DIGITAL CULTURE
ITMO UNIVERSITY

Information Retrieval

High School of Digital Culture
ITMO University
dc@itmo.ru

Contents

1 Bag of Words and Zipf's Law

The last time we were discussing natural language processing. We reviewed its history and learned to normalize words and prepare texts for advanced processing.

This time we will look closely at a couple of tools used for natural language processing.

The processing of a large text set requires either dealing with the concept of text similarity or using it in practice.

Suppose, for example, that we want to find texts of a similar topic or those that describe one thing. For a computer, a piece of text is nothing but a string, a sequence of characters. However, we've learned that a text represented as a string can be tokenized and that a lemmatizer can return the base or dictionary form of each word in the text.

Thus, what we have is not just a string but a sequence of tokens. What texts are similar? The answer to this question depends on the problem posed. Let's start with a simple approach. If the texts are about the same subject, the sets of words in these texts will likely intersect. The more matching words, the more similar the texts.

But is it always the case?

You're probably thinking that we can use different words to express the same idea and use synonyms, i.e. words that have the same or nearly the same meaning as another, for example, 'big' and 'large'. It is a challenging problem that requires different solutions depending on a task. We will not dwell upon it this time. However, we would like to note that simple approaches sometimes well handle texts that are large enough.

Let's take three texts for ease of demonstration. The original texts are available on `https://nplus1.ru/`:

1. Astrophysicists have estimated the sizes of gravitational lenses to refine the Hubble constant, which describes the universe's expansion rate.

2. The algorithm detected not the most probable diagnosis, but three most probable out of 26 possible by assigning weights to them.

3. Astrophysicists have detected a pulsar which has the smallest orbit and is the brightest in the universe.

The first and third texts are about space exploration, and the second is not. For us, it's clear because of the terminology used and some keywords, for example, astrophysicists, gravitational, pulsar, diagnosis. So, our task is to make it clear for a machine too.

The texts are too short, and the examples are synthetic. Anyway, we see that the chosen approach works well. The first and third texts use the same two words that are the field-specific terms. The first and the second texts have no matching meaningful words (except articles and prepositions). But the second and third include the same instance (the word detected).

Even short texts describing different things may contain identical words — usually pronouns, numerals, prepositions, conjunctions, auxiliary verbs, and so on. They are either function words or frequently used words. Because of that, compared text sets almost always intersect. However, intersecting grammatical words say nothing about meaningful similarities. That's why they are called stop words. The common practice is to neglect them or filter them out.

Well, let's take a look at an important law. Zipf's law is not a strict rule. It's rather an empirical observation popularized by George Kingsley Zipf. Before Zipf, the German physicist Felix Auerbach and French stenographer Jean-Baptiste Estoup appear to have noticed the regularity.

Zipf's law states that if we take a large set of texts in a natural language, calculate the frequencies of each word, and sort the words by their frequency in descending order, the frequency of a word will be inversely proportional to its rank in the frequency table. If we plot the word frequencies along the vertical axis and the words along the horizontal axis, we will see a similar picture for any language. The graph shows the most frequent words of Wikipedia in Russian. If we take the logarithm of the frequency and the word number in the list, the graph will be different. Each curve corresponds to a language in the legend. The frequencies are also derived from Wikipedia.

Probably you'll agree that such an invariant for different languages is an interesting observation. But what does it mean to us?

Firstly, some words can be found in almost any text. They are shown on the left. As you might have guessed, it's a stop-word region.

Secondly, the long tail on the right indicates words that are absent in the dataset. Thus, there should be a way to process such words. When solving problems of subject classification, subject modeling, and information retrieval, we are looking for words somewhere in between these two extremes.

Let's return to the word comparison. Now you know that the most frequent words should be neglected. Besides, it is useful to compare not only the fact of the word presence in a text but also the number of its occurrences. Thus, the goal is to consider texts as multisets instead of sets of words. The representation of the text as a multiset of its constituent tokens is called the bag-of-words model.

Simple problems of text classification, retrieval, and modeling require only this model. The subsequent steps depend on representations and algorithms.

You're probably wondering why we use such a simple, unintelligent model.

After all, this model doesn't consider the word order that can be very useful sometimes. Think, for example, about the proper name "New York". The word order is necessary to process the city name correctly. However, a bag-of-words model will lose this meaningful information because it will separately count each word in the city name. Because of that, a machine will not understand that "New" is a part of the city name and will treat it as the adjective in the phrase "new dress". A machine will not be able to process all the word sequences due to their high number. The good news is that some neural networks can handle word sequences. However, we will skip this theme in this course.

Well, there is a compromise solution that can be called a "charity" bag of words. We will not look at tokens (or look not only at tokens) but at the token pairs that appear in the text in a row. They can also be triples, and so on. Such subsequences of N tokens are called n-grams. The model processing a text as an N-gram multiset is called the N-gram bag.

Here, it's important to choose the right N. If N is large (for example, twenty), even an internet search will yield poor results due to the lack of texts containing such engram. Meanwhile, too small N (for example 1) will through us back to the bag of words and result in the information loss of the model.

2 Boolean search

Internet search is one of the most well-known and successful natural language processing systems. Web search offers different tools, such as image searching or music recognition using, for example, Shazam. This time, we will only consider text retrieval because it is relevant for the course.

Information retrieval is sometimes attributed to natural language processing tasks, even though not all typical search tasks are language-related. We cannot deny that they have a lot of in common, and many important concepts, methods, and scientific practices came to natural language processing from the information retrieval science. Anyway, let's not dive into history.

From a practical point of view, the most useful information retrieval task is ad-hoc search. It aims to return the most suitable, i.e. relevant, documents for a given query. In our case, such documents are texts. This example is easy to understand because it explains how top search engines work.

Well, how to develop a search engine? We can use a bag-of-words model and, for example, process queries as a text, or as a propositional formula that decides what terms should or should not be present in a document we are looking for. The approach that presents all documents and a query as sets, evaluates the similarity between each document and query, as well as provides the most relevant documents is called Boolean search.

By the way, how can we evaluate the similarity of two sets? Since we are going to compare each set with another pairwise, it would be convenient if the similarity of the two sets is expressed by a number. Thus, we will be able to use the obtained number for further sorting. A common way to do it is the Jaccard similarity index (sometimes called the Jaccard similarity coefficient). For two sets, we simply divide the ratio of the cardinality (the size) of the intersection between the sets by the cardinality of the union between the sets.

In other words, the larger the rate of elements shared by both sets, the greater the Jaccard index. When sets perfectly match, the union equals the intersection, and the coefficient is one. With no shared elements, the intersection cardinality is zero, and, consequently, the resulting similarity is zero too.

Here we have a term-document matrix that we will use later in this course. Each row corresponds to a document represented as a set, each column corresponds to a term of the dictionary (a set of all possible terms). The first document contains the terms "nail" and "tellurium" and no "nickel" or "to dig". That explains how the positions are filled out with the ones and zeros.

Firstly, on paper, we can run a query for all the lines, for example, ""nail" AND ("tellurium" OR "nickel") to extract all the documents satisfying this query.

Secondly, to calculate the intersection between the sets of document terms, we can multiply the lines of the necessary documents by the corresponding coordinates and sum them up. Thus, we can calculate the dot product of the corresponding rows of the matrix. The union can be calculated similarly. It means that the Jaccard index of the query and the considered document (or document pairs) is in the bag.

You'd think that we had everything we need for a simple search engine, including document and query representations, as well as a measure of similarity between a query and document. The only thing remained was to calculate this measure.

Unfortunately, it's not so easy in reality because the number of documents is usually large.

The graph shows the estimated change in size of the World Wide Web part being searched by Google. The plan to calculate the Jaccard index 60 billion times and sort the documents based on the obtained number doesn't sound like a good idea.

That's why document search uses an entirely different approach. For each term, we keep a sorted list of documents containing this term.

Additional information is also often stored in this list along with the document number. For example, the position in the text (to consider the proximity of terms requested by the query), the actual form of a word, the number of times it appears in the text, and so on).

A dictionary (a term list) can be stored in memory, and document lists are often stored on disk. The AND operator in the Boolean expression corresponding to the query leads to the intersection between the document lists, and the same is done for the ordered sets in linear time depending on their length. The OR operator leads to the union of sorted lists, or, to put it differently, an algorithm of the same complexity.

There is still room for improvement of this structure, both in terms of performance and memory usage. However, some truly amazing solutions can save CPU time and disk usage. For example, lists are often loaded with additional links to several items ahead. Think, for example, how this helps in calculating the intersection between lists.

Having the sorted lists, we can save resources by storing only the number of the first documents and then keeping only the differences instead of the complete identifiers. Etc.

Developers made a lot of effort to decrease storage demand and increase performance. So, if you ever want to do something like this, explore existing solutions first. Many free open-source tools offer effective solutions.

Well, suppose that we collected and indexed data to run a query. Using the methods discussed earlier, we extract potentially relevant documents.

3 Ranking

Ranking, i.e. document ordering or sorting, is, so to say, the brain of a search engine and, perhaps, the most important feature of the latter. Those who used search engines in the early 2000s well remember why you cannot rely only on keyword intersections only. Some website owners may use dirty handed tactics of adding only popular keywords to their webpages to deliver irrelevant or fraudulent content.

We can overcome all these challenges with a ranking formula that considers all the variables.

We have already discussed one classical theoretical model of information retrieval called Boolean search. So, let's consider a VECTOR SPACE MODEL. Documents and queries in this model are represented as vectors, and the relevance of documents is calculated using a distance measure based on the proximity. In theory, we make a vector for each document in the database and a vector for each incoming query, then we try to find, for example, ten closest document vectors (based on a certain distance) and provide them to a user.

In practice, it's very different because a mindless search for the nearest neighbors among several billion vectors is not manageable. Finding the closest neighbors is a resource-consuming activity. We will discuss how to embed a

vector space model in a real search a bit later.

Vectors are made in different ways. Being familiar with a bag-of-words model, we can construct a long vector where a cell corresponds to a word and a cell value corresponds to the word frequency in the document or query. Alternatively, we can also use one instead of the frequency if the term occurs in the text and zero otherwise. Well, the simplest text and query representation to build a vector model is ready.

The dimensionality of such a vector equals the dictionary size, i.e. the number of unique terms in hand. Usually, it amounts to dozens or hundreds of thousands.

By the way, there's no difference for a machine whether vectors are long or short. Most of the vector values are zeros, and nowadays there are many effective ways to represent the data of this type. For example, we can store a list of non-zero values, as in the inverted indexes considered earlier. You can find the examples of sparse matrices and vector representation in the sparse module of the SciPy package.

The slide shows an example of a vector part corresponding to the given text. There, stop words were excluded and terms were lemmatized. Keep it in mind. We will return to it later.

We've learned that some words are more important than others. Let's move one step away from common counters and frequencies and create a more complex formula that considers the importance of a word both in the entire corpus and the document.

4 TF-IDF

No doubt that a word appearing in almost every document doesn't contribute to the search. Usually, a rare word is highly valuable if it is not a typo. For example, a small number of documents may contain domain-specific terms. Thus, we can balance the term frequency in a document with a respective penalty.

For example, we calculate the ratio of the total number of documents in our database and the number of documents containing the term, then we take the logarithm of this value. The less often the term occurs in the corpus, the greater the value will be, and vice versa.

Multiplying the penalty by the relative term frequency in the document leads us to the well-known ftf-idf formula. Assume we have the following texts:

```
"... the managing director of the company for production ...."
"... financial director of ShoreCo LLC ..."
"... crisis affects all life domains..."
```

```
"... Deputy Director for Science of the FAQ Research Institute..."
"... being a director of a travel agency in a crisis time ..."
```

Real collections usually include thousands of texts, but we will consider this synthetic example. The word "director" is very common in articles about science, industry, and technology. In our case, this term is found in 80% of texts, and its value for the search is small. Fewer articles are about the crisis (after all, it will not last forever). In the example, the word "crisis" occurs less often. In reality, 40% is a significant value but bear in mind that it is still a synthetic example.

