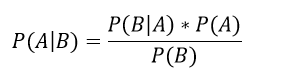
# Hello, my name is Ernesto Lee and I want us to build our first Naive Bayes Classifier with Python.

Naive Bayes Classifiers are one of the most intuitive and popular algorithms used in supervised learning, whenever the task is a classification problem. In addition to it being both intuitive and popular - it is also the EASIEST algorithm to learn and use. I’ve been talking about the differences between supervised and unsupervised learning, as well as between classification and regression, in my previous episodesand, if you are not familiar with this terminology, I suggest you to have a look at there earlier episodes.

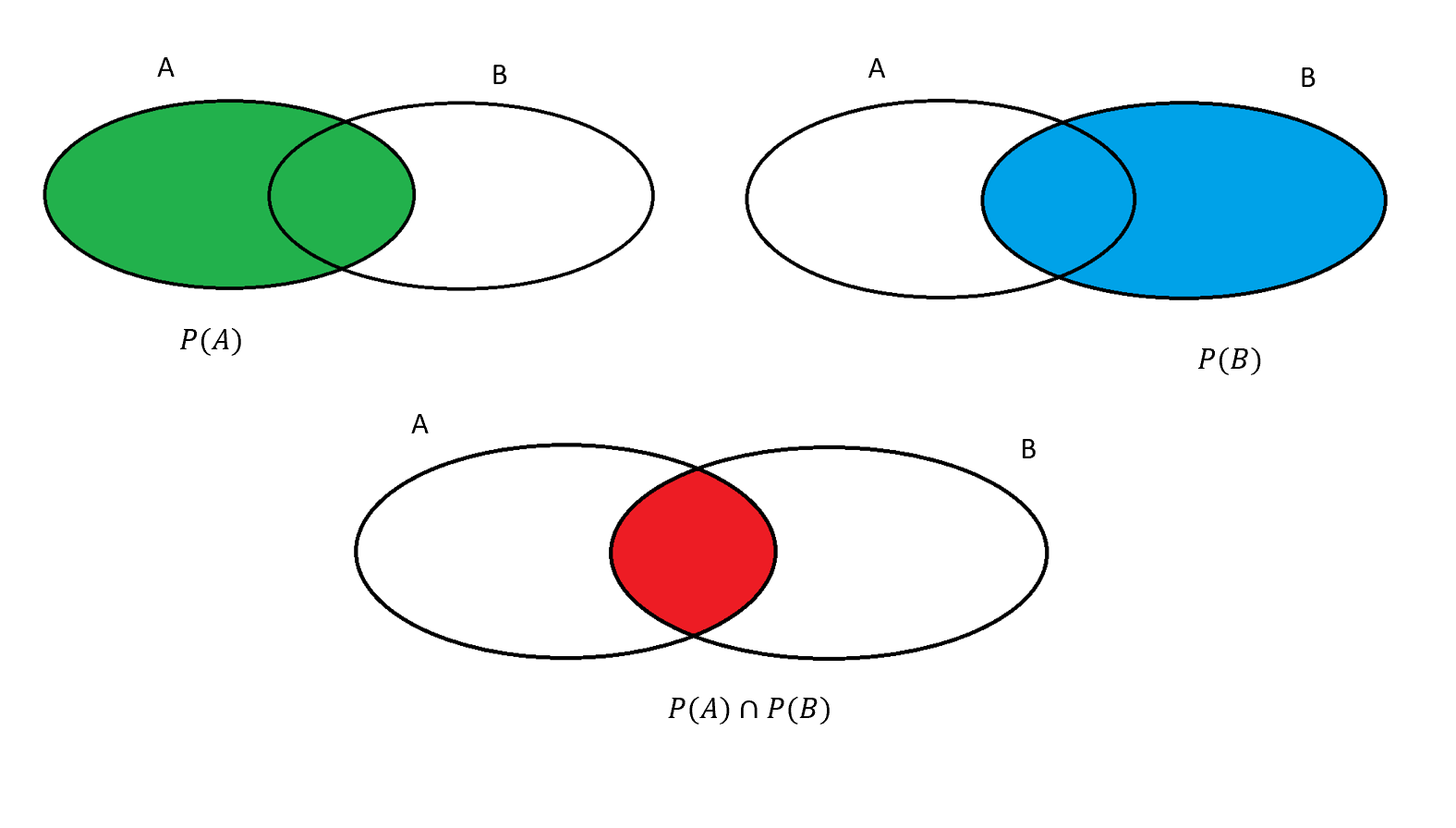
Here, I’m going to discuss the surprisingly easy math behind the Naive Bayes Classifier and then I will implement it from scratch with Python and scikit learn, using the well-known Iris Dataset.

