



Implementing a Multinomial Naive Bayes Classifier from Scratch with Python



John Michael Kovachi

[Follow](#)

7 min read · Jan 21, 2018



148



1



For sentiment analysis, a Naive Bayes classifier is one of the easiest and most effective ways to hit the ground running for sentiment analysis. My goal of this post is to show how to implement this classifier **from scratch** and without help from great libraries such as [NLTK](#). I will be doing an implementation of the algorithm found in Jurafsky and Martin's [Speech and Natural Language Processing \(3rd ed\)](#). This will be a learning experience for me as much as it will be for you. Let's get started.

Understanding Naive Bayes

The Naive Bayes classification algorithm is based off of [Bayes' Theorem](#). Bayes' Theorem is a beautiful yet simple theorem developed primitively by English statistician [Thomas Bayes](#) in the early 1700s. At its most basic level,

Bayes' Theorem uses conditional probabilities to “predict” the outcome of later probabilities. The Bayesian concept of *a priori* probability is important: it is used as a ground truth from which we can use to obtain the outcome of other probabilities. Let's take a look at Bayes' theorem:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

The diagram illustrates the components of Bayes' Theorem. At the top, 'Likelihood' is labeled above $P(x|c)$, 'Class Prior Probability' is labeled above $P(c)$, and 'Posterior Probability' is labeled below $P(c|x)$. Arrows point from these three labels to the terms in the formula. At the bottom, 'Predictor Prior Probability' is labeled below $P(x)$, and an arrow points from it to the denominator of the formula.

Bayes Theorem

$P(c)$ indicates the *a priori* probability of the given **class**, while $P(x)$ represents the probability of a given **predictor**. Below the main formula, we can see that the theorem can be derived to show that the probability of a class given a predictor can be obtained by multiplying the probability of individual features that make up document X given a class c times the *a priori* probability of c . We will see this in action later in the article.

With a little thought, it is clear to see that this theorem could potentially have many applications in natural language processing, especially classification problems. With the Naive Bayes algorithm, our ultimate goal is to train our model to learn the probabilities needed in order to make a classification decision about a given document (which can be a sentence, entire corpus, etc). This leads to the problem of training our model, which we will now explore.

Training the Naive Bayes Model

To train our model, we will first need a training corpus. For sentiment analysis, there a plethora of suitable corpora out there, but I will personally be using the [Pitt MPQA opinion corpus](#) for use as a pre-annotated dataset. My ultimate goal in this sentiment analysis project is to be able to analyze sentiments from stock market news article corpora, so this news article corpora is a good starting point. This dataset can be used to train our model, as it contains a number of documents (words, phrases) that are classified as “positive”, “neutral”, and “negative”. For use in the Naive Bayes’ algorithm, these simple classifications work perfectly for our purposes.

To find the most probable class given a document d , we need to find the probability of d given c times the prior probability of c .

$$\hat{c} = \operatorname{argmax}_{c \in C} \underbrace{P(d|c)}_{\text{likelihood}} \underbrace{P(c)}_{\text{prior}}$$

To do this, for each class c in the set of available classes C we find the probability of each feature in d and multiply them together.

$$P(f_1, f_2, \dots, f_n | c) = P(f_1 | c) \cdot P(f_2 | c) \cdot \dots \cdot P(f_n | c)$$

The reason we can multiply the probability of the features together is because we assume that each feature is independent of one another: this is called **Bayes’ Assumption**. The Naive Bayes model is named as such because we are “naively” assuming conditional independence of features.

Deriving the prior probability of a class is rather trivial, as it is simply the sum of all words in d that are assigned to c divided by the number of words in d .

$$\hat{P}(c) = \frac{N_c}{N_{doc}}$$

Now, here comes the crucial question: how do we learn all of the probabilities of that make up each feature? The solution is again rather simple: for a given word w in words W from d we count how many of w belong in class c . We then divide this by ALL the words in d that belong to c . This gives us a probability for a word w given c :

$$\hat{P}(w_i|c) = \frac{\text{count}(w_i, c)}{\sum_{w \in V} \text{count}(w, c)}$$

We can make a minor adjustment to this formula to smooth the data (this accounts for the possibility that some words could have a zero-count for a class, which would make eliminate our work done so far):

$$\hat{P}(w_i|c) = \frac{\text{count}(w_i, c) + 1}{\sum_{w \in V} (\text{count}(w, c) + 1)} = \frac{\text{count}(w_i, c) + 1}{(\sum_{w \in V} \text{count}(w, c)) + |V|}$$

With the ideas set forth, we can now begin to train our data and calculate our probabilities.

