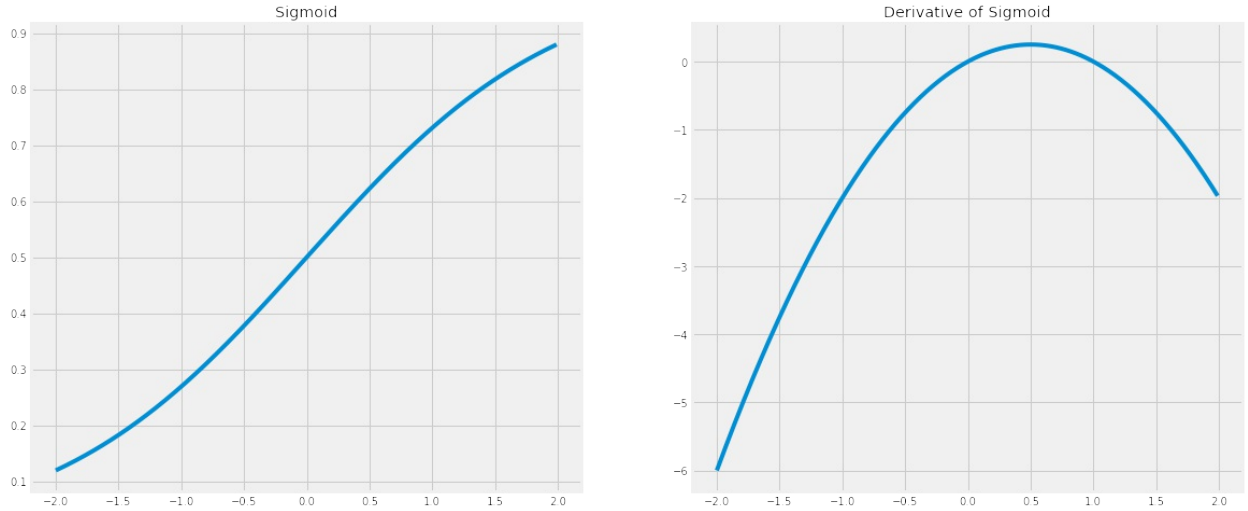
```
    plt.plot(x, function.getTransferFunction(x))

    ax=plt.subplot(122)
    ax.set_title('Derivative of ' + function.__name__)
    plt.plot(x, function.getTransferFunctionDerivative(x))
```

Sigmoid or logistic function

A sigmoid or logistic function is the canonical activation function and is well-suited for calculating probabilities in classification properties. Firstly, let's prepare a function that will be used to graph all the transfer functions with their derivatives, from a common range of `-2.0` to `2.0`, which will allow us to see the main characteristics of them around the *y* axis. The classical formula for the sigmoid function is as follows:

```
classSigmoid(TransferFunction):#Squash 0,1defgetTransferFunction(x):return1/(1+np.exp(-x))
    defgetTransferFunctionDerivative(x):return x*(1-x)

graphTransferFunction(Sigmoid)
```

Take a look at the following graph:

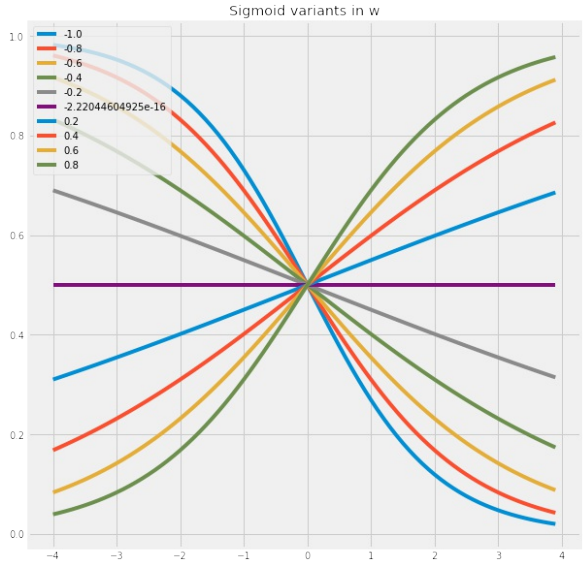


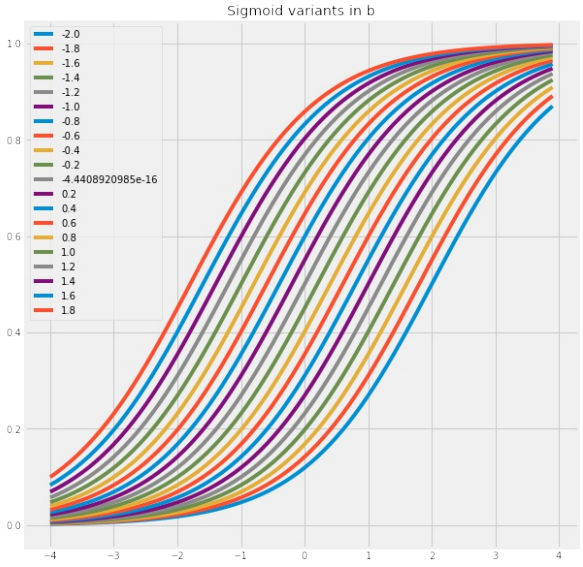
Playing with the sigmoid

Next, we will do an exercise to get an idea of how the sigmoid changes when multiplied by the weights and shifted by the bias to accommodate the final function towards its minimum. Let's then vary the possible parameters of a single sigmoid first and see it stretch and move:

```
ws=np.arange(-1.0, 1.0, 0.2)
  bs=np.arange(-2.0, 2.0, 0.2)
  xs=np.arange(-4.0, 4.0, 0.1)
  plt.figure(figsize=(20,10))
  ax=plt.subplot(121)
  for i in ws:
      plt.plot(xs,  Sigmoid.getTransferFunction(i *xs),label= str(i));
  ax.set_title('Sigmoid variants in w')
  plt.legend(loc='upper left');

  ax=plt.subplot(122)
  for i in bs:
      plt.plot(xs,  Sigmoid.getTransferFunction(i +xs),label= str(i));
  ax.set_title('Sigmoid variants in b')
  plt.legend(loc='upper left');
```

Take a look at the following graph:

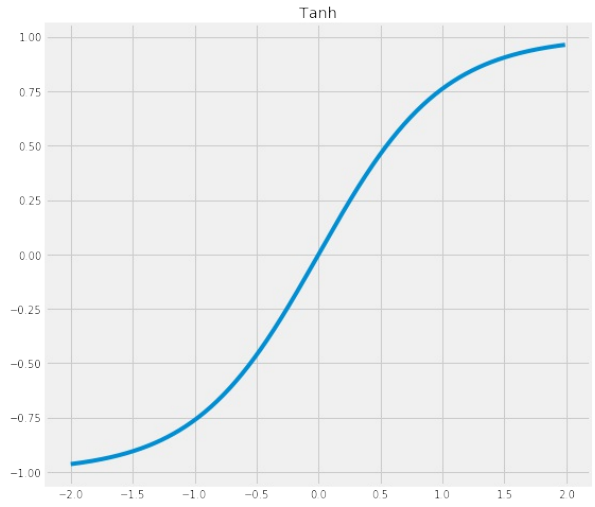


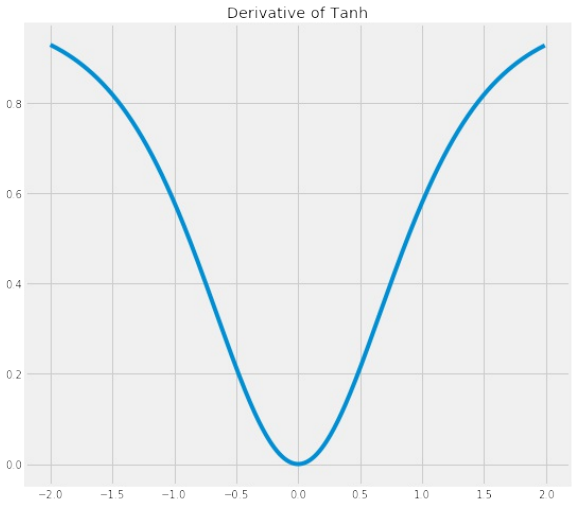


Let's take a look at the following code snippet:

```
classTanh(TransferFunction):#Squash -1,1defgetTransferFunction(x):return np.tanh(x)
    defgetTransferFunctionDerivative(x):return np.power(np.tanh(x),2)
graphTransferFunction(Tanh)
```

Lets take a look at the following graph:



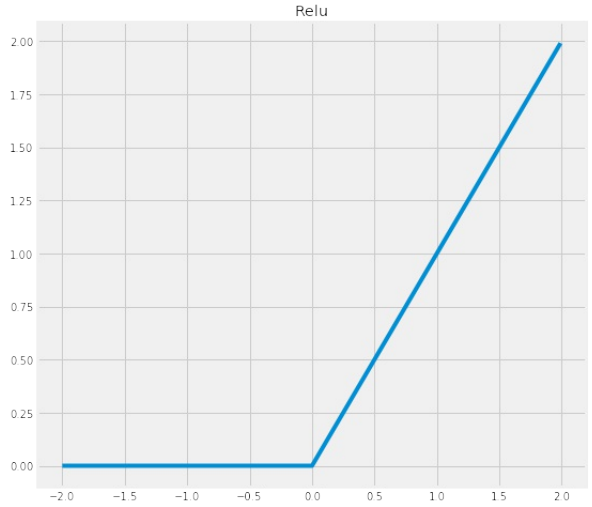


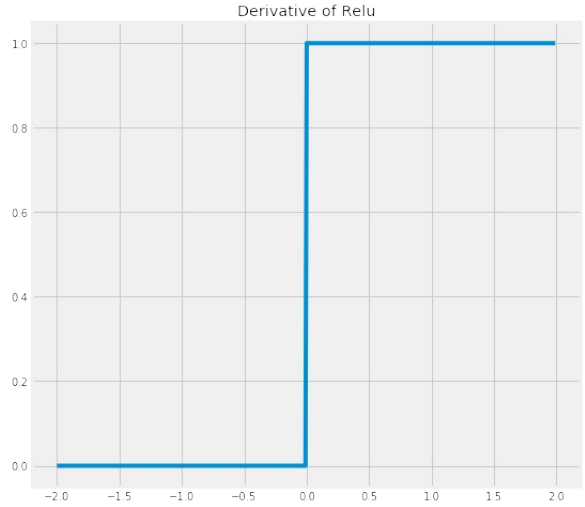
Rectified linear unit or ReLU

ReLU is called a rectified linear unit, and one of its main advantages is that it is not affected by vanishing gradient problems, which generally consist of the first layers of a network tending to be values of zero, or a tiny epsilon:

```
classRelu(TransferFunction):defgetTransferFunction(x):return x * (x>0)
    defgetTransferFunctionDerivative(x):return1 * (x>0)
graphTransferFunction(Relu)
```

Let's take a look at the following graph:



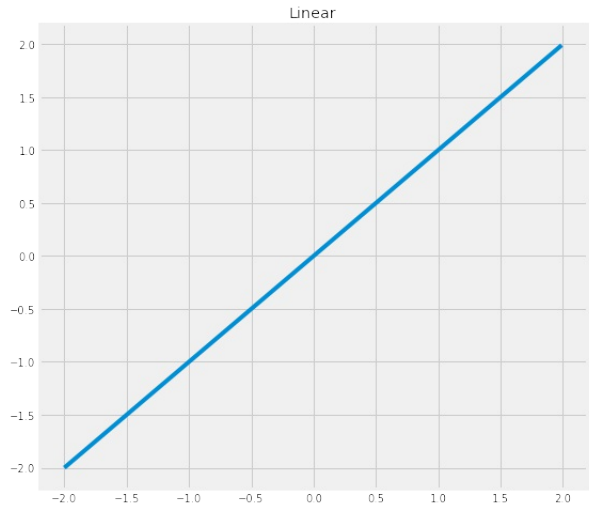


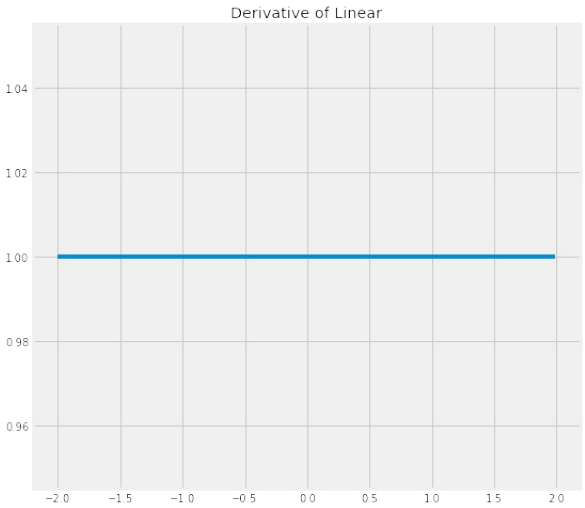
Linear transfer function

Let's take a look at the following code snippet to understand the linear transfer function:

```
classLinear(TransferFunction):defgetTransferFunction(x):return x
    defgetTransferFunctionDerivative(x):return np.ones(len(x))
graphTransferFunction(Linear)
```

Let's take a look at the following graph:





Defining loss functions for neural networks

As with every model in machine learning, we will explore the possible functions that we will use to determine how well our predictions and classification went.

The first type of distinction we will do is between the L1 and L2 error function types.

L1, also known as **least absolute deviations** (**LAD**) or **least absolute errors** (**LAE**), has very interesting properties, and it simply consists of the absolute difference between the final result of the model and the expected one, as follows:

$$S = \sum_{i=1}^{n} |y_i - f(x_i)|.$$

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

L1 versus L2 properties

Now it's time to do a head-to-head comparison between the two types of loss function:

- **Robustness**: L1 is a more robust loss function, which can be expressed as the resistance of the function when being affected by outliers, which projects a quadratic function to very high values. Thus, in order to choose an L2 function, we should have very stringent data cleaning for it to be efficient.
- **Stability**: The stability property assesses how much the error curve jumps for a large error value. L1 is more unstable, especially for non-normalized datasets (because numbers in the `[-1, 1]` range diminish when squared).
- **Solution uniqueness**: As can be inferred by its quadratic nature, the L2 function ensures that we will have a unique answer for our search for a minimum. L2 always has a unique solution, but L1 can have many solutions, due to the fact that we can find many paths with minimal length for our models in the form of piecewise linear functions, compared to the single line distance in the case of L2.

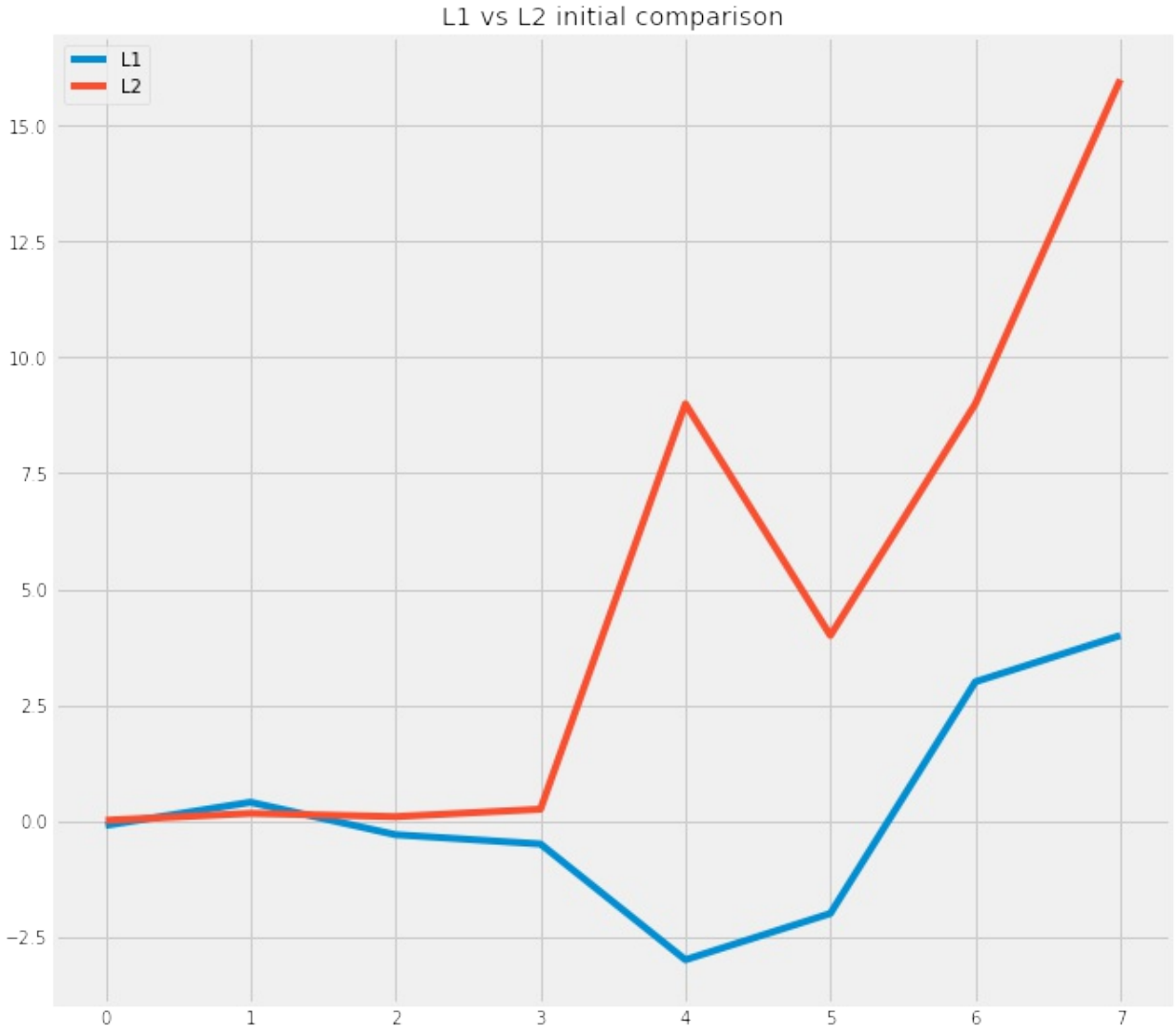
Regarding usage, the summation of the past properties allows us to use the L2 error type in normal cases, especially because of the solution uniqueness, which gives us the required certainty when starting to minimize error values. In the first example, we will start with a simpler L1 error function for educational purposes.

Let's explore these two approaches by graphing the error results for a sample L1 and L2 loss error function. In the next simple example, we will show you the very different nature of the two errors. In the first two examples, we have normalized the input between `-1` and `1` and then with values outside that range.

As you can see, from samples `0` to `3`, the quadratic error increases steadily and continuously, but with non-normalized data it can explode, especially with outliers, as shown in the following code snippet:

```
sampley_=np.array([.1,.2,.3,-.4, -1, -3, 6, 3])
sampley=np.array([.2,-.2,.6,.10, 2, -1, 3, -1])

ax.set_title('Sigmoid variants in b')
plt.figure(figsize=(10,10))
ax=plt.subplot()
plt.plot(sampley_ - sampley, label='L1')
plt.plot(np.power((sampley_ - sampley),2), label="L2")
ax.set_title('L1 vs L2 initial comparison')
plt.legend(loc='best')
plt.show()
```

Let's take a look at the following graph:



Let's define the loss functions in the form of a `LossFunction` class and a `getLoss` method for the L1 and L2 loss function types, receiving two NumPy arrays as parameters, `y_`, or the estimated function value, and `y`, the expected value:

```
classLossFunction:defgetLoss(y_ , y ):raise NotImplementedError

classL1(LossFunction):defgetLoss(y_, y):return np.sum (y_ - y)

classL2(LossFunction):defgetLoss(y_, y):return np.sum (np.power((y_ - y),2))
```

Now it's time to define the goal function, which we will define as a simple `Boolean`. In order to allow faster convergence, it will have a direct relationship between the first input variable and the function's outcome:

```
# input dataset
X = np.array([  [0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1] ])

# output dataset
y = np.array([[0,0,1,1]]).T
```

The first model we will use is a very minimal neural network with three cells and a weight for

each one, without bias, in order to keep the model's complexity to a minimum:

```
# initialize weights randomly with mean 0
W = 2*np.random.random((3,1)) - 1
print (W)
```

Take a look at the following output generated by running the preceding code:

```
[[ 0.52014909]
 [-0.25361738]
 [ 0.165037  ]]
```

Then we will define a set of variables to collect the model's error, the weights, and training results progression:

```
errorlist=np.empty(3);
weighthistory=np.array(0)
resultshistory=np.array(0)
```

Then it's time to do the iterative error minimization. In this case, it will consist of feeding the whole true table 100 times via the weights and the neuron's transfer function, adjusting the weights in the direction of the error.

Note that this model doesn't use a learning rate, so it should converge (or diverge) quickly:

```
for iter in range(100):

    # forward propagation
    l0 = X
    l1 = Sigmoid.getTransferFunction(np.dot(l0,W))
    resultshistory = np.append(resultshistory , l1)

    # Error calculation
    l1_error = y - l1
    errorlist=np.append(errorlist, l1_error)

    # Back propagation 1: Get the deltas
    l1_delta = l1_error * Sigmoid.getTransferFunctionDerivative(l1)

    # update weights
    W += np.dot(l0.T,l1_delta)
    weighthistory=np.append(weighthistory,W)
```

Let's simply review the last evaluation step by printing the output values at `l1`. Now we can see that we are reflecting quite literally the output of the original function:

```
print (l1)
```

Take a look at the following output, which is generated by running the preceding code:

```
[[ 0.11510625]
 [ 0.08929355]
 [ 0.92890033]
 [ 0.90781468]]
```

To better understand the process, let's have a look at how the parameters change over time. First, let's graph the neuron weights. As you can see, they go from a random state to accepting the whole values of the first column (which is always right), going to almost `0` for the second column (which is right 50% of the time), and then going to `-2` for the third (mainly because it has to trigger `0` in the first two elements of the table):

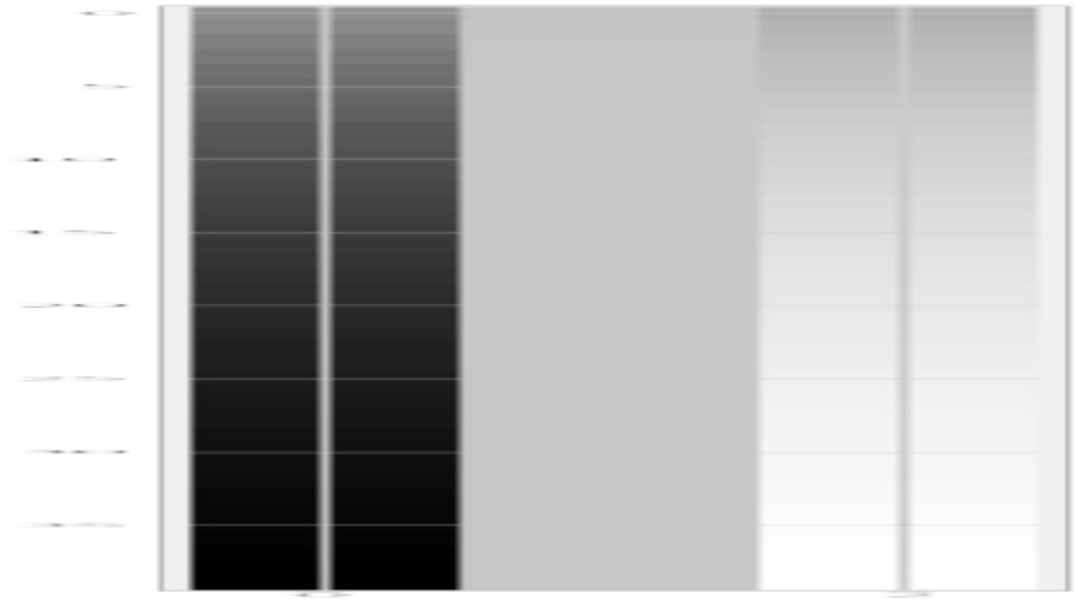
```
plt.figure(figsize=(20,20))
print (W)
plt.imshow(np.reshape(weighthistory[1:],(-1,3))[:40], cmap=plt.cm.gray_r,
```

```
    interpolation='nearest');
```

Take a look at the following output, which is generated by running the preceding code:

```
[[ 4.62194116]
 [-0.28222595]
 [-2.04618725]]
```

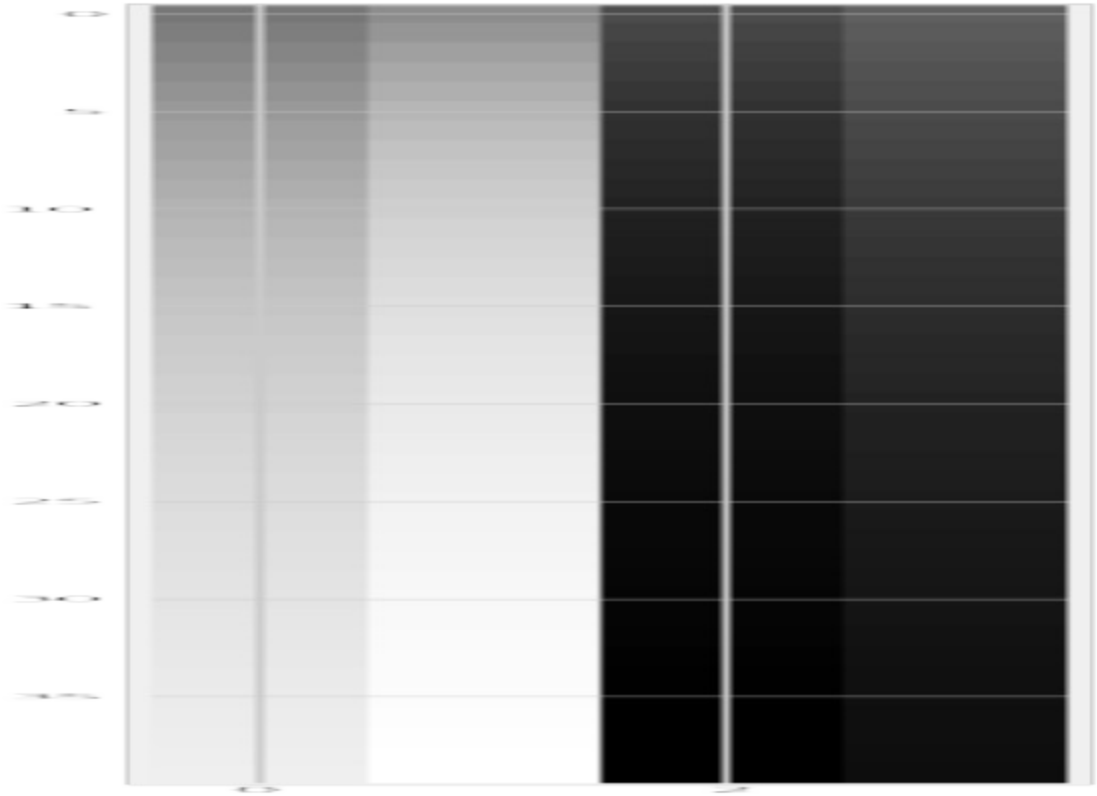
Let's take a look at the following screenshot:



Let's also review how our solutions evolved (during the first 40 iterations) until we reached the last iteration; we can clearly see the convergence to the ideal values:

```
    plt.figure(figsize=(20,20))
    plt.imshow(np.reshape(resultshistory[1:], (-1,4))[:40], cmap=plt.cm.gray_r,
    interpolation='nearest');
```

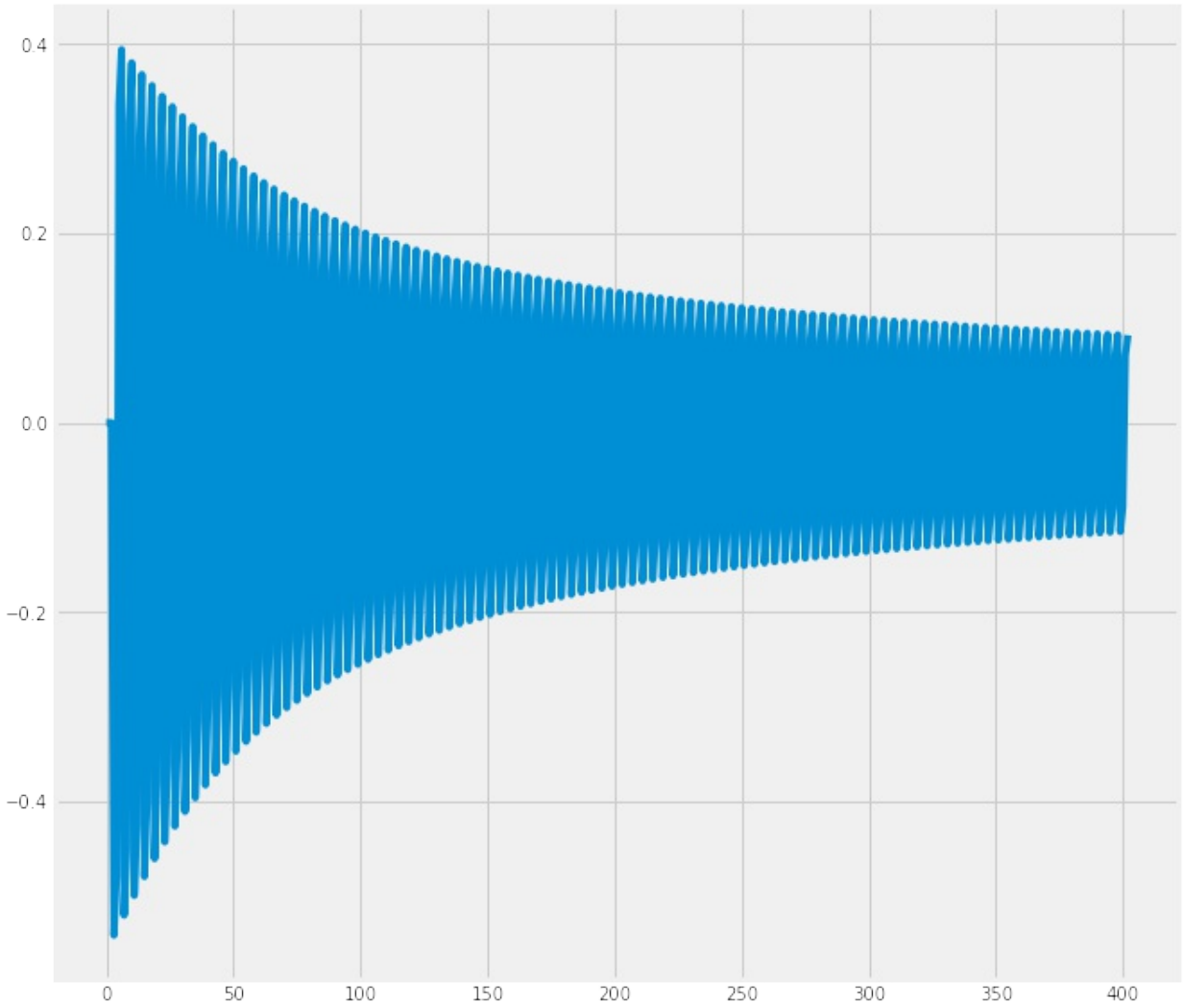
Let's take a look at the following screenshot:



We can see how the error evolves and tends to be zero through the different epochs. In this case, we can observe that it swings from negative to positive, which is possible because we first used an L1 error function:

```
plt.figure(figsize=(10,10))
plt.plot(errorlist);
```

Let's take a look at the following screenshot:



Depiction of the decreasing training error of our simple Neural Network

Summary

In this chapter, we took a very important step towards solving complex problems together by means of implementing our first neural network. Now, the following architectures will have familiar elements, and we will be able to extrapolate the knowledge aquired on this chapter, to novel architectures.

In the next chapter, we will explore more complex models and problems, using more layers and special configurations, such as convolutional and dropout layers.

References

Refer to the following content:

- McCulloch, Warren S., and Walter Pitts,. *A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics 5.4 (1943): 115-133.* Kleene, Stephen Cole. Representation of events in nerve nets and finite automata. No. RAND-RM-704. RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.
- Farley, B. W. A. C., and W. Clark,*Simulation of self-organizing systems by digital computer.* Transactions of the IRE Professional Group on Information Theory 4.4 (1954): 76-84.
- Rosenblatt, Frank, *The perceptron: A probabilistic model for information storage and organization in the brain*, Psychological review 65.6 (1958): 386.Rosenblatt, Frank. x.
- Principles of Neurodynamics: perceptrons and the Theory of Brain Mechanisms. Spartan Books, Washington DC, 1961
- Werbos, P.J. (1975), *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.*
- Preparata, Franco P., and Michael Ian Shamos,. *"Introduction." Computational Geometry. Springer New York, 1985. 1-35.*
- Rumelhart, David E., Geoffrey E. Hinton, and Ronald J, *Williams. Learning internal representations by error propagation. No. ICS-8506. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.*
- Rumelhart, James L. McClelland, and the PDP research group. *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundation. MIT Press, 1986.*
- Cybenko, G. 1989. *Approximation by superpositions of a sigmoidal function Mathematics of Control, Signals, and Systems, 2(4), 303–314.*
- Murtagh, Fionn. *Multilayer perceptrons for classification and regression. Neurocomputing 2.5 (1991): 183-197.*
- Schmidhuber, Jürgen. *Deep learning in neural networks: An overview. Neural networks 61 (2015): 85-117.*

Chapter 6. Convolutional Neural Networks

Well, now things are getting fun! Our models can now learn more complex functions, and we are now ready for a wonderful tour around the more contemporary and surprisingly effective models

After piling layers of neurons became the most popular solution to improving models, new ideas for richer nodes appeared, starting with models based on human vision. They started as little more than research themes, and after the image datasets and more processing power became available, they allowed researchers to reach almost human accuracy in classification challenges, and we are now ready to leverage that power in our projects.

The topics we will cover in this chapter are as follows:

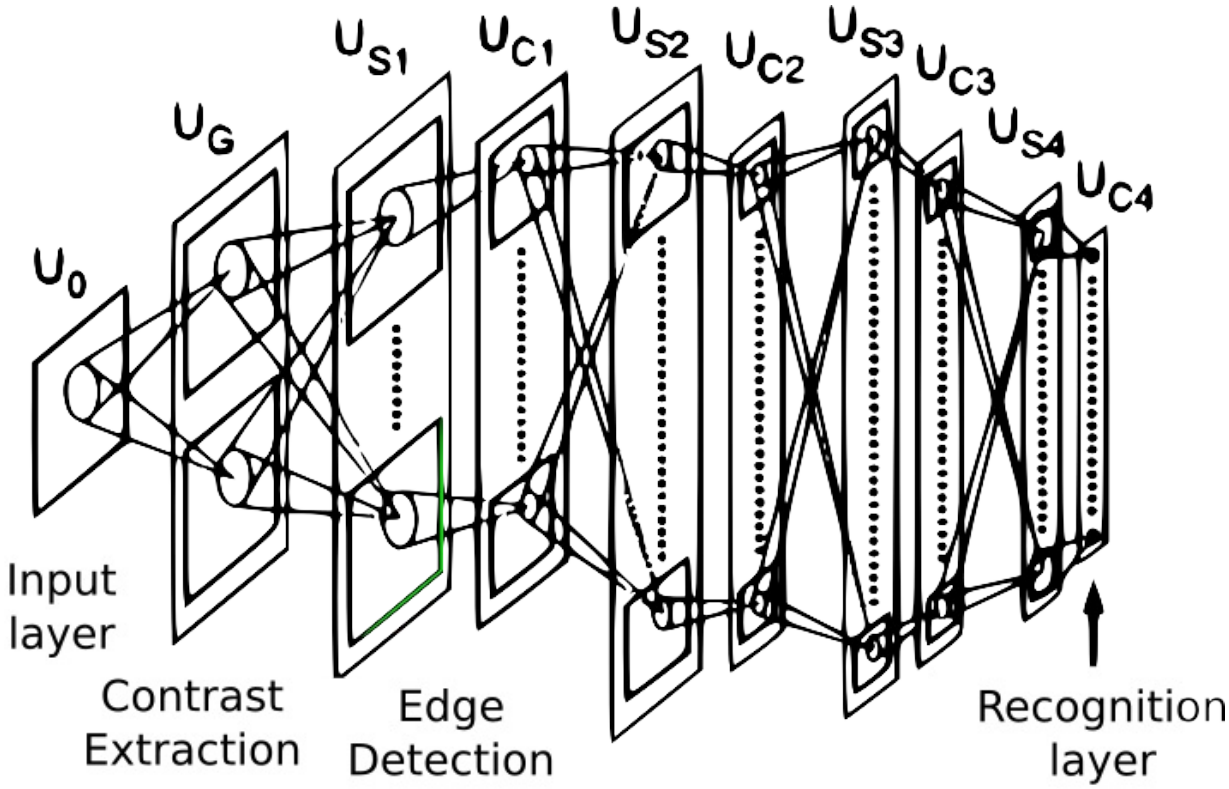
- Origins of convolutional neural networks
- Simple implementation of discrete convolution
- Other operation types: pooling, dropout
- Transfer learning

Origin of convolutional neural networks

Convolutional neural networks (**CNNs**) have a remote origin. They developed while **multi-layer perceptrons** were perfected, and the first concrete example is the **neocognitron**.

The neocognitron is a hierarchical, multilayered **Artificial Neural Network** (**ANN**), and was introduced in a 1980 paper by Prof. Fukushima and has the following principal features:

- Self-organizing
- Tolerant to shifts and deformation in the input



This original idea appeared again in 1986 in the book version of the original backpropagation paper, and was also employed in 1988 for temporal signals in speech recognition.

The design was improved in 1998, with a paper from Ian LeCun,*Gradient-Based Learning Aapplied to Document Recognition*, presenting the LeNet-5 network, an architecture used to classify handwritten digits. The model showed increased performance compared to other existing models, especially over several variations of SVM, one of the most performant operations in the year of publication.

Then a generalization of that paper came in 2003, with "*Hierarchical Neural Networks for Image Interpretation*". But in general, almost all kernels followed the original idea, until now.

Getting started with convolution

In order to understand convolution, we will start by studying the origin of the convolution operator, and then we will explain how this concept is applied to the information.

Convolution is basically an operation between two functions, continuous or discrete, and in practice, it has the effect of filtering one of them by another.

It has many uses across diverse fields, especially in digital signal processing, where it is the preferred tool for shaping and filtering audio, and images, and it is even used in probabilistic theory, where it represents the sum of two independent random variables.

And what do these filtering capabilities have to do with machine learning? The answer is that with filters, we will be able to build network nodes that can emphasize or hide certain characteristics of our inputs (by the definition of the filters) so we can build automatic custom detectors for all the features, which can be used to detect a determinate pattern. We will cover more of this in the following sections; now, let's review the formal definition of the operation, and a summary of how it is calculated.

Continuous convolution

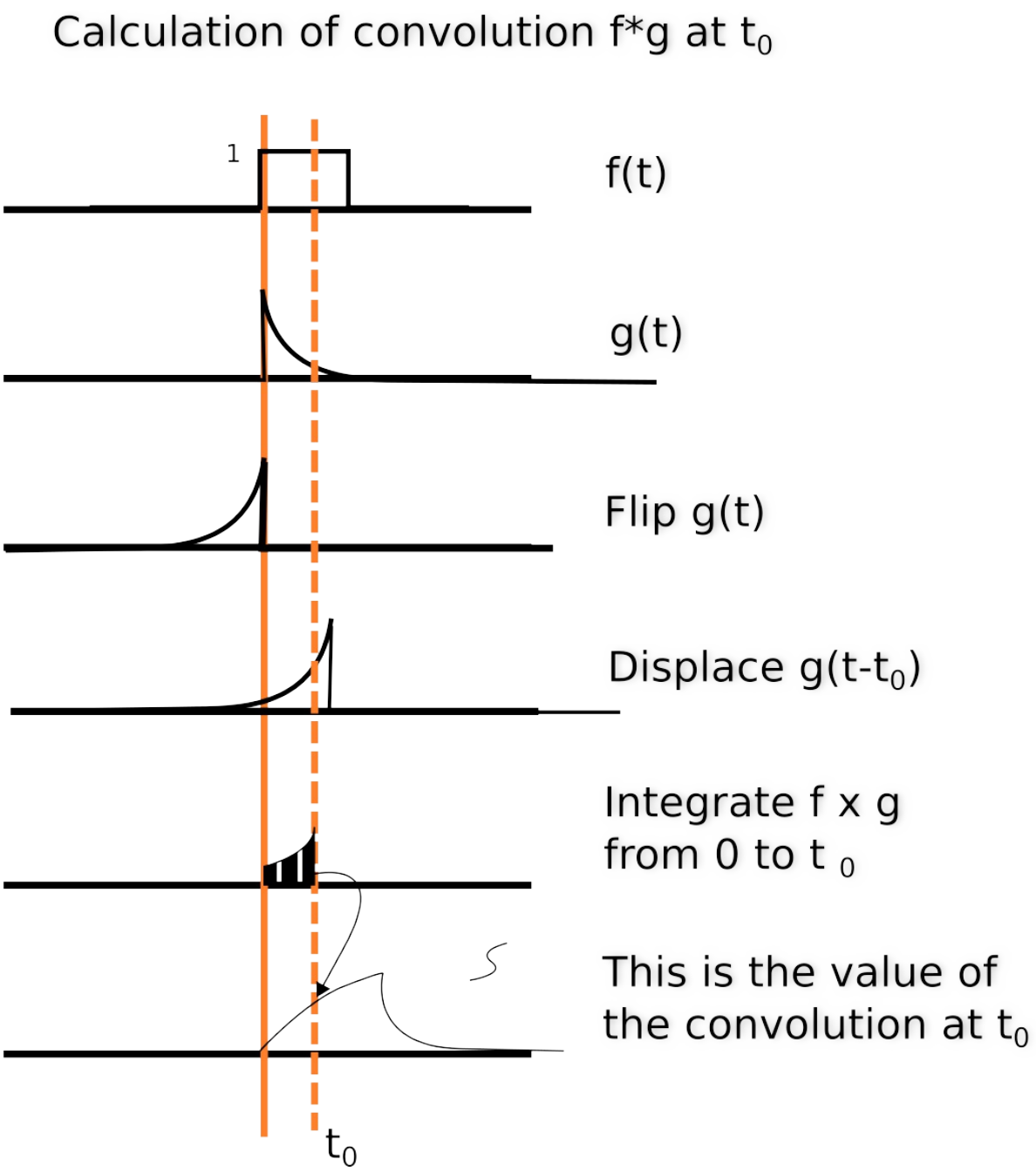
Convolution as an operation was first created during the 18 century by d'Alembert during the initial developments of differential calculus. The common definition of the operation is as follows:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau)\, d\tau$$

If we try to describe the steps needed to apply the operation, and how it combines two functions, we can express the mathematical operation involved in the following detail:

- Flip the signal: This is the (-τ) part of the variable
- Shift it: This is given by the t summing factor for $g(\tau)$
- Multiply it: The product of f and g
- Integrate the resulting curve: This is the less intuitive part, because each instantaneous value is the result of an integral

In order to understand all the steps involved, let's visually represent all the steps involved for the calculation of the convolution between two functions, f and g, at a determinate point t_0:



This intuitive approximation to the rules of the convolution also applies to the discrete field of functions, and it is the real realm in which we will be working. So, let's start by defining it.

Discrete convolution

Even when useful, the realm in which we work is eminently digital, so we need to translate this operation to the discrete domain. The convolution operation for two discrete functions *f* and *g* is the translation of the original integral to an equivalent summation in the following form:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

This original definition can be applied to functions of an arbitrary number of dimensions. In particular, we will work on 2D images, because this is the realm of a large number of applications, which we will describe further in this chapter.

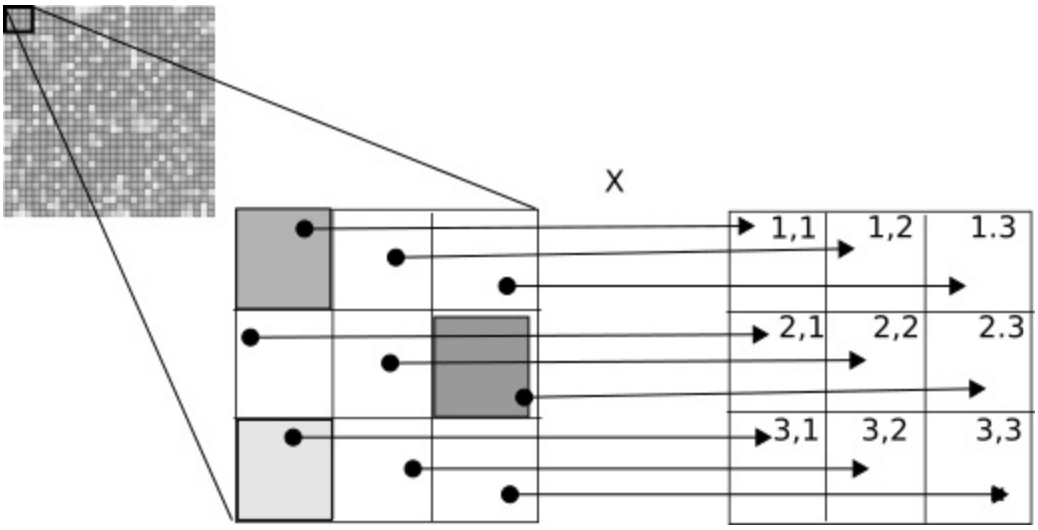
Now it's time to learn the way in which we normally apply the convolution operator, which is through kernels.

Kernels and convolutions

When solving practical problems in the discrete domain, we normally have 2D functions of finite dimensions (which could be an image, for example) that we want to filter through another image. The discipline of filter development studies the effects of different kinds of filters when applied via convolution to a variety of classes. The most common types of function applied are of two to five elements per dimension, and of 0 value on the remaining elements. These little matrices, representing filtering functions, are called **kernels**.

The convolution operation starts with the first element of an *n*-dimensional matrix (usually a 2D matrix representing an image) with all the elements of a kernel, applying the center element of the matrix to the specific value we are multiplying, and applying the remaining factors following the kernel's dimensions. The final result is, in the case of an image, an equivalent image in which certain elements of it (for example, lines and edges) have been highlighted, and others (for example, in the case of blurring) have been hidden.

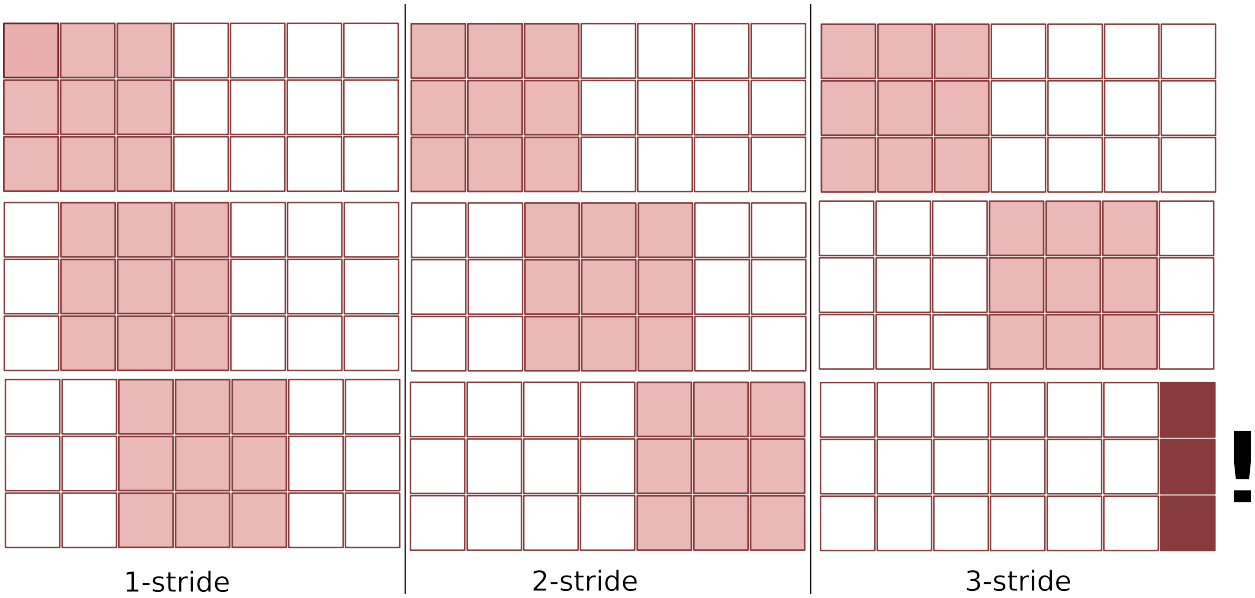
In the following example, you will see how a particular 3 x 3 kernel is applied to a particular image element. This is repeated in a scan pattern to all elements of the matrix:



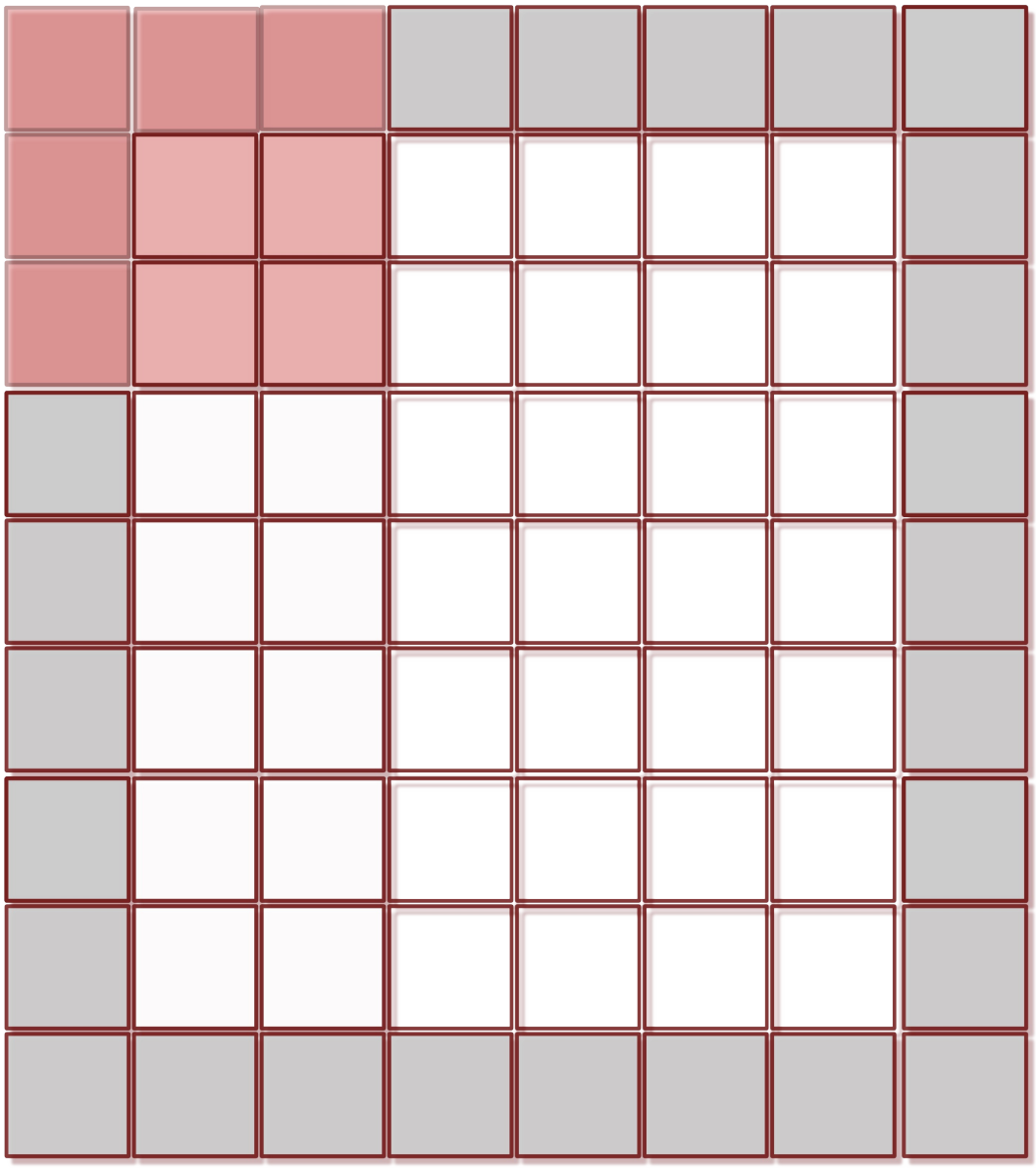
Kernels also have a couple of extra elements to consider when applying them, specifically stride and padding, which complete the specification for accommodating special application cases. Let's have a look at stride and padding.

Stride and padding

When applying the convolution operation, one of the variations that can be applied to the process is to change the displacement units for the kernels. This parameter, which can be specified per dimension, is called **stride**. In the following image, we show a couple of examples of how stride is applied. In the third case, we see an incompatible stride because the kernel can't be applied to the last step. Depending on the library, this type of warning can be dismissed:



The other important fact when applying a kernel is that the bigger the kernel, the more units there are on the border of the image/matrix that won't receive an answer because we need to cover the entire kernel. In order to cope with that, the **padding** parameters will add a border of the specified width to the image to allow the kernel to be able to apply evenly to the edge pixels/elements. Here, you have a graphical depiction of the padding parameter:



After describing the fundamental concepts of convolution, let's implement convolution in a practical example to see it applied to real image and get an intuitive idea of its effects.

