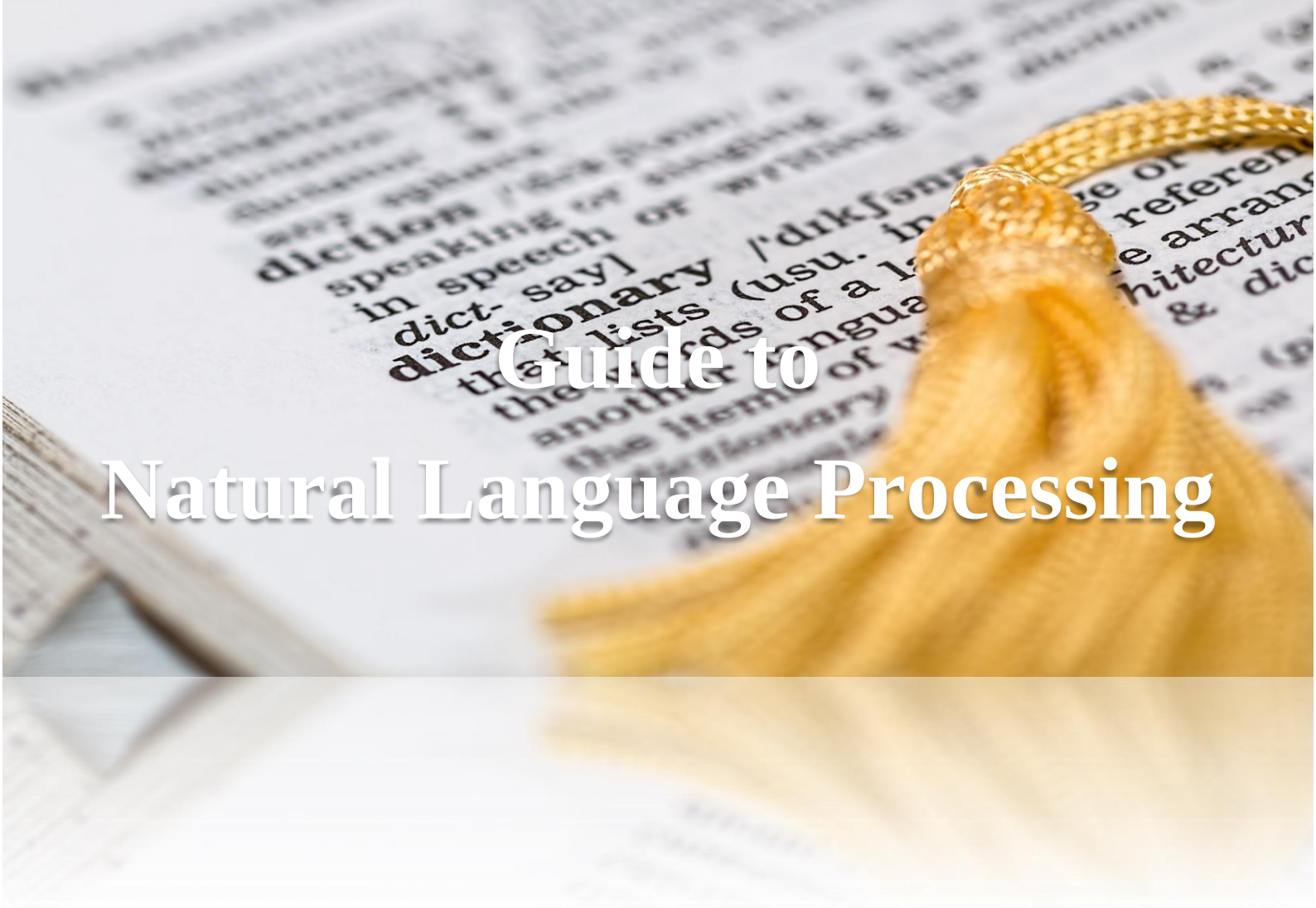

Guide to Natural Language Processing

Analyze and Understand Text

STRU
MEN
TA

Learn more at https://tomassetti.me

What Can You Use Natural Language Processing for?

Natural Language Processing (NLP) comprises a set of techniques that can be used to achieve many different objectives. Take a look at the following table to figure out which technique can solve your particular problem.

What You Need	Where To Look
Grouping similar words	Stemming, Splitting Words, Parsing Documents
Finding words with the same meaning	Latent Semantic Analysis
Generating realistic names	Splitting Words
Understanding how much time it takes to read a text	Reading Time
Understanding how difficult to read is a text	Readability of a Text
Identifying the language of a text	Identifying a Language
Generating a summary of a text	SumBasic (word-based), Graph-based Methods: TextRank (relationship-based), Latent Semantic Analysis (semantic-based)
Finding similar documents	Latent Semantic Analysis
Identifying entities (e.g., cities, people) in a text	Parsing Documents
Understanding the attitude expressed in a text	Parsing Documents
Translating a text	Parsing Documents

This article uses terms like parsing and understanding in a loose way, at least compared to traditional meaning of these terms. We are going to talk about parsing in the general sense of analyzing a document and extracting its meaning**.** So, we are going to talk about actual parsing of natural languages, but we will spend most of the time on other techniques. When it comes to understanding programming languages, to parsing is the way to go. However, for natural languages you can pick specific alternatives. In other words, we are mostly going to talk about what you would use instead of parsing, to accomplish your goals.

For instance, if you wanted to find all for statements a programming language file, you would parse it and then count the number of for *statements*. Instead, to find all mentions of cats in a natural language document you are probably going to use something like *stemming*.

This is necessary because the theory behind the parsing of natural languages might be the same one that is behind the parsing of programming languages, however the practice is very dissimilar. In fact, you are not going to build a parser for a natural language. That is unless you work in artificial intelligence or as researcher. You are even rarely going to use one. Rather you are going to find an algorithm that work a simplified model of the document that can only solve your specific problem.

In short, you are going to find tricks to avoid to actually having to parse a natural language. That is why this area of computer science is usually called **natural language processing** rather than natural language parsing.

Algorithms That Require Data

We are going to see specific solutions to each problem. Mind you that these specific solutions can be quite complex themselves. The more advanced they are, the less they rely on simple algorithms. Usually they need a vast database of data about the language. A logical consequence of this is that it is rarely easy to adopt a tool for one language to be used for another one. Or rather, the tool might work with few adaptations, but to build the database would require a lot of investment. So, for example, you would probably find a ready to use tool to create a summary of an English text, but maybe not one for an Italian one.

For this reason, in this article we concentrate mostly on English language tools. Although we mention if these tools work for other languages. You do not need to know the theoretical differences between languages, such as the number of genders or cases they have. However, you should be aware that the more different a language is from English, the harder would be to apply these techniques or tools to it.

For example, you should not expect to find tools that can work with Chinese (or rather the *Chinese writing system*). It is not necessarily that these languages are harder to understand programmatically, but there might be less research on them or the methods might be completely different from the ones adopted for English.

The Structure of This Guide

This article is organized according to the tasks we want to accomplish. Which means that the tools and explanation are grouped according to the task they are used for. For instance, there is a section about measