bmc

LEARNING MACHINE LEARNING

A Hands-On Introduction to Math, Stats, and Machine Learning

Table of Contents

Introduction

The purpose of this book is to teach machine learning in the simplest way possible. Examples from online communities, like Stack Overflow, Beyond Data Science, and open source ML websites, tend to be too difficult to understand. The contributors often assume you already understand all the concepts they rely on. We make no such assumptions.

We wrote this book for managers, technical directors, programmers, product managers, and others who want to learn more about machine learning. Perhaps you've read something about neural networks, regression, TensorFlow, or classification, and now you want to know how to use those tools to solve problems in your own organization. Or, maybe you're looking to move into this field as a new career or to earn a higher salary.

We start with the very basics—basic stats and algebra—and then build upon that. This is because machine learning is applied mathematics. If you don't understand basic algebra, then you can never understand ML. Luckily, you don't need any concepts more complicated than that.

To that end, we have organized this book like this:

- **Basic descriptive statistics.** We review the normal curve, standard deviation, mean, and variance. These statistical concepts measure the accuracy of machine learning models. Plus, they're often the first step in looking at a data set—before writing code, before picking an ML algorithm.

- **Basic algebra and correlation.** This is the study of the relationship between data. You can build a predictive model only when data is positively correlated, so we'll cover how to determine whether the output is somehow related to the input. Otherwise you're just wasting your time.

- **Regression.** Once you've determined that there is some correlation between your input and output data sets, you look for a function that describes that relationship. In the simplest case, it's just y = mx + b. This is simple linear regression. It's important to have a solid understanding of this as most machine learning, even neural networks, are an expansion on the basic regression idea.

- **Classification.** This is a predictive model where the outcome is discrete rather than a real number. In other words, y = mx + b produces a floating point (real number). But your outcomes might be one of a set of numbers, like 1,2,3,4, or 5. An example of that would looking at handwritten digits and seeing if they are a 0, 1, 2, ..., or 9.

- **Clustering.** Divides data points into groups. For example, you might want to group people by some common characteristics, such as their medical condition based on blood pressure, blood sugar, etc. Or perhaps you want to divide cities based upon different quality of life issues.

- **Decision trees.** A formal way of making a decision, like whether to grant someone a loan based upon factors that would determine their ability and likelihood to repay the loan.

- **Neural networks.** A neural network can do classification or regression. The difference is it is designed to work at a very large scale. For example, when Facebook tags someone's photo with a name this is because you have given them enough examples where they can do that. Similarly, an app that can identify plants uses a neural network. It does this by taking a picture and reducing it to pixels and then making predictions based upon the arrangement of those pixels. Voice recognition works the same way.

We conclude this book with additional resources from BMC Blogs.

Requisite Skills & Software

To understand machine learning, you need to understand mathematics. Ideally, you've had some university exposure to algebra.

To get the most out of this book, you should know basic Python (or another programming language at the very least). All our programming examples are in Python because Python is the easiest programming language to learn, it's among the most widely used languages in business, and, most importantly, it's the most widely used language in machine learning.

Fortunately, advanced Python knowledge isn't necessary, because we explain every line of code.

To follow the tutorials in this book, you'll need to have certain software installed.

- Install **Python** version 3.x.
- Install **Apache Zeppelin**.
- Use **pip** to install these Python packages.
  - **TensorFlow**
  - **Keras**
  - **Seaborn**
  - **Matplotlib**
  - **NumPy**
  - **scikit-learn**

What is Machine Learning?

Machine learning is, essentially, applied mathematics. What makes ML different today from a decade ago is that only mathematicians and statisticians understood how to apply mathematics to data to build predictive and other models. Such persons tended to live in the realm of academia or work in industries such as pharmaceuticals. They tended to use the R-programming language, SaaS statistical software, or MATLAB.

That required a deep understanding of linear algebra, calculus, differential equations, etc.

Today, the mathematics and algorithms that power such thinking have been turned into APIs that regular programmers and anyone else can download. Now, it's no longer necessary to understand the calculus to find the minimum value of the stochastic gradient descent (SGD) algorithm, for instance. All the programmer needs to know is when and how to use this black-box function.

Of course, a programmer is more valuable if they understand the black box. That way, they can look at the output of a programming model and say with confidence whether it is producing accurate or misleading results. We call such people data scientists.

Types of Machine Learning

Machine learning models tend to fit into one of two categories:

Predictive Models

A predictive model uses a mathematical formula to predict a particular outcome given a particular set of inputs. In simple terms, it is the line equation $y = mx + b$. But in actual practice m is not a simple number. Instead m is usually written as W, or weights, which is a matrix of numbers, because the model considers more than one input.

Classification

Classification draws conclusions about something you are looking at right now. For example, you might have a set of data on engine vibration and temperature. Then you could group those into categories such as worn, broken, soon-to-be-broken, etc.

Problems Machine Learning Can Solve

Machine learning can solve problems that range from the day-to-day to the esoteric. Here are some examples:

The transportation problem. Think Federal Express and UPS. ML figures out how to make the required deliveries while driving the minimum kilometers and at least cost. This is called linear programming.

Regression analysis. This is the basis of most predictive models. Regression analysis finds the mathematical formula that models a given relationship. For example, you suppose that sales and the advertising budget are positively correlated. ML can verify, for instance, that every $1 increase in advertising might result in $25 in additional sales.

Cybersecurity and performance monitoring. These fields have a signal-to-noise problem, which makes it difficult to determine which events are statistically significant. ML ignores conventional thresholds that alert too frequently and instead only flag statistically significant events. ML reduces noise, flagging fewer, but more relevant, signals—so an analyst can take action on the right events and not waste time on the noisy ones.

Social media data mining. Facebook, Twitter, and most social media use obtuse data structures like directed graphs to model the relations between people. ML algorithms can mine this data.

Tools for Machine Learning

There are a variety of machine learning tools available. Below are the most common. In this book we use most, but not all, of these tools. We'll note when that's the case.

- **Python.** By far the most widely used tool for ML. Its growth is propelled forward by the wide variety of ML APIs and SDKs.

- **scikit-learn.** Despite being the easiest ML model to use, it's also one of the most powerful thanks to its breadth of algorithms. What we like best about scikit-learn, though, is you don't have to waste time putting data into a particular format. scikit-learn does that for you.

- **TensorFlow.** TensorFlow is a Google product. We find it too complicated for beginners, so we have not provided any examples in this book.

- **Keras.** Keras sits on tops of TensorFlow and it's much easier for neural networks, especially for beginners. We recommend starting with Keras instead of TensorFlow.

- **Matplotlib.** This is a sophisticated graphing (charting) tool that works with Pandas and NumPy.

- **Seaborn.** This sits on top of Matplotlib, adding statistical features of interest to the data scientist.

- **Pandas.** Pandas makes working with spreadsheet-like row and column data far easier than working with that as a regular array. For example, it allows column headings, does conversion from CSV to JSON format, supports filters, and lets you select subsets of the data without having to write much code.

- **NumPy.** A tool for working efficiently with very large matrices.

- **Spark ML.** This ML framework works with Apache Spark to handle data that is too large to fit into one machine. It's very simple to use. It can be used with Scala, Python, or Java, but it does not support Pandas, Matplotlib, or NumPy.

- **Jupyter and Zeppelin notebooks.** These "notebooks" are like web pages for Python code. They support big data databases and other languages, and they offer a graphical environment so you can draw graphs (charts).

The Basics: Descriptive Statistics

In order to understand machine learning, you need to understand basic descriptive statistics. Stats are usually the first step when trying to understand a set of data. Even after you select the algorithm, the **normal curve** is often used.

For example, the error in a machine learning model typically follows a **normal curve**. We describe how wide a curve is, and thus how much variance (error) is in our model, using the terms **mean**, **variance**, and **standard deviation**. This information helps us measure the average error in order to minimize it.

The normal curve

All statistics is mostly based on the idea of the **normal curve**, aka the bell curve.

Every student understands the normal curve concept even if they don't know what it's called. Grades on an exam generally hover about some average, known as the **mean** in statistics. The distribution of grades about the mean shows how widely the grades are spread. That variation is called **variance**.

The normal curve expresses this with the metrics **mean** and **standard deviation**. The standard deviation (denoted by the Greek letter δ, or delta) is the square root of the **variance**, v. The mean is given by the Greek letter **mu** μ.

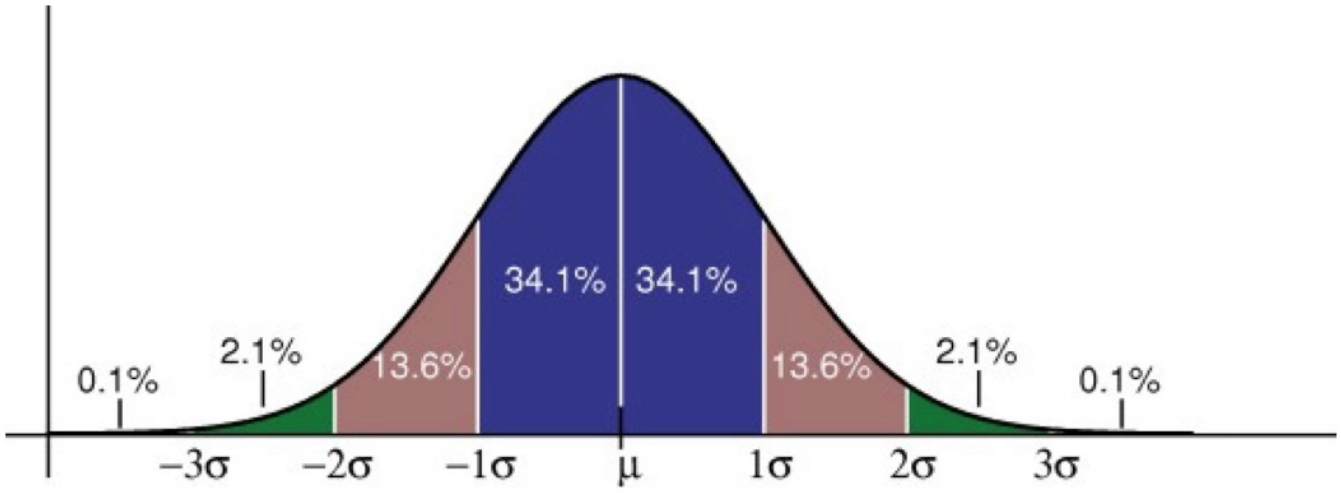
The normal curve normalizes the grades by putting them on a common scale. It does this by calculating the z-score. x is the variable. μ is the mean and δ is the standard deviation:

$$\frac{x-\mu}{\delta}$$

The area under the curve sums to 100%. The normal curve is a probability distribution.

If you look at the chart below, you see that the probability that a data point (grade) is within one standard deviation (μ +- δ) is 34.1% + 34.1 % = 68.2%. So, most observations are in the middle. The farther out you go, the less likely the event. For example, the probability that a student's grade is more than 2 standard deviations above the average is 2.1% + 0.1% + (an infinite string of increasingly tiny numbers) ≤2.2%.

in the normal curve the normalized values have a mean of 0 and variance of 1:



Source

Student Grades and the Normal Curve

Now we use a spreadsheet to illustrate this concept. We will use spreadsheets throughout this book as they are an easy way to show the math used by the algorithms. Using spreadsheets also reinforces the idea that most ML is based on a few simple concepts from algebra and statistics. Understand those concepts thoroughly and the rest becomes easier.

We plot student grades. We use the American system, where grades range from 0 to 100 and 60 is a passing grade. This worksheet is online **here** in the **normal** worksheet tab.

grade	z-score	cumulative prob	probability density
	=(A2-$A$13)/$A$15	=NORMDIST(A2,$A$13,$A$15,true)	=NORMDIST(A2,$A$13,$A$15,FALSE)
65	-2.11	0.01736	0.00472
75	-1.01	0.15605	0.02635
78	-0.68	0.24808	0.03484
80	-0.46	0.32263	0.03950
82	-0.24	0.40509	0.04267
87	0.31	0.62181	0.04186
87	0.31	0.62181	0.04186
90	0.64	0.73908	0.03577
90	0.64	0.73908	0.03577
92	0.86	0.80530	0.03032
100	1.74	0.95919	0.00964

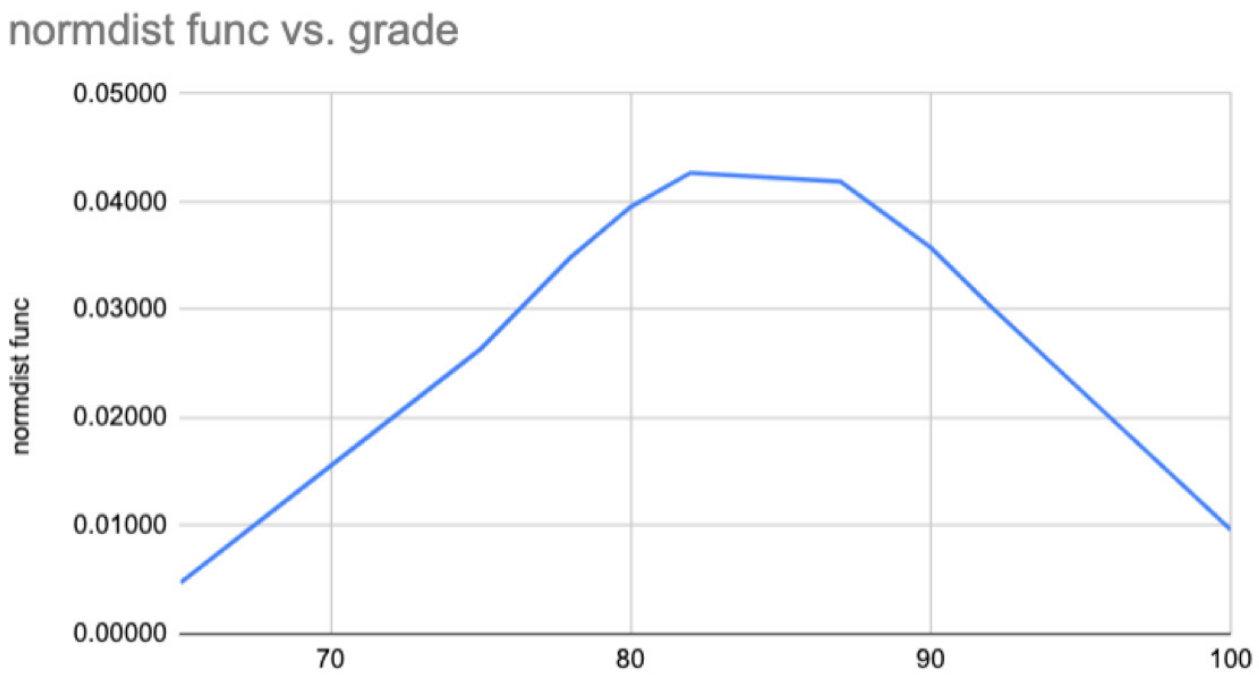
Here we calculate the average, variance, and standard distribution using the Google Sheets functions. The function names are almost the same with Microsoft Excel :

84.18181818	=AVERAGE(A2:A12)
82.51239669	=VAR.P(A2:A12)
9.083633452	=STDEV.P(A2:A12)

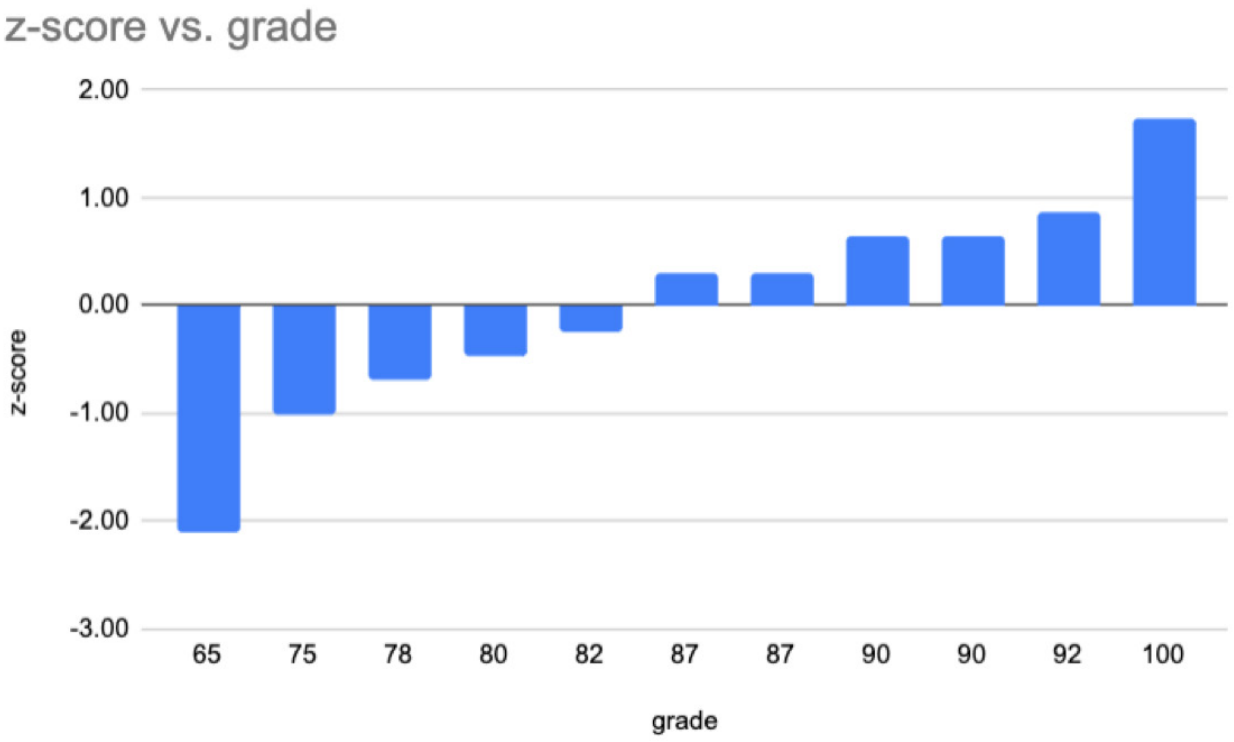
Now we plot the normal curve, which is the gaussian distribution function:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

As you can see, the grades look like a bell (normal curve), hovering about the middle, which is what we would expect. This is because, in general, students' grades hover around some midpoint.



Let's look at the **z-score**, which is the number of standard deviations away from the mean. It's easier to see the z score if we plot it as a bar chart. You can see that it's symmetrical. Symmetry is the whole point when using the normal distribution.



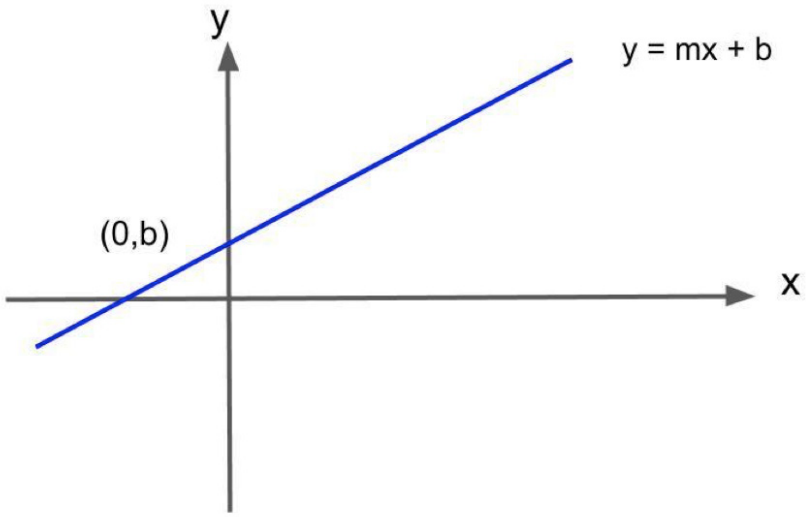
The Basics: Algebra & Correlation

In order to understand correlation, you need to understand the equation for a line. The line shows the relation between an independent (x) and dependent (y) variable. When you do machine learning, you know x and y. You are searching for the coefficients (m or sometimes called w) and the bias (b, in algebra called the y-intercept).

All machine learning is based on this simple formula:

$$f(x) = y = mx + b$$

where **m** is the **slope** of the line (aka **coefficient**) and b is the **y-intercept** (aka bias).