Working with the data

We are almost set to train the data and build our classifier. First, however, we need to extract the data from whatever corpus we are using. The MPQA corpus can be a little challenging to extract, as the documents are presented as an XML structure. For parsing documents like this, I recommend using a

Python module called [ElementTree](#). ElementTree does all the dirty work for you when parsing, and allows you to access individual XML tree elements with ease. Whatever you do, just DO NOT try and [parse XML with regular expressions](#). It won't end well. (Hint: XML is a context-free grammar, while regexes are confined to dealing with finite automata). I won't get into my XML parsing script, as it is a little messy, but just know that I converted the XML to a list of 2-tuples (ex. ("happy", "positive")) containing the documents that were annotated with sentiment data.

Get John Michael Kovachi's stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

Now that we have parsed our data, we need to build counts of each individual word and corresponding with its class. This will later be used to calculate the probabilities used in our Naive Bayes formula. I have implemented this idea below:

I have used the Counter Python module to count the number of words that belong to each class and return them in a dictionary. Since we are employing the bag-of-words model for our Naive Bayes implementation, we are unconcerned with the ordering of the words in our documents, and thus can use an unordered dictionary to store the counts of words for each class. This code loops through our list of 2-tuples (“pol_list”) and creates a temporary dictionary holding the counts of words for each class from our two-tuple “pol”. Notice in my list comprehension I am lemmatizing each word using

NLTK's very own lemmatizer (I said I wouldn't use NLTK at all here, but I couldn't help myself here). I am also creating a set of stopwords and checking that each word in our document is not a stopword. If it is, I am throwing it out to improve the accuracy of our sentiment classification. Each dictionary counts positive, negative, and neutral words, and each are returned at the end of the function.

Next up, we are going to use these dictionaries to actually employ our training model. I have implemented this in a separate class called MN_NaiveBayes (Multinomial Naive Bayes). Let's take a look at its constructor:

Keep in mind the MPQA.count() method simply is a helper method that counts the number of features in a given class. The constructor here is pretty self-explanatory.

Now, let's get to the meat of the class: the training function.

This function implements Jurafsky's MN Naive Bayes training algorithm, which can be seen below:

```

function TRAIN NAIVE BAYES(D,C) returns log  $P(c)$  and log  $P(w|c)$ 
    for each class  $c \in C$           # Calculate  $P(c)$  terms
         $N_{doc}$  = number of documents in D
         $N_c$  = number of documents from D in class c
         $logprior[c] \leftarrow \log \frac{N_c}{N_{doc}}$ 
         $V \leftarrow$  vocabulary of D
         $bigdoc[c] \leftarrow \text{append}(d)$  for  $d \in D$  with class  $c$ 
        for each word  $w$  in  $V$           # Calculate  $P(w|c)$  terms
             $count(w,c) \leftarrow$  # of occurrences of  $w$  in  $bigdoc[c]$ 
             $loglikelihood[w,c] \leftarrow \log \frac{count(w,c) + 1}{\sum_{w' \text{ in } V} (count(w',c) + 1)}$ 
    return  $logprior, loglikelihood, V$ 

```

I have modified the algorithm a little bit to fit the purposes of the implementation, but the general idea is the same. I don't consider each word in the vocabulary V (all words in all classes), because I have already calculated the word counts per class. I have accounted for smoothing, though, by adding one to the numerator of the feature-given-class calculation and adding the cardinality of V to the denominator of the calculation. This idea is equivalent to the formula provided in this algorithm. We can see the equivalence here:

$$P(w_i|c) = \frac{count(w_i, c) + 1}{\sum_{w \in V} (count(w, c) + 1)} = \frac{count(w_i, c) + 1}{(\sum_{w \in V} count(w, c)) + |V|}$$

Keep in mind that we are using the log probability to compute these calculations, because log probability generally guarantees better accuracy and speed. A more verbose explanation of log probability can be found [here](#).

Testing our model

Finally, we will move on to testing our model. Testing the model is fairly straightforward, and is done by considering our *a priori* class probabilities in conjunction with our individual feature-given-class probabilities. Remember, with Naive Bayes we are considering each respective feature-given-class probability to be independent of the other, which allows for this

“naive” implementation. This idea is expressed here, where a position indicates an individual word (“feature”) in a test document:

$$c_{NB} = \operatorname{argmax}_{c \in C} \log P(c) + \sum_{i \in positions} \log P(w_i | c)$$

Jurafsky refers to this classification decision algorithm as a **linear classifier**, because we are making a classification decision based off a linear combination of probabilities.

Here is the implementation in code of this idea:

And there we have it. We have now implemented and are able to test a fully-operational MN Naive Bayes model! A lot of work, right? Hopefully you enjoyed reading this article as much as I have writing it. In the next blog, we will focus on testing this model and checking for statistical significance. I might attempt to test the model with a different data source, as well. We will see...