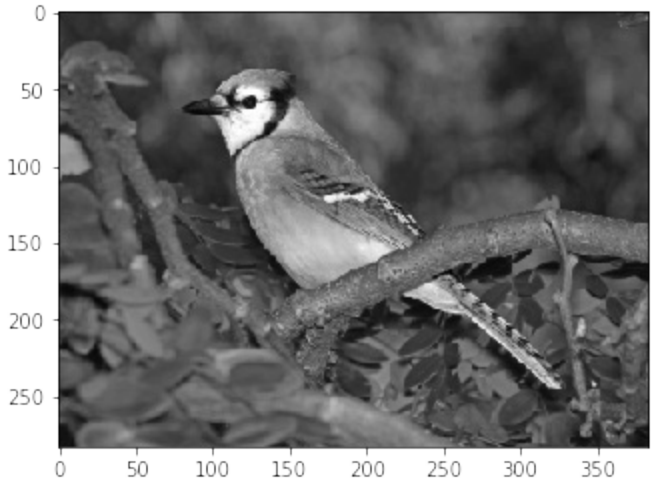
Implementing the 2D discrete convolution operation in an example

In order to understand the mechanism of the discrete convolution operation, let's do a simple intuitive implementation of this concept and apply it to a sample image with different types of kernel. Let's import the required libraries. As we will implement the algorithms in the clearest possible way, we will just use the minimum necessary ones, such as NumPy:

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

Using the `imread` method of the `imageio` package, let's read the image (imported as three equal channels, as it is grayscale). We then slice the first channel, convert it to a floating point, and show it using matplotlib:

```
arr = imageio.imread("b.bmp") [:,:,0].astype(np.float)
plt.imshow(arr, cmap=plt.get_cmap('binary_r'))
plt.show()
```



Now it's time to define the kernel convolution operation. As we did previously, we will simplify the operation on a 3 x 3 kernel in order to better understand the border conditions. `apply3x3kernel` will apply the kernel over all the elements of the image, returning a new equivalent image. Note that we are restricting the kernels to 3 x 3 for simplicity, and so the 1 pixel border of the image won't have a new value because we are not taking padding into consideration:

```
class ConvolutionalOperation:
    def apply3x3kernel(self, image, kernel):  # Simple 3x3 kernel operation
        newimage=np.array(image)
        for m inrange(1,image.shape[0]-2):
            for n inrange(1,image.shape[1]-2):
                newelement =0for i inrange(0, 3):
                    for j inrange(0, 3):
                        newelement = newelement + image[m -1+ i][n -1+
                        j]*kernel[i][j]
                newimage[m][n] = newelement
        return (newimage)
```

As we saw in the previous sections, the different kernel configurations highlight different elements and properties of the original image, building filters that in conjunction can specialize in very high-level features after many epochs of training, such as eyes, ears, and doors. Here, we will generate a dictionary of kernels with a name as the key, and the coefficients of the kernel arranged in a 3 x 3 array. The `Blur` filter is equivalent to calculating the average of the 3 x 3 point neighborhood, `Identity` simply returns the pixel value as is, `Laplacian` is a classic derivative filter that highlights borders, and then the two `Sobel` filters will mark horizontal edges in the first case, and vertical ones in the second case:

```
kernels = {"Blur":[[1./16., 1./8., 1./16.], [1./8., 1./4., 1./8.], [1./16., 1./8., 1./16.]]
           ,"Identity":[[0, 0, 0], [0., 1., 0.], [0., 0., 0.]]
           ,"Laplacian":[[1., 2., 1.], [0., 0., 0.], [-1., -2., -1.]]
           ,"Left Sobel":[[1., 0., -1.], [2., 0., -2.], [1., 0., -1.]]
```

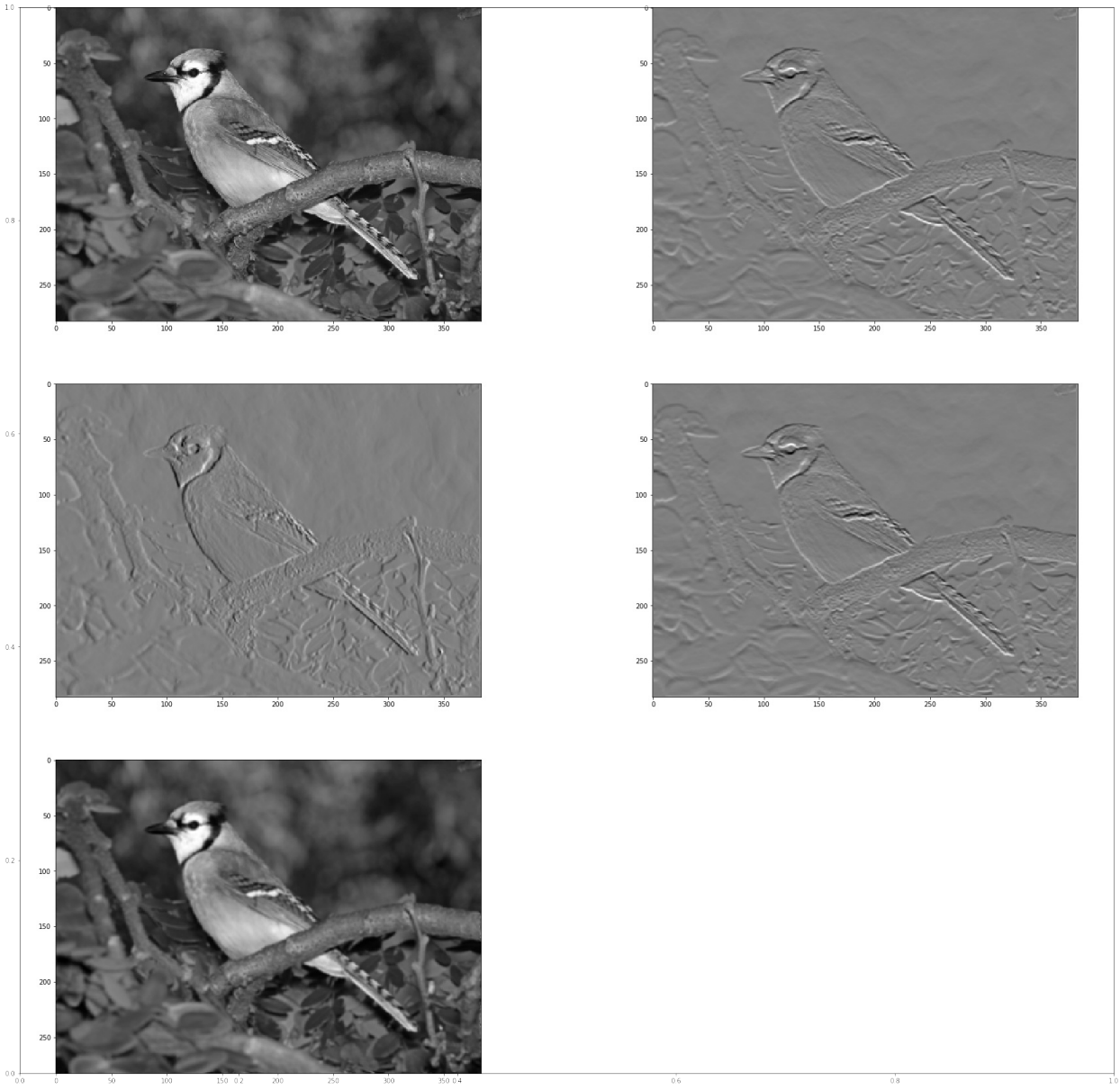
```
          ,"Upper Sobel":[[1., 2., 1.], [0., 0., 0.], [-1., -2., -1.]]}
```

Let's generate a `ConvolutionalOperation` object and generate a comparative kernel graphical chart to see how they compare:

```
conv = ConvolutionalOperation()
plt.figure(figsize=(30,30))
fig, axs = plt.subplots(figsize=(30,30))
j=1for key,value in kernels.items():
    axs = fig.add_subplot(3,2,j)
    out = conv.apply3x3kernel(arr, value)
    plt.imshow(out, cmap=plt.get_cmap('binary_r'))
    j=j+1
plt.show()
```

```
<matplotlib.figure.Figure at 0x7fd6a710a208>
```

In the final image you can clearly see how our kernel has detected several high-detail features on the image—in the first one, you see the unchanged image because we used the unit kernel, then the Laplacian edge finder, the left border detector, the upper border detector, and then the blur operator:



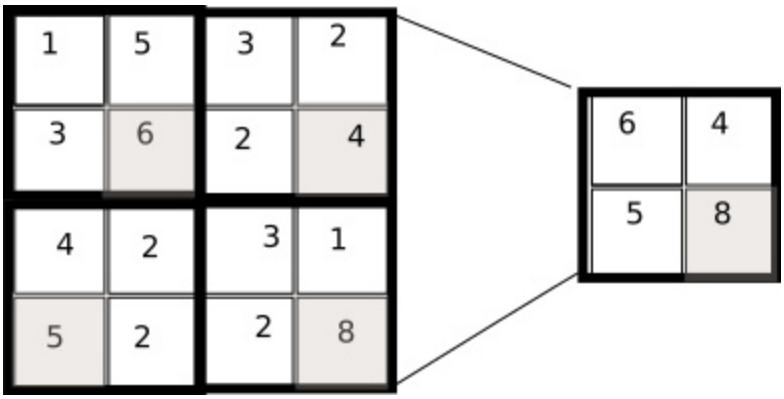
Having reviewed the main characteristics of the convolution operation for the continuous and discrete fields, we can conclude by saying that, basically, convolution kernels highlight or hide patterns. Depending on the trained or (in our example) manually set parameters, we can begin to discover many elements in the image, such as orientation and edges in different dimensions. We may also cover some unwanted details or outliers by blurring kernels, for example. Additionally, by piling layers of convolutions, we can even highlight higher-order composite elements, such as eyes or ears.

This characteristic of convolutional neural networks is their main advantage over previous data-processing techniques: we can determine with great flexibility the primary components of a certain dataset, and represent further samples as a combination of these basic building blocks.

Now it's time to look at another type of layer that is commonly used in combination with the former—the pooling layer.

Subsampling operation (pooling)

The subsampling operation consists of applying a kernel (of varying dimensions) and reducing the extension of the input dimensions by dividing the image into *mxn* blocks and taking one element representing that block, thus reducing the image resolution by some determinate factor. In the case of a 2 x 2 kernel, the image size will be reduced by half. The most well-known operations are maximum (max pool), average (avg pool), and minimum (min pool). The following image gives you an idea of how to apply a 2 x 2 `maxpool` kernel, applied to a one-channel 16 x 16 matrix. It just maintains the maximum value of the internal zone it covers:



Now that we have seen this simple mechanism, let's ask ourselves, what's the main purpose of it? The main purpose of subsampling layers is related to the convolutional layers: to reduce the quantity and complexity of information while retaining the most important information elements. In other word, they build a ***compact representation*** of the underlying information.

Now it's time to write a simple pooling operator. It's much easier and more direct to write than a convolutional operator, and in this case we will only be implementing max pooling, which chooses the brightest pixel in the 4 x 4 vicinity and projects it to the final image:

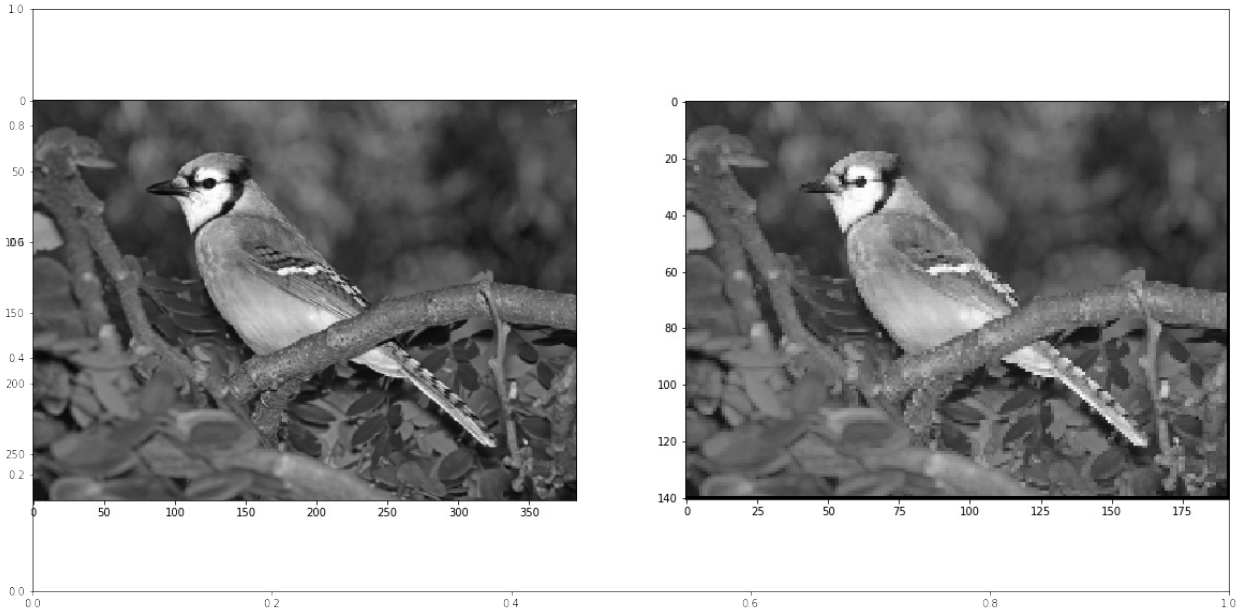
```
class PoolingOperation:
    def apply2x2pooling(self, image, stride):  # Simple 2x2 kernel operation
        newimage=np.zeros((int(image.shape[0]/2),int(image.shape[1]/2)),np.float32)
        for m inrange(1,image.shape[0]-2,2):
            for n inrange(1,image.shape[1]-2,2):
```

```
                newimage[int(m/2),int(n/2)] = np.max(image[m:m+2,n:n+2])
        return (newimage)
```

Let's apply the newly created pooling operation, and as you can see, the final image resolution is much more blocky, and the details, in general, are brighter:

```
plt.figure(figsize=(30,30))
pool=PoolingOperation()
fig, axs = plt.subplots(figsize=(20,10))
axs = fig.add_subplot(1,2,1)
plt.imshow(arr, cmap=plt.get_cmap('binary_r'))
out=pool.apply2x2pooling(arr,1)
axs = fig.add_subplot(1,2,2)
plt.imshow(out, cmap=plt.get_cmap('binary_r'))
plt.show()
```

Here you can see the differences, even though they are subtle. The final image is of lower precision, and the chosen pixels, being the maximum of the environment, produce a brighter image:



Improving efficiency with the dropout operation

As we have observed in the previous chapters, overfitting is a potential problem for every model. This is also the case for neural networks, where data can do very well on the training set but not on the test set, which renders it useless for generalization.

For this reason, in 2012, a team led by Geoffrey Hinton published a paper in which the dropout operation was described. Its operation is simple:

- A random number of nodes is chosen (the ratio of the chosen node from the total is a parameter)
- The values of the chosen weights are reviews to zero, invalidating their previously connected peers at the subsequent layers

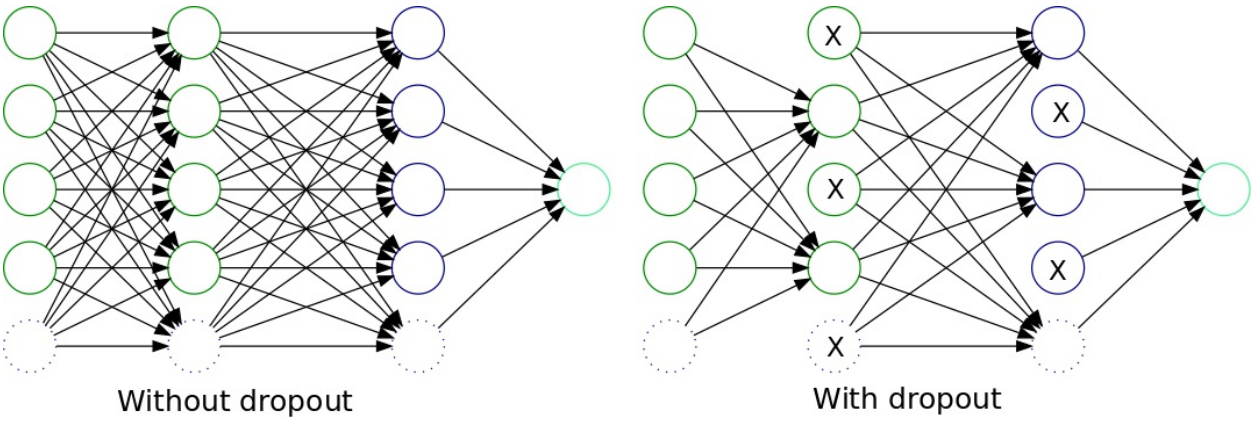
Advantages of the dropout layers

The main advantage of this method is that it prevents all neurons in a layer from synchronously optimizing their weights. This adaptation, made in random groups, prevents all the neurons from converging to the same goal, thus decorrelating the weights.

A second property discovered for the application of dropout is that the activations of the hidden units become sparse, which is also a desirable characteristic.

In the following diagram, we have a representation of an original, fully-connected multi-layer neural network, and the associated network with dropout:

>



Deep neural networks

Now that we have a rich number of layers, it's time to start a tour of how the neural architectures have evolved over time. Starting in 2012, a rapid succession of new and increasingly powerful combinations of layers began, and it has been unstoppable. This new set of architectures adopted the term **deep learning**, and we can approximately define them as complex neural architectures that involve at least three layers. They also tend to include more advanced layers than the **Single Layer Perceptrons**, like convolutional ones.

Deep convolutional network architectures through time

Deep learning architectures date from 20 years ago and have evolved, guided for the most part by the challenge of solving the human vision problem. Let's have a look at the main deep learning architectures and their principal building blocks, which we can then reuse for our own purposes.

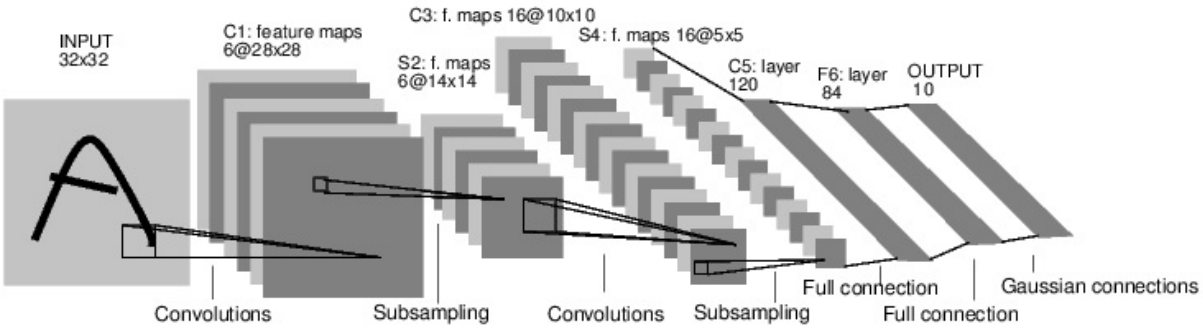
Lenet 5

As we saw in the historical introduction of the convolutional neural networks, convolutional layers were discovered during the 1980s. But the available technology wasn't powerful enough to build complex combinations of layers until the end of the 1990s.

Around 1998, in Bell Labs, during research around the decodification of handwritten digits, Ian LeCun formed a new approach—a mix of convolutional, pooling, and fully connected layers—to solve the problem of recognizing handwritten digits.

At this time, SVM and other much more mathematically defined problems were used more or less successfully, and the fundamental paper on CNNs shows that neural networks could perform comparatively well with the then state-of-the-art methods.

In the following diagram, there is a representation of all the layers of this architecture, which received a grayscale 28 x 28 image as input and returned a 10-element vector, with the probability for each character as the output:



Alexnet

After some more years of hiatus (even though Lecun was applying his networks to other tasks,

such as face and object recognition), the exponential growth of both available structured data and raw processing power allowed the teams to grow and tune the models to an extent that could have been considered impossible just a few years before.

One of the elements that fostered innovation in the field was the availability of an image recognition benchmark called **Imagenet**, consisting of millions of images of objects organized into categories.

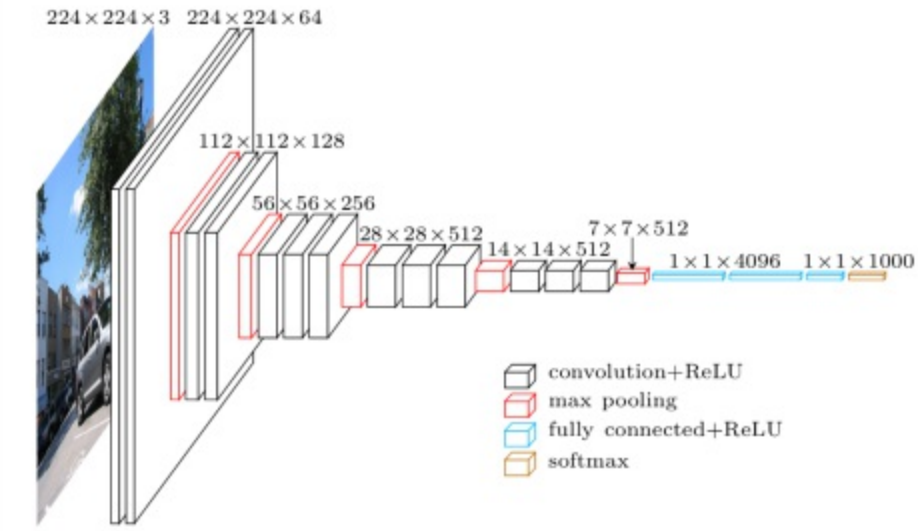
Starting in 2012, the **Large Scale Visual Recognition Challenge (LSVRC)** ran every year and helped researchers to innovate in network configurations, obtaining better and better results every year.

Alexnet, developed by Alex Krizhevsky, was the first deep convolutional network that won this challenge, and set a precedent for years to come. It consisted of a model similar in structure to Lenet-5, but the convolutional layers of which had a depth of hundreds of units, and the total number of parameters was in the tens of millions.

The following challenges saw the appearance of a powerful contender, the **Visual Geometry Group (VGG)** from Oxford University, with its VGG model.

The VGG model

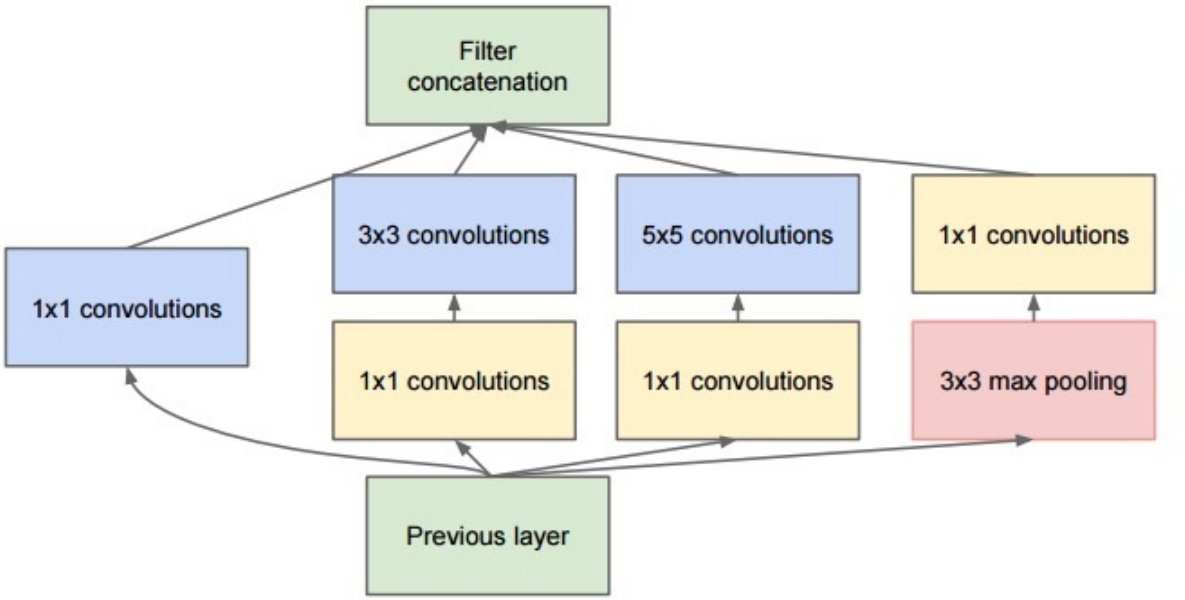
The main characteristic of the VGG network architecture is that it reduced the size of the convolutional filters to a simple 3 x 3 matrix and combined them in sequences, which was different to previous contenders, which had large kernel dimensions (up to 11 x 11). Paradoxically, the series of small convolutional weights amounted to a really large number of parameters (in the order of many millions), and so it had to be limited by a number of measures.



GoogLenet and the Inception model

GoogLenet was the neural network architecture that won the LSVRC in 2014, and was the first really successful attempt by one of the big IT companies in the series, which has been won mostly by corporations with giant budgets since 2014.

GoogLenet is basically a deep composition of nine chained Inception modules, with little or no modification. Each one of these Inception modules is represented in the following figure, and it's a mix of tiny convolutional blocks, intermixed with a 3 x 3 max pooling node:



Even with such complexity, GoogLenet managed to reduce the required parameter number (11 millon compared to 60 millon), and increased the accuracy (6.7% error compared to 16.4%) compared to Alexnet, which was released just two years previously. Additionally, the reuse of the Inception module allowed agile experimentation.

But it wasn't the last version of this architecture; soon enough, a second version of the Inception module was created, with the following characteristics.

Batch-normalized inception V2 and V3

In December 2015, with the paper *Rethinking the Inception Architecture for Computer Vision*, Google Research released a new iteration of the Inception architecture.

The internal covariance shift problem

One of the main problems of the original GoogLenet was training instability. As we saw earlier, input normalization consisted basically of centering all the input values on zero and dividing its value by the standard deviation in order to get a good baseline for the gradients of the backpropagations.

What occurs during the training of really large datasets is that after a number of training examples, the different value osculations begin to amplify the mean parameter value, like in a resonance phenomenon. This phenomenon is called **covariance shift**.

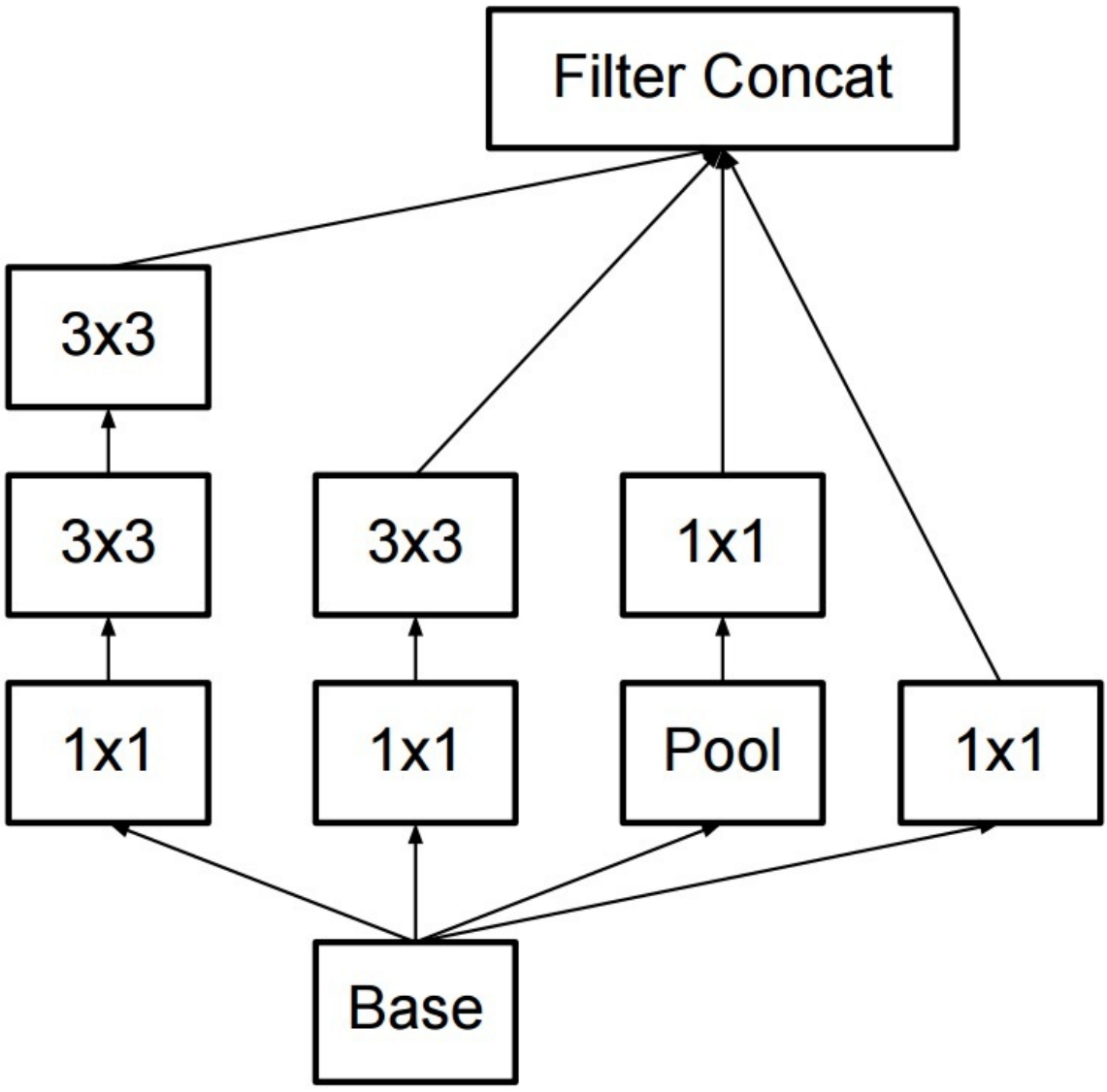
To mitigate this, the solution was to apply normalization not only to the original input values, but also to the output values at each layer, avoiding the instabilities appearing between layers before they begin to drift from the mean.

Apart from batch normalization, there were a number of additions proposed incrementally to V2:

- Reduce the number of convolutions to maximum of 3 x 3
- Increase the general depth of the networks
- Use the width increase technique on each layer to improve feature combination
- Factorization of convolutions

Inception V3 basically implements all the proposed innovations on the same architecture, and adds batch normalization to auxiliary classifiers of the networks.

In the following diagram, we represent the new architecture. Note the reduced size of the convolutional units:

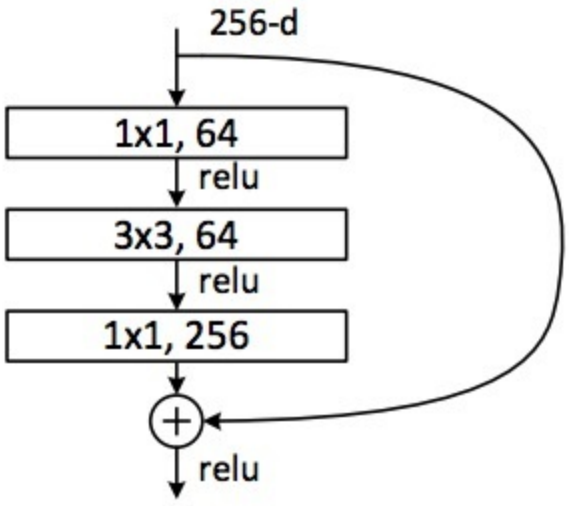


At the end of 2015, the last fundamental improvement in this series of architectures came from another company, Microsoft, in the form of **ResNets**.

Residual Networks (ResNet)

This new architecture appeared in December 2015 (more or less the same time as Inception V3), and it had a simple but novel idea—not only should the output of each constitutional layer be used but the architecture should also combine the output of the layer with the original input.