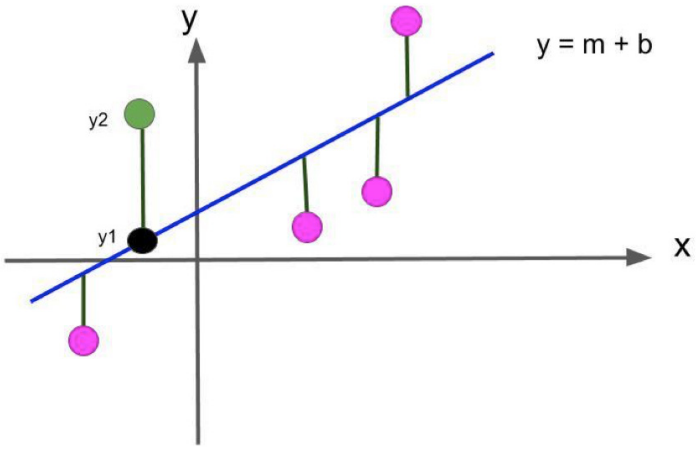
In machine learning the m is used replaced with a **W**, which is a matrix of coefficients, since there is more than one input. A matrix is an n-dimensional set of numbers, like this:

$$\begin{bmatrix} a, b, c \\ d, e, f \\ g, h, i \\ j, k, l \end{bmatrix}$$

Linear Regression

Most machine learning models are simply the linear regression problem. In most situations, the model will involve multiple independent variables. In the example below, we have just one independent variable, x. Solving that is so easy that you can simply use a spreadsheet. Generally, you have to write a program to solve when you have more than one independent variable.

Look at the chart below. At first glance, you might suspect that the pink dots are correlated—they generally fit along the line we have drawn in the middle.



But what is that blue line, slicing down the middle of the pink circles? That is the **linear regression line**, the predictive model, the line that most nearly plots the relation between x and y. We find this line by trial and error: We first put a line and then calculate the distance of the line from each of the points. Move the line again and then repeat the process.

For example, the distance from the green point to the black points is:

$$\sqrt{(y_1 - y_2)^2}$$

Statisticians call that distance the **error**. They say **least square error (LSE)** because you square each of these numbers so that positive distances don't cancel out negative ones. Then you pick another line and calculate LSE again. Stop when LSE has reached its minimum value.

Solving a linear regression problem with Google Sheets

We can illustrate linear regression with a simple example using Google Sheets. (It is located online **here** in the **linear** worksheet.) Google will use the LSE method to solve this problem. We invoke this using the **slope()** and **intercept()** functions.

Let's make up some data. Suppose we have these **x-y** values: 1, 2, 3, ... , 7. We make up some data that is roughly double that (i.e., 2*x) by multiplying each x times 2, then adding or subtracting a little to simulate some randomness.

Looking at this (and not knowing we made up this data), we might observe that the coefficient of this line is somewhere near 2.

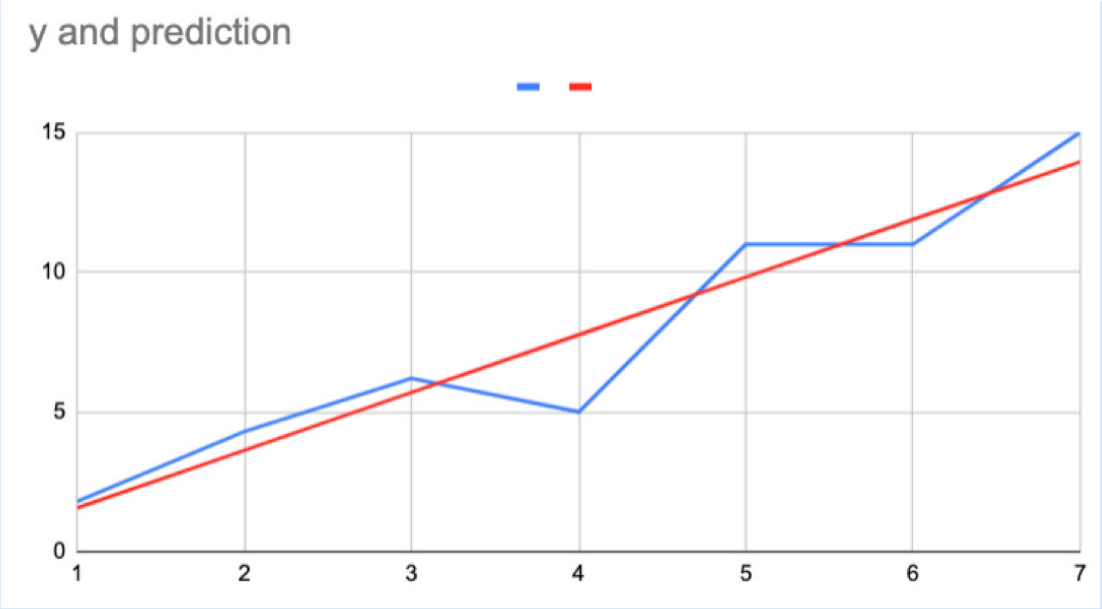
In y = mx + b, Google Sheets calculates the coefficient m and the y-intercept b. The y-intercept is the point (0,b), which is the point where the line crosses the y (vertical axis).

Then we show the values of the linear regression line by putting into the prediction column 2.06x - 0.5, which is **slope(b2:b8,a2:a8) * x + intercept(b2:b8,a2;a8)**

slope(B2:B8,A2:A8)	2.06
INTERCEPT(B2:B8,A2:A8)	-0.5

x	y	y=mx + b
		=($G$1*A3)+$G$3
1	1.8	1.6
2	4.3	3.6
3	6.2	5.7
4	5	7.8
5	11	9.8
6	11	11.9
7	15	14.0

Now we plot the x, y, and predicted values by adding a chart to the sheet. You can see that the orange line y = (2.06 * x) - 0.5 almost splits the values right down the middle.



NumPy

Unfortunately, before we go any further, we must introduce you to NumPy. (We say unfortunately because NumPy can be difficult to understand.) But, it's used everywhere, so you need to understand the basics.

NumPy is a Python API for constructing very large arrays of numbers with different degrees of precision, like 32-bit integers. Machine learning frameworks usually require that you put the data into a NumPy array.

To create a two-dimensional array in Python, you can simply write:

```
arr = [1, 2, 3]
```

But with machine learning, we are often working with higher order arrays, meaning arrays with 2 or more dimensions. Multiply these together (called the **dot-product**) takes a lot of memory.

NumPy was invented to make this multiplication more efficient, so your computer won't run out of memory. NumPy also handles complex array operations like slicing arrays (taking a subset along some

dimension) and different numeric types (small and large integers, floats, etc.). Because Python is not a typed language, there is no distinction between number types, integers versus floats, and maximum sizes, like 32-bit versus 64-bit numbers. But you waste memory if you give all numeric values the same size.

To illustrate how to work with NumPy, let's make a NumPy array:

```
import numpy as np
arr =  np.array([1,2,3,4,5])
print(arr)
arr.shape
```

The computer responds:

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

(5,) means that this is an array of 5 elements. It is of size (5,) which is more easily understood as (5,0)—meaning a single array with no extra dimensions.

This changes this 1-dimensional array of 5 elements into a 5-dimensional array of 5 arrays elements, each of which has 1 element.

```
arr.reshape(5,1)
```

Which looks like this:

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

Slicing arrays

Slicing is equally perplexing. Slicing is used to pull array elements along a dimension, like, for example, all the elements in the third column.

Let's create a new array:

```
arr =  np.array([[1,2,3,4,5], [2,4,6,8,10]])
print(arr)
```

When you print it, NumPy drops the commas, which is annoying. You would have to write code to put them back:

```
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]]
```

Let's take the three elements from the first axis. The index is the first element. The 0:3 means the number of elements to take. If we just put the colon (:) it would take all elements on the 0 axis.

```
arr[0,0:3]
```

Results in:

```
array([1, 2, 3])
```

Let's take the 3 elements from the second index:

```
arr[1,0:3]
```

Results in:

```
array([2, 4, 6])
```

All elements from the 1st index:

```
arr[0,:]
```

Results in:

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

All elements from the 2nd index:

```
arr[1:]
```

Results in:

```
array([[ 2,  4,  6,  8, 10]])
```

4th element from both indices:

```
arr[:,3]

array([4, 8])
```

Shaping arrays
The concept of shape is a bit complicated. You need to understand shape, as you often have to use **reshape()** as some ML algorithms might require.

Shape is roughly analogous to using the x to say, for example, you have a 2x3 or 3x3 matrix. Think of both of those as numbers arranged in a cube, whereas a 2x2 matrix has no third dimensions, so it's like the x,y coordinates on the x-y plane. Dimensions higher than three you cannot draw—only imagine.

You can reshape an array with equal number of elements into another array with equal number of elements. For example, you can reshape an 12x1 array into a 3x4. It still has the same number of elements, 12. You only put those into additional dimensions (axes). Conversely, you can flatten an x-y array into a 1-dimensional array of xy elements.

Like this: 12x1 [written as (12,)]:

3x4, written as (3,4):

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

For example, the below array has shape(3,4):

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

- 1x3 means 1 column and 3 rows
- 3x1 means 3 rows and 1 column

Results in a vector of shape 3x0:

```
arr=np.array([1,2,3])
print("arr=", arr)
print("arr shape", arr.shape)
```

Prints:

```
arr= [1 2 3]
arr shape (3,)
```

The -1 tells NumPy to figure out what to put there. For example, in our previous example of (12,) we did arr.reshape(3,4). But, since 3*4=12, we could have written arr.reshape(3,-1) and let NumPy figure out the 4.

```
b=arr.reshape(1,-1)
print("b=",b)

print("b shape", b.shape)
```

Prints:

```
b= [[1 2 3]]
b shape (1, 3)
```

Results in an array of 3x1:

```
c=arr.reshape(-1,1)
print("c=",c)
print("c shape", c.shape)
```

```
c= [[1]
 [2]
 [3]]
c shape (3, 1)
```

If that's all NumPy can do, you might wonder what's the big deal? But NumPy is packed with other functions useful to the data scientist, like the histogram.

Histograms

The **np.histogram()** function calculates the frequency distribution.

Below we have four bins: 0, 2, 4.1, and 6. There are seven entries less than 4.1 and 3 between that and 6.

```
import numpy as np

arr = np.array([2,2,2,2,4,4,4,6,6,6])
hist,bins = np.histogram(arr, bins=[0,2,4.1,6])
print("count=", hist, " bins=",bins)
```

Responds:

```
count= [0 7 3]  bins= [0.  2.  4.1 6. ]
```

Solving a linear regression problem with scikit-learn

We use scikit-learn because it is so easy (although, really, nothing is easy about data science) and has such a wide following among data scientists. The only complicated part of scikit-learn is understanding NumPy.