A full representation of the code for this project can be viewed on my [github](#):

<https://github.com/jmkovachi/sent-classifier>

Sources:

Speech and Language Processing

Speech and Language Processing

Speech and Language Processing web.stanford.edu

http://mpqa.cs.pitt.edu/corpora/mpqa_corpus/

Machine Learning

Naive Bayes

Natural Language Process

Statistics

Sentiment Analysis



Written by John Michael Kovachi

11 followers · 2 following

Follow

Responses (1)



Write a response

What are your thoughts?



Backend developer

Jun 12, 2019

...

Hi John,

I have gone through the GitHub link (<https://github.com/jmkovachi/sent-classifier>) which you have provided in the post. But “McDonaldDict.xlsx” is not available anywhere due to which I’m getting error File not found when trying to execute... [more](#)



[Reply](#)

Recommended from Medium

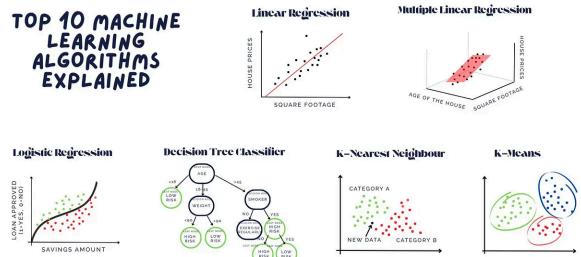


In Towards Explainable AI by Subhani Shaik

Uncovering Shopping Patterns: Market Basket Analysis Using the...

Ever walked into a store for just one thing and somehow walked out with five?

TOP 10 MACHINE LEARNING ALGORITHMS EXPLAINED



In Learning Data by Rita Angelou

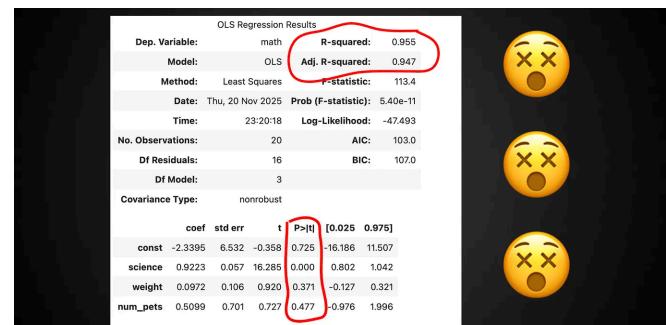
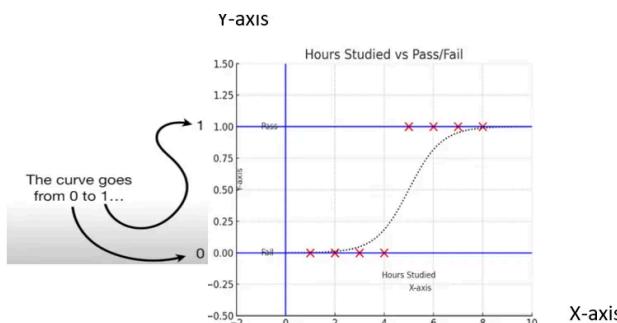
10 ML Algorithms Every Data Scientist Should Know—Part 1

I understand well that machine learning might sound intimidating. But once you break dow...

Jun 18 ⚡ 50



Jun 10 ⚡ 104 🎙 2



 Akanksha Verma, MSc Data Science

Logistic Regression in ML

Logistic regression is similar to linear regression, except logistic regression predic...

6d ago ⚡ 14 🎙 4



 Liu Zuo Lin

4 Things I Used To Overlook In Linear Regression

Linear Regression Is More Than Just
`model.fit(x_train, y_train)`

6d ago ⚡ 67

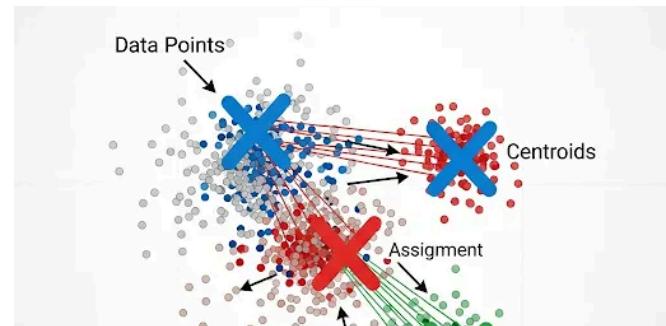


 Saurav Mandal

Do Hard Things if You Want an Easy Life

The one skill that changes everything

Jun 13 ⚡ 24K 🎙 999



 Shankar Angadi

AI :Unsupervised Learning with K-Means Clustering

Sep 14 ⚡ 131 🎙 5



[See more recommendations](#)