In the following diagram, we observe a simplified view of one of the ResNet modules. It clearly shows the sum operation at the end of the convolutional series, and a final ReLU operation:



The convolutional part of the module includes a feature reduction from 256 to 64 values, a 3 x 3 filter layer maintaining the feature numbers, and a feature augmenting the 1 x 1 layer from 64 x 256 values. Originally, it spanned more than 100 layers, but in recent developments, ResNet is also used in a depth of fewer than 30 layers, but with a wider distribution.

Now that we have seen a general overview of the main developments of recent years, let's go directly to the main types of application that researchers have discovered for CNNs.

Types of problem solved by deep layers of CNNs

CNNs have been employed to solve a wide variety of problems in the past. Here is a review of the main problem types, and a short reference to the architecture:

- Classification
- Detection
- Segmentation

Classification

Classification models, as we have seen previously, take an image or other type of input as a parameter and return one array with as many elements as the number of the possible classes, with a corresponding probability for each one.

The normal architecture for this type of solution is a complex combination of convolutional and pooling layers with a logistic layer at the end, showing the probability any of the pretrained

classes.

Detection

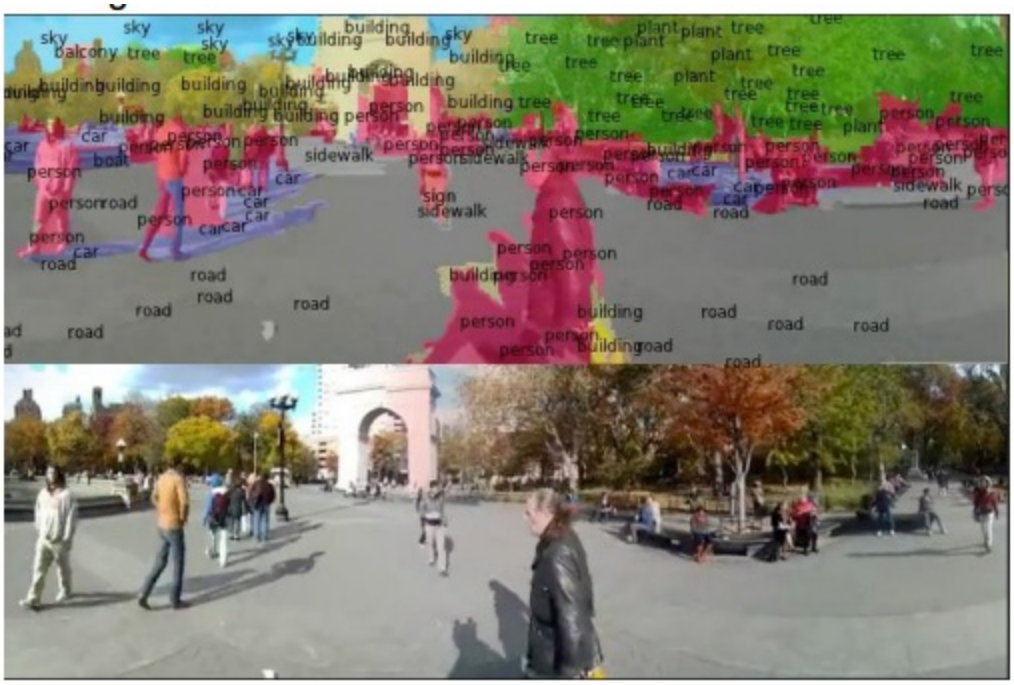
Detection adds a level of complexity because it requires guessing the location of one or more elements pertaining to the image, and then trying to classify each of these elements of information.

For this task, a common strategy for the individual localization problem is to combine a classification and regression problem—one (classification) for the class of the object, and the remaining one (regression) for determining the coordinates of the detected object—and then combining the losses into a common one.

For multiple elements, the first step is to determine a number of regions of interest, searching for places in the image that are statistically showing blobs of information belonging to the same object and then only applying the classification algorithms to the detected regions, looking for positive cases with high probabilities.

Segmentation

Segmentation adds an additional layer of complexity to the mix because the model has to locate the elements in an image and mark the exact shapes of all the located objects, as in the following illustration:



One of the most common approaches for this task is to implement sequential downsampling and upsampling operations, recovering a high-resolution image with only a certain number of possible outcomes per pixel that mark the class number for that element.

Deploying a deep neural network with Keras

In this exercise, we will generate an instance of the previously described Inception model, provided by the Keras application library. First of all, we will import all the required libraries, including the Keras model handling, the image preprocessing library, the gradient descent used to optimize the variables, and several Inception utilities. Additionally, we will use OpenCV libraries to adjust the new input images, and the common NumPy and matplotlib libraries:

```
from keras.models import Model
from keras.preprocessing import image
from keras.optimizers import SGD
from keras.applications.inception_v3 import InceptionV3, decode_predictions, preprocess_input

import matplotlib.pyplot as plt
import numpy as np
import cv2

Using TensorFlow backend.
```

Keras makes it really simple to load a model. You just have to invoke a new instance of the `InceptionV3` class, and then we will assign an optimizer based on **stochastic gradient descent**, and the categorical cross-entropy for the loss, which is very suitable for image classification problems:

```
model=InceptionV3()
model.compile(optimizer=SGD(), loss='categorical_crossentropy')
```

Now that the model is loaded into memory, it's time to load and adjust the photo using the `cv` library, and then we call the preprocess input of the Keras application, which will normalize the values:

```
# resize into VGG16 trained images' format
im = cv2.resize(cv2.imread('blue_jay.jpg'), (299, 299))
im = np.expand_dims(im, axis=0)
im = im /255.
im = im -0.5
im =  im *2
plt.figure (figsize=(10,10))
plt.imshow(im[0], cmap=plt.get_cmap('binary_r'))
plt.show()
```

This is what the image looks like after it's normalized—note how our structural understanding of the image has changed, but from the point of view of the model, this is the best way of allowing the model to converge:

Now we will invoke the `predict` method of the model, which will show the results of the last layer of the neural network, an array of probabilities for each of the categories. The `decode_predictions` method reads a dictionary with all the category numbers as indexes, and the category name as the value, and so it provides the name of the detected item classification, instead of the number:

```
out = model.predict(im)
print('Predicted:', decode_predictions(out, top=3)[0])
print (np.argmax(out))
```

```
Predicted: [('n01530575', 'brambling', 0.18225007), ('n01824575', 'coucal', 0.13728797),
('n01560419', 'bulbul', 0.048493069)]
10
```

As you can see, with this simple approach we have received a very approximate prediction from a list of similar birds. Additional tuning of the input images and the model itself could lead to more precise answers because the blue jay is a category included in the 1,000 possible classes.

Exploring a convolutional model with Quiver

In this practical example, we will load one of the models we have previously studied (in this case, `Vgg19`) with the help of the Keras library and Quiver. Then we will observe the different stages of the architecture, and how the different layers work, with a certain input.

Exploring a convolutional network with Quiver

Quiver (https://github.com/keplr-io/quiver) is a recent and very convenient tool used to explore models with the help of Keras. It creates a server that can be accessed by a contemporary web browser and allows the visualization of a model's structure and the evaluation of input images from the input layers until the final predictions.

With the following code snippet, we will create an instance of the `VGG16` model and then we will allow Quiver to read all the images sitting on the current directory and start a web application that will allow us to interact with our model and its parameters:

```
from keras.models import Model
from keras.preprocessing import image
from keras.optimizers import SGD
from keras.applications.vgg16 import VGG16
import keras.applications as apps

model=apps.vgg16.VGG16()

from quiver_engine.server import launch
launch(model,input_folder=".")
```

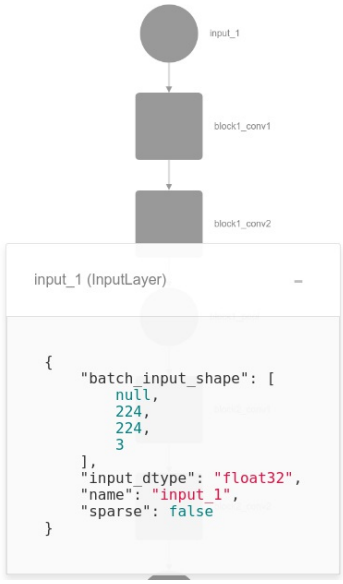
The script will then download the `VGG16` model weights (you need a good connection because it weighs in the hundreds of megabytes). Then it loads the model in memory and creates a server listening on port 5000.

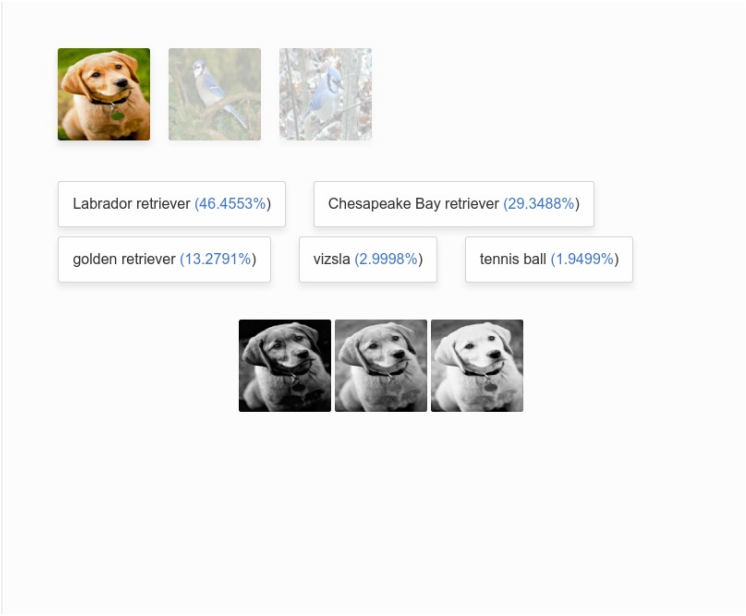
Note

The model weights that the Keras library downloads have been previously trained thoroughly with Imagenet, so it is ready to get very good accuracy on the 1,000 categories in our dataset.

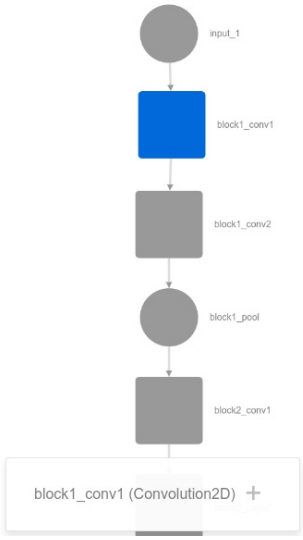
In the following screenshot, we see the first screen we will see after loading the index page of the web application. On the left, an interactive graphical representation of the network architecture is shown. On the center right, you can select one of the pictures in your current directory, and the application will automatically feed it as input, printing the five most likely outcomes of the input.

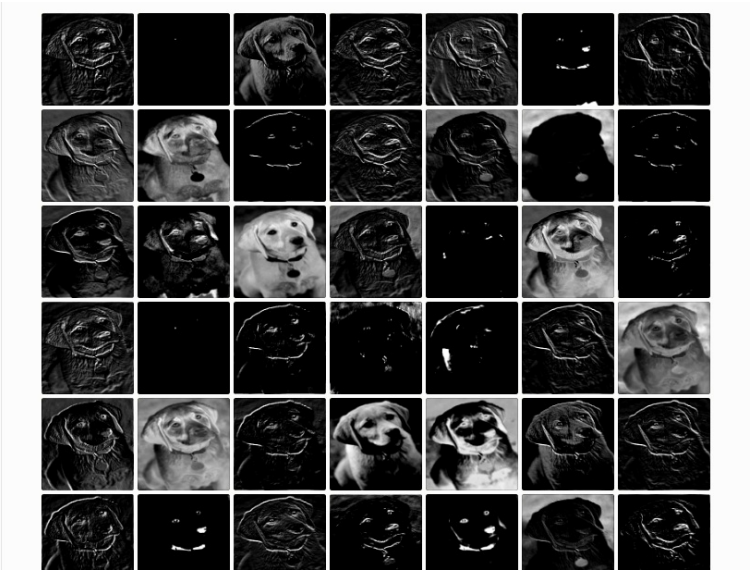
The screenshot also shows the first network layer, which basically consists of three matrices representing the red, green, and blue components of the original image:



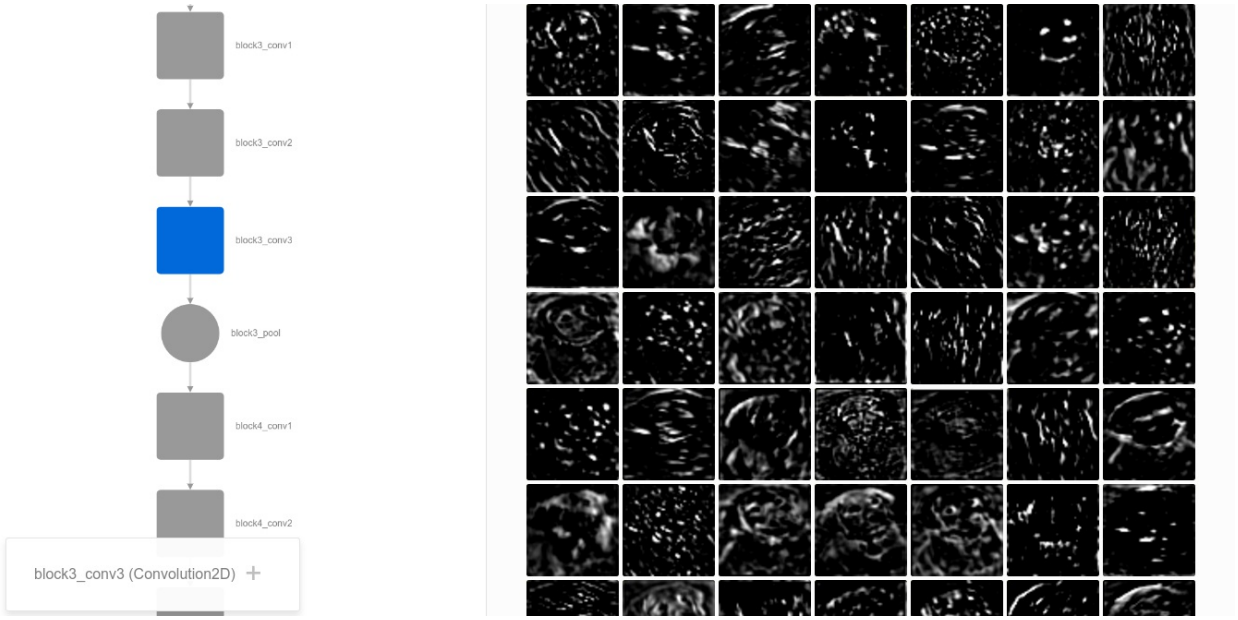


Then, as we advance into the model layers, we have the first convolutional layer. Here we can see that this stage highlights mainly high-level features, like the ones we set up with our 3 x 3 filters, such as different types of border, brightness, and contrast:

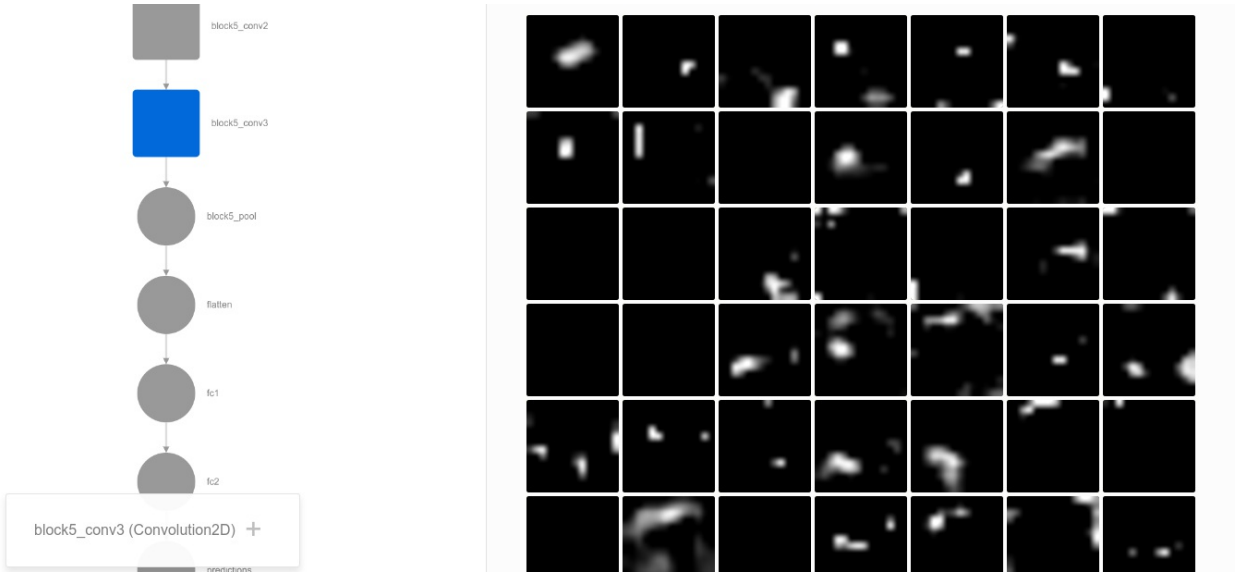




Let's advance a bit more. We now can see an intermediate layer that isn't focused on global features. Instead, we see that it has trained for intermediate features, such as different sets of textures, angles, or sets of features, such as eyes and noses:



When arriving at the last convolutional layers, really abstract concepts are appearing. This stage shows how incredibly powerful the models we are now training are, because now we are seeing highlighted elements without any useful (for us) meaning. These new abstract categories will lead, after some fully connected layers, to the final solution, which is a 1,000-element array with a float probability value, the probability value for each category in ImageNet:



We hope you can explore different examples and the layers' outputs, and try to discover how they highlight the different features for different categories of images.

Now it's time to work on a new type of machine learning, which consists of applying previously trained networks to work on new types of problem. This is called **transfer learning**.

Implementing transfer learning

In this example, we will implement one of the previously seen examples, replacing the last stages of a pretrained convolutional neural network and training the last stages for a new set of

elements, applying it to classification. It has the following advantages:

- It builds upon models with proved efficiency for image classification tasks
- It reduces the training time because we can reuse coefficients with an accuracy that could take weeks of computing power to reach

The dataset classes will be two different flower types from the flower17 dataset. It is a 17-category flower dataset with 80 images for each class. The flowers chosen are some common flowers in the UK. The images have large scale, pose, and light variations, and there are also classes with large variations of images within the class and close similarity to other classes. In this case, we will gather the first two classes (daffodil and coltsfoot), and build a classifier on top of the pretrained VGG16 network.

First, we will do image data augmentation, because the quantity of images may not be enough to abstract all the elements of each species. Let's start by importing all the required libraries, including applications, preprocessing, the checkpoint model, and the associated object, to allow us to save the intermediate steps, and the `cv2` and `NumPy` libraries for image processing and numerical base operations:

```
from keras import applications
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers
from keras.models import Sequential, Model
from keras.layers import Dropout, Flatten, Dense, GlobalAveragePooling2D
from keras import backend as k
from keras.callbacks import ModelCheckpoint, LearningRateScheduler, TensorBoard, EarlyStopping
from keras.models import load_model
from keras.applications.vgg16 import VGG16, decode_predictions,preprocess_input
import cv2
import numpy as np
```

```
Using TensorFlow backend.
```

In this section, we will define all the variables affecting the input, data sources, and training parameters:

```
img_width, img_height =224, 224
train_data_dir ="train"
validation_data_dir ="validation"
nb_train_samples =300
nb_validation_samples =100
batch_size =16
epochs =50
```

Now we will invoke the VGG16 pretrained model, not including the top flattening layers:

```
model = applications.VGG16(weights ="imagenet", include_top=False, input_shape = (img_width,
img_height, 3))

# Freeze the layers which you don't want to train. Here I am freezing the first 5 layers.for
layer in model.layers[:5]:
    layer.trainable =False#Adding custom Layers
x = model.output
x = Flatten()(x)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.5)(x)
x = Dense(1024, activation="relu")(x)
predictions = Dense(2, activation="softmax")(x)

# creating the final model
model_final = Model(input= model.input, output = predictions)
```

Now it's time to compile the model and create the image data augmentation object for the training and testing dataset:

```
# compile the model
model_final.compile(loss ="categorical_crossentropy", optimizer = optimizers.SGD(lr=0.0001,
momentum=0.9), metrics=["accuracy"])

# Initiate the train and test generators with data Augumentation
train_datagen = ImageDataGenerator(
rescale =1./255,
horizontal_flip =True,
fill_mode ="nearest",
zoom_range =0.3,
width_shift_range =0.3,
height_shift_range=0.3,
rotation_range=30)

test_datagen = ImageDataGenerator(
rescale =1./255,
horizontal_flip =True,
fill_mode ="nearest",
zoom_range =0.3,
width_shift_range =0.3,
height_shift_range=0.3,
rotation_range=30)
```

Now we will generate the new augmented data:

```
train_generator = train_datagen.flow_from_directory(
train_data_dir,
target_size = (img_height, img_width),
batch_size = batch_size,
class_mode ="categorical")

validation_generator = test_datagen.flow_from_directory(
validation_data_dir,
target_size = (img_height, img_width),
class_mode ="categorical")

# Save the model according to the conditions
checkpoint = ModelCheckpoint("vgg16_1.h5", monitor='val_acc', verbose=1, save_best_only=True,
save_weights_only=False, mode='auto', period=1)
early = EarlyStopping(monitor='val_acc', min_delta=0, patience=10, verbose=1, mode='auto')

Found 120 images belonging to 2 classes.
Found 40 images belonging to 2 classes.
```

It's time to fit the new final layers for the model:

```
model_final.fit_generator(
train_generator,
samples_per_epoch = nb_train_samples,
nb_epoch = epochs,
validation_data = validation_generator,
nb_val_samples = nb_validation_samples,
callbacks = [checkpoint, early])

Epoch 1/50
288/300 [============================>..] - ETA: 2s - loss: 0.7809 - acc: 0.5000

/usr/local/lib/python3.5/dist-packages/Keras-1.2.2-py3.5.egg/keras/engine/training.py:1573:
UserWarning: Epoch comprised more than `samples_per_epoch` samples, which might affect
learning results. Set `samples_per_epoch` correctly to avoid this warning.
  warnings.warn('Epoch comprised more than '


Epoch 00000: val_acc improved from -inf to 0.63393, saving model to vgg16_1.h5
304/300 [==============================] - 59s - loss: 0.7802 - acc: 0.4934 - val_loss: 0.6314
- val_acc: 0.6339
Epoch 2/50
296/300 [============================>.] - ETA: 0s - loss: 0.6133 - acc: 0.6385Epoch 00001:
val_acc improved from 0.63393 to 0.80833, saving model to vgg16_1.h5
312/300 [==============================] - 45s - loss: 0.6114 - acc: 0.6378 - val_loss:
0.5351 - val_acc: 0.8083
Epoch 3/50
288/300 [============================>..] - ETA: 0s - loss: 0.4862 - acc: 0.7986Epoch 00002:
```

```
val_acc improved from 0.80833 to 0.85833, saving model to vgg16_1.h5
304/300 [==============================] - 50s - loss: 0.4825 - acc: 0.8059 - val_loss: 0.4359
- val_acc: 0.8583
Epoch 4/50
296/300 [============================>.] - ETA: 0s - loss: 0.3524 - acc: 0.8581Epoch 00003:
val_acc improved from 0.85833 to 0.86667, saving model to vgg16_1.h5
312/300 [==============================] - 48s - loss: 0.3523 - acc: 0.8590 - val_loss:
0.3194 - val_acc: 0.8667
Epoch 5/50
288/300 [============================>..] - ETA: 0s - loss: 0.2056 - acc: 0.9549Epoch 00004:
val_acc improved from 0.86667 to 0.89167, saving model to vgg16_1.h5
304/300 [==============================] - 45s - loss: 0.2014 - acc: 0.9539 - val_loss: 0.2488
- val_acc: 0.8917
Epoch 6/50
296/300 [============================>.] - ETA: 0s - loss: 0.1832 - acc: 0.9561Epoch 00005:
val_acc did not improve
312/300 [==============================] - 17s - loss: 0.1821 - acc: 0.9551 - val_loss:
0.2537 - val_acc: 0.8917
Epoch 7/50
288/300 [============================>..] - ETA: 0s - loss: 0.0853 - acc: 0.9792Epoch 00006:
val_acc improved from 0.89167 to 0.94167, saving model to vgg16_1.h5
304/300 [==============================] - 48s - loss: 0.0840 - acc: 0.9803 - val_loss: 0.1537
- val_acc: 0.9417
Epoch 8/50
296/300 [============================>.] - ETA: 0s - loss: 0.0776 - acc: 0.9764Epoch 00007:
val_acc did not improve
312/300 [==============================] - 17s - loss: 0.0770 - acc: 0.9776 - val_loss:
0.1354 - val_acc: 0.9417
Epoch 9/50
296/300 [============================>.] - ETA: 0s - loss: 0.0751 - acc: 0.9865Epoch 00008:
val_acc did not improve
312/300 [==============================] - 17s - loss: 0.0719 - acc: 0.9872 - val_loss:
0.1565 - val_acc: 0.9250
Epoch 10/50
288/300 [============================>..] - ETA: 0s - loss: 0.0465 - acc: 0.9931Epoch 00009:
val_acc did not improve
304/300 [==============================] - 16s - loss: 0.0484 - acc: 0.9901 - val_loss: 0.2148
- val_acc: 0.9167
Epoch 11/50
296/300 [============================>.] - ETA: 0s - loss: 0.0602 - acc: 0.9764Epoch 00010:
val_acc did not improve
312/300 [==============================] - 17s - loss: 0.0634 - acc: 0.9744 - val_loss:
0.1759 - val_acc: 0.9333
Epoch 12/50
288/300 [============================>..] - ETA: 0s - loss: 0.0305 - acc: 0.9931
```

Now let's try this with a daffodil image. Let's test the output of the classifier, which should output an array close to `[1.,0.]`, indicating that the probability for the first option is very high:

```
im = cv2.resize(cv2.imread('test/gaff2.jpg'), (img_width, img_height))
im = np.expand_dims(im, axis=0).astype(np.float32)
im=preprocess_input(im)


out = model_final.predict(im)

print (out)
print (np.argmax(out))

[[  1.00000000e+00   1.35796010e-13]]
0
```

So, we have a very definitive answer for this kind of flower. You can play with new images and test the model with clipped or distorted images, even with related classes, to test the level of accuracy.

References

- Fukushima, Kunihiko, and Sei Miyake, *Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition.* Competition and cooperation in neural nets. Springer, Berlin, Heidelberg, 1982. 267-285.
- LeCun, Yann, et al. *Gradient-based learning applied to document recognition.* Proceedings of the IEEE 86.11 (1998): 2278-2324.
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks.* Advances in neural information processing systems. 2012.
- Hinton, Geoffrey E., et al, *Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors.* arXiv preprint arXiv:1207.0580 (2012).
- Simonyan, Karen, and Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv preprint arXiv:1409.1556 (2014).
- Srivastava, Nitish, et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* Journal of machine learning research15.1 (2014): 1929-1958.
- Szegedy, Christian, et al, *Rethinking the Inception Architecture for Computer Vision.* Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.
- He, Kaiming, et al, *Deep Residual Learning for Image Recognition.* Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.
- Chollet, François, *Xception: Deep Learning with Depthwise Separable Convolutions.* arXiv preprint arXiv:1610.02357 (2016).

Summary

This chapter provides important insights into one of the technologies responsible for the amazing new applications you see in the media every day. Also, with the practical example provided, you will even be able to create new customized solutions.

As our models won't be enough to solve very complex problems, in the following chapter, our scope will expand even more, adding the important dimension of time to the set of elements included in our generalization.

Chapter 7. Recurrent Neural Networks

After we reviewed the recent developments in deep learning, we are now reaching the cutting-edge of machine learning, and we are now adding a very special dimension to our model (time, and hence sequences of inputs) through a recent series of algorithms called **recurrent neural networks (RNNs)**.

Solving problems with order — RNNs

In the previous chapters, we have examined a number of models, from simple ones to more sophisticated ones, with some common properties:

- They accept unique and isolated input
- They have unique and fixed size output
- The outputs will depend exclusively on the current input characteristics, without dependency on past or previous input

In real life, the pieces of information that the brain processes have an inherent structure and order, and the organization and sequence of every phenomenon we perceive has an influence on how we treat them. Examples of this include speech comprehension (the order of the words in a sentence), video sequence (the order of the frames in a video), and language translation. This prompted the creation of new models. The most important ones are grouped under the RNN umbrella.