The code for this example is available **here**. Below the code below, we explain each section.

```
import matplotlib.pyplot as plt
from sklearn import linear_model
import numpy as np
from sklearn.metrics import mean_squared_error, r2_score

reg = linear_model.LinearRegression(fit_intercept=True)

data = np.array([[1, 2, 3, 4, 5, 6, 7],[1.8, 4.3, 6.2,  5, 11, 11,
15]])

x = data[0,:].reshape(7,1)
y = data[1,:]


reg.fit(x,y)

print('Coefficients: \n', reg.coef_)
print('Intercept: \n', reg.intercept_)
print('Score: \n' , reg.score(x, y))
print('Mean squared error: %.2f' % mean_squared_error(x, y))
print('Coefficient of determination: %.2f' % r2_score(x, y))
```

The code explained: Linear regression in scikit-learn

We first create the linear regression object:

```
reg = linear_model.LinearRegression(fit_intercept=True)
```

We use np.array(). Here we create(x,y).

```
data = np.array([[1, 2, 3, 4, 5, 6, 7],[1.8, 4.3, 6.2,  5, 11, 11,
15]]
```

Then we separate x and y into separate arrays using the NumPy slice functions [ ]:

- [0:] means take all the elements on the 0 axis.
- [1,:] means take all the elements on the 1 axis.

```
x = data[0,:].reshape(7,1)
y = data[1,:]
```

We reshape it to be an array of 1 dimensional arrays, a requirement of scikit-learn:

```
x = data[0,:].reshape(7,1)
```

Now we fit the curve to the linear model, meaning find the coefficient m and y-intercept b:

```
reg.fit(x,y)
```

Here are the results:

```
Coefficients:
 [2.06428571]
Intercept:
 -0.4999999999999991
Score:
 0.9111050507254281
Mean squared error: 20.31
Coefficient of determination: -4.08
```

So, your function is y = `2.06428571x – 0.4999999999999991`

Remember that ML uses the normal curve in expressing errors. Mean Squared Error (MSE) is used here.

The coefficient of determination (R^2) is the proportion of variance that can be explained by the independent variables. If it is 0, then the two are not in any way correlated.

Logistic Regression

Logistic regression and linear regression are very similar. The difference is that logistic regression outputs a **1 (true)** or **0 (false)**. The output of linear regression can be any number.

With **linear regression** we solve some formula:

$$y = mx + b$$

for the coefficient **m** and the y-intercept **b**.

For **logistic regression** we have a discrete and not a continuous outcome. We still have a function of the form:

$$y = mx + b$$

but we want to know what is the probability that y is some value, i.e., we want to solve $p(y)$.

Take a deep breath. The probability of y is called the **log odds**. It is the natural logarithm of the odds. If that sounds complicated, it's not. We will explain. Relax and absorb this. It's important that you understand this completely as you find linear and logistic regression throughout even the most advanced ML.

Gambling odds and logistic regression

When a bookmaker says that the odds of a horse winning a race are 9 to 1, that means there is a 10% chance of the horse winning. This is because **odds** are:

$$\frac{\text{Probability Success}}{\text{Probability Failure}}$$

For that horse to win, the odds are 9 to 1 meaning the chance the horse will win are:

$$\frac{0.1}{1 - 0.1}$$

$$= \frac{1}{9}$$

But bookmakers write the denominator ahead of the numerator:

9:1

And they pay $9 for every $1 bet on that horse if it wins.

With that little bit of gambling knowledge, we will show how to find the probability of a positive (true or 1) result given a logistic regression equation.

Finding the probability of a logistic regression equation given the odds

Suppose our linear regression model is y= mx + b. The log odds, then, is the natural logarithm of the odds.

$$ln(\frac{p}{1-p}) = mx + b$$

Let's simplify this. Raise both sides to the power **e** and we get:

$$(\frac{p}{1-p}) = e^{mx+b}$$

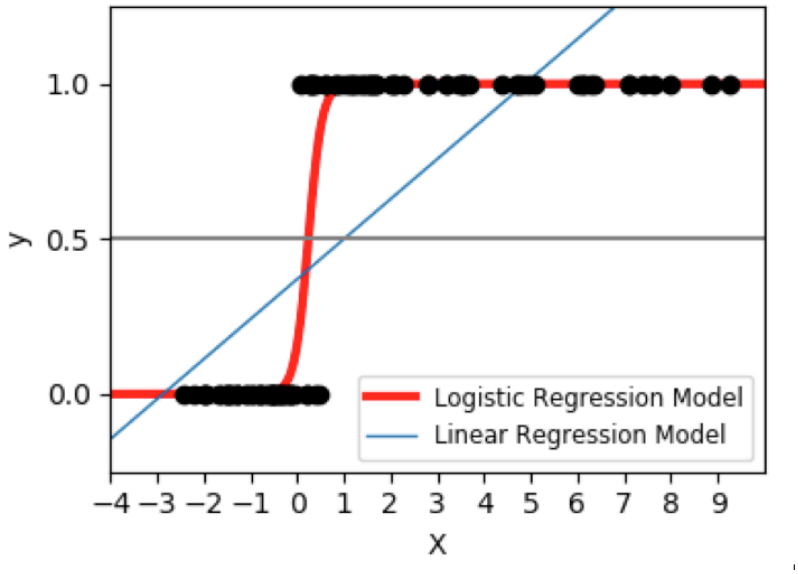
And simplify and we get P, the probability function.

$$p = \frac{e^{mx+b}}{1 - e^{mx+b}}$$

This is a number that is very useful for statistics because it ranges from 0 to 1. When p(y) is greater than 50% then we set the outcome of logistic regression to 1. Otherwise it is zero. That way we have a discrete true or false outcome.

$$p \geq 0.5 \Rightarrow 1$$

We use 50% because that is the point where the derivative of the logistic function starts to decrease. You can see that below in the plot of the logistic regression function. Below the black line, the rate of change (called the derivative in calculus) is increasing. Above the line it is decreasing. Another way of saying that is a tangent to the red line starts tilting up then starts tilting down until it goes horizonal.



Source

We said before that an increase of x to x+1 in the linear model results in an increase in:

$$\hat{y} = (m + 1)x + b$$

where $\hat{y}$ (pronounced y-hat) is the increase in a regression predictive model.

When the model is logistic regression that increase of m in:

$$ln(\frac{p}{1-p}) = mx + b$$

results in an increase in the log odds. What does that mean? It means an increase in the probability of a positive event.

Logistic regression using Google Sheets

Here we can illustrate this idea simply with Google Sheets, showing how to calculate the probability. As we said, P is the probability of the event (x):

$$p = \frac{e^{mx+b}}{1 - e^{mx+b}}$$

Suppose we have this data for x and y:

```
x = [.50, 0.75, 1.00,
1.25,1.50,1.75,1.75,2.00,2.25,2.50,2.75,3.00,3.25,3.50,4.00,4.25,4.50,4.75,5.00,5.50])

y =  ([0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,1,1,1,1,1])
```

As we show in the Python code below that creates this logistic regression model:

$$y = 0.61 * x - 1.36550178$$

cofficient (m)	**0.61**
intercept (b)	**-1.36550178**

The probability p for each value of x is:

$$p = \frac{e^{mx+b}}{1 - e^{mx+b}}$$

And if that value is greater than or equal to 50% then

$$p \geq 0.5 \Rightarrow 1$$

We write these formulae in Google Sheets:

column name	function
probability	=exp(B2)/(1+exp(B2))
mx + b	=($F$1*A2)+$F$2
log regression	=if(C2>=0.5,1,0)

And calculate the logistic regression and probability in the spreadsheet:

x	mx + b	probability	log regression
1	-0.8	0.32	0
2	-0.1	0.46	0
3	0.5	0.61	1
4	1.1	0.75	1
5	1.7	0.84	1
6	2.3	0.91	1
7	2.9	0.95	1
8	3.5	0.97	1
9	4.1	0.98	1
10	4.7	0.99	1
11	5.4	1.00	1
12	6.0	1.00	1

The code explained: Logistic regression

In Python code, this is:

```
import matplotlib.pyplot as plt
from sklearn import linear_model
import numpy as np

x = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
x1 = x.reshape(x.size,1)
y = np.array([0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

reg = linear_model.LogisticRegression(fit_intercept=True)
p=np.array([x1.size]).reshape(1,-1)
reg.fit(x1,y)


for i in range (0,x1.size):
    p=np.array(x[i]).reshape(1,-1)
    reg.predict(p)
    print("x[i]=", x[i] , " actual value=" , y[i] ," predicted value="
, reg.predict(p))
```

Echoes the actual value and predicted value.

```
x[i]= 1  actual value= 0  predicted value= [0]
x[i]= 2  actual value= 0  predicted value= [0]
x[i]= 3  actual value= 1  predicted value= [1]
x[i]= 4  actual value= 1  predicted value= [1]
x[i]= 5  actual value= 1  predicted value= [1]
x[i]= 6  actual value= 1  predicted value= [1]
x[i]= 7  actual value= 1  predicted value= [1]
x[i]= 8  actual value= 1  predicted value= [1]
x[i]= 9  actual value= 1  predicted value= [1]
x[i]= 10  actual value= 1  predicted value= [1]
x[i]= 11  actual value= 1  predicted value= [1]
x[i]= 12  actual value= 1  predicted value= [1]
```

Breaking this down, first import scikit-learn and NumPy. (Matplotlib is for drawing charts with NumPy arrays, which we don't use yet.)

```
import matplotlib.pyplot as plt
from sklearn import linear_model
import numpy as np
```

Now make a NumPy array of the values 1 through 12:

```
x = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
```

Give the vector (12,) a dimension of 12x1:

```
x1 = x.reshape(x.size,1)
```

Now provide the output of logistic regression. Scikit-learn will find the coefficient and y intercept:

```
y = np.array([0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Now create the logistic regression **model. fit_intercept=True** means set the y-intercept to some value besides 0. After all, there are an infinite number of lines parallel to the regression line so you could put one of them down to the point where it crosses (0,x).

```
reg = linear_model.LogisticRegression(fit_intercept=False)
```

Here we run the fit() function to train the model meaning, calculate the coefficients and the y-intercept.