To understand this algorithm, we first need to refresh some concepts of probability theory. Indeed, Naive Bayes Classifier is based on the Bayes’ Theorem of conditional probability:

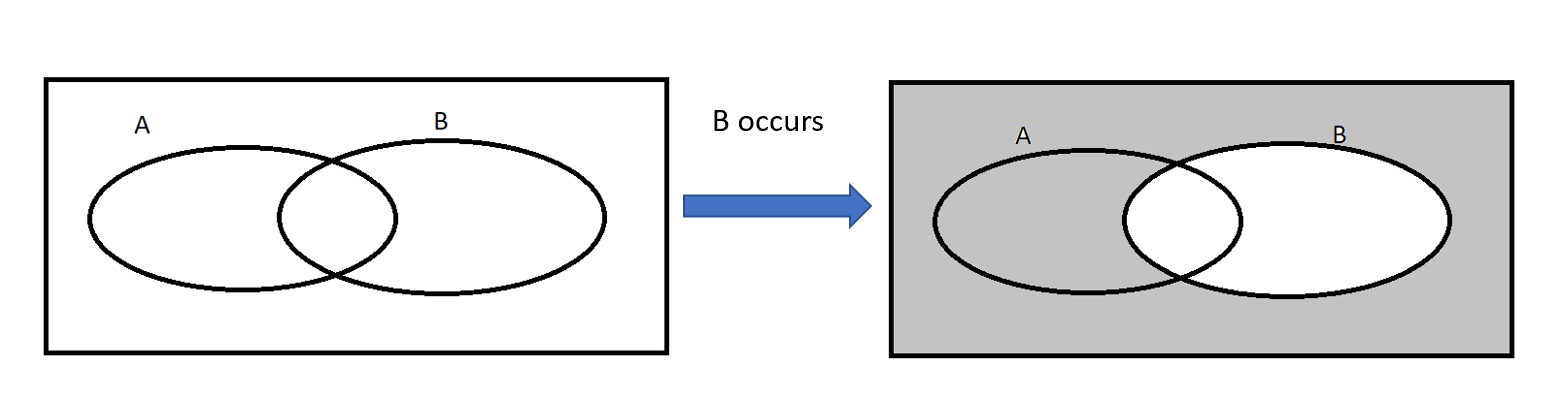


Where A and B are events and P(B)≠0. We talk about conditional probability of A with respect to B when we want to know the likelihood of A event, given that B event has occurred.

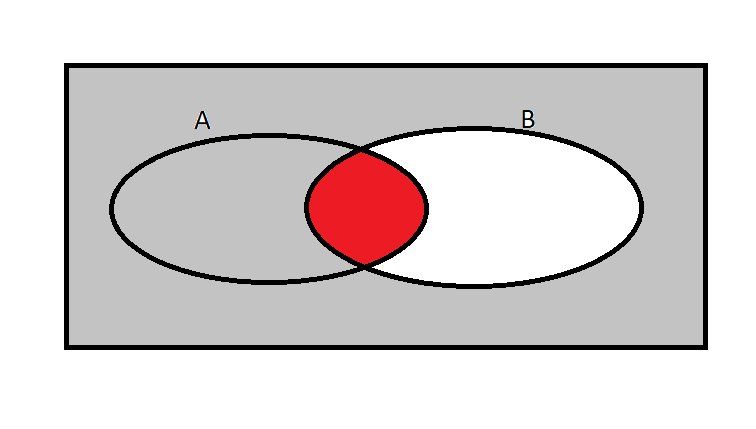
Let’s visualize it with a Venn Diagram:

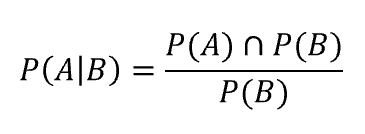


Basically, once the event B has occurred, the probability space of event A is reduced to the intersection between A and B… in other words the UNION of A and B, since everything that is not B cannot occur (indeed, we already know that B has already occurred!). The more there is a UNION of A and B - the greater the likelihood that when B occurs then A will occur. The situation is the following:

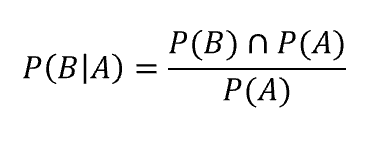


Now, all the grey area no longer exists. So what we want to compute is the probability of the intersection, but we have to divide it by the probability of the new domain, which is not 1 as before (the rectangular area) but the probability of B. So as you can see - the probability of event A being true given that event B has already occured is equal to the union of P of A and P of B… all of that divided by the probability of event b.

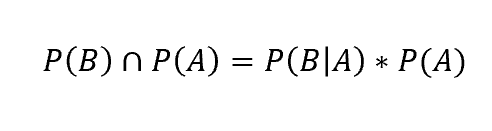




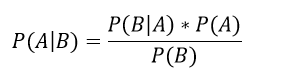
Now, since the same reasoning holds for the conditional probability of B as you can see from the equation on your screen:



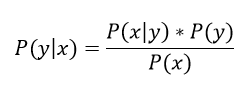
And knowing that the union of P of B and P of A is equal to the Probability of B given A times Probability of A as you can see:



We can then substitute the latter in the numerator of P(A|B), obtaining the exact the formula for Bayes Theorem:



So that is all that there is to Naive Bayes with respect to the math of Naive Bayes classification. The obvious question becomes how do we apply this theorem to a classification task? In classification problems, the theorem on your screen is used to predict the likelihood of a class, given a feature (or a vector of features). So, if I want to predict the class *y* of a new observation *x*, the algorithm will look like what you see:



Let’s consider the following example:

|  |  |
| --- | --- |
| **Feature** | **Target** |
| ‘sale’ | ‘spam’ |
| ‘sale’ | ‘spam’ |
| ‘name’ | ‘not spam’ |
| ‘balance’ | ‘not spam’ |
| ‘balance’ | ‘spam’ |
| ‘confirm’ | ‘spam’ |
| ‘confirm’ | ‘not spam’ |
| ‘congratulations’ | ‘not spam’ |
| ‘congratulations’ | ‘spam’ |
| ‘congratulations’ | ‘spam’ |

We are considering a binary classification problem with only one feature: the object of the e-mail. Then we receive a new e-mail with a given object and we want to predict whether or not it’s spam.

Imagine that the object of our new e-mail is ‘congratulations’: how can we proceed? We want to know the likelihood of the class spam given our object, hence we have to compute some probabilities:

P(congratulations|spam) =

P(spam) =

P(congratulations) =

And finally compute the conditional probability:

P(spam|congratulations) = [P(congratulations|spam) \* P(spam)]/P(congratulations) =

So the probability of our new e-mail of being spam, given that its object is ‘congratulations’, is 66.7%: we conclude that it is more likely that it’s not spam.

Python libraries offer three kinds of Naïve Bayes classifiers:

* Gaussian:it assumes that features follow a normal distribution.
* Multinomial**:** it is used for discrete counts
* Bernoulli**:** the binomial model is useful if your feature vectors are binary (i.e. zeros and ones)

Of course, here the example refers only to one feature and two classes. However, the procedure is the same even with a vector of features and multiple classes (with the only difference of applying an *argmax* function to the probability output, so that the class with the highest probability is returned).

