Document Classification Part 2: Text Processing (N-Gram Model & TF-IDF Model)

Chan Woo Kim Feb 23, 2018 · 7 min read ★

Photo by Luca Bravo on Unsplash

In this article I will explain some core concepts in text processing in conducting machine learning on documents to classify them into categories. This is the part 2 of a series outlined below:

Part 1: Intuition & How Do We Work With Documents?

Part 2: Text Processing (N-Gram Model & TF-IDF Model)

Part 3: Detection Algorithm (Support Vector Machines & Gradient Descent)

Part 4: Variations of This Approach (Malware Detection Using Document Classification)

In part 1, we went over representing documents as numerical vectors. But there are many problems with this simplistic approach. Let me put what we ended up with in the last article:

```
Sentence 1 (S1): “vectorize this text to have numbers!”
Sentence 2 (S2): "what does it mean to vectorize?"
Sentence 3 (S3): "document classification is cool"
```

Then we created the vocabulary set:

```
V = ['classification', 'cool', 'document', 'does', 'have', 'is',
'it', 'mean', 'numbers', 'text', 'this', 'to', 'vectorize', 'what']
```

Then we represented each sentence as a vector where each index corresponds to the index in the vocabulary set, where the number is how many times the word occurs in that sentence:

```
S1 = [0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0]
S2 = [0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]
S3 = [1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

This is pretty cool and we will still use this approach in the end, but can you figure out what the glaring problem is?

There is no concept of order or sequence here! Ordering things differently can create the meaning of sentences dramatically but this approach does not take this into account. For instance:

```
S1 = "Car was cleaned by Jack."
S2 = "Jack was cleaned by Car."
```

We can all agree that these sentences are saying pretty different things and our mental image of them contrast drastically. But according to our model:

```
V = ["by", "car", "cleaned", "Jack", "was"]
```

…they are the exact same sentences!

This is not okay, those are different sentences — sequences of words are important as well, not just if certain words are present. Let me introduce a model that takes a step towards fixing this issue.

N-Gram Model

Under the n-gram model, a vocabulary set like:

```
V = ["by", "car", "cleaned", "Jack", "was"]
```

would be considered a set of ***uni-grams.*** If you haven't guessed already, the ***'n'*** in the ***n***-gram approach represents a number, and it represents how many words there are in one gram. For instance, if we take our set of sentences again:

```
S1 = "Car was cleaned by Jack."
S2 = "Jack was cleaned by Car."
```

a **bi-gram** approach will create a vocabulary set like the following:

```
V = ["car was", "was cleaned", "cleaned by", "by jack", ... ,"by
car"]
```

A **bi-gram** is a sequence of two words. With this n-grams approach, our previous conundrum is a nonissue:

```
V = ['by car', 'by jack', 'car was', 'cleaned by', 'jack was', 'was
cleaned']
```

```
S1 = [0 1 1 1 0 1]
S2 = [1 0 0 1 1 1]
```

Now they are different, unlike how it was when we first began, which is to say that the program now recognizes that those are two pretty different sentences because it now takes into account the sequences of words not just singular words.

It doesn't need to be limited to bi-grams. It can be trigrams, four-grams, five-grams, whatever you think is necessary for your specific document classification task:

```
# 4 grams:
V = ['car was cleaned by', 'jack was cleaned by', 'was cleaned by
car', 'was cleaned by jack']

# 5 grams:
V = ['car was cleaned by jack', 'jack was cleaned by car']

# Combination of 4 and 5 grams:
V = ['car was cleaned by', 'car was cleaned by jack', 'jack was
cleaned by', 'jack was cleaned by car', 'was cleaned by car', 'was
cleaned by jack'
```

This n-gram model is integrated in most document classification tasks and it almost always boosts accuracy. This is because the n-gram model lets you take into account the sequences of words in contrast to what just using singular words (unigrams) will allow you to do.

TF-IDF Weighting

Okay, so now we take into account sequences of words. But let us explore this example of classifying sentences to be either a positive or a negative review:

```
S1 = "This movie is bad"
S2 = "This movie is good"
```

in common except for just one word. That one word ("good", "bad") changes its category dramatically. But with our current model, our classifiers take each word as seriously as it does with every other word, which is to say that each word is ***considered*** the same, when it should really be that the word "good" and "bad" should be considered more.

But how do we do that? If you are working with thousands of documents that are thousands of words long (which is most cases when you would want to use machine learning to classify documents), you do not have the luxury for a human being to sort through and find the most significant words. Even if we did that, even though some word are more worth noting than others, how do we possibly quantify that?

This is where TF-IDF weighting comes in and it is a very popular and standard tool in document classification. It stands for Term Frequency-Inverse Document Frequency. The formula is the following:

$$tf(w,d) = f_d(w)\text{: frequency of } w \text{ in document } d$$
$$idf(w,D) = log\frac{1+|D|}{1+df(d,w)}$$

$$tfidf(w,d,D) = tf(w,d) \times idf(w,D)$$

Do not be scared by the formulas! You do not yet know what certain symbols represent so you'd have to be magical to understand it already.

Let us first explore what 'tf(w,d)' is. This essentially is what we have been working with before in our previous examples before we introduced TF-IDF weighting: the number of times certain words (w) appear in a document (d). Remember that our numerical representation of text was just based on how many times certain words appeared in a document, indexes corresponding to the vocabulary set. That is just what the tf(w,d) value is, this is nothing new.

The absolute value sign on 'D' represents the size of the corpus, how many documents there are in total. In the bottom, 'df(d,w)' , represents how many ***documents*** the ***word*** appears in. We then end up with a logarithmically scaled value of the number of documents in the corpus divided by the number of times word w appears throughout the corpus for the idf(w, D) value.

Let me explain that in english. The intuition is this: if a certain word occurs everywhere throughout all the documents, it is less likely that it will be significant. Think about it, the word "the" will appear in almost all documents without giving anybody any insight if it belongs to a positive or a negative review. And because that is true, we can assume, to an extent, that it is less significant than other words, and therefore should be given less consideration. So to do this, the denominator in the IDF value is the number of documents a certain word appears in, and as that number gets greater, the IDF value decreases because the value in the denominator is getting bigger (1/2 is greater than 1/6)! So a word that is prevalent throughout the entire corpus will have a lesser TFIDF value because the IDF value it would multiply with the TF value will be smaller than others.

But How Does This Work With Vectors?

See, it's pretty intuitive how this value is calculated, but how does it apply to vectors? How do we apply this to our current representation of sentences?

We simply put the TF-IDF values instead of just pure counts of how many times certain words (or n-grams) appear in a sentence. We then normalize these values using the Euclidean norm:

$$v_{norm} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}}$$

With our example:

```
V = ['by car', 'by jack', 'car was', 'cleaned by', 'jack was', 'was
cleaned']

# TF values
S1 = [0 1 1 1 0 1]
S2 = [1 0 0 1 1 1]

# TF-IDF weighted:
S1 = [0, 1.40546511, 1.40546511, 1, 0, 1]
S2 = [1.40546511, 0, 0, 1, 1.40546511, 1]
```

And taking one of the values as an example and applying the normalization:

$$\frac{1.40546511}{\sqrt{1.40546511^2+1.40546511^2+1^2+1^2)}} = 0.57615236$$

```
S1 = [ 0, 0.57615236, 0.57615236, 0.40993715, 0, 0.40993715]
S2 = [ 0.57615236, 0, 0, 0.40993715, 0.57615236, 0.40993715]
```

The value of normalization is put simply by the following:

> *"leaving variances unequal is equivalent to putting more weight on variables with smaller variance" — Franck*

Now we not only have a numerical representation of our data, we have a weighted representation that also takes into account sequences of words. All this process has been leading up to finally training the model based on this data. This exciting concept will be the topic of the next article!

Python Implementation

I know when I wanted to learn about this stuff I wanted to know how to implement this myself in Python. I won't get into stuff about implementing the entire machine learning process here but just what I've outlined in this article. Each article in this series will have a sample python implementation doing tasks discussed in the article. This code is actually what I used to compute some of the demonstrations shown earlier in the article.

```
2
3   arr = ["Car was cleaned by Jack",
4          "Jack was cleaned by Car."]
5
6   # If you want to take into account just term frequencies:
7   vectorizer = CountVectorizer(ngram_range=(2,2))
8   # The ngram range specifies your ngram configuration.
9
10  X = vectorizer.fit_transform(arr)
11  # Testing the ngram generation:
12  print(vectorizer.get_feature_names())
13  # This will print: ['by car', 'by jack', 'car was', 'cleaned by', 'jack was', 'was cleaned']
14
15  print(X.toarray())
16  # This will print:[[0 1 1 1 0 1], [1 0 0 1 1 1]]
17
18  ## And now testing TFIDF vectorizer:
19  vectorizer = TfidfVectorizer(ngram_range=(2,2)) # You can still specify n-grams here.
20  X = vectorizer.fit_transform(arr)
21
22
23  # Testing the TFIDF value + ngrams:
24  print(X.toarray())
25  # This will print:  [[ 0.          0.57615236  0.57615236  0.40993715  0.          0.40993715]
26  # [ 0.57615236  0.          0.          0.40993715  0.57615236  0.40993715]]
27
28
29  ## Testing TFIDF vectorizer without normalization:
30  vectorizer = TfidfVectorizer(ngram_range=(2,2), norm=None) # You can still specify n-grams here
31  X = vectorizer.fit_transform(arr)
32
33  # Testing TFIDF value before normalization:
34  print(X.toarray())
35  # This will print: [[ 0.          1.40546511  1.40546511  1.          0.          1.        ]
36  # [ 1.40546511  0.          0.          1.          1.40546511  1.        ]]
```

tfidf.py hosted with ❤ by **GitHub** **view raw**

Naturallanguageprocessing NLP Python Machine Learning Classification