```
reg.fit(x1,y)
```

Loop through each value in the x array and use the trained model to predict y.

```
for i in range (0,x1.size):
    p=np.array(x[i]).reshape(1,-1)
    reg.predict(p)
    print("x[i]=", x[i] , " actual value=" , y[i] ," predicted value="
, reg.predict(p))
```

Solver algorithm

The above example uses one independent variable and only a handful of data points. If we use the approach we mentioned above, which is to guess the linear line and the calculate the LSE (least square error) and move the line again, that would take all day for a model with hundreds of variables and thousands of data points. So, we use a solver algorithm, which does this in the most efficient and fastest way.

The default algorithm used to solve the logistic regression model is **Broyden–Fletcher–Goldfarb–Shanno (BFGS)**. It uses calculus and like the others loops over the result until it finds the minimum error. Of course, there are many more algorithms.

Pandas

Now we introduce another Python structure. Pandas Dataframe is widely used in ML. Think of Pandas as a NumPy array with two notable differences:

1. It has an index.
2. Columns and, optionally, the index have names.

And because it is an extension of NumPy and Matplotlib, Pandas also:

- Supports NumPy operations, like slicing
- Draws charts

Look at the code below. This creates a Pandas dataframe from a NumPy array of size (3,). Notice that we assign the column name **a** to the first column. So, you can refer to columns by name rather than array position, which is the rule with NumPy. That's obviously easier and useful.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.array([1,2,3]),columns=['a'])
```

This creates a structure like this, where 0, 1, and 2 are the row indexes.

```
   a
0  1
1  2
2  3
```

If we pass it a dictionary it takes the names from the labels (keys):

```
dict = {"a": [1,2,3], "b": [3,4,5], "c": [7,8,9]}
df = pd.DataFrame(dict)
df
```

Results in:

```
      a     b     c
0     1     3     7
1     2     4     8
2     3     5     9
```

We can also give it an explicit index instead of letting it choose its own:

```
dict = {"a": [1,2,3], "b": [3,4,5], "c": [7,8,9]}
df = pd.DataFrame(dict,index=['Mike','Sam','Sally'])
df
```

Results in:

```
      a     b     c
Mike  1     3     7
Sam   2     4     8
Sally 3     5     9
```

This slice makes a Panda **series**, which is a dataframe with one column:

```
df['b']
```

Results in:

```
Mike     3
Sam      4
Sally    5
Name: b, dtype: int64
```

This gives us the column names:

```
df.columns
```

Results in:

```
Index(['a', 'b', 'c'], dtype='object')
```

Now you should have enough understanding of Pandas to understand the following code.

Logistic regression with Python Pandas

The data we are looking at is glucose, body mass index (BMI), etc., taken from two sets of people: those who are diabetic and those who are not. The classification is 1 (diabetic) and 0 (not) is in the **Outcome** column. The goal is to use that data to train a predictive model that will show given certain health indicators whether a person is likely to have or will get diabetes.

We use **this data set** from Kaggle which tracks diabetes in Pima Native Americans. The code is **here**.

The code explained: Logistic regression with Pandas

The first step is to read the data into a dataframe.

```
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

data = pd.read_csv('/home/ubuntu/Downloads/diabetes.csv',
delimiter=',')
```

Then, take a look at it.

```
data.head()
```

Predictive models require that you split the data into **features** (inputs) and **labels** (outputs):

- **Features** are the characteristics of what you are looking at, aka independent variables.
- **Labels** are what you are trying to predict, aka dependent variables.

Classification means there are a finite set of outcomes. Here there are two, so you could call it a **binary classification problem**.

As you can see, the **outcome**—whether someone has diabetes—is the last column. The rest are features. It will be easy to split this data since the labels are on the end.

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI
0	6	148	72	35	0	33.6
1	1	85	66	29	0	26.6
2	8	183	64	0	0	23.3
3	1	89	66	23	94	28.1
4	0	137	40	35	168	43.1

	DiabetesPedigreeFunction	Age	Outcome
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

- The Pandas **drop()** command creates a new dataframe by taking an existing dataframe and dropping one or more columns.
- **axis=1** means we are referring to the columns and not the rows (which, for Pandas, is aka the **index**).
- **Note**: We make the output x as dataframe operations usually result in the creation of a new dataframe. We could have used the argument **inPlace=True**, but that does not work with all Pandas operations.

```
x = data.drop("Outcome", axis=1)
```

data['Outcome'] is a Pandas **series** and not a Pandas dataframe. This means it has one column only, but it still has the index column. **np.ravel()** will flatten that to a vector. In other words, if we did not use np.ravel() it would look like this, with the first column is the index and the second column is the value:

```
0       1
1       0
2       1
3       0
4       1
5       0
```

```
y=np.ravel(data['Outcome'])
```

Now the index is gone, so it's easier to work with:

```
array([1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0,
       1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1,
       0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
       1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0,
```

Moving on, the standard procedure is to take the input data and create training and test datasets by splitting them by some amount. Here we pick 50%.

- **Training datasets** are used to train the model.
- **Test datasets** are used to make predictions based on that trained model. So, they test the accuracy of the model.

We use x and y since the familiar equation for a line is y = mx + b. For machine learning y is a vector and m and x are matrices, meaning n-dimensional. b is bias, which is a single real number. Think of it like the y-intercept: it's just a constant to shift the rule up or down or whatever direction.

```
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size =
0.5, random_state=50)
```

Now we normalize the data. This is standard machine learning practice. Basically, this calculates the value $((x - \mu) / \delta)$ where μ is the mean and δ is the standard deviation. This puts all the features on the same scale. In other words, it makes large numbers small so that all the numbers are about the same size.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(x_train)

x_train = scaler.transform(x_train)

x_test = scaler.transform(x_test)
```

First, we declare the model. We are using a support vector machine. This is an efficient way to divide data into two categories, so it's useful for binary regression or classification models.

```
from sklearn.svm import SVC
svc_model = SVC()
```

Then we train it. It's that simple when you use scikit-learn: there's no other data manipulation required.

```
svc_model.fit(x_train, y_train)
```

The **fit()** function responds with this information:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

Now we use the test data to create predictions by feeding into it the x_test values. Those are the values we took from our 50-50 splits.

```
y_predict = svc_model.predict(x_test)
```

To determine the accuracy of those predictions, we'll create a **confusion matrix**. This matrix visualizes how many times the model was right versus how many times it was incorrect.

```
from sklearn.metrics import classification_report, confusion_matrix

cm = np.array(confusion_matrix(y_test, y_predict, labels=[0,1]))

confusion = pd.DataFrame(cm, index=['Diabetic', 'Not Diabetic'],
columns=['Predicted Diabetes', 'Predicted Healthy'])
print(confusion)
```

```
                     Predicted Diabetes  Predicted Healthy
Diabetic                            225                 23
Not Diabetic                         68                 68
```

If it's a little difficult to understand that display, think of it like this:

Diabetic (Outcome = 1)

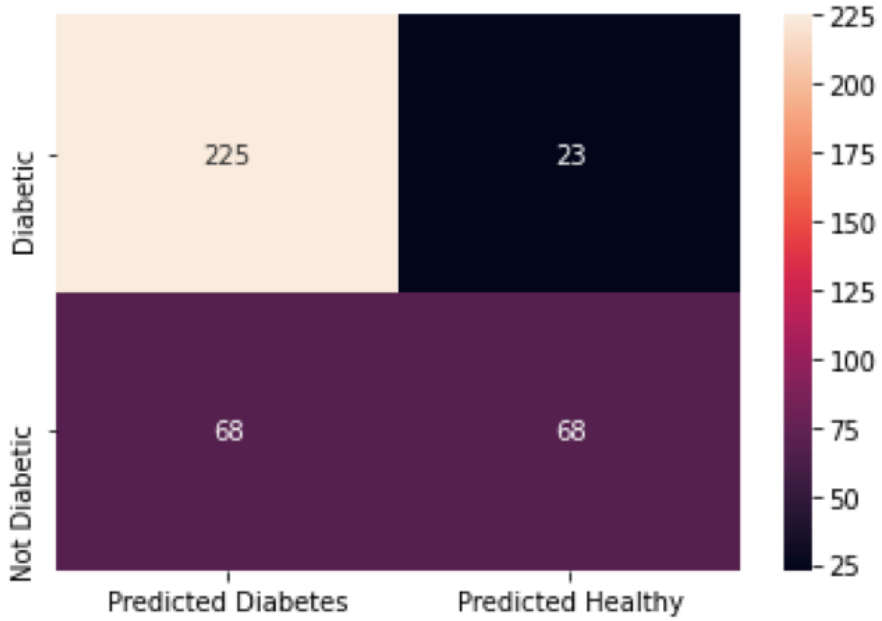
- **True positive.** Patient is diabetic, and the model correctly predicted that.
- **False positive.** Patient is not diabetic, but the model incorrectly said the patient is diabetic.

Not Diabetic (Outcome = 0)

- **False positive.** Patient is not diabetic, but the model incorrectly said the patient is diabetic.
- **True negative.** Patient is not diabetic, and the model correctly predicted that.

Here is a graphical way to show the same results using the powerful Seaborn extension to Matplotlib:

```
sns.heatmap(confusion,annot=True,fmt='g')
```



The classification report prints a summary of the model, showing a 77% precision. This means our model accurately predicts diabetes 77% of the time.

