@author Brian Hassett

Naïve Bayes for Text Classification

Notes on Background Theory and Java Implementation

Preprocessing

Before the training documents are sent to the learning algorithm, they undergo a process of tokenization where they are broken up into meaningful terms. This implementation uses a simple tokenization process whereby whitespaces are used to delimit words, and the majority of punctuation is removed. In addition, all words are converted to lower case to remove the distinction between a word and its capitalized equivalent. In this implementation the apostrophe (') is preserved during tokenization, as are extremely common words (stop words) such as "to", "the", and "I". However it is possible to include further rules in a tokenization algorithm to determine how apostrophes and stop words should be treated.

Bag-of-Words Model (mostly taken from [Wikipedia])

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity. The bag-of-words model is commonly used in methods of document classification where the (frequency of) occurrence of each word is used as a feature for training a classifier.

To illustrate the model, consider the three sample text documents below:

1. *The dog likes to play with the ball.*
2. *The boy likes to play ball with the dog.*
3. *The cat likes to sleep. The dog likes to sleep too.*

There are 11 unique words in these three documents (ignore capitalization). Therefore the dictionary is of size 11, and is constructed as follows:

{
"the" = 1
"dog" = 2
"likes" = 3
"to" = 4
"play" = 5
"with" = 6
"ball" = 7
"boy" = 8
"cat" = 9
"sleep" = 10
"too" = 11
}

Using the indexes of the dictionary, each document can be represented by an 11-entry vector:

[2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]

[2, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]

[2, 1, 2, 2, 0, 0, 0, 0, 1, 2, 1]

where each entry of the vectors refers to the count of the corresponding entry in the dictionary. For example, in the first vector (which represents document 1) the first entry is “2”. This corresponds to the word "the" - the first word in the dictionary -and its value is "2" because "the" appears in the first document two times. Likewise, the second entry in the first vector is “1” indicating that the second word in the dictionary – “dog” – appears once in the document. An entry of “0” in a vector indicates that the corresponding entry in the dictionary does not appear in that document. A key aspect of the bag-of-words model and this vector representation is that the order of the words in the original sentences is **not** preserved.

@author Brian Hassett

Dealing with Sparse Arrays:

When a *term-document matrix* is constructed by combining the three vectors representing the training documents we have an array with three rows and 11 columns. As we scale up the size of our training set the inefficiencies of this representation become apparent. To illustrate, say we have 10,000 training documents with a corresponding dictionary of 20,000 words. Our term-document matrix is now a 2-D array of size 10,000 x 20,000. If we are dealing with short sentences, in a vector containing 20,000 entries there may be only 10 or 20 entries that actually give us any information!

To avoid such sparse arrays, this Java implementation uses a map to represent each training document. Each map only has as many entries as its corresponding training document, whereas before we had a vector with the same number of entries as the dictionary. The key (an integer) for each map is the dictionary index for the corresponding feature in the training document. The value (also an integer) is the count for that feature in the training document. For example, the first document which was previously represented by the following vector:

[2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]

can now be represented by the following map:

{1=2, 2=1, 3=1, 4=1, 5=1, 6=1, 7=1}

Now only those words that appear in the document are included in the vectorised representation. This method of representing the documents was found to vastly outperform the sparse array representation (for ~16,000 training documents, the training time goes from ~3 minutes to ~3 seconds).

Another method known as the hashing trick can be used to avoid sparse arrays. This method also circumvents the need for a dictionary by directly mapping words to indices with a hashing function. However, for this implementation the use of a dictionary is preferred and the map representation described above is used.

@author Brian Hassett

Naïve Bayes Text Classification (mostly taken from Manning et al (2009))

In text classification, we are given a description $\boldsymbol{d} \in \mathbf{X}$ of a document, where $\mathbf{X}$ is the *document space*; and a fixed set of *classes* $\mathbf{C} = \{c1, c2, \ldots, cJ\}$. Classes are also called *categories* or *labels*. Typically, the document space $\mathbf{X}$ is some type of high-dimensional space, and the classes are human-defined for the needs of an application. We are given a *training set* $\mathbf{D}$ of labelled documents $\langle d, c\rangle$, where $\langle d, c\rangle \in \mathbf{X} \times \mathbf{C}$. For example:

$$\langle d, c\rangle = \langle \text{excellent movie with great actors, } \textit{positive}\rangle$$

represents the one-sentence document "excellent movie with great actors" and the class (or label) "China".

Using a learning algorithm we then wish to learn a classifier or classification function $\boldsymbol{\gamma}$ that maps documents to classes:

$$\gamma: X \rightarrow C$$

In this implementation, a learning algorithm based on the *multinomial* Naïve Bayes model is used. Class membership for a document is decided by assigning it to the class with the *maximum a posteriori* probability, which is computed as follows:

$$C_{map} = \underset{c \in C}{\operatorname{argmax}}\ P(d|c)P(c)$$

As we are using a multinomial model, we will interpret the term $\boldsymbol{P(d|c)}$ as:

$$P(d|c) = P(\langle x_1, x_2, x_3, \ldots, x_n\rangle|c)$$

where $\langle x_1, x_2, x_3, \ldots, x_n\rangle$ is the sequence of terms as it occurs in the tokenized document $\mathbf{d}$. In fact, the document space $\mathbf{X}$ is comprised of all possible sequences of terms after tokenization, and we represent each document as $\langle x_1, x_2, x_3, \ldots, x_n\rangle$.