RNN definition

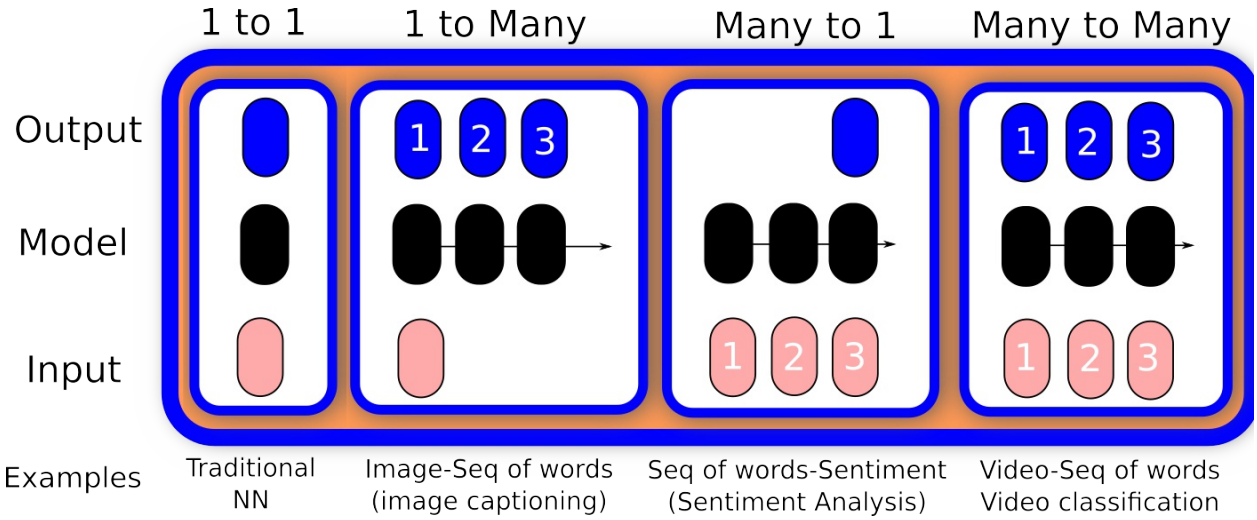
RNNs are **Artificial Neural Network** (**ANN**) models whose inputs and outputs are sequences. A more formal definition can be expressed as follows:

> *"An RNN represents a sequence with a high-dimensional vector (called the hidden state) of a fixed dimensionality that incorporates new observations using a complex non-linear function."*

RNNs are highly expressive and can implement an arbitrary memory-bounded computation, and as a result, they can be configured to achieve non-trivial performance on difficult sequence tasks.

Types of sequence to be modeled

RNNs work with sequences models, both in the input and the output realm. Based on this, we can have all the possible combinations to solve different kinds of problems. In the following diagram, we illustrate the main architectures used in this field, and then a reference of the recursive ones:

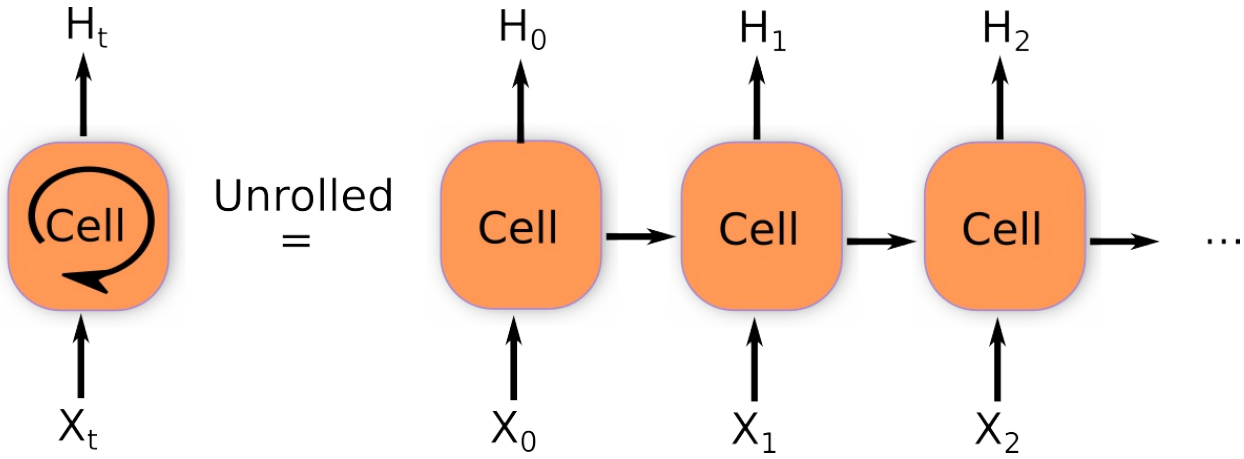


Type of sequences modes

Development of RNN

The origin of RNNs is surprisingly common to the other modern neural network architectures, dating back to Hopfield networks from the 1980s, but with counterparts in the 1970s.

The common structure for the first iterations of the recurrent architecture can be represented in the following way:



Recurrent cell unrolling

Classic RNN nodes have recurrent connections to themselves, and so they can evolve their weights as the input sequence progresses. Additionally, on the right of the diagram, you can see how the network can be *unrolled* to generate a set of outputs based on the stochastic model it has saved internally. It stores representations of recent input events in the form of activation (**short-term memory**, as opposed to **long-term memory**, embodied by slowly changing weights). This is potentially significant for many applications, including speech processing, music composition (such as *Mozer, 1992*), **Natural Language Processing** (**NLP**), and many other fields.

Training method — backpropagation through time

After the considerable number of model types we've been studying, it's possible that you can already see a pattern in the implementation of the training steps.

For recurrent neural networks, the most well-known error minimization technique is a variation of the well-known (for us) backpropagation methods, with a simple element – **backpropagation through time** (**BPTT**) works by unrolling all input timesteps. Each timestep has one input timestep, one copy of the whole network, and one output. Errors are calculated and accumulated for each timestep, and finally the network is rolled back up and the weights are updated.

Spatially, each timestep of the unrolled recurrent neural network can be seen as an additional layer, given the dependence from one timestep to another and how every timestep output is taken as an input for the subsequent timestep. This can lead to really complex training performance requirements, and thus, **truncated backpropagation through time** was born.

The following pseudocode represents the whole process:

```
Unfold the network to contain k instances of the cell
While (error < ε or iteration>max):
    x = zeros(sequence_legth)
    for t in range (0, n-sequence_length)  # initialize the weights
        copy sequence_length input values into the input x
        p = (forward-propagate the inputs over the whole unfolded network)
        e = y[t+k] - p;           # calculate error as target - prediction
        Back-propagate the error e, back across the whole unfolded network
        Sum the weight changes in the k model instances together.
            Update all the weights in f and g.
            x = f(x, a[t]);    # compute new input for the next time-step
```

Main problems of the traditional RNNs — exploding and vanishing gradients

However, RNNs have turned out to be difficult to train, especially on problems with complicated long-range temporal structures – precisely the setting where RNNs ought to be most useful. Since their potential has not been realized, methods that address the difficulty of training RNNs are of great importance.

The most widely used algorithms for learning what to put in short-term memory, however, take too much time or do not work well at all, especially when minimal time lags between inputs and corresponding teacher signals are long. Although theoretically fascinating, the existing methods did not provide clear practical advantages over traditional feedforward networks.

One of the main problems with RNNs happens in the backpropagation stage. Given its recurrent nature, the number of steps that the backpropagation of the errors has corresponds to a very deep network. This cascade of gradient calculations could lead to a very insignificant value in the last stages, or on the contrary, to ever-increasing and unbounded parameters. Those phenomena receive the names of vanishing and exploding gradients. This is one of the reasons for which LSTM was created.

The problem with conventional BPTT is that error signals going backwards in time tend to either blow up or vanish – the temporal evolution of the backpropagated error exponentially depends on the size of the weights. It could lead to oscillating weights, or taking a prohibitive amount of time, or it could not work at all.

As a consequence of many different attempts to solve the problem with vanishing and exploding gradients, finally in 1997, *Schmidhuber* and *Sepp* published a fundamental paper on RNNs, and LSTM, which paved the way for all modern developments in the area.

LSTM

LSTMs are a fundamental step in RNNs, because they introduce long-term dependencies into the cells. The unrolled cells contain two different parameter lines: one long-term status, and the other representing short-term memory.

Between steps, the long-term forgets less important information, and adds filtered information from short-term events, incorporating them into the future.

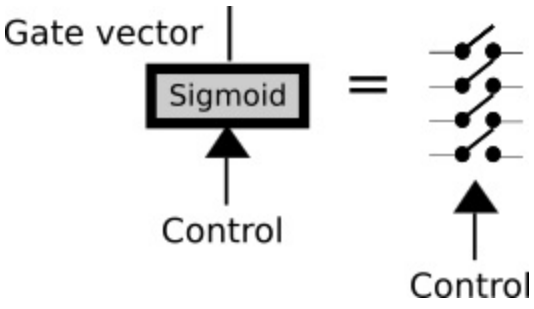
LSTMs are really versatile in their possible applications, and they are the most commonly employed recurrent models, along with GRUs, which we will explain later. Let's try to break down an LSTM into its components to get a better understanding of how they work.

The gate and multiplier operation

LSTMs have two fundamental values: remembering important things from the present, and slowly forgetting unimportant things from the past. What kind of mechanism can we use to apply this kind of filtering? It's called the **gate** operation.

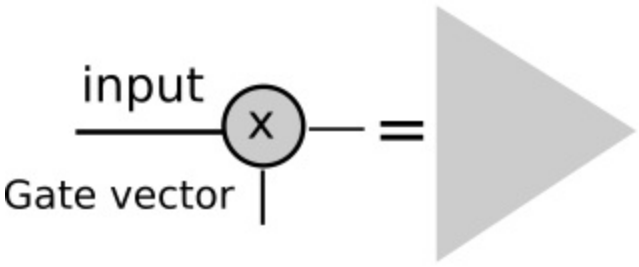
The gate operation basically has a multivariate vector input and a filter vector, which will be dot multiplied with the input, allowing or rejecting the elements from the inputs to be transferred. How do we adjust this gate's filters? This multivariate control vector (marked with an arrow on the diagram) is connected with a neural network layer with a sigmoid activation function. If we apply the control vector and pass through the sigmoid function, we will get a binary-like output vector.

In the following diagram, the gate will be represented by a series of switches:



LSTM gate

Another important part of the process is the multiplications, which formalize the trained filters, effectively multiplying the inputs by an included gate. The arrow icon indicates the direction in which the filtered information will flow:



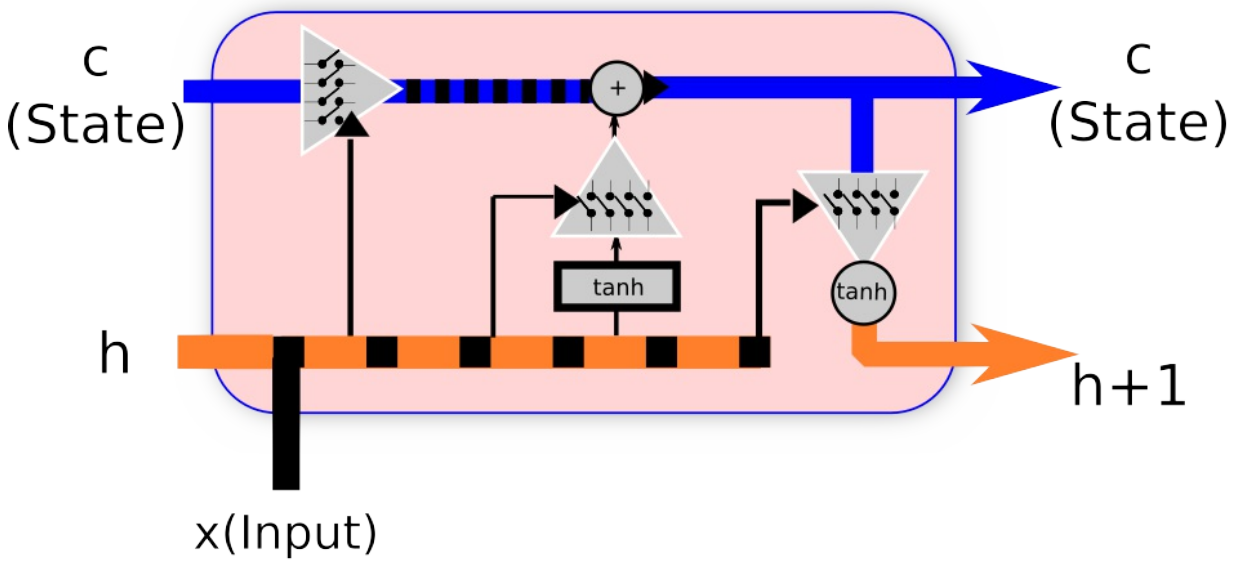
Gate multiplication step

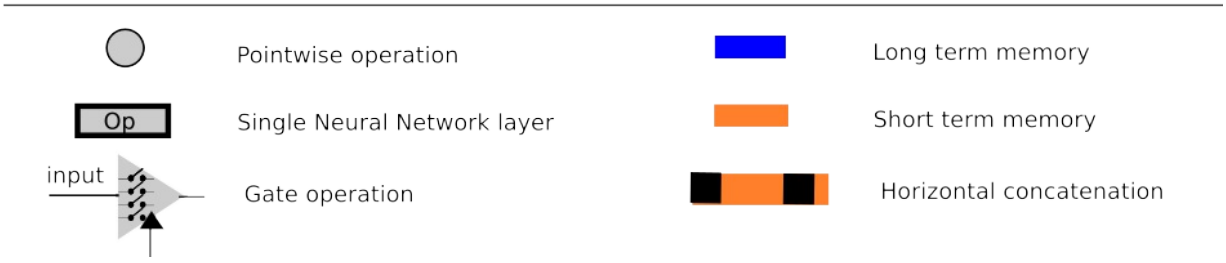
Now, it's time to describe an LSTM cell in more detail.

An LSTM has three gates to protect and control the cell state: one at the start of the data flow, another in the middle, and the last at the end of the cell's informational boundaries. This operation will allow both discarding (hopefully not important) low-important state data and incorporating (hopefully important) new data to the state.

The following diagram shows all the concepts in the operation of one LSTM cell. As inputs, we have the following:

- The cell state, which will store long-term information, because it carries the optimized weights dating from the origin of the cell training
- The short-term state, *h(t)*, which will be directly combined with the current input on each iteration, and so it will have a much bigger influence on the latest values of the inputs:



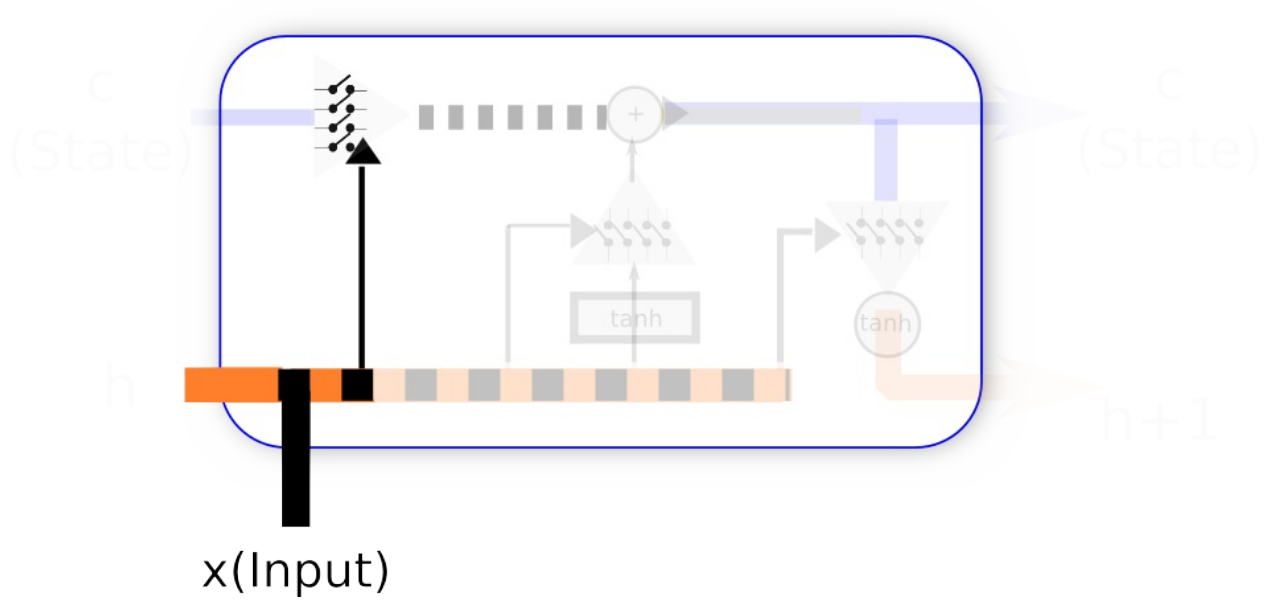


Depiction of an LSTM cell, with all its components

Now, let's explore the data flow on this LSTM unit in order to better understand how the different gates and operations work together in a single cell.

Part 1 — set values to forget (input gate)

In this stage, we take the values coming from the short-term memory combined with the input itself, and these values will then output a binary function, represented by a multivariable sigmoid. Depending on the input and short-term memory values, the sigmoid output will filter the long-term knowledge, represented by the weights of the cell's state:

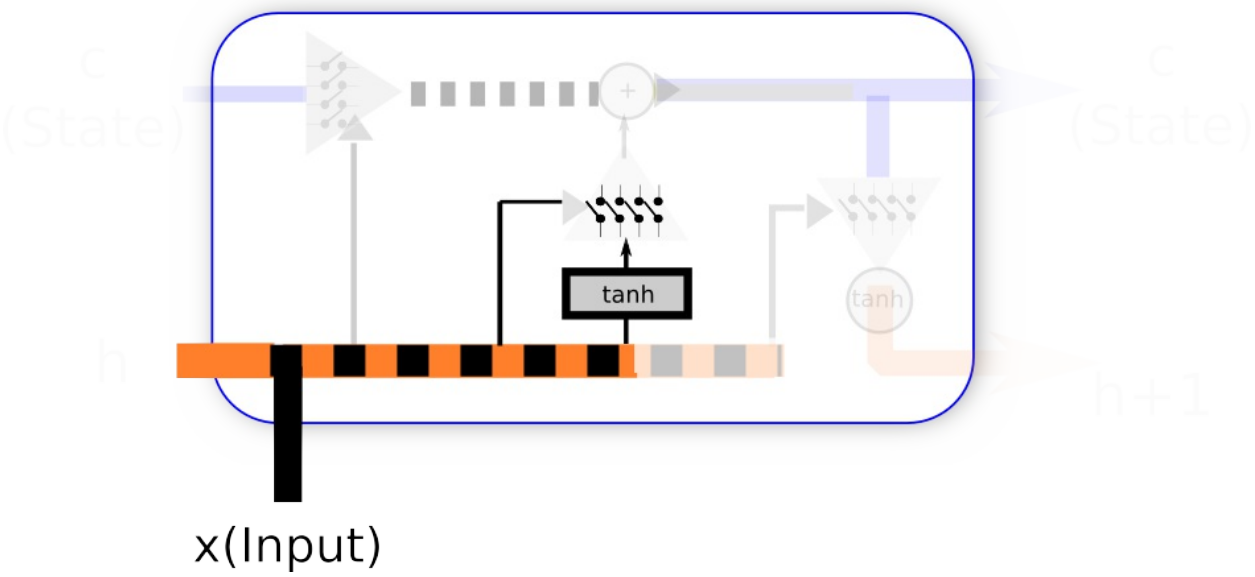


State forget parameters setup

Part 2 — set values to keep

Now, it's time to set the filter that will allow or reject the incorporation of new and short-term memory to the cell's semi-permanent state. Then it is time to set the filter that will allow or reject the incorporation of new and short-term memory to the cell's semi-permanent state.

So, at this stage, we will determine how much of the new and semi-new information will be incorporated in the cell's new state:

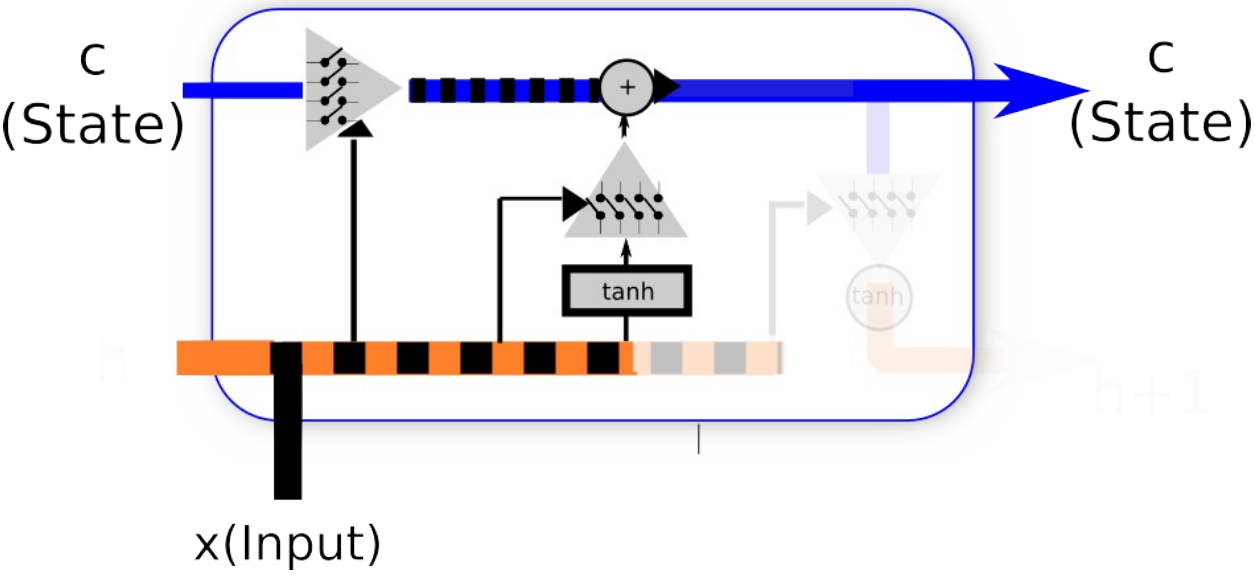


Short-term values selection step

Part 3 — apply changes to cell

In this part of the sequence, we will finally pass through the information filter we have been configuring, and as a result, we will have an updated long-term state.

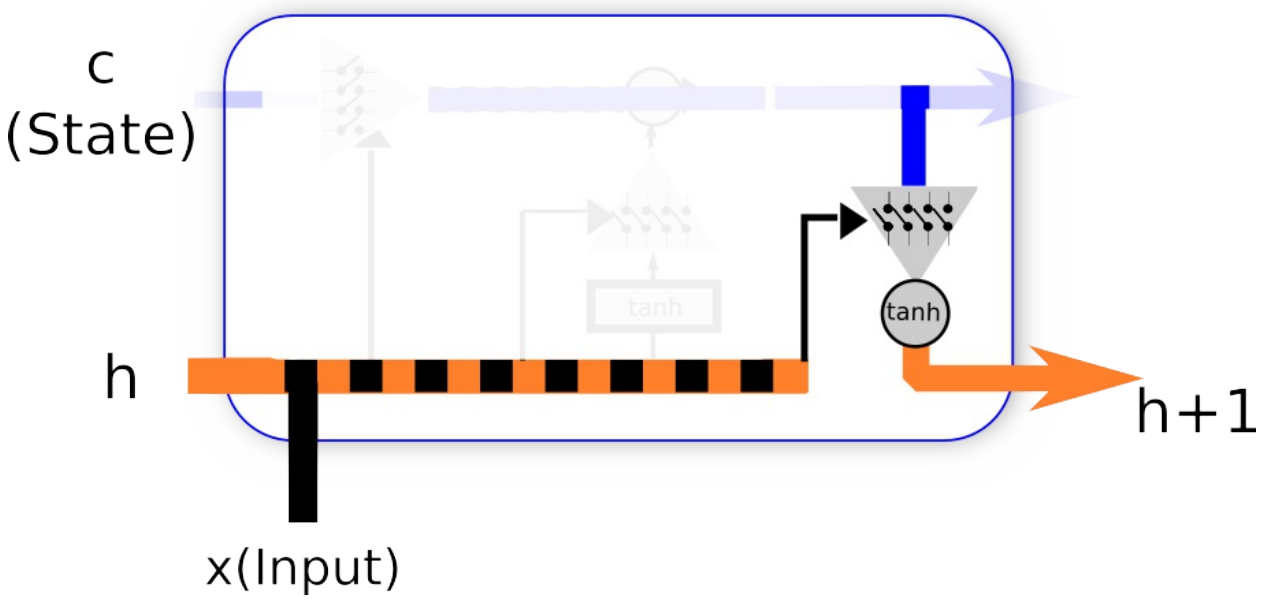
In order to normalize the new and short-term information, we pass the new input and the short-term state via a neural network with **tanh** activation. This will allow us to feed the new information in a normalized *[-1,1]* range:



State changes persistence step

Part 4 — output filtered cell state

Now, it's the turn of the short-term state. It will also use the new and previous short-term state to set up the filter that will pass the long-term state, dot multiplied by a tanh function, again to normalize the information to incorporate a *(-1,1)* range:



New short-term generation step

Univariate time series prediction with energy consumption data

In this example, we will be solving a problem in the domain of regression. For this reason, we will build a multi-layer RNN with two LSTMs. The type of regression we will do is of the *many to one* type, because the network will receive a sequence of energy consumption values and will try to output the next value based on the previous four registers.

The dataset we will be working on is a compendium of many measurements of the power consumption of one home over a period of time. As we might infer, this kind of behavior can easily follow patterns (it increases when the occupants use the microwave to prepare breakfast and use computers during the day, it decreases a bit in the afternoon, and then increases in the evening with all the lights, finally decreasing to zero when the occupants are asleep).

Let's start by setting the appropriate environment variables and loading the required libraries:

```
%matplotlib inline
%config InlineBackend.figure_formats = {'png', 'retina'}

import numpy as np
import pandas as pd
import tensorflow as tf
from matplotlib import pyplot as plt

from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.layers.recurrent import LSTM
from keras.layers import Dropout
```

```
Using TensorFlow backend.
```

Dataset description and loading

In this example, we will be using the **Electricity Load Diagrams Data Sets**, from *Artur Trindade* (https://archive.ics.uci.edu/ml/datasets/ElectricityLoadDiagrams20112014). This is the description of the original dataset:

> *"Data set has no missing values. Values are in kW of each 15 min. To convert values in kWh values must be divided by 4. Each column represent one client. Some clients were created after 2011. In these cases consumption were considered zero. All time labels report to Portuguese hour. However all days present 96 measures (24*15). Every year in March time change day (which has only 23 hours) the values between 1:00 am and 2:00 am are zero for all points. Every year in October time change day (which has 25 hours) the values between 1:00 am and 2:00 am aggregate the consumption of two hours."*

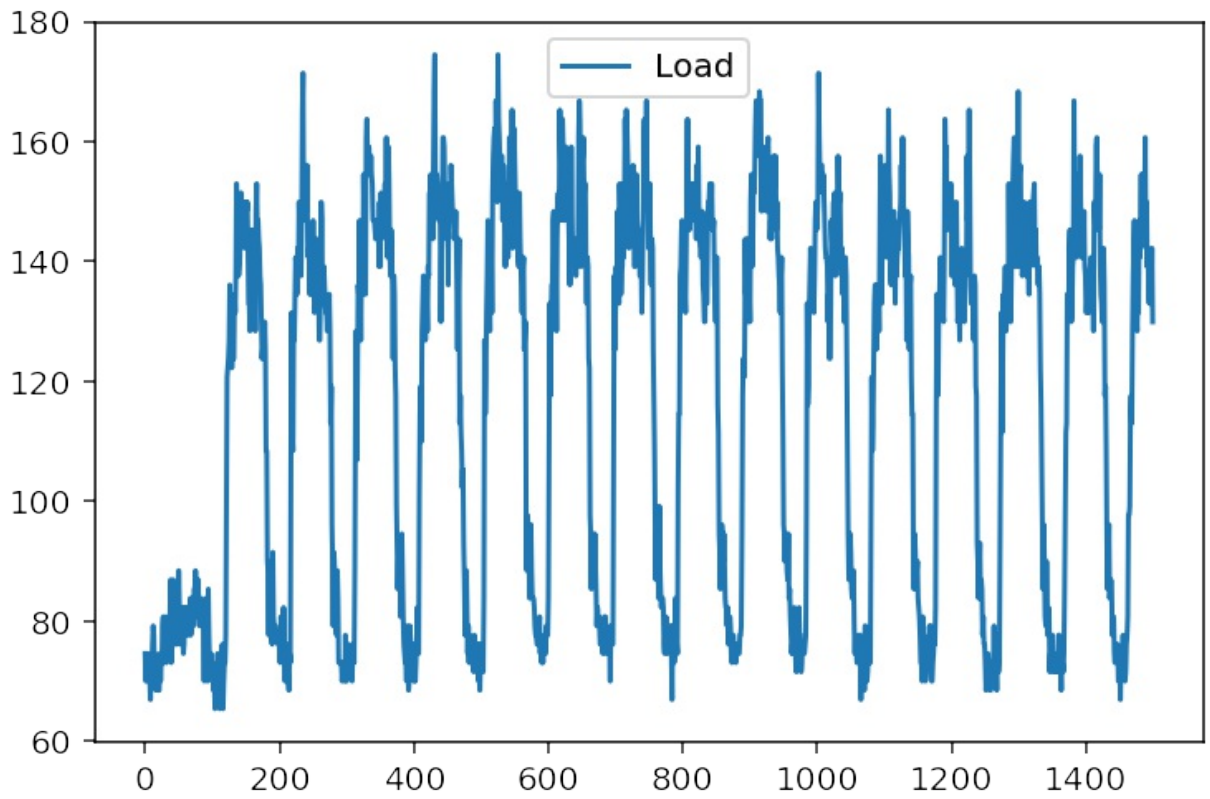
In order to simplify our model description, we took just one client's complete measurements and converted its format to standard CSV. It is located in the `data` subfolder of this chapter's code folder.