```
print(classification_report(y_test, y_predict))
```

```
              precision    recall  f1-score   support

           0       0.77      0.91      0.83       248
           1       0.75      0.50      0.60       136

   micro avg       0.76      0.76      0.76       384
   macro avg       0.76      0.70      0.72       384
weighted avg       0.76      0.76      0.75       384
```

f1-score is the weighted average of recall and recall:

- **precision** (aka positive predictive value): (number of correct positives / number of all positives) = 225/(225+68) = 0.77
- **recall:** (true positives) / (true positives + false negatives) = (225)/(225+23)

Classification: K-means Clustering

K-means clustering is an example of a model that is not trained. So, there is no training set, which is called an **unsupervised** model, meaning without labels. K-means clustering is a type of **classification** problem. The goal is to put data into groups by something they have in common, meaning put them in clusters. Example use cases are:

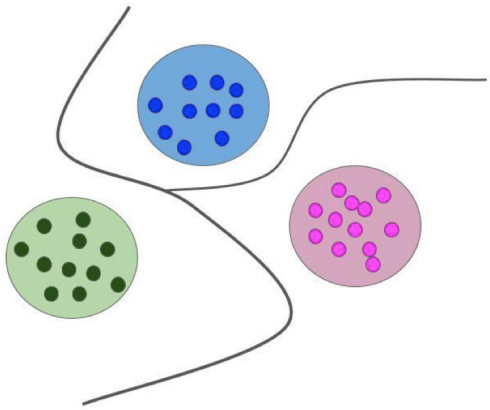
- Potential cybersecurity event identification
- High credit risk borrowers
- Customer segmentation
- IT performance monitoring alert automation

With K-means clustering, we look at a set of data points and try to separate them into clusters. For example, we might have customers in a sales system. We might want to figure out which are most sensitive to price increases, coupons, discounts, advertising, etc. Or we could look at truck driver data and figure out which drivers are driving unsafely based upon speed approaching a stop sign, brake wear, fuel consumption per kilometer, etc.

A mathematician would say when we are doing classification that we are trying to fit a **hyperplane** in n dimensional space, which they call R^n. The hyperplane divides all the points on either side.

We start by guessing. Then we figure out the distance of each data point from every other data point. That's called the **nearest neighbor**. We move the hyperplane again and try again. So, it's the same approach as regression, to iteratively try to fit a curve or line among data points—except we're not looking for a coefficient. We just want to separate points on either side of the hyperplane.

The graphic below shows a curve that bends between the green, pink, and blue clusters.

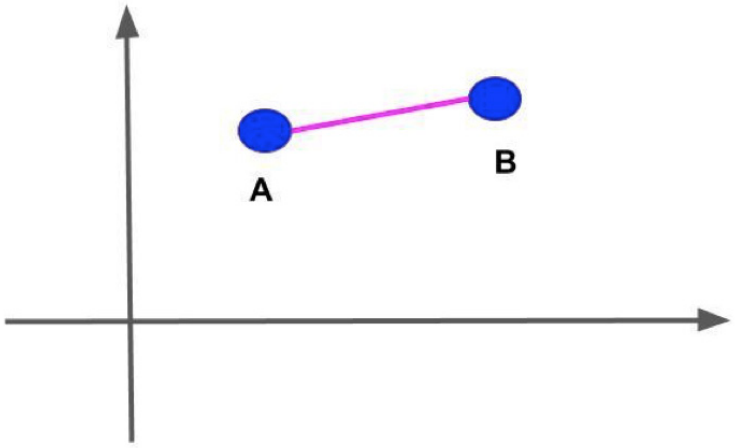


Of course, the algorithm might not converge. Then you would say you cannot divide the data into clusters. That means whatever hypothesis you are following is not true. So, quit. That's why when you run classification you tell the algorithm how many times to try before giving up.

Here is how you calculate the distance between the points **a** and **b**. Those of you with a math background will recognize this as the formula for the distance between points in **Euclidean Space**. You subtract the coordinates of one from the other then square that. Then take the square root. You square the number to make all distances positive.

$$\sqrt{(a-b)^2}$$

Below you can see the distance from A and B. It is the length of the purple line segment.



The code explained: K-means clustering

Now we show how to use K-means clustering using scikit-learn. We make up some data to keep things simple. The code is **here** in a Jupyter notebook.

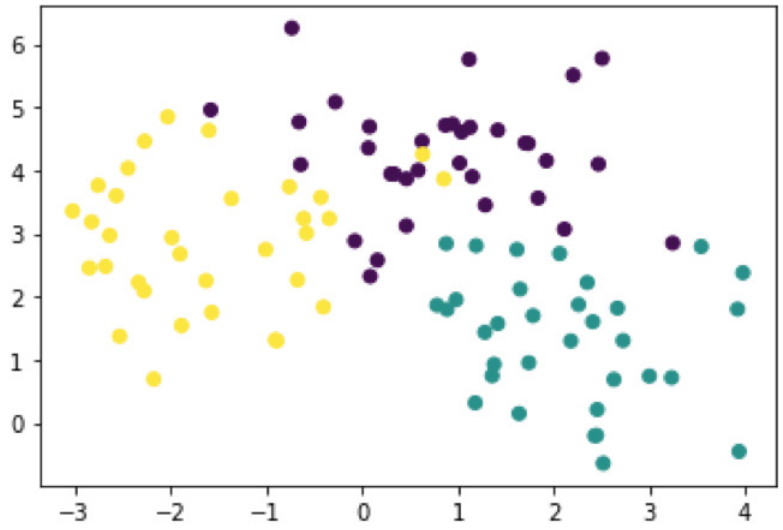
First, we import the required Python libraries. Then we use the scikit-learn make_blobs function. That's designed to generate clusters. We make 3 clusters with 100 samples. The number of features is 3, meaning 0, 1, and 2.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X , y = make_blobs(n_samples=100, centers=3, random_state=0)
```

Now let's use the Matplotlib scatter function to plot these so that we can see that they truly cluster together. c is the color array. It must be the same size as the data you are plotting. The values are [0, 1, 2] corresponding to the three features.

```
plt.scatter(X[:,0], X[:,1], c=y)
```



Now we see how easy it is to use scikit-learn. We just declare the K-means algorithm and tell it how many clusters to divide the data into. Then we call fit() and scikit-learn does that.

Then we print the labels that show for each value in the array X, what cluster does it belong to.

```
Kmean = KMeans(n_clusters=3)
Kmean.fit(X)
Kmean.labels_
```

Here is the array of clusters.

```
array([2, 1, 2, 0, 0, 1, 0, 0, 2, 1, 1, 1, 2, 1, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 0, 2, 2, 2, 2, 0, 0, 1, 2, 2, 1, 0, 1, 1, 2, 2, 0, 0, 2, 2, 1,
       1, 1, 2, 2, 0, 0, 1, 2, 1, 2, 0, 0, 2, 2, 1, 2, 2, 0, 0, 0, 0, 2,
       1, 0, 2, 1, 0, 2, 0, 2, 1, 1, 1, 1, 0, 2, 1, 1, 2, 1, 1, 1, 1, 1,
       2, 1, 2, 2, 0, 0, 0, 0, 1, 1, 0, 0], dtype=int32)
```

Now we make one point(2,3) and ask scikit-learn to use the model and tell us what cluster the point belongs to.

```
test=np.array([2,3]).reshape(1,-1)
Kmean.predict(test)
```

Here are the results. (Note that it says the data type is a 32-bit integer. NumPy supports a long list of numeric types.)

```
array([1], dtype=int32)
```

Now we show the cluster centers:

```
Kmean.cluster_centers_
```

Each is a pair of x-y coordinates.

```
array([[-1.5510878 ,  2.88827923],
       [ 0.9801637 ,  4.30837857],
       [ 2.2427373 ,  1.30652003]])
```

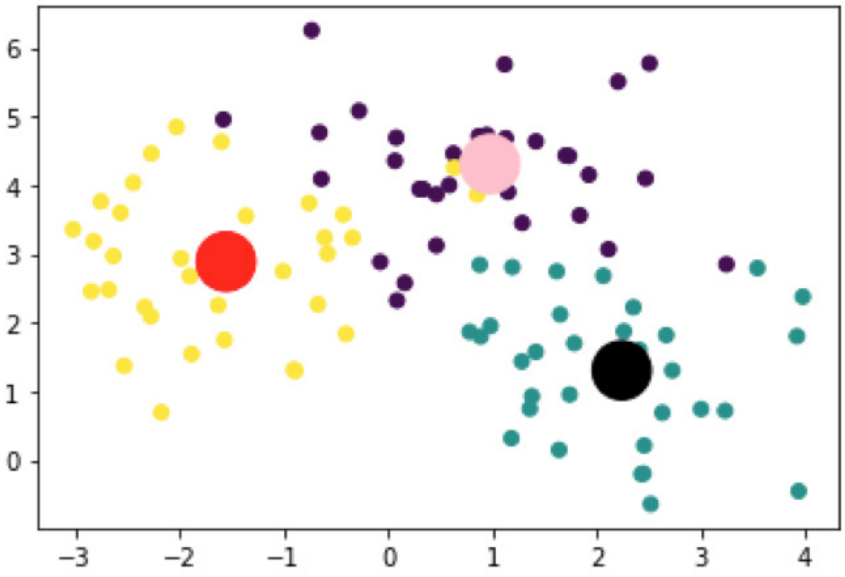
Now let's plot the centers on top of the original chart. We take all the x values and put them in **a** and all the y values and put them in **b**.

```
a=Kmean.cluster_centers_[:,0]
b=Kmean.cluster_centers_[:,1]
```

Here we plot the centers onto the data, making them large red, pink, and black dots.

```
plt.scatter(X[:,0], X[:,1], c=y)
plt.scatter(a,b,c=['red', 'pink', 'black'],s=600)
plt.show()
```

Here is the chart. As you can see the centers are right in the middle of each cluster:



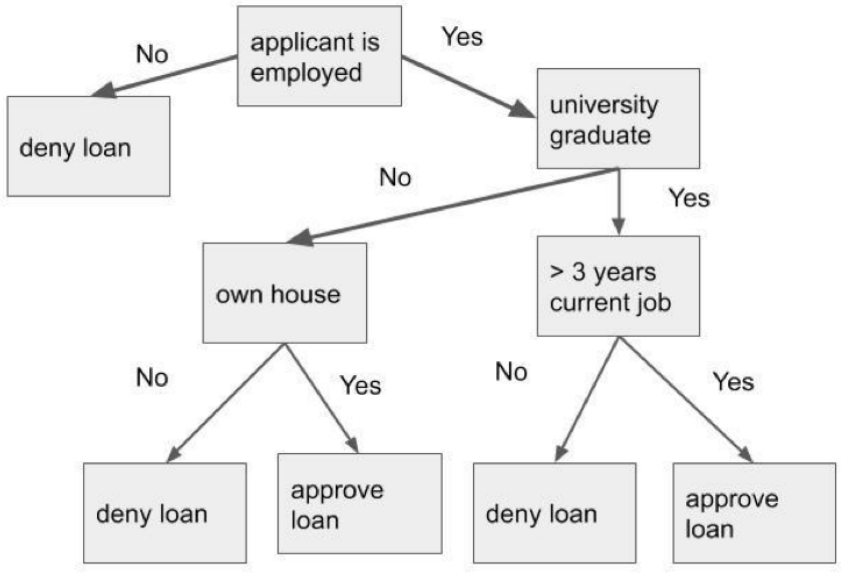
Classification: Decision Trees

A **decision tree** classifies data by asking questions at each node. You can use entropy or information gain, Gini, which measures impurity.

