home

Bag of Words (BoW) - Natural Language Processing

2014-12-05 | **Explanation** | machine-learning, natural-language-processing

Introduction

Bag of Words (BoW) is a model used in natural language processing. One aim of BoW is to categorize documents. The idea is to analyse and classify different "bags of words" (corpus). And by matching the different categories, we identify which "bag" a certain block of text (test data) comes from.

Putting into context

One excellent way to explain this is to put this model into content. One classic use of BoW is for spam filtering. Through the use of the BoW model, the system is trained to differentiate between spam and ham (actual message). To extend the metaphor, we are trying to guess which bag the document comes from, the "bag of spam" or the "bag of ham".

Note: I will not be explaining the logic behind how the spam filter works (though I might do it in a different post). I am just giving the example so you can understand the rationale of categorizing different text.

How BoW works

Forming the vector

Take for example 2 text samples:
og
and `Never jump over the lazy dog qui`

The corpus(text samples) then form

did you find what you need?
0 1 2 3 4 5 6 7 8 9 10
Not likely at all Extremely likely
Not using Hotjar yet? Send

```
{
    'brown': 0,
    'dog': 1,
    'fox': 2,
    'jump': 3,
    'jumps': 4,
    'lazy': 5,
    'never': 6,
    'over': 7,
    'quick': 8,
    'quickly': 9,
    'the': 10,
}
```

Vectors are then formed to represent the count of each word. In this case, each text sample (i.e. the sentences) will generate a 10-element vector like so:

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

Each element represent the number of occurrence for each word in the corpus(text sample). So, in the first sentence, there is 1 count for “brown”, 1 count for “dog”, 1 count for “fox” and so on (represented by the first array). Whereas, the vector shows that there is 0 count of “brown”, 1 count for “dog” and 0 count for “fox”, so on and so forth

Weighting the terms: tf-idf

As of most languages, some words tend to appear more often than other. Words such as “is”, “the”, “a” are very common words in the English language. If we take consider their raw frequency, we might not be able to effectively differentiate between different classes of documents.

A common fix for this is to use a statistical method known as the tf-idf to make the data more accurate, reflecting the context of the text sample better. TF-IDF, short for term frequency-inverse document frequency takes into account 2 values: **term freque** **frequency**(idf).

There are a few different ways to n way to determine the value of the ake the raw frequency of a term divide m in the document, like so:

did you find what you need?

0 1 2 3 4 5 6 7 8 9 10

Not likely at all Extremely likely

Not using Hotjar yet? Send

```
0.5 + 0.5 * freq(term in document)/max(freq(all word in document))
```

One common way to determine the **inverse document frequency** is to take the log of the inverse of the proportion of documents containing the term, like so:

```
log( document_count/len(documents containing term) )
```

And by multiplying both values, we get the magic value, **term frequency-inverse document frequency (tf-idf)**, which reduces the value of common words that are used across different documents.

An additional step after obtaining is the tf-idf is to normalize the vector, will makes it less troublesome to apply different operators to.

Taking it further: Feature hashing / Hashing trick

The basic concept explained here may work for small sample of text where the dictionary size is rather small. But for the actual training process, there are text with tens of thousands of unique words, we would need some way to represent the document more efficiently.

By hashing the terms(i.e. the individual words), we obtain a index, which corresponds to the element in the generated vector. So instead of having to store the words in a dictionary and having a ten-thousands-elements long vector for each document, we have a N-size vector instead(N determined by whoever makes the decision).

To account for hash collisions, an additional hash function would then be implemented as a operator choosing function (returns 1 or 0). This helps ensure that when different entries collide, they will cancel each other out, giving us a expected value of 0 for each element(More on that here). And with that, the final vectors of the documents would then be used to classify different types of documents(e.g. spam VS ham).

For those who see better through c ook like:

```
function hashing_vectorizer(feature
    x = new Vector[N]
    for f in features:
```

did you find what you need?

0 1 2 3 4 5 6 7 8 9 10

Not likely at all Extremely likely

Not using Hotjar yet? Send

```
        h = hash(f)
        ### sign operator
        if hash2(f):
            x[h%N] += 1
        else:
            x[h%N] -= 1
    return x
```

Conclusion

After the BoW is completed, what is obtained would be a vector for each individual document. These documents will then be passed through different machine learning algorithms to determine the features that separates the different documents.

That is where the actual "machine learning" comes in. BoW is basically just a tool to convert text documents into a vector that describes its features and content.

Note: Do drop me a note if there is any unclear portions. Trust me, I will get better at this.

Readings:

- http://deeplearning4j.org/bagofwords-tf-idf.html
- http://en.wikipedia.org/wiki/Tf%E2%80%93idf
- http://en.wikipedia.org/wiki/Bag-of-words_model
- http://en.wikipedia.org/wiki/Feature_hashing

Categories: Explanation
Tags: machine-learning, natural-language-processing

<-- home

did you find what you need?

0 1 2 3 4 5 6 7 8 9 10

Not likely at all Extremely likely

Not using Hotjar yet? Send