It is unfeasible to estimate $\boldsymbol{P(d|c)}$ as formulated above without a very large training corpus, but by making two assumptions we can make the model more tractable. Firstly, by employing the bag-of-words representation of our training corpus we assume positional independence of the words in each document. Secondly we make the naïve assumption that the feature probabilities $\boldsymbol{P(x_i|c_j)}$ are independent given the class $\mathbf{c}$. We can now rewrite $\boldsymbol{P(d|c)}$:

$$P(d|c) = P(x_1|c)P(x_2|c)P(x_3|c) \ldots P(x_n|c)$$

i.e. $P(d|c) = \prod_{x \in X} P(x|c)$

By substituting this alternative formulation of ***P(d|c)*** into the equation for C_{map} we have C_{NB}, i.e. the most likely class based on our assumptions:

$$C_{NB} = \underset{c \in C}{\text{argmax}}\ P(c) \prod_{x \in X} P(x|c)$$

In building the learning algorithm it is helpful to use a set which we will term ***positions*** to represent all word positions a document:

positions ← all word positions in test document

We can now rewrite $\mathbf{C_{NB}}$ as:

$$C_{NB} = \underset{c_j \in C}{\text{argmax}}\ P(c_j) \prod_{i \in positions} P(x_i|c_j)$$

and proceed with the implementation.

@author Brian Hassett

Learning Algorithm and Implementation

We now define a method to actually calculate the maximum likelihood estimates; as per Juarafsky we can use *frequencies* in the data to do this. The below table illustrates how the two estimated probabilities - $\hat{P}(c)$ and $\hat{P}(w_i|c_j)$ - are calculated, and what Java classes and fields in the implementation are used to represent them:

Parameter

$$\hat{P}(c) = \frac{docCount(C = c_j)}{N_{doc}}$$

Description

The number of documents in the training corpus labelled c_j divided by the total number of documents.

Java classes/fields

`Category.frequency` - a double representing the prior probability for the class.

Parameter

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

Description

The fraction of times word w_i appears among all words of class c_j. To account for the case whereby a term which hasn't been seen in the training data appears in a test document we apply Laplace smoothing. This involves adding 1 to the numerator and |V| (the size of the dictionary) to the denominator. This way we avoid probabilities of zero for unseen terms.

Java classes/fields

`Category.frequency` - a double representing the prior probability for the class.

`NB_Classifier.wordFrequencies` - a list of arrays of doubles. Each array of doubles represents a document in the training corpus, and each entry in that array represents $P(w_i|c_j)$ for each word in the document.

@author Brian Hassett

The below example taken from illustrates steps required to calculate these two parameters and ultimately C_{NB}.

The following table shows a training corpus with four training documents each given a label from {c, j} where "c" corresponds to "Chinese" and "j" corresponds to "Japanese":

	doc	Words	Class
Training	1	Chinese Beijing Chinese	c
	2	Chinese Chinese Shanghai	c
	3	Chinese Macao	c
	4	Tokyo japan Chinese	j
Test	5	Chinese Chinese Chinese Tokyo Japan	?

Document number 5 (d5) is a test document which has no class assigned as yet. To assign a class to d5 we calculate the prior probabilities and conditional probabilities for each word in the test document:

Priors:

P(c) = 3 / 4

P(j) = 1 / 4

Conditional probabilities:

P(Chinese|c) = (5 + 1) / (8 + 6) = 6 / 14 = 3 / 7

P(Tokyo|c) = (0 + 1) / (8 + 6) = 1 / 14

P(Japan|c) = (0 + 1) / (8 + 6) = 1 / 14

P(Chinese|j) = (1 + 1) / (3 + 6) = 2 / 9

P(Tokyo|j) = (1 + 1) / (3 + 6) = 2 / 9

P(Japan|j) = (1 + 1) / (3 + 6) = 2 / 9

Choosing a class:

$P(c|d5) \propto 3 / 4 * (3 / 7)^3 * 1 / 14 * 1 / 14 \approx 0.0003$

$P(j|d5) \propto 1 / 4 * (2 / 9)^3 * 2 / 9 * 2 / 9 \approx 0.0001$

Given these values for we can now choose ***P(c|d5)*** as the most likely class.

For full implementation details the java source code can be referenced. One additional point to note is that to prevent floating-point underflow the Java implementation uses log space in the model. Since log(ab) = log(a) + log(b), we can write:

$$C_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \, logP(c_j) + \sum_{i \in positions} logP(x_i|c_j)$$

and the chosen class is the class with the highest log probability score.