A sample set is said to be **pure** if all the elements are in the same label. Here our labels are 0, 1, 2:

- If Gini = 1 (i.e., 100%) then the elements are randomly distributed.
- If Gini = 50% then the elements are equally distributed.
- If Gini = 0, stop. The calculation is done; all the labels are in separate sets.

A decision tree works like this bank loan example. (The code is available **here**.) You branch based on answers until you reach the terminus node, which is the decision.



Gini is the probability of an object being wrongly classified. Each p is the probability of an object being classified to a label. When Gini = 0, the chance of wrong classification is 0 so stop. It is 0 when the probabilities sum to 1.

$$Gini = 1 - \sum_{i=1}^{n} p_i^2$$

The code explained: Decision tree

Here we go. Let's make up some data. Here in this simple example we just have a nx1 array, i.e. a vector.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.datasets import make_blobs

x,y = make_blobs(n_samples=100, n_features=3)

clf = tree.DecisionTreeClassifier()

clf = clf.fit(x, y)
```

x has shape (100,3) since we asked for n_features=3. The make_blobs generated these data clusters for us.

```
array([[ -4.46065188,  -6.4944518 ,   5.52121682],
       [ -3.81505102,  -2.99198226,   5.09465203],
       [  7.03002858,  -5.08368717,  -9.86018268],
       [  5.37722699,  -4.83812689,  -9.8746919 ],
       [ -2.05919531,  -1.29351217,   3.97821907],
       [  5.02286836,  -6.42049141, -11.26941882],
...
```

and y is the label:

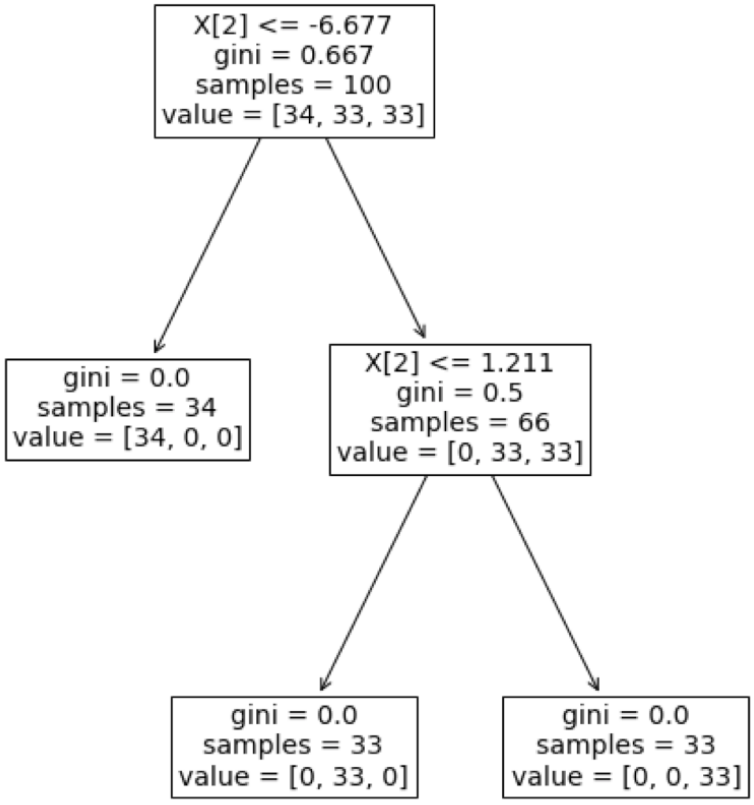
```
array([2, 2, 0, 0, 2, 0, 0, 0, 2, 1, 2, 0, 0, 1, 2, 2, 0, 1, 1, 0, 1,
1, ...
```

When you run fit(), the algorithm keeps going until Gini=0, that is where it reaches purity, as the equation below goes to 1 - 0 (chance of an error) = 1, pure.

```
plt.figure(figsize= [8, 10])
tree.plot_tree(clf,fontsize=14)
```

In the tree, we can see the decisions the algorithm made and the samples it drew at each node.

Note that at each label it takes the 100 input samples and assigns them to either the terminating node or the next batch of samples to repeat the process.



Neural Networks

We are going to show you how to create a **neural network** using Keras.

We will use as an example data from the **2016 Scania Industrial Challenge**, a data science competition hosted by the Swedish truck manufacturer. Here is **the data** and here is **the code**.

What is Keras?

Keras is an API that sits on top of Google's TensorFlow, Microsoft Cognitive Toolkit (CNTK), and other machine learning frameworks. The goal is to have a single API to work with all of those and to make that work easier.

Keras vs TensorFlow

Our advice is to use Keras, not TensorFlow. There is a tendency among beginning data scientists to want to use TensorFlow. They assume that because it was developed by Google, it has a certain panache, but, unfortunately, it's needlessly complicated.

If you use TensorFlow without understanding it, your model will produce incorrect results. This will make you draw the wrong conclusion. Keras does all the complicated set up, like rearranging NumPy arrays and creating one-hot vectors for you. So, save yourself the misery of doing that.

For advanced users, you can use TensorFlow functions and Keras at the same time. You might need to do this, for example, if you want to write your own activation function.

How neural networks work

You should have a basic understanding of the logic behind neural networks before you study the code below. Here is a quick review; you'll need a basic understanding of linear algebra to follow the discussion.

A **neural network** is like our old friend, the linear regression model. It's the same principle. The only difference is the coefficients in the model are functions and not numbers. Because this is difficult to visualize, it's called a **black box**.

The algorithm is fairly the same as with linear regression. You try to minimize the error and calculate the coefficients in the model y = mx + b. Except that's the simple case of one independent variable. Here we are dealing with more than one independent variable. So, we replace **m** and **x** with the matrices capital W (weights) and X.

Then our regression problem is this:

$$\sum_{i=1}^{m} w_i x_i + b$$

Now, to introduce some terminology.

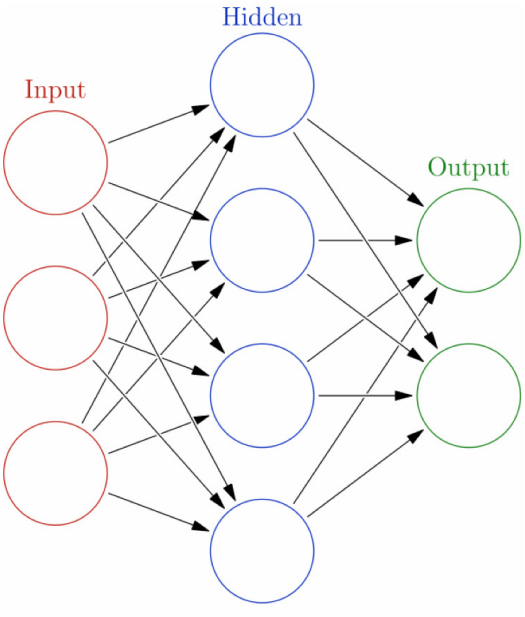
Basically, a **neural network** is a connected graph of **perceptrons**. (It's called a neural network because it looks like a picture of the synapses in a brain.) Each perceptron is just a function. It returns true or false.

A neural network has a minimum of three layers:

- The **input layer**, where we have m number of inputs (x values).
- The **output layer**, where we have the labels such as 1 or 0.
- The **hidden layer** is part of the black box concept. You can't visualize it other than to show and specify how many there are. Just think of the perceptrons like coefficients in the linear model.

For example, if we were working with something more complex, like handwriting recognition, the input would be pixels (dots in a graphic) and the output labels would be letters.

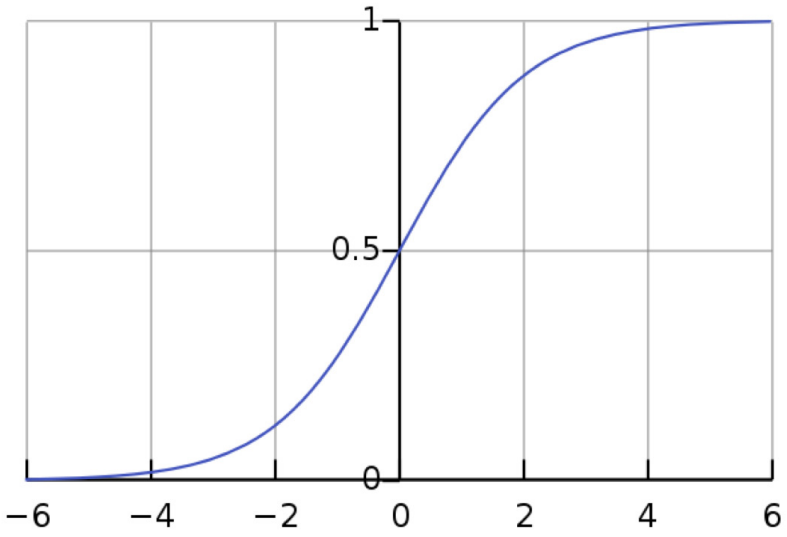
The algorithm runs until it finds the coefficients (i.e., functions) that most accurately represent the relation between the inputs and outputs.



(Graphic source)

Each perceptron makes a calculation and hands that off to the next perceptron. This calculation is really a probability. In the case of a classification problem a threshold **t** is arbitrarily set such that if the probability of event x is > t then the result is 1 (true) otherwise false (0). It loops back through the layers until it iteratively finds the optimal solution.

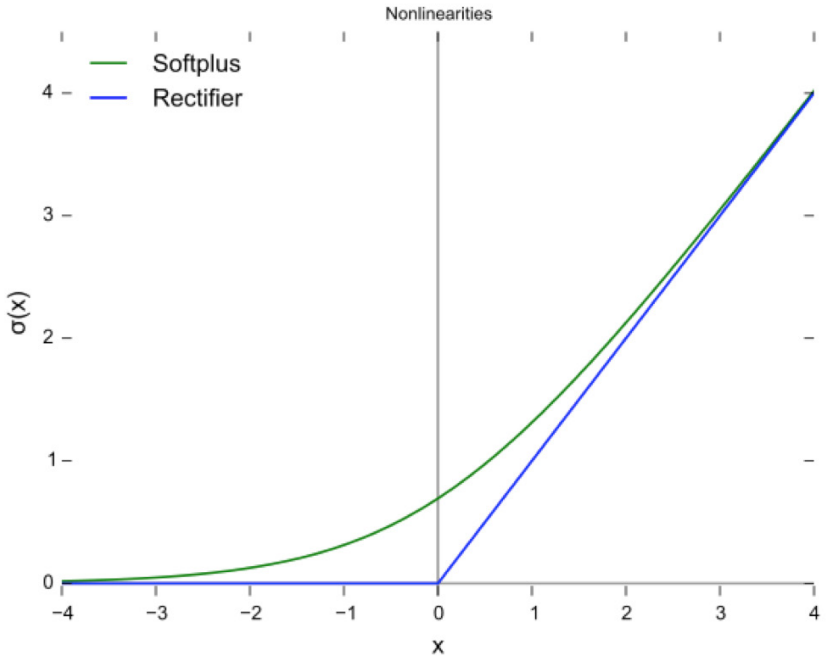
The functions used are a **sigmoid** function, meaning a curve, just like logistic regression, except this curve ranges from -1 to 1 and not 0 to 1:



(Graphic source)

Besides the threshold approach, another activation—that is, decision function—is **relu** (linear rectifier function). This determines whether the perceptron should have off a true (1) or false (0) to the next neuron in the network.

This process repeats iteratively until the optimal solution is found or until you tell it to quit searching.



(Graphic source)

The code explained: Keras neural network

How let's try an example. The Scania data is **here** and the code is **here**.

The data is about failure in the air pressure system (APS) on Scania heavy trucks, including APS powers brakes and the gear shifter. The features are metrics from different components that make up the system. Scania has coded the component names to keep them confidential.

Here is part of the data:

- The first column is the index.
- Class is either positive or negative, indicating failure or not.
- The other values are components and metrics. That could be heat, vibration, hours used, etc.
- There are 171 columns. We drop 28 of them because those are not numbers, they are histograms computed by Scania.

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	a
0	neg	76698	na	2130706438	280	0	0	
1	neg	33058	na	0	na	0	0	
2	neg	41040	na	228	100	0	0	
3	neg	12	0	70	66	0	10	
4	neg	60874	na	1368	458	0	0	

Note that the data has a lot of missing data. We drop those rows to make things simpler. (A data scientist would say opt for a **dense** matrix to a **sparse** one. Keras and TensorFlow can handle the situation of missing values, aka sparse values.)

Now we proceed.

First, load the data into a Pandas dataframe using **pd.read_csv()**. The first row in column is column headers, so Pandas will use that for column names in the Pandas dataframe.