Assume that each of these pieces of text is a document. Let's calculate the tf-idf of the term DIRECTOR in the last document. To begin with, we calculate the IDF since the term DIRECTOR is found in 80% of the documents. Hence, the IDF is the logarithm of five fourths. As for the TF, the number of references in the document is divided by the size of the document, that is, one fifth. Thus, if we take the binary logarithm, the tf-idf approximately equals six hundredths.

In the same way, we calculate the tf-idf for the term "crisis" in the same document. Its rate in the documents is forty percent, that is, the IDF is the logarithm of two and one fifth. And the TF value is the same as that of the director. The result is more than twenty-six hundredths, which is more than the tf-idf of the director.

In addition to the intuitive reasoning, it also has a probabilistic interpretation. To obtain a larger sum of tf-idf values for all terms from the query for a text, we minimize for a given model the probability that all these terms appear in this document by accident.

Next, instead of usual frequencies, the term importance for the document will consider the general distribution and impose penalties for too frequent ones. A document represented as a vector with tf-idf weights better copes with finding the most relevant documents than a regular bag of words.

This general searching algorithm is common, but it is not the only one.

Raw texts are collected and uploaded into a database. Database indexation is carried out at intervals or continuously, which converts terms and texts, for example, in inverted indexes. Each query is converted into a query in an index.

For example, the query "green slippers" requires extracting the lists of intersecting documents related to the inverted index of the terms "green" and "slippers". The obtained list of documents is sorted by a ranking algorithm.

Ranking algorithms use not only tf-idf but also machine learning to calculate the relevance function. For example, we can use statistics of the most clickable elements for the most frequent queries. In this case, a search engine can provide a user not only the documents where the words "green" and "slippers" are found but also suggest movies or shops.

Then best results are preprocessed and returned to the user. Almost all stages require algorithms that can be related to natural language processing.

If we think about real-life usage, users, and search engines, we will soon realize that search is not limited to the discussed tasks and approaches. User queries often contain errors or typos that need to be corrected in a moment.

To increase the output volume, i.e. a set of documents provided to a user in response to a query, the query is often expanded with synonyms or alternative phrases that have similar meanings. It helps to preserve potentially useful texts that don't contain the original terms when texts are selected for ranking.

5 Personalization

Personalization. If, for example, you are a car dealer often searching on the internet for things related to cars, a request about Jaguar will return webpages about Jaguar cars, not animals or other brands with the same name.

When typing in a search bar, we often see suggestions picked by a search engine. If a guess is correct, we can click the link to save some time. It's extremely useful on mobile devices because every user wants to type less on a small keyboard. Such clever hints are given by a suggestion tool. Developing a good algorithm for this tool using the query statistics and things we can do with natural language is a challenge.

Often, instead of a webpage, we simply need an answer to the question. For example, if you ask, "When was the Woodstock Festival held?", Google will give you the dates. You will see the answer and source, and there's no need to check the webpage, which is very convenient. Solving such problems requires an ability to determine the type of the query, for example, a factual question. The next step is to convert the query into a structured one, such as "display the value of the field "Start date" and "End date ", where the event is equal to "Woodstock"", and then use the knowledge base prepared for that in advance. It relates to the tasks of natural language processing.

Because of spam, plagiarism, and complicated search engine optimization schemes, the same content is duplicated multiple times. Thus, there should be a way to find copies. Doing so will make the output readable and speed up the performance at all stages of searching.

Well, there are many searching tasks, and those that we discussed cannot be considered secondary. And we haven't even touched upon those that entire departments are trying to solve. There is plenty of work ahead of language processing professionals.

Information systems need a user interface. You've probably seen search bars labeled "On-Site Search". Sometimes it's enough to search for the exact

match, which any database can handle, but searching based on natural language processing methods is also common.

Of course, there's no need to develop from scratch the things we have discussed so far. Since ad-hoc searching is there for some time, you can use the ready-to-go and well-developed search algorithms and a wide range of ready--made free tools to create your search engine.

Probably the most popular open-source solutions for page collecting are Apache Software Foundation projects written in Java, such as Nutch for page collecting, Lucene for indexing and searching, and Solr search engine. The ElasticSearch system, which is based on Lucene, is also popular.

There are also Python solutions, such as Scrapy for collecting pages and Xapian for indexing and searching.