For this purpose, I’m going to use the Gaussian Naive Bayes Classifier. Let’s implement it with Python:

from sklearn import datasets

import matplotlib.pyplot as plt

import pandas as pd

#importing the necessary packages

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

#downloading the iris dataset, splitting it into train set and validation set

iris = datasets.load\_iris()

class\_names = iris.target\_names

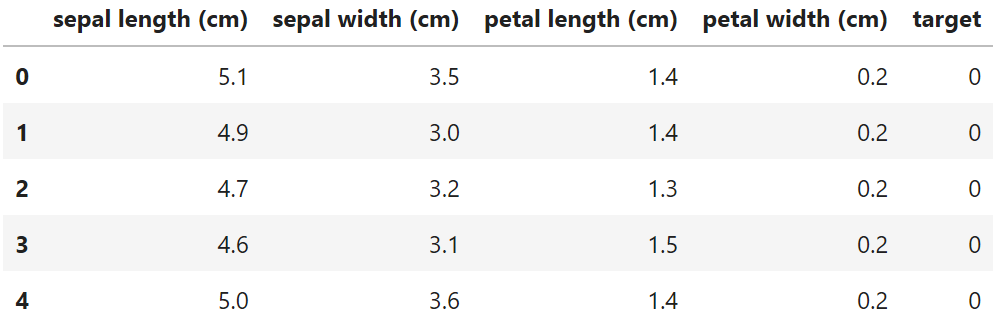
iris\_df=pd.DataFrame(iris.data, columns=iris.feature\_names)

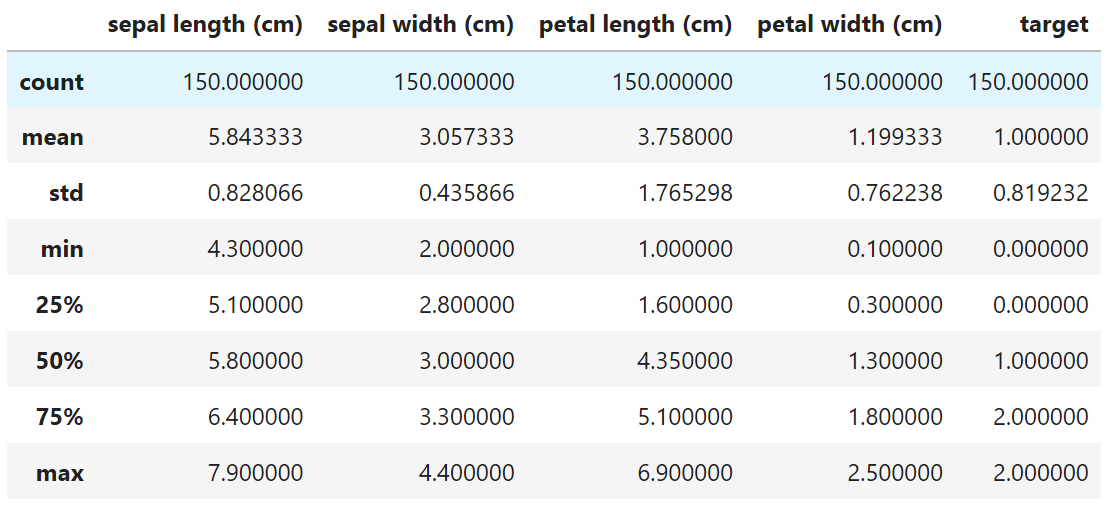
iris\_df['target']=iris.target

Let’s have a look at our dataset:

iris\_df.head()

iris\_df.describe()





As you can see, we have 4 features and one target, which is categorical since we have 3 classes ( Setosa, Versicolor and Virginica, encoded with 0, 1 and 2). Now we can build our classifier as follows:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(iris\_df[['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']], iris\_df['target'], random\_state=0)

NB = GaussianNB()

NB.fit(X\_train, y\_train)

y\_predict = NB.predict(X\_test)

print("Accuracy NB: {:.2f}".format(NB.score(X\_test, y\_test)))



As you can see, our algorithm correctly classified all the data of the test set. It is a pretty good result, actually the best we can aspire to. However, keep in mind the poor dimension of our dataset (only 150 observations).

It is normally hard to obtain a 100% accuracy on real data. Furthermore, it might also be counterproductive aspiring to an error equal to zero. Indeed, since the last goal of any ML algorithm is making reliable predictions on new, unknown data, we’d prefer an algorithm which is able to generalize rather than perfectly fit the test data. The risk of overfitting (using too many parameters in order to have a model which perfectly fit test data) might lead to a useless model, if it is not able to fit new data.

Please complete this exercise on your own and with a different data set. Thank you.