So, we will load the first 1,500 values of the consumption of a sample home from the dataset:

```
df = pd.read_csv("data/elec_load.csv", error_bad_lines=False)
```

```
plt.subplot()
plot_test, = plt.plot(df.values[:1500], label='Load')
plt.legend(handles=[plot_test])
```

The following graph shows the subset of data that we need to model:



If we take a look at this representation (we took the first 1,500 samples) we can see an initial transient state, probably when the measurements were put in place, and then we see a really clear cycle of high and low consumption levels. From simple observation, we can also see that the cycles are of more or less 100 samples, pretty close to the 96 samples per day this dataset has.

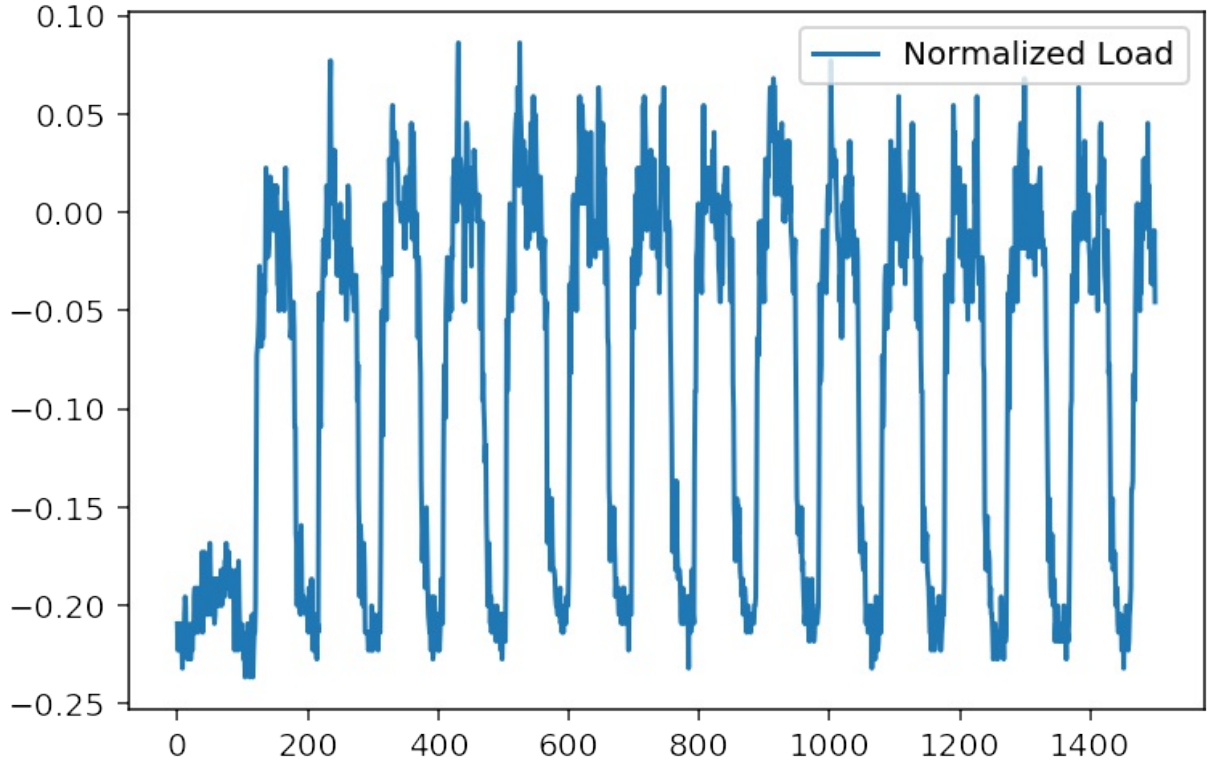
Dataset preprocessing

In order to assure a better convergence of the backpropagation methods, we should try to normalize the input data. So, we will be applying the classic scale and centering technique, subtracting the mean value, and scaling by the `floor()` of the maximum value. To get the required values, we use the pandas `describe()` method:

```
print(df.describe())
array=(df.values -145.33) /338.21
plt.subplot()
plot_test, = plt.plot(array[:1500], label='Normalized Load')
plt.legend(handles=[plot_test])
```

```
                Load
count  140256.000000
mean      145.332503
std        48.477976
min         0.000000
25%       106.850998
50%       151.428571
75%       177.557604
max       338.218126
```

This is the graph of our normalized data:



In this step, we will prepare our input dataset, because we need an input `x` (the previous 5 values) with a corresponding input `y` (the value after 5 timesteps). Then, we will assign the first 13,000 elements to the training set, and then we will assign the following 1,000 samples to the testing set:

```
listX = []
listy = []
X={}
y={}

for i inrange(0,len(array)-6):
    listX.append(array[i:i+5].reshape([5,1]))
    listy.append(array[i+6])

arrayX=np.array(listX)
arrayy=np.array(listy)

X['train']=arrayX[0:13000]
X['test']=arrayX[13000:14000]

y['train']=arrayy[0:13000]
y['test']=arrayy[13000:14000]
```

Now, we will build the model, which will be a dual LSTM with a dropout layer at the end of each:

```
#Build the model
model = Sequential()

model.add(LSTM( units=50, input_shape=(None, 1), return_sequences=True))

model.add(Dropout(0.2))

model.add(LSTM( units=200, input_shape=(None, 100), return_sequences=False))
model.add(Dropout(0.2))
```

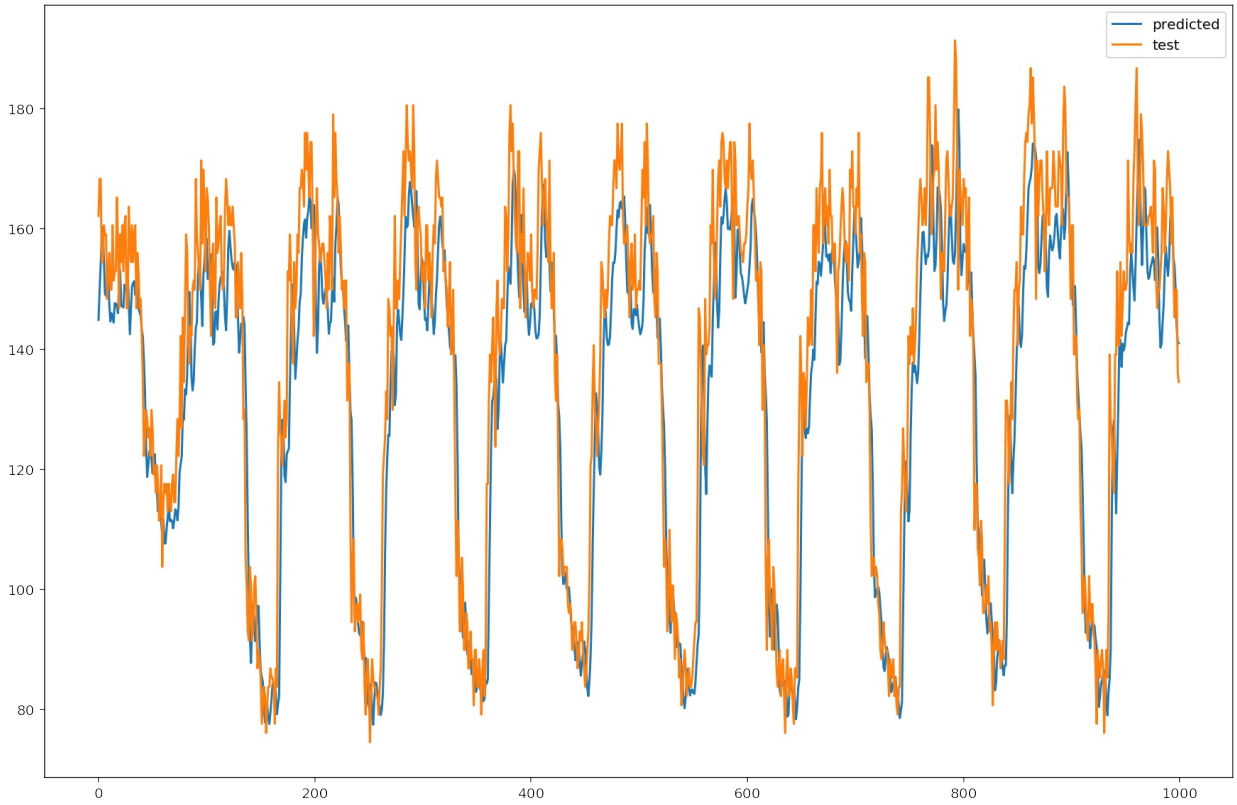
```
model.add(Dense(units=1))
model.add(Activation("linear"))

model.compile(loss="mse", optimizer="rmsprop")
```

Now it's time to run the model and adjust the weights. The model fitter will use 8% of the dataset values as the validation set:

```
#Fit the model to the data

model.fit(X['train'], y['train'], batch_size=512, epochs=10, validation_split=0.08)

Train on 11960 samples, validate on 1040 samples
Epoch 1/10
11960/11960 [==============================] - 41s - loss: 0.0035 - val_loss: 0.0022
Epoch 2/10
11960/11960 [==============================] - 61s - loss: 0.0020 - val_loss: 0.0020
Epoch 3/10
11960/11960 [==============================] - 45s - loss: 0.0019 - val_loss: 0.0018
Epoch 4/10
11960/11960 [==============================] - 29s - loss: 0.0017 - val_loss: 0.0020
Epoch 5/10
11960/11960 [==============================] - 30s - loss: 0.0016 - val_loss: 0.0015
Epoch 6/10
11960/11960 [==============================] - 28s - loss: 0.0015 - val_loss: 0.0013
Epoch 7/10
11960/11960 [==============================] - 43s - loss: 0.0014 - val_loss: 0.0012
Epoch 8/10
11960/11960 [==============================] - 37s - loss: 0.0013 - val_loss: 0.0013
Epoch 9/10
11960/11960 [==============================] - 31s - loss: 0.0013 - val_loss: 0.0012
Epoch 10/10
11960/11960 [==============================] - 25s - loss: 0.0012 - val_loss: 0.0011


<keras.callbacks.History at 0x7fa435512588>
```

After rescaling, it's time to see how our model predicts the values compared with the actual test values, which didn't participate in the training of the models, to understand how the model generalizes the behavior of the sample home:

```
# Rescale the test dataset and predicted data

test_results = model.predict( X['test'])

test_results = test_results *338.21+145.33
y['test'] = y['test'] *338.21+145.33

plt.figure(figsize=(10,15))
plot_predicted, = plt.plot(test_results, label='predicted')

plot_test, = plt.plot(y['test']  , label='test');
plt.legend(handles=[plot_predicted, plot_test]);
```



Final regressed data

Summary

In this chapter, our scope has expanded even more, adding the important dimension of time to the set of elements to be included in our generalization. Also, we learned how to solve a practical problem with RNNs, based on real data.

But if you think you have covered all the possible options, there are many more model types to see!

In the next chapter, we will talk about cutting edge architectures that can be trained to produce very clever elements, for example, transfer the style of famous painters to a picture, and even play video games! Keep reading for reinforcement learning and generative adversarial networks.

References

- Hopfield, John J, *Neural networks and physical systems with emergent collective computational abilities.* Proceedings of the national academy of sciences 79.8 (1982): 2554-2558.
- Bengio, Yoshua, Patrice Simard, and Paolo Frasconi, *Learning long-term dependencies with gradient descent is difficult.* IEEE transactions on neural networks 5.2 (1994): 157-166.
- Hochreiter, Sepp, and Jürgen Schmidhuber, *long short-term memory*. Neural Computation 9.8 (1997): 1735-1780.
- Hochreiter, Sepp. *Recurrent neural net learning and vanishing gradient.* International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems 6.2 (1998): 107-116.
- Sutskever, Ilya, *Training recurrent neural networks.* University of Toronto, Toronto, Ont., Canada (2013).
- Chung, Junyoung, et al, *Empirical evaluation of gated recurrent neural networks on sequence modeling.* arXiv preprint arXiv:1412.3555 (2014).

Chapter 8. Recent Models and Developments

In the previous chapters, we have explored a large number of training mechanisms for machine learning models, starting with simple pass-through mechanisms, such as the well-known feedforward neural networks. Then we looked at a more complex and reality-bound mechanism, accepting a determined sequence of inputs as the training input, with **Recurrent Neural Networks** (**RNNs**).

Now it's time to take a look at two recent players that incorporate other aspects of the real world. In the first case, we will have not only a single network optimizing its model, but also another participant, and they will both improve each other's results. This is the case of **Generative Adversarial Networks** (**GANs**).

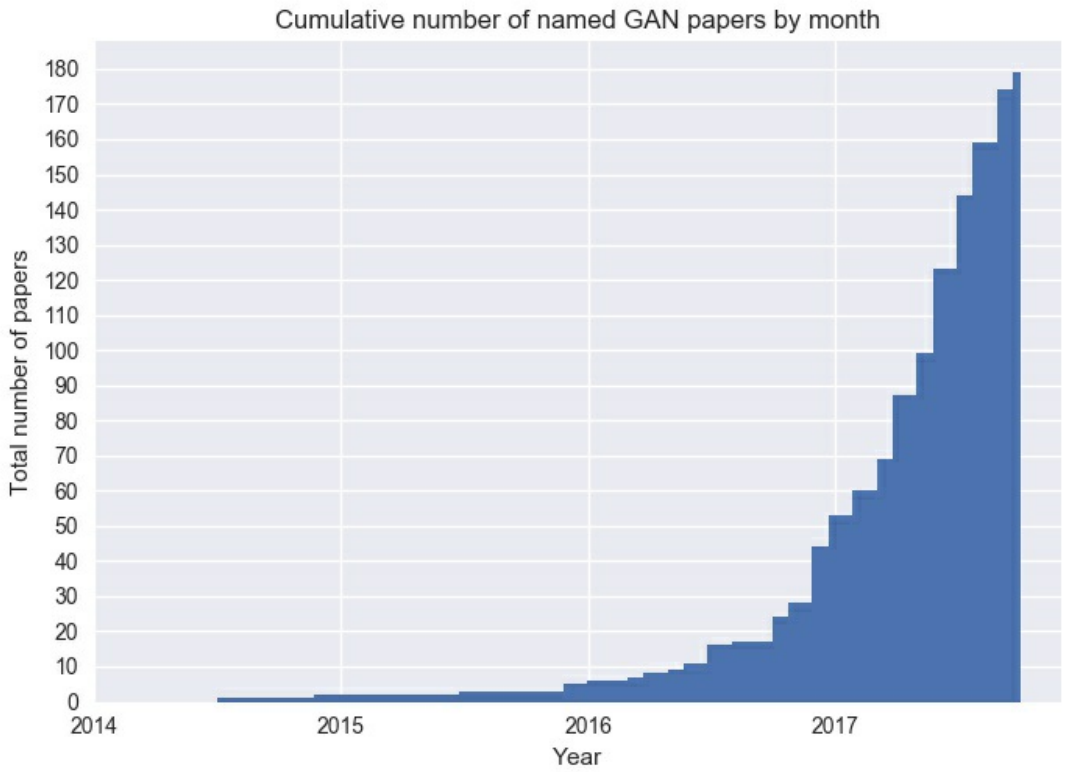
In the second case, we will talk about a different kind of model, which will try to determine the optimal set of steps to maximize a reward: **reinforcement learning**.

GANs

GANs are a new kind of unsupervised learning model, one of the very few disrupting models of the last decade. They have two models competing with and improving each other throughout the iterations.

This architecture was originally based on supervised learning and game theory, and its main objective is to basically learn to generate realistic samples from an original dataset of elements of the same class.

It's worth noting that the amount of research on GANs is increasing at an almost exponential rate, as depicted in the following graph:

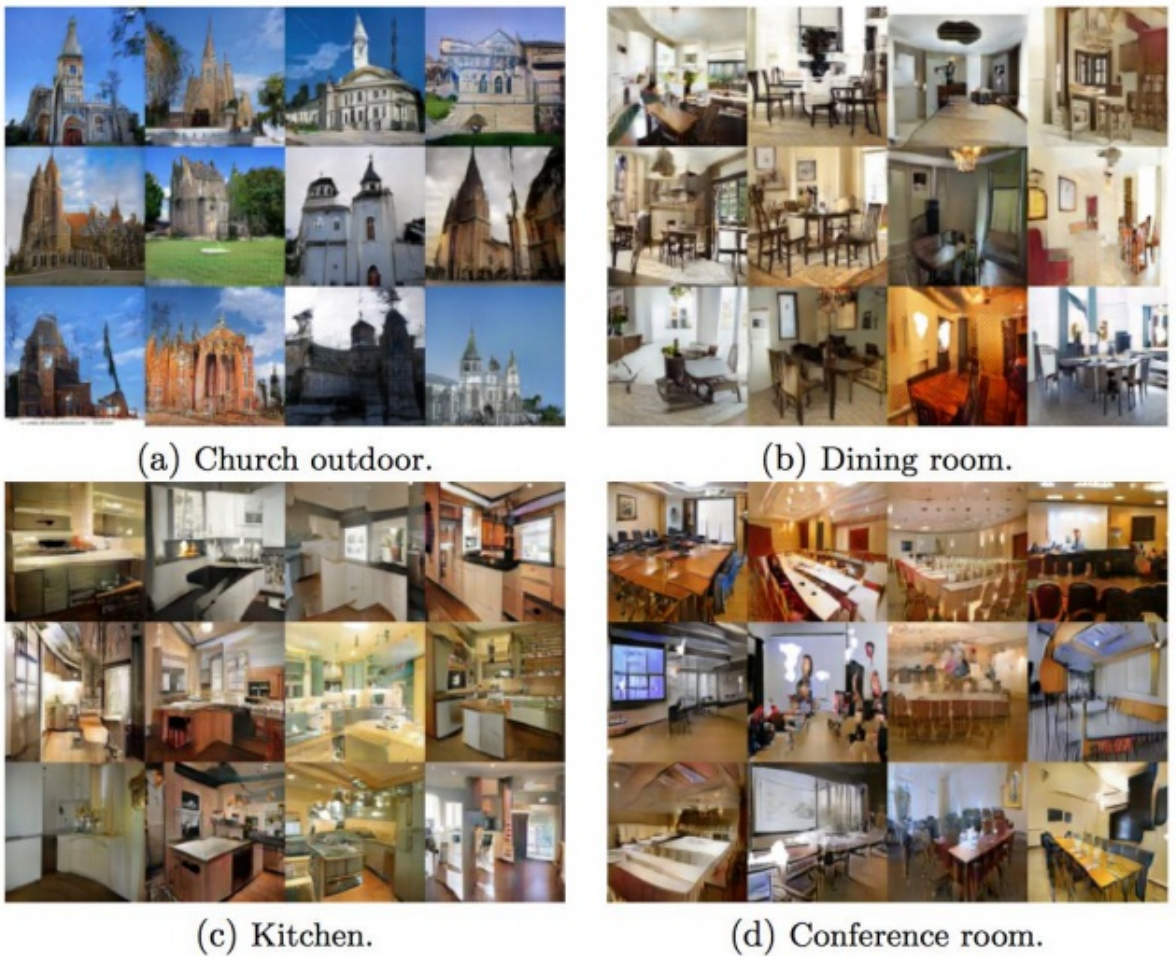


Source: The GAN Zoo (https://github.com/hindupuravinash/the-gan-zoo)

Types of GAN applications

GANs allow new applications to produce new samples from a previous set of samples, including completing missing information.

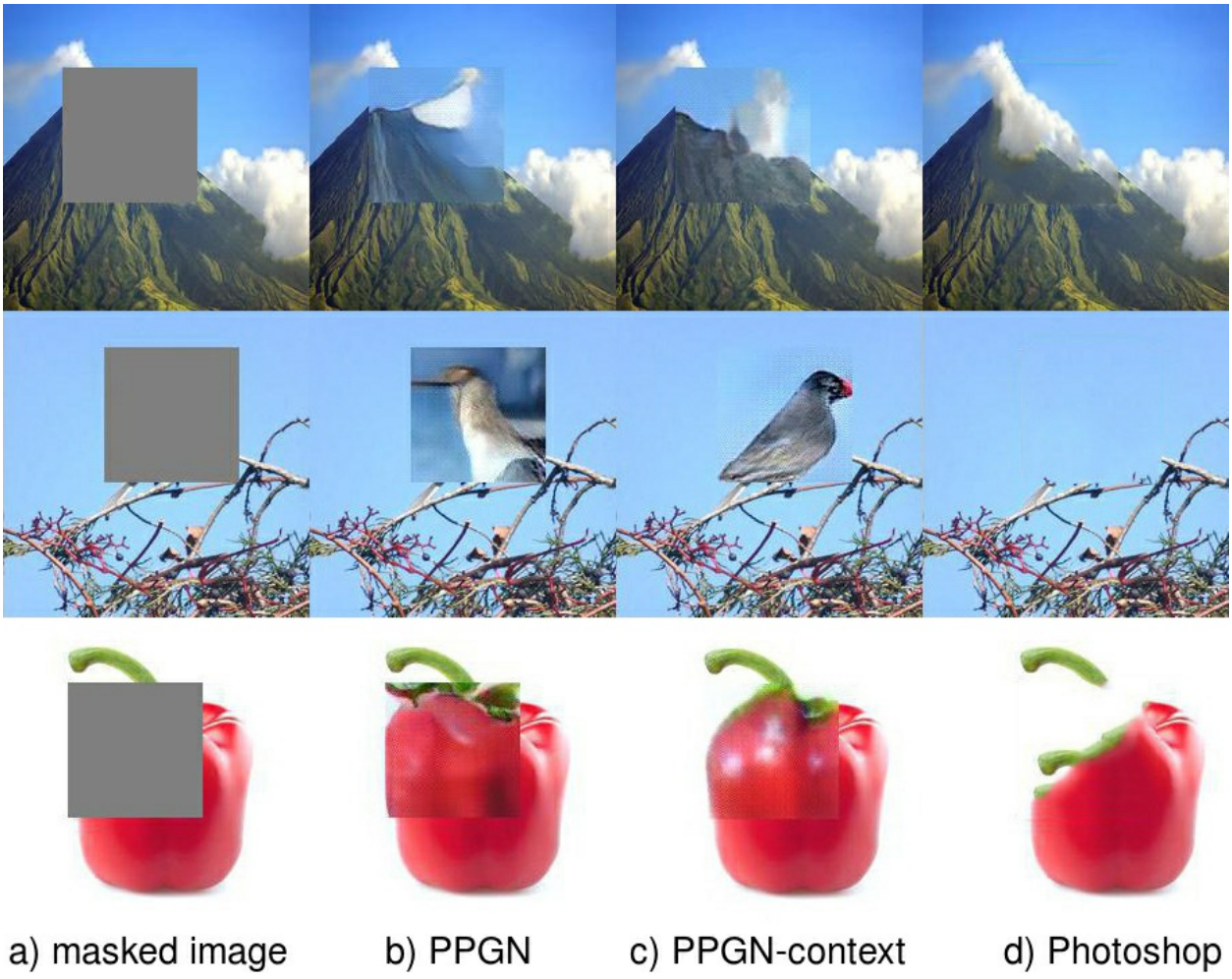
In the following screenshot, we depict a number of samples created with the **LSGAN architecture** on five scene datasets from LSUN, including a kitchen, church, dining room, and conference room:



LSGAN created models

Another really interesting example is class-conditional image sampling using the **Plug and Play Generative Network (PPGN)** to fill in 100 x 100 missing pixels in a real 227 x 227 image.

The following screenshot compares PPGN variations and the equivalent Photoshop image completion:

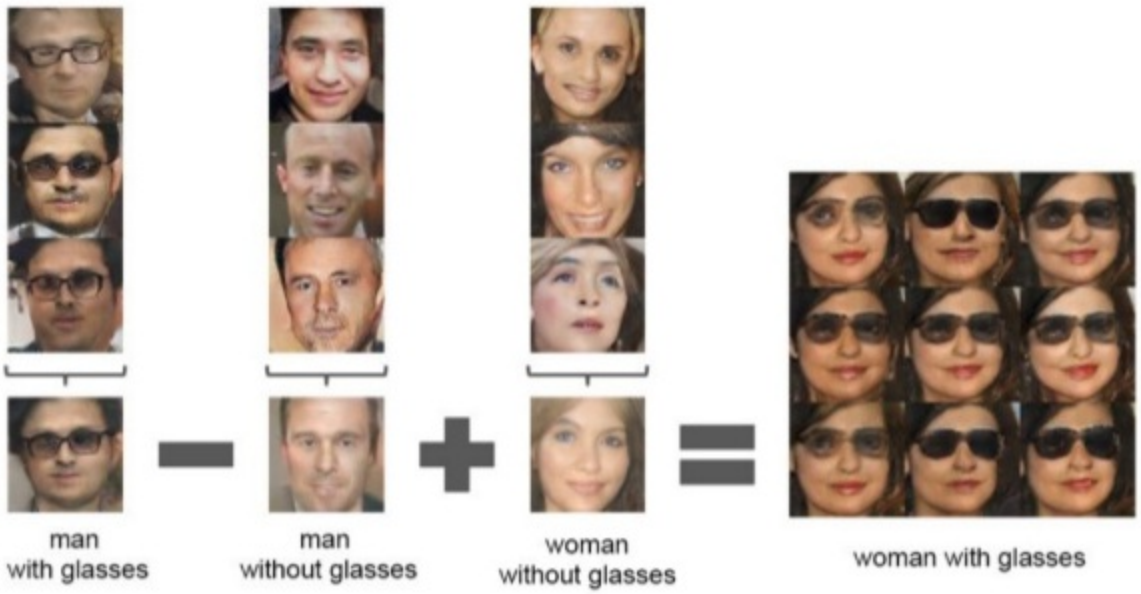


PPGN infilling example

The PPGN can also generate images synthetically at a high resolution (227 x 227) for the volcano class. Not only are many images nearly photo-realistic, but the samples within this class are diverse:

PPGN generated volcano samples

The following screenshot illustrates the process vector arithmetic for visual concepts. It allows the use of operands representing objects in the image and can add or remove them, moving in a feature space:



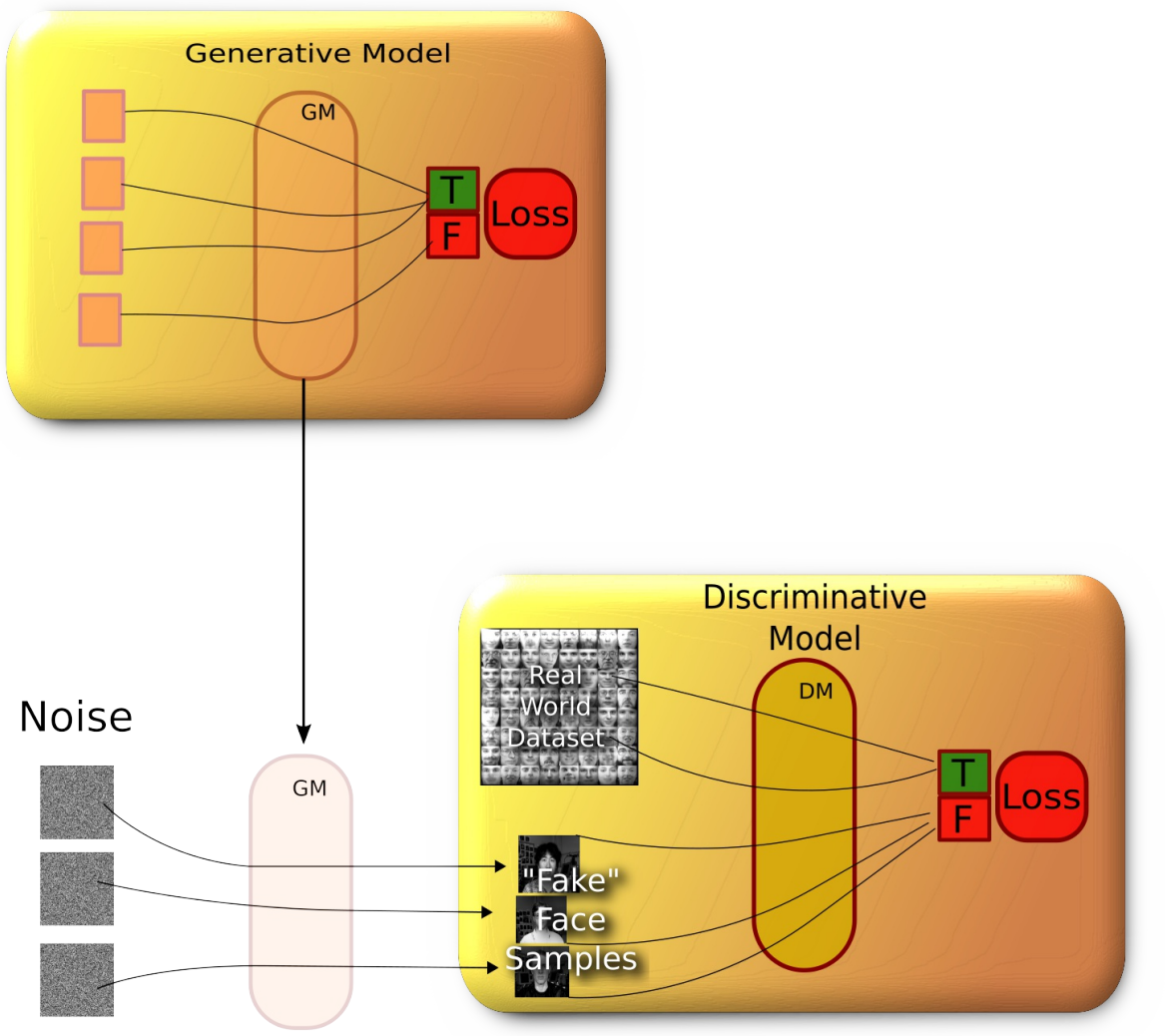
Vector arithmetic operations for the feature space

Discriminative and generative models

To understand the concept of adversarial networks, we will first define the two models that interact in a typical GAN setup:

- **Generator**: This is tasked with taking in a sample from a standard random distribution (for example, a sample from an n-dimensional Gaussian) and producing a point that looks sort of like it could come from the same distribution as *XX*. It could be said that the generator wants to fool the discriminator to output 1. Mathematically, it learns a function that maps the input data (*x*) to some desired output class label (*y*). In probabilistic terms, it learns the conditional distribution *P(y|x)* of the input data. It discriminates between two (or more) different classes of data—for example, a convolutional neural network that is trained to output 1 given an image of a human face and 0 otherwise.

- **Discriminator**: This is tasked with discriminating between samples from the true data and the artificial data generated by the generator. Each model will try to beat the other (the generator's objective is to fool the discriminator and the discriminator's objective is to not be fooled by the generator):



Training process for a GAN

More formally, the generator tries to learn the joint probability of the input data and labels simultaneously, that is, *P(x, y)*. So, it can be used for something else as well, such as creating likely new *(x, y)* samples. It doesn't know anything about classes of data. Instead, its purpose is to generate new data that fits the distribution of the training data.

In the general case, both the generator and the discriminator are neural networks and they are trained in an alternating manner. Each of their objectives can be expressed as a loss function that we can optimize via gradient descent.

The final result is that both models improve themselves in tandem; the generator produces better images, and the discriminator gets better at determining whether the generated sample is fake. In practice, the final result is a model that produces really good and realistic new samples (for example, random pictures of a natural environment).

Summarizing, the main takeaways from GANs are:

- GANs are generative models that use supervised learning to approximate an intractable cost function
- GANs can simulate many cost functions, including the one used for maximum likelihood
- Finding the Nash equilibrium in high-dimensional, continuous, non-convex games is an important open research problem
- GANs are a key ingredient of PPGNs, which are able to generate compelling high-resolution samples from diverse image classes

Reinforcement learning

Reinforcement learning is a field that has resurfaced recently, and it has become more popular in the fields of control, finding the solutions to games and situational problems, where a number of steps have to be implemented to solve a problem.

A formal definition of reinforcement learning is as follows:

> *"Reinforcement learning is the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment." (Kaelbling et al. 1996).*

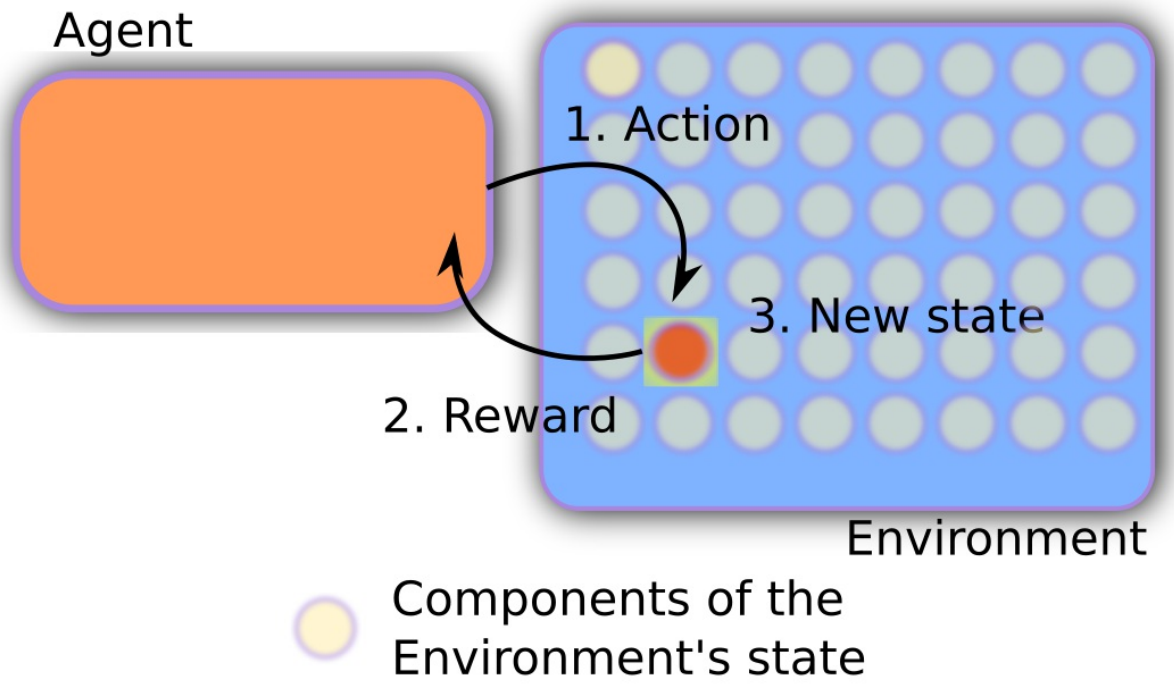
In order to have a reference frame for the type of problem we want to solve, we will start by going back to a mathematical concept developed in the 1950s, called the **Markov decision process**.

Markov decision process

Before explaining reinforcement learning techniques, we will explain the type of problem we will attack with them.

When talking about reinforcement learning, we want to optimize the problem of a Markov decision process. It consists of a mathematical model that aids decision making in situations where the outcomes are in part random, and in part under the control of an agent.

The main elements of this model are an **Agent**, an **Environment**, and a **State**, as shown in the following diagram:



Simplified scheme of a reinforcement learning process

The agent can perform certain actions (such as moving the paddle left or right). These actions can sometimes result in a reward r_t, which can be positive or negative (such as an increase or decrease in the score). Actions change the environment and can lead to a new state s_{t+1}, where the agent can perform another action a_{t+1}. The set of states, actions, and rewards, together with the rules for transitioning from one state to another, make up a Markov decision process.

Decision elements

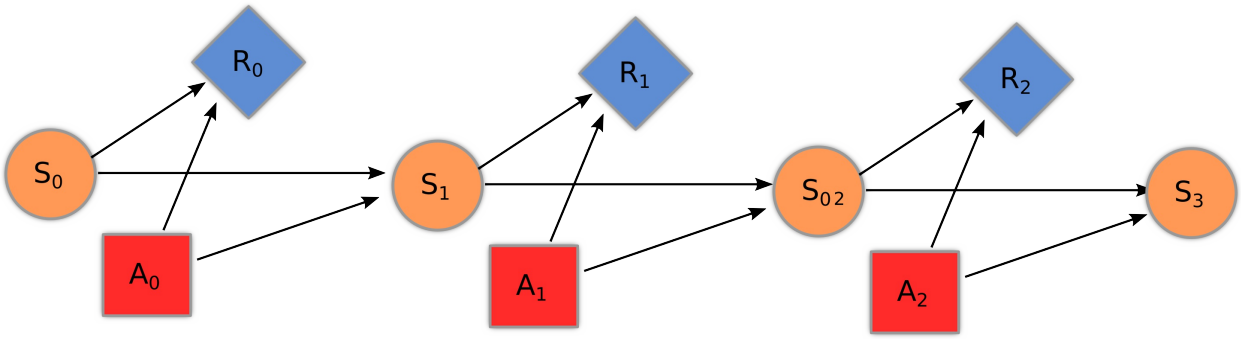
To understand the problem, let's situate ourselves in the problem solving environment and look at the main elements:

- The set of states
- The action to take is to go from one place to another
- The reward function is the value represented by the edge
- The policy is the way to complete the task
- A discount factor, which determines the importance of future rewards

The main difference with traditional forms of supervised and unsupervised learning is the time taken to calculate the reward, which in reinforcement learning is not instantaneous; it comes after a set of steps.

Thus, the next state depends on the current state and the decision maker's action, and the state is not dependent on all the previous states (it doesn't have memory), thus it complies with the Markov property.

Since this is a Markov decision process, the probability of state s_{t+1} depends only on the current state s_t and action a_t:



Unrolled reinforcement mechanism

The goal of the whole process is to generate a policy P, that maximizes rewards. The training samples are tuples, $<s, a, r>$.

Optimizing the Markov process

Reinforcement learning is an iterative interaction between an agent and the environment. The following occurs at each timestep:

- The process is in a state and the decision-maker may choose any action that is available in that state
- The process responds at the next timestep by randomly moving into a new state and giving the decision-maker a corresponding reward
- The probability that the process moves into its new state is influenced by the chosen action in the form of a state transition function

Basic RL techniques: Q-learning

One of the most well-known reinforcement learning techniques, and the one we will be implementing in our example, is **Q-learning**.

Q-learning can be used to find an optimal action for any given state in a finite Markov decision process. Q-learning tries to maximize the value of the Q-function that represents the maximum discounted future reward when we perform action *a* in state *s*.

Once we know the Q-function, the optimal action *a* in state *s* is the one with the highest Q-value. We can then define a policy $\pi(s)$, that gives us the optimal action in any state, expressed as follows:

$$\pi^* = \sum_{t \geq 0} \gamma^t r_t$$

We can define the Q-function for a transition point (s_t, a_t, r_t, s_{t+1}) in terms of the Q-function at the next point (s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2}), similar to what we did with the total discounted future reward. This equation is known as the **Bellman equation for Q-learning:**

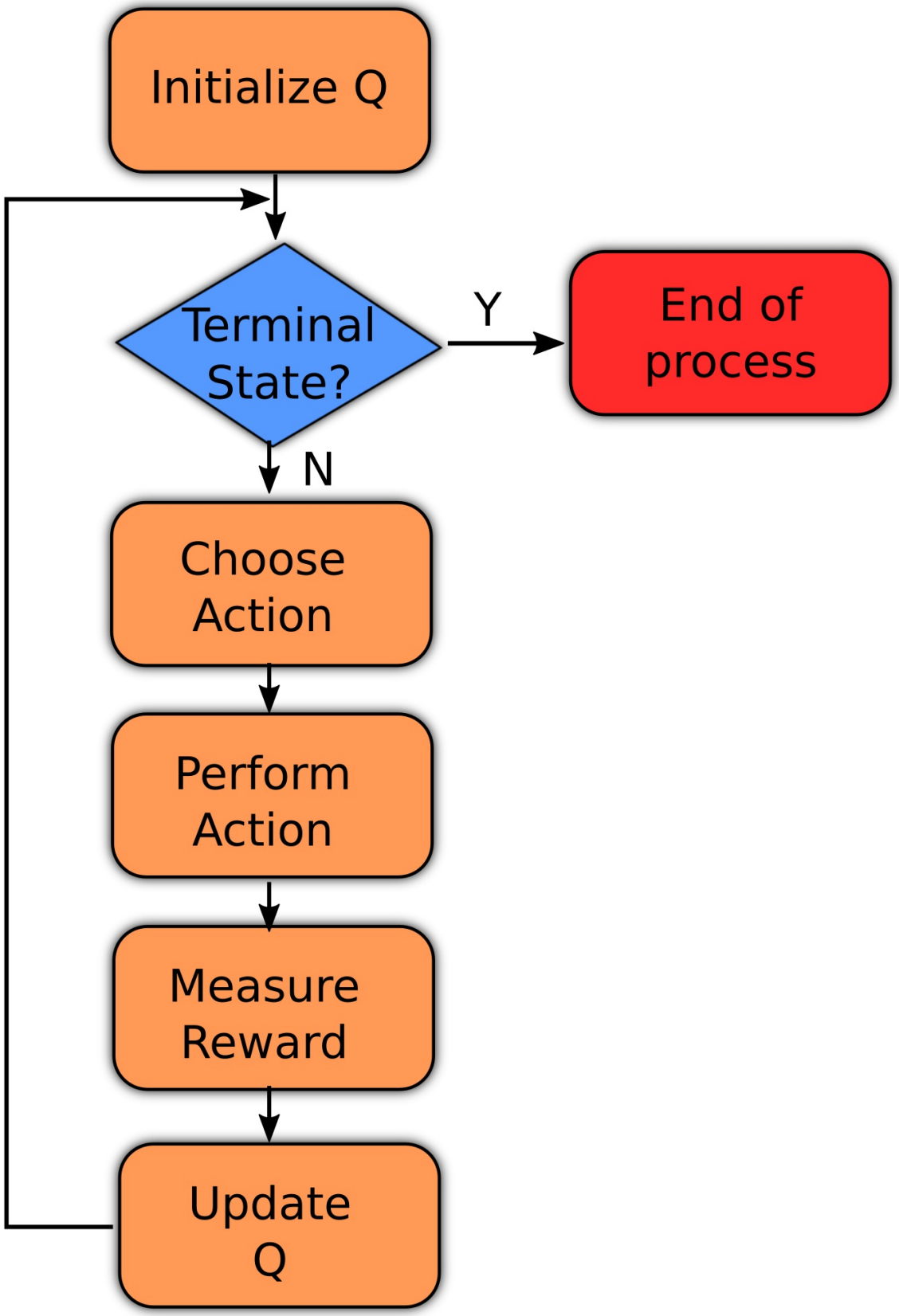
$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

In practice, we can think of the Q-function as a lookup table (called a **Q-table**) where the states (denoted by *s*) are rows and the actions (denoted by *a*) are columns, and the elements (denoted by *Q(s, a)*) are the rewards that you get if you are in the state given by the row and take the action given by the column. The best action to take at any state is the one with the highest reward:

```
initialize Q-table Q
observe initial state s
while (! game_finished):
   select and perform action a
   get reward r
   advance to state s'
   Q(s, a) = Q(s, a) + α(r + γ max_a' Q(s', a') - Q(s, a))
   s = s'
```

You will realize that the algorithm is basically doing stochastic gradient descent on the Bellman equation, backpropagating the reward through the state space (or episode) and averaging over many trials (or epochs). Here, α is the learning rate that determines how much of the difference between the previous Q-value and the discounted new maximum Q-value should be incorporated.

We can represent this process with the following flowchart:


Initialize Q
Terminal State?
Y
End of process
N
Choose Action
Perform Action
Measure Reward
Update Q

References

- Bellman, Richard, *A Markovian decision process.* Journal of Mathematics and Mechanics (1957): 679-684.
- Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore, *Reinforcement learning: A survey.* Journal of artificial intelligence research 4 (1996): 237-285.
- Goodfellow, Ian, et al., *Generative adversarial nets, advances in neural information processing systems*, 2014
- Radford, Alec, Luke Metz, and Soumith Chintala, *Unsupervised representation learning with deep convolutional generative adversarial networks.* arXiv preprint arXiv:1511.06434 (2015).
- Isola, Phillip, et al., *Image-to-image translation with conditional adversarial networks*, arXiv preprint arXiv:1611.07004 (2016).
- Mao, Xudong, et al., *Least squares generative adversarial networks.* arXiv preprint ArXiv:1611.04076 (2016).
- Eghbal-Zadeh, Hamid, and Gerhard Widmer, *Likelihood Estimation for Generative Adversarial Networks.* arXiv preprint arXiv:1707.07530 (2017).
- Nguyen, Anh, et al., *Plug & play generative networks: Conditional iterative generation of images in latent space*. arXiv preprint arXiv:1612.00005 (2016).

Summary

In this chapter, we have reviewed two of the most important and innovative architectures that have appeared in recent. Every day, new generative and reinforcement models are applied in innovative ways, whether to generate feasible new elements from a selection of previously known classes or even to win against professional players in strategy games.

In the next chapter, we will provide precise instructions so you can use and modify the code provided to better understand the different concepts you have acquired throughout the book.

Chapter 9. Software Installation and Configuration

So, it's time to begin our farewell! In this chapter, we will detail all the instructions for building our development environment.

In this chapter, you will learn about:

- Anaconda and pip environment installation on Linux
- Anaconda and easy install environment installation on macOS
- Anaconda environment installation on Windows

So, let's start with the step-by-step instructions, starting with Linux.

Linux installation

Linux is, in our view, the most flexible platform on which you can work for machine learning projects. As you probably know, there is a really large number of alternatives in the Linux realm, and they all have their own particular package management.

Given that writing instructions for working on all these distributions would take a large number of pages, we will settle on instructions for the **Ubuntu 16.04** distribution.

Ubuntu is undoubtedly the most widespread Linux distribution, and in the particular case of Ubuntu 16.04 it is an LTS version, or long term support. This means that the base software we will run in this book will have support until the year 2021.

> ### Note
>
> You will find more information about the meaning of LTS at https://wiki.ubuntu.com/LTS.

Regarding the feasibility of Ubuntu as a scientific computing distribution, even if it is considered by many to be newbie-oriented, it has the necessary support for all the technologies that a current machine learning environment requires, and has a very large user base.

> ### Note
>
> The instructions in this chapter are very similar to those for Debian-based distributions; they will work with minimal to no changes.

Initial distribution requirements

For the installation of the Python environment, you will need the following things:

- An AMD64 instruction-capable computer (commonly called a 64-bit processor)
- An AMD64-based image running on the cloud

> ### Note
>
> On AWS, a suitable **Amazon Machine Image** (**AMI**) is code ami-cf68e0d8.

Installing Anaconda on Linux

A very popular way of installing a Python distribution is via the software package Anaconda. It contains a complete high performance Python, Scala, and R ecosystem, including the most well-known packages used in data science. It also includes a host of other packages available through `conda`, which is the main utility of the package and manages the environment, packages, and

dependencies.

Note

Anaconda is built and distributed by **Continuum Analytics** (continuum.io), a company which also maintains the packages and their dependencies.

In order to install Anaconda, let's first download the installer package, version 4.2.0 at the time of writing.

Note

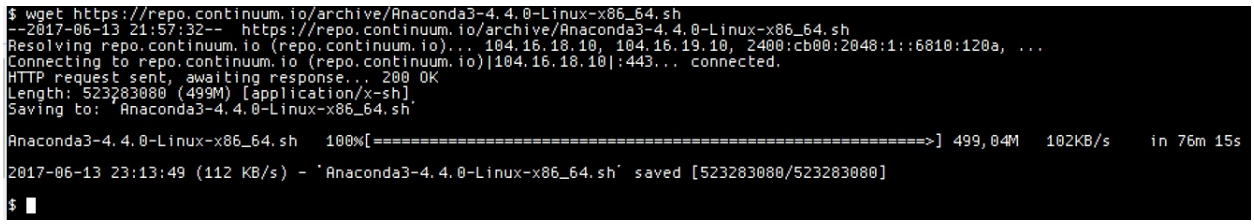
You can find the latest package version at https://www.anaconda.com/download/.

Take a look at the following steps to install Anaconda on Linux:

1. Let's run the following command:

```
curl -O https://repo.continuum.io/archive/Anaconda3-4.2.0-Linux-
x86_64.sh
```

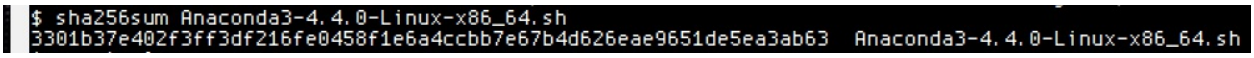
The preceding command will generate the following output:



2. Then, we need to verify the data integrity of the package, using a checksum or SHA-256 type. The Linux command to perform this operation is `sha256sum` and is shown as follows:

```
sha256sum Anaconda3-4.4.0-Linux-x86_64.sh
```

The preceding command will generate the following output:



3. Then, let's execute the installer using a `bash` interpreter as follows:

```
bash Anaconda3-4.2.0-Linux-x86_64.sh
```

The preceding command generates the following output:

```
$ bash Anaconda3-4.4.0-Linux-x86_64.sh

Welcome to Anaconda3 4.4.0 (by Continuum Analytics, Inc.)

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
==================================
Anaconda End User License Agreement
==================================

Copyright 2017, Continuum Analytics, Inc.
```

4. After pressing *Enter*, we will be presented with the license, which you can accept by typing `yes` after reading it, as shown in the following screenshot:

```
pyopenssl
A thin Python wrapper around (a subset of) the OpenSSL library.

kerberos (krb5, non-Windows platforms)
A network authentication protocol designed to provide strong authentication
for client/server applications by using secret-key cryptography.

cryptography
A Python library which exposes cryptographic recipes and primitives.

Do you approve the license terms? [yes|no]
>>> yes
Please answer 'yes' or 'no':
>>>
```

5. Then, it's time to choose the location and start the installation of all the packages, as shown in the following screenshot:

```
>>> yes

Anaconda3 will now be installed into this location:
/home/bonnin/anaconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/bonnin/anaconda3] >>>
PREFIX=/home/bonnin/anaconda3
installing: python-3.6.1-2 ...
installing: _license-1.1-py36_1 ...
installing: alabaster-0.7.10-py36_0 ...
installing: anaconda-client-1.6.3-py36_0 ...
```

6. After this, we will be asked to add the installed Anaconda installation to the path, mainly to have the libraries and binaries, especially the `conda` utility, integrated into the system. Then the installation is finished, as shown in the following screenshot:

```
Do you wish the installer to prepend the Anaconda3 install location
to PATH in your /home/bonnin/.bashrc ? [yes|no]
[no] >>> yes

Prepending PATH=/home/bonnin/anaconda3/bin to PATH in /home/bonnin/.bashrc
A backup will be made to: /home/bonnin/.bashrc-anaconda3.bak


For this change to become active, you have to open a new terminal.

Thank you for installing Anaconda3!

Share your notebooks and packages on Anaconda Cloud!
Sign up for free: https://anaconda.org
```

7. Let's then test the current Anaconda installation by running the following command:

```
source ~/.bashrc
conda list
```

The preceding command will generate the following output:

```
$ source ~/.bashrc
$ conda list
# packages in environment at /home/bonnin/anaconda3:
#
_license                  1.1                      py36_1
alabaster                 0.7.10                   py36_0
anaconda                  4.4.0                np112py36_0
anaconda-client           1.6.3                    py36_0
anaconda-navigator        1.6.2                    py36_0
anaconda-project          0.6.0                    py36_0
asn1crypto                0.22.0                   py36_0
astroid                   1.4.9                    py36_0
astropy                   1.3.2                np112py36_0
babel                     2.4.0                    py36_0
backports                 1.0                      py36_0
```

8. Now it's time to create a Python 3 environment by running the following command:

```
conda create --name ml_env python=3
```

The preceding command will generate the following output:

```
$ conda create --name ml_env python=3
Fetching package metadata .........
Solving package specifications: .

Package plan for installation in environment /home/bonnin/anaconda3/envs/ml_env:

The following NEW packages will be INSTALLED:

    openssl:    1.0.2l-0
    pip:        9.0.1-py36_1
    python:     3.6.1-2
    readline:   6.2-2
    setuptools: 27.2.0-py36_0
    sqlite:     3.13.0-0
    tk:         8.5.18-0
    wheel:      0.29.0-py36_0
    xz:         5.2.2-1
    zlib:       1.2.8-3

Proceed ([y]/n)?

openssl-1.0.2l 100% |################################################################
```

9. To activate this new environment, let's use the `source` command by running the following command:

```
source activate ml_env
```

The preceding command will generate the following output:

```
$ source activate ml_env
(ml_env) $
```

10. With your environment activated, your Command Prompt prefix will change as follows:

```
python --version
```

The preceding command will generate the following output:

```
$ source activate ml_env
(ml_env) $ python --version
Python 3.6.1 :: Continuum Analytics, Inc.
(ml_env) $
```

11. When you don't want to use the environment anymore, run the following command:

```
source deactivate
```

The preceding command will generate the following output:

```
(ml_env) $ source deactivate
$
```

12. If you want to inspect all the conda environments, you can use the following conda command:

```
conda info --envs
```

The preceding command will generate the following output:

```
$ conda info --envs
# conda environments:
#
ml_env                   /home/bonnin/anaconda3/envs/ml_env
root                  *  /home/bonnin/anaconda3
```

The asterisk (`*`) indicates the currently active environment.

13. Install additional packages by running the following command:

```
conda install --name ml_env numpy
```

The preceding command will generate the following output:

```
$ conda install --name ml_env numpy
Fetching package metadata .........
Solving package specifications: .

Package plan for installation in environment /home/bonnin/anaconda3/envs/ml_env:

The following NEW packages will be INSTALLED:

    mkl:   2017.0.1-0
    numpy: 1.13.0-py36_0

Proceed ([y]/n)?

mkl-2017.0.1-0  31% |############################
```

14. To delete an environment, you can use the following command:

```
conda remove --name ml_env --all
```

15. Add the remaining libraries:

```
conda install tensorflow
conda install -c conda-forge keras
```

pip Linux installation method

In this section, we will use the pip (`pip` installs packages) package manager to install all the required libraries for the project.

Pip is the default package manager for Python, and has a very high number of available libraries, including almost all the main machine learning frameworks.

Installing the Python 3 interpreter

Ubuntu 16.04 has Python 2.7 as its default interpreter. So our first step will be to install the Python 3 interpreter and the required libraries:

```
sudo apt-get install python3
```

Installing pip

In order to install the `pip` package manager, we will install the `python3-pip` package, using the

native `apt-get` package manager from Ubuntu:

```
sudo apt-get install python3-pip
```

Installing necessary libraries

Execute the following command to install the remaining necessary libraries. Many of them are needed for the practical examples in this book:

```
sudo pip3 install pandas
sudo pip3 install tensorflow
sudo pip3 install keras
sudo pip3 install h5py
sudo pip3 install seaborn
sudo pip3 install jupyter
```

macOS X environment installation

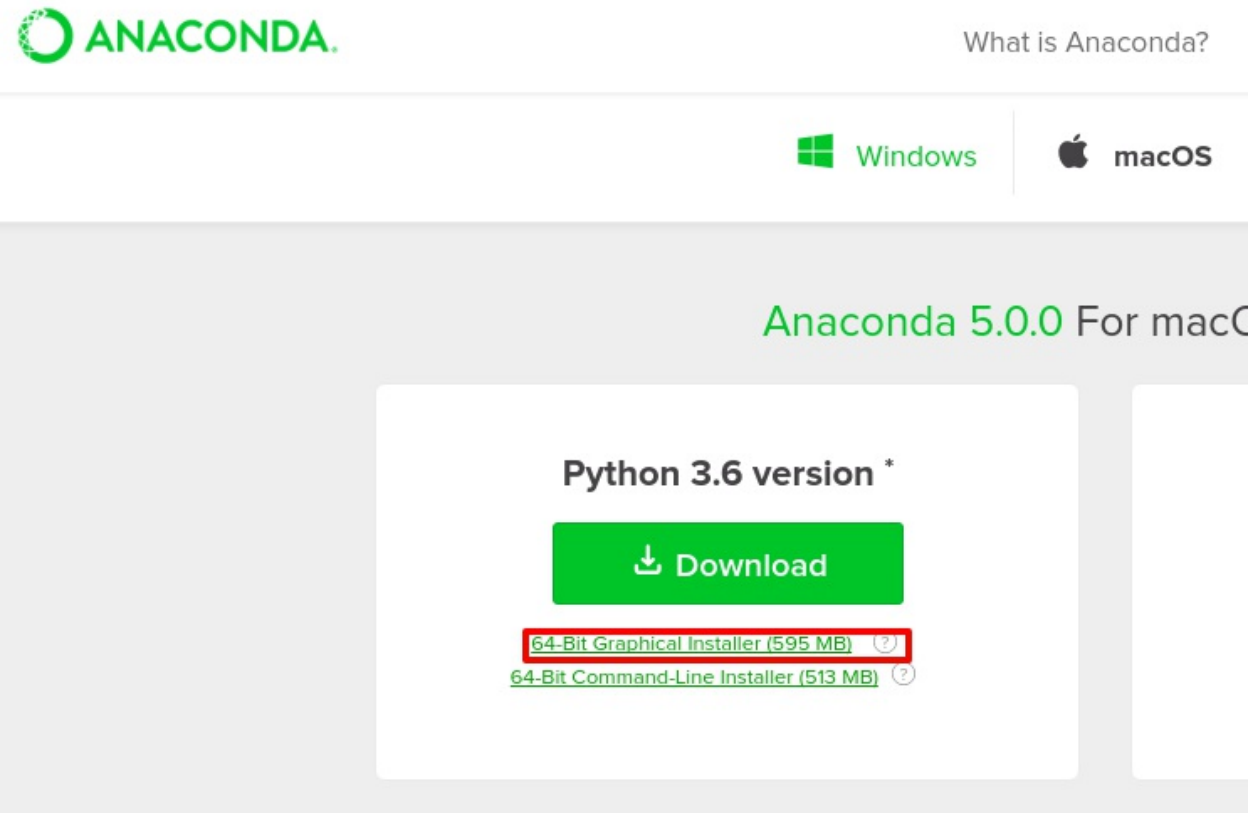
Now it's the turn of macOS X installation. The installation process is very similar to Linux, and is based on the **OS X High Sierra** edition.

Note

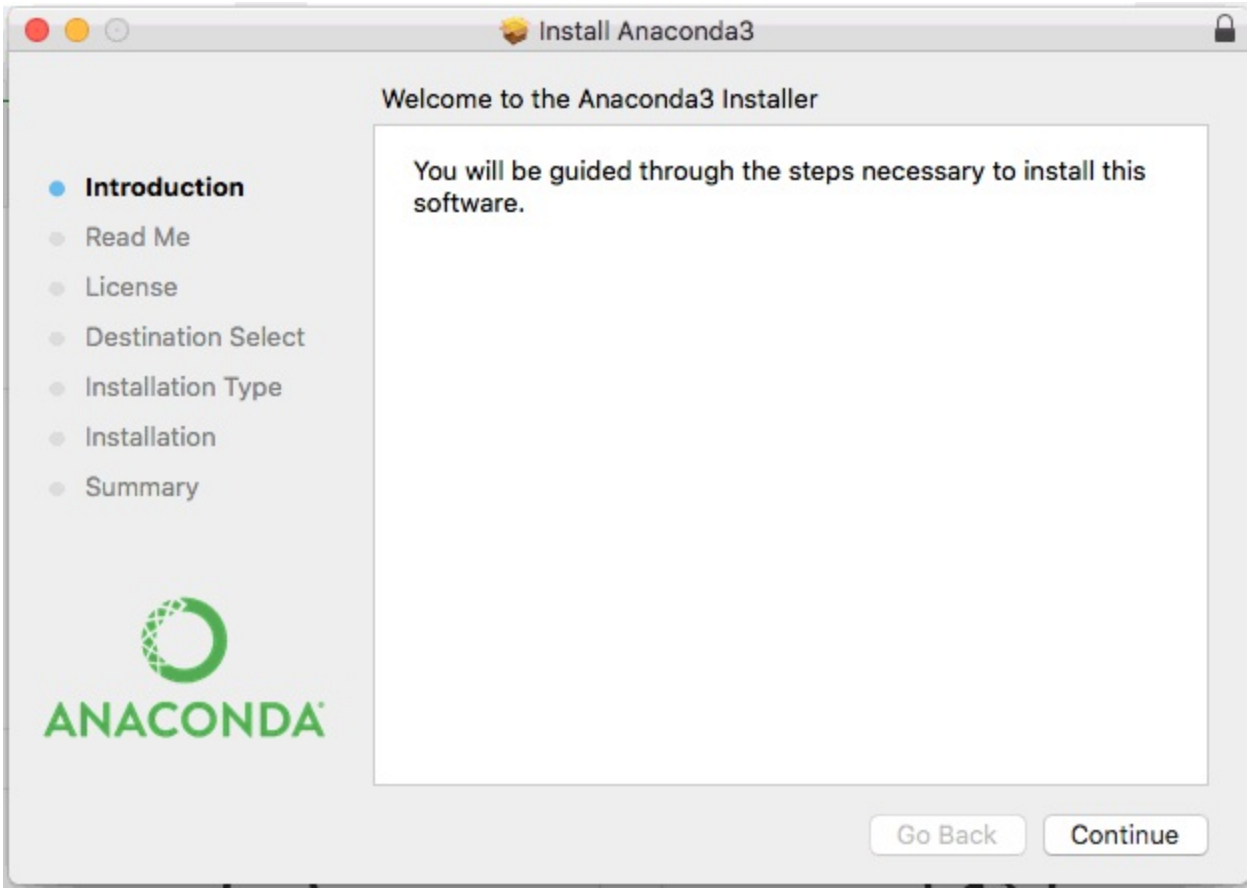
The installation requires `sudo` privileges for the installing user.

Anaconda installation

Anaconda can be installed via a graphical installer or a console-based one. In this section, we will cover the graphical installer. First, we will download the installer package from https://www.anaconda.com/download/ and choose the 64-bit package:

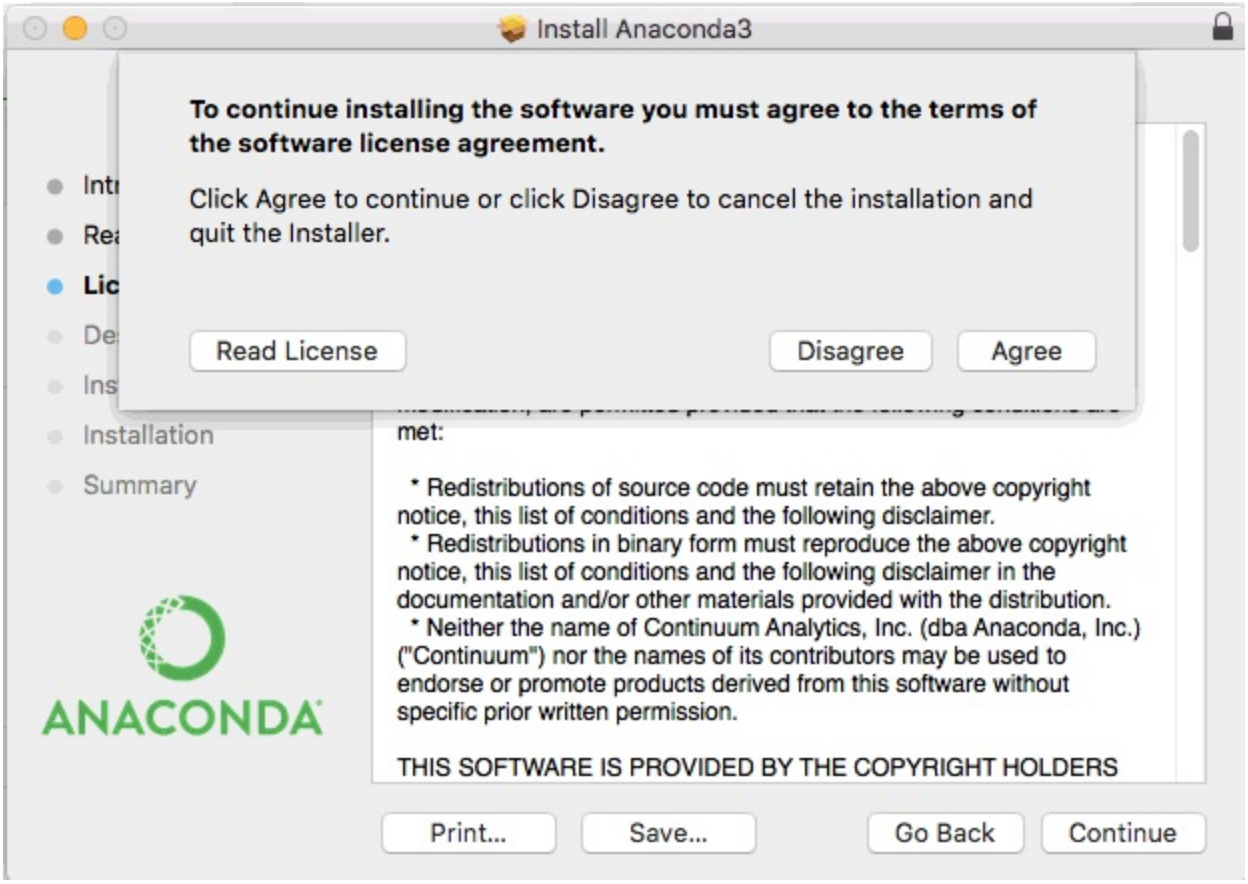


Once we have downloaded the installer package, we execute the installer, and we are presented with the step-by-step GUI:

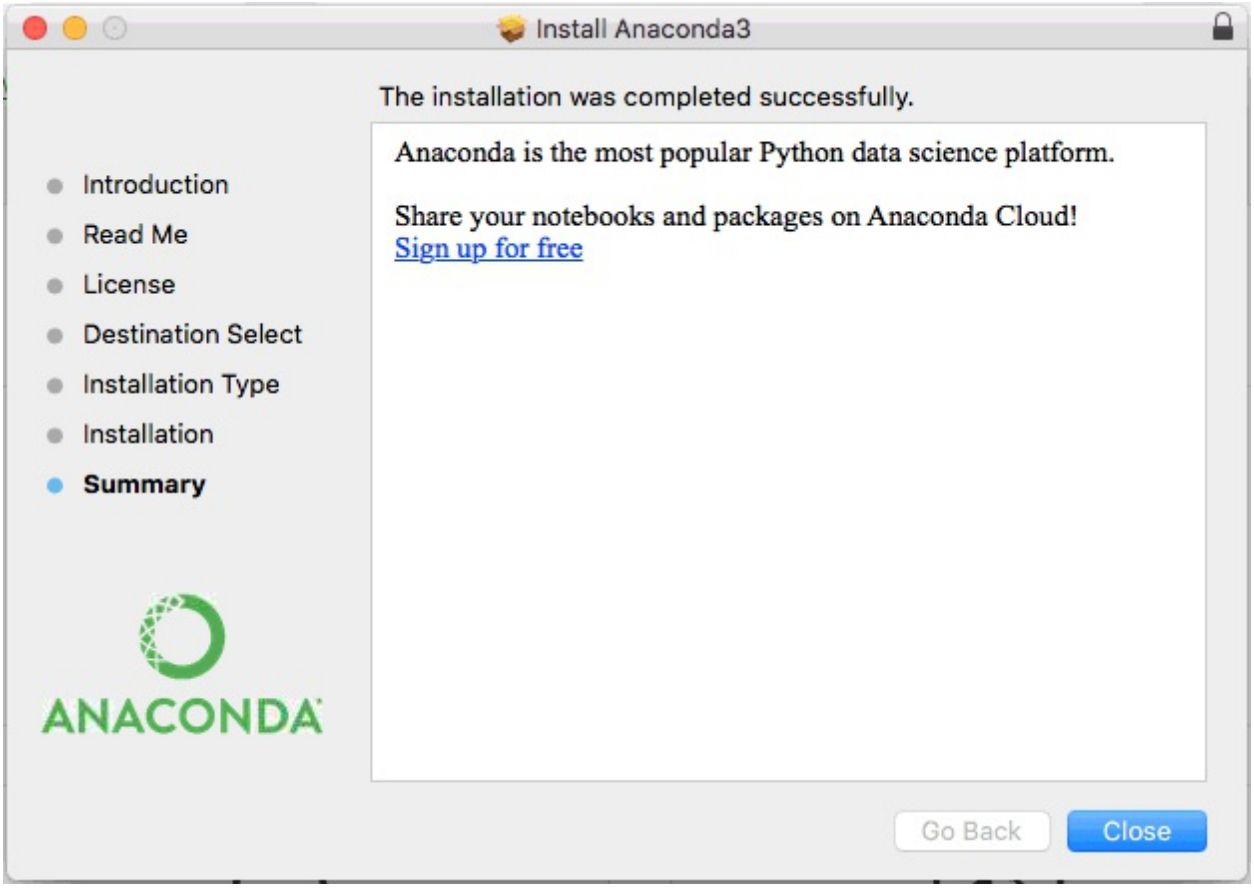


Then, we choose the installation location (take into account that the whole package needs almost 2 GB of disk to install):

Firstly, we accept the license, before actually installing all the required files:



After the necessary processes of file decompression and installing, we are ready to start using the Anaconda utilities:



A last step will be to install the missing packages from the Anaconda distribution, with the `conda` command:

```
conda install tensorflow
conda install -c conda-forge keras
```

Installing pip

In this section, we will install the `pip` package manager, using the `easy_install` package manager which is included in the `setuptools` Python package, and is included by default in the operating system.

For this process, we will execute the following command in a Terminal:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"

$ sudo brew install python3
```

Installing remaining libraries via pip

Then it is the turn to install all the remaining libraries:

```
sudo pip3 install pandas
sudo pip3 install tensorflow
sudo pip3 install keras
sudo pip3 install h5py
sudo pip3 install seaborn
sudo pip3 install jupyter
```

So this ends the installation process for Mac; let's move on to the Windows installation process.

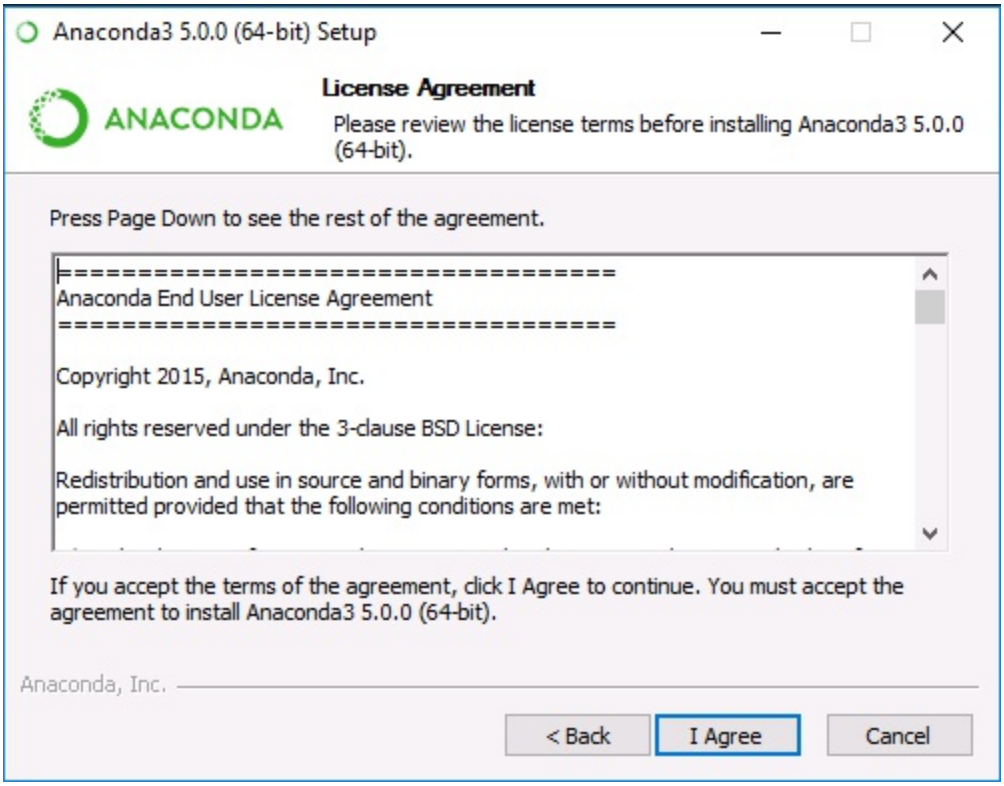
Windows installation

Windows is a platform on which Python can run without problems. In this section, we will cover the installation of Anaconda on the Windows platform.

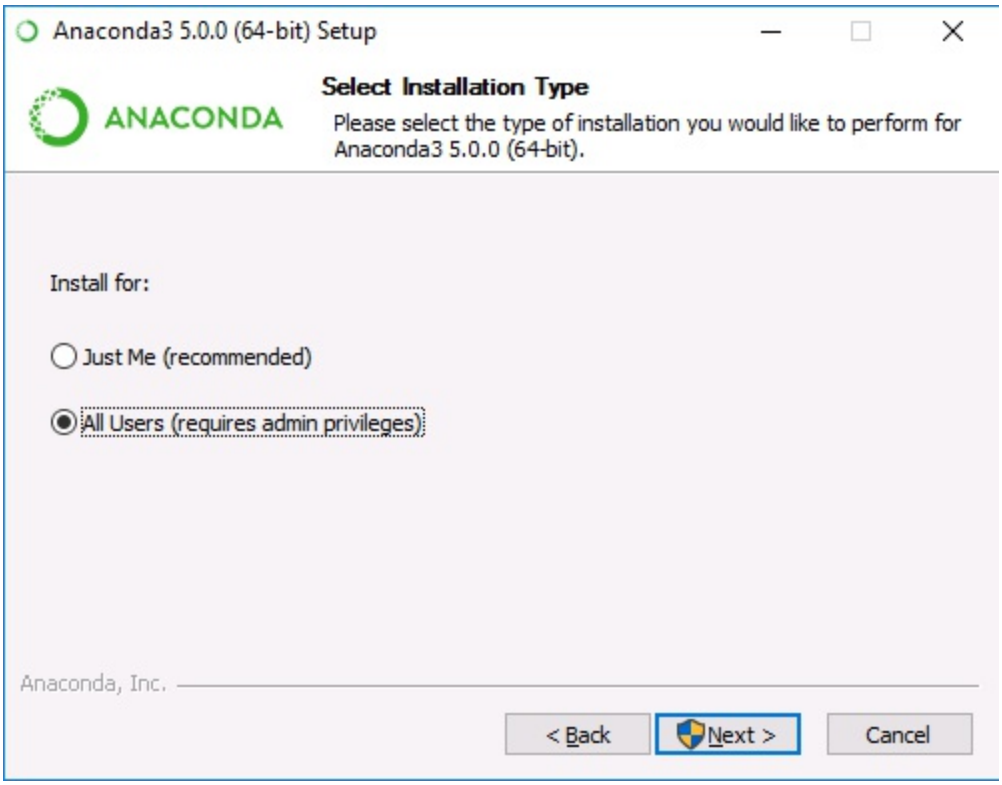
Anaconda Windows installation

The process to install Anaconda is pretty similar to the macOS one, because of the graphic installer. Let's start by downloading the installer package
from https://www.anaconda.com/download/ and choosing the 64-bit package:

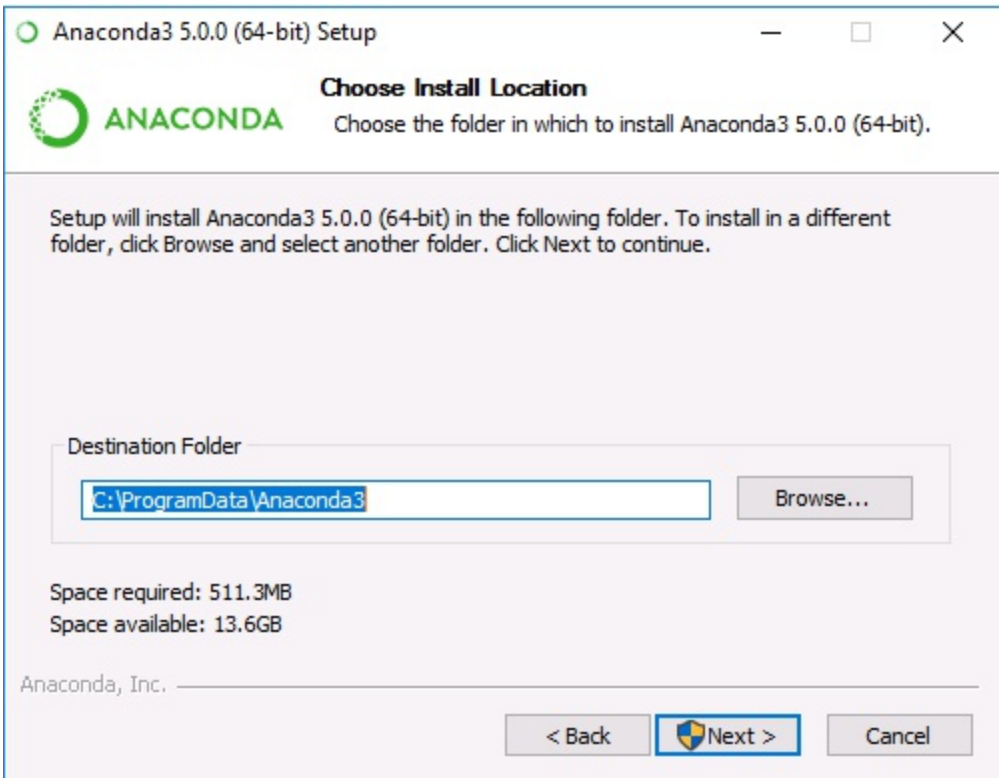
After downloading the installer, accept the license agreement, and go to the next step:



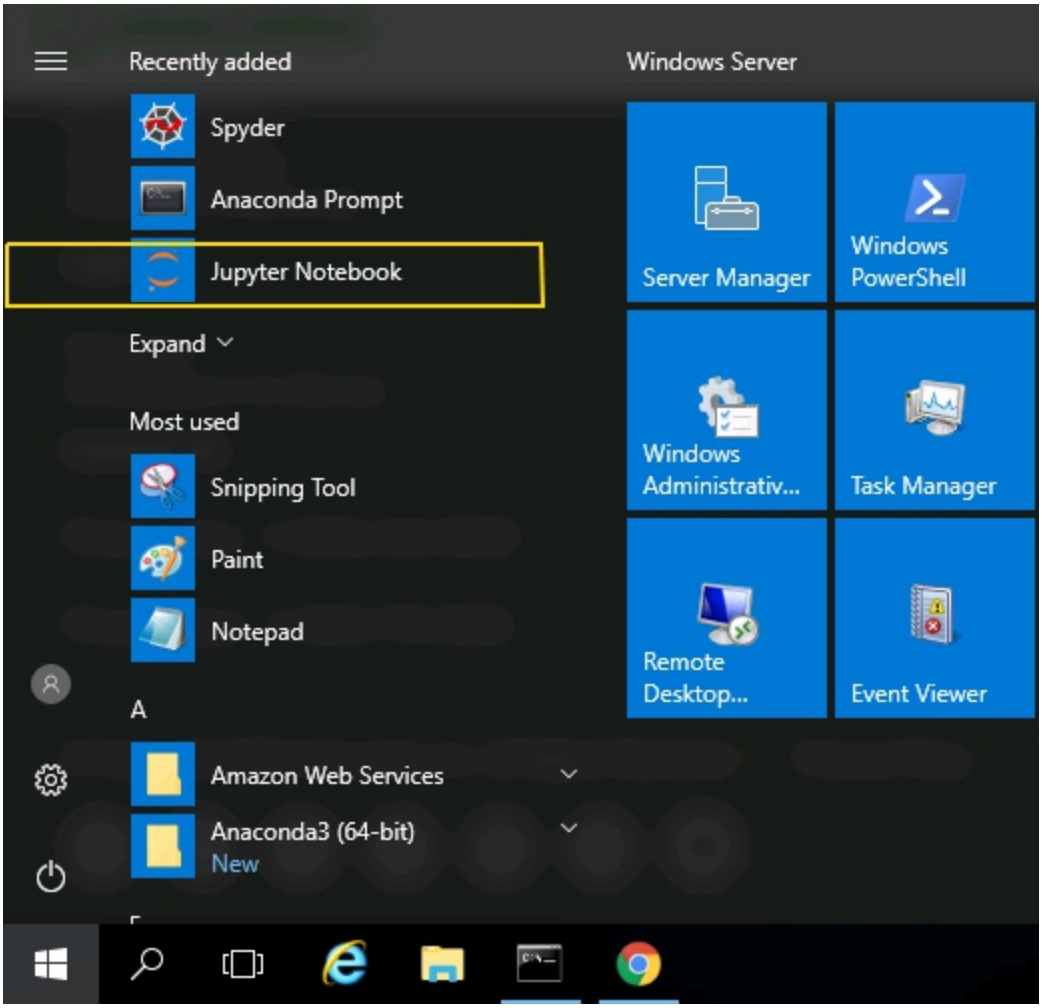
Then, you can choose to install the platform for your current user, or for all users:



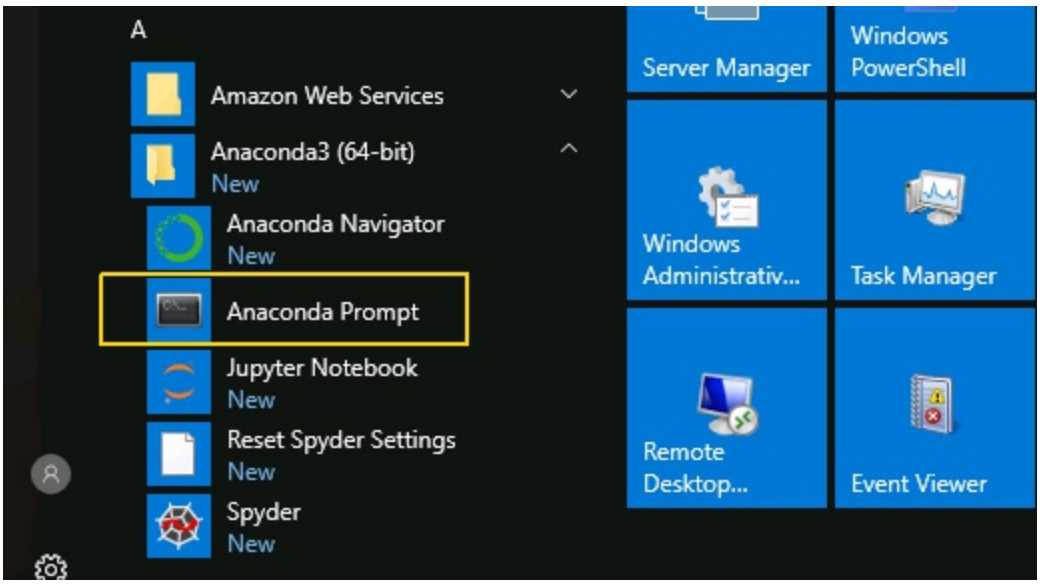
Then, you choose the installation directory for the whole installation. Remember, this will take close to 2 GB of disk to install:



After the environment is installed, you will find the Jupyter Notebook shortcut in the main Windows menu:



In order to use the Python commands and the `conda` utility, there is a convenient Anaconda prompt, which will load the required paths and environment variables:



The last step consists of executing the following `conda` commands from the Anaconda prompt to install the missing packages:

```
conda install tensorflow
conda install -c conda-forge keras
```

Summary

Congratulations! You have reached the end of this practical summary of the basic principles of machine learning. In this last chapter, we have covered a lot of ways to help you to build your machine learning computing environment.

We want to take the opportunity to sincerely thank you for your attentive reading, and we hope you have found the material presented interesting and engaging. Hopefully you are now ready to begin tackling new challenging problems, with the help of the tools we have presented, as well as new tools being developed all the time, and with the knowledge we have striven to provide.

For us, it has been a blast to write this book and search for the best ways to help you understand the concepts in a practical manner. Don't hesitate to write with questions, suggestions, or bug reports to the channels made available by the publisher.

Best regards, and happy learning!