```
import tensorflow as tf
from keras.models import Sequential
import pandas as pd
from keras.layers import Dense
import numpy as np

def isAps(s):
    if s == 'neg':
        return 1
    else:
        return 0
def isNa(s):
    if s == 'na':
        return 'Nan'
    else:
        return s

train =
pd.read_csv('/Users/walkerrowe/Documents/mlbook/Scania/ida_2016_challenge_update/ida_2016_training_set_update.csv',
delimiter=',')
```

Note that we can work with numbers only—not text (although Keras and TensorFlow can handle labels as text). We change **na** in the input data to **NaN** which means missing value in NumPy. Then we drop the rows with NaN values and change neg to 0 and pos to 0 using this:

```
train.replace('na',np.NaN,inplace=True)
train.dropna(inplace=True)
```

inplace=True replaces the data in the existing dataframe. If you don't put that then Pandas will create a new dataframe.

Next uses the **apply()** function to run the function **isAps()** in-line, which is what lambda means. Apply does not have the **inplace** option, so we create the dataframe **y**. We want that to be the label, so it naturally belongs in its own dataframe (It's actually called a panda **Series** since it has only one column, plus the index.)

```
y=train['class'].apply(lambda x: isAps(x))
```

Next we drop all the columns that are histograms.

```
train.replace('na',np.NaN,inplace=True)
train.dropna(inplace=True)
y=train['class'].apply(lambda x: isAps(x))


dropColumns=['class', 'ag_001',
'ag_002',
'ag_003',
'ag_004',
'ag_005',
'ag_006',
'ag_007',
'ag_008',
'ag_009',
'ay_001',
'ay_002',
'ay_003',
'ay_004',
'ay_005',
'ay_006',
'ay_007',
'ay_008',
'ay_009',
'az_001',
'az_002',
'az_003',
'az_004',
'az_005',
'az_006',
'az_007',
'az_008',
'az_009',
'ba_001',
'ba_002',
'ba_003',
'ba_004',
'ba_005',
'ba_006',
'ba_007',
'ba_008',
'ba_009',
'cn_001',
'cn_002',
'cn_003',
'cn_004',
'cn_005',
'cn_006',
'cn_007',
'cn_008',
'cn_009',
'cs_001',
'cs_002',
'cs_003',
'cs_004',
'cs_005',
'cs_006',
'cs_007',
'cs_008',
'cs_009']


train.drop(dropColumns,axis=1,inplace=True)
```

Now we set up and train the model. The parameters to note are:

- **input_shape**. We only have to give it the shape (number of columns) of the input on the first layer. Take the second number in **train.shape**, which is (60000, 116) or 116 feature columns. (Remember we dropped many of them.)

- **Dense** applies the activation function over ((w • x) + m).The activation functions we are using are **relu** and **sigmoid**. Without going into the technical details, it's close enough to relate this to linear regression and say that w is the coefficients, x the input values, and m is the offset (more correctly called b, bias). The actual definition is more rigorous than that. You can read about that in the **Keras reference manual**.

- **Loss**. The goal of the neural network is to minimize the loss function, i.e., the difference between predicted and observed values. There are many functions we can use. We pick **binary_crossentropy** because our label data is binary.

- **Optimizer**. We use the optimizer function sgd, **Stochastic Gradient Descent**. It's an algorithm designed to minimize the loss function in the quickest way possible. There are others.

- **Epoch** means how many times to run the model. Remember that it is an iterative process. You could add additional epochs, but, as we see below, this data will converge with just one epoch.

- **Metrics** means what metrics to display as it runs. **Accuracy** means how accurately the evolving model predicts the outcome.

- **Batch size n** means divide the input data into n batches and process each in parallel. Again, this is just a trial and error metric that you can adjust to try to increase the accuracy or reduce the time it takes to train the model.

So, we declare a sequential model and the input layer, one hidden layer, and the output later. The first number in the **Dense()** function is the number of nodes. Other than the input and output layer that number too can be adjusted up or down.

The first argument in **model.add()** is the number of layers. As we just said, the first input layer must be the size of the input. The output layer is 1 as we want 1 or 0 for the result, meaning just 1 output class (and not 2 as in 2 possible values).

```
model = Sequential()

model.add(Dense(116, activation='relu', input_shape=(116,)))

model.add(Dense(8, activation='relu'))

model.add(Dense(1, activation='sigmoid'))
```

Now we compile the model and run **fit()** to kick off the calculation.

```
model.compile(loss='binary_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(train, y,epochs=8, batch_size=1, verbose=1)
```

Here is the run. As you can see the accuracy does not improve for each run. With other data it might and, in fact, usually does.

```
Epoch 1/8
591/591 [==============================] - 2s 3ms/step - loss: 2.1319
- acc: 0.8663
Epoch 2/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663
Epoch 3/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663A: 0s - loss: 2.1930 - acc: 0.86
Epoch 4/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663
Epoch 5/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663A: 0s - loss: 1.83
Epoch 6/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663A: 0s - loss: 2.1244 - acc: 0.866
Epoch 7/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663A: 0s - loss: 2.5469 -
Epoch 8/8
591/591 [==============================] - 1s 2ms/step - loss: 2.1319
- acc: 0.8663
```

Normally when you work with machine learning, you split the data into training and test data sets. **Train** is used to train the model. **Test** is used to see how accurate it is. But Scania has already done that. They gave us two files.

You test a model's accurate accuracy by running predictions over the test data set. Remember that it is calculated (trained) using the training dataset. Then you take the labels already present in the data (a) and the predicted labels (b). The accuracy then is:

Number of correct predictions / number of all predictions

Proceeding, we read the test data set into a dataframe then run the **predict()** function against that.

```
test =
pd.read_csv('/Users/walkerrowe/Documents/mlbook/Scania/ida_2016_challenge_update/ida_2016_test_set_update.csv', delimiter=',')

z=pd.Series(test['class'].apply(lambda x: isAps(x)))

test.drop(dropColumns,axis=1,inplace=True)
test.drop(['id'],axis=1,inplace=True)

test.replace('na',np.NaN,inplace=True)

test['prediction']=model.predict_classes(test)
```

I am not sure why, but the test Scania dataset includes only data on trucks for which the APS system did not fail. We can see that like this:

```
test['class']=z
test['class'].groupby(test['class']).count()
```

```
     class
0    16000
```

Here we show counts by prediction:

```
test['prediction'].groupby(test['prediction']).count()
```

So, our model is accurate 15,968 times out of 16,000 for a rate of 99.8% accuracy:

```
prediction
0       15968
1       32
```

Additional Resources

BMC Blogs has a wealth of information on machine learning and related topics. Whether you want to better understand concepts or practice with more tutorials, here are a few worthy reads to continue your ML education:

Blogs & Guides

BMC Machine Learning & Big Data Blog

Apache Spark Guide

Data Visualization Guide

Machine Learning with TensorFlow & Keras

scikit-learn Guide

Data Science

What Is a Data Pipeline?

Structured vs Unstructured Data: A Shift in Privacy

Dark Data: The Basics and The Challenges

Enabling the Citizen Data Scientists

Using Python for Big Data and Analytics

Machine Learning & Artificial Intelligence

Machine Learning, Data Science, Artificial Intelligence, Deep Learning, and Statistics

Machine Learning: Hype vs Reality

4 Types of Artificial Intelligence

How Machine Learning Benefits Businesses

What is a Neural Network? An Introduction with Examples

What's a Deep Neural Network? Deep Nets Explained

AIOps Machine Learning: Supervised vs Unsupervised

Interpretability vs Explainability: The Black Box of Machine Learning

Bias and Variance in Machine Learning

Top 5 Machine Learning Algorithms for Beginners

Author Bio

Walker Rowe is an American freelance technical writer and computer programmer living in Cyprus. He writes SDK documentation and code tutorials. He is the founder of **Hypatia Academy Cyprus**, which teaches computer programming to secondary school students.

You can find him on **LinkedIn**, **Upwork**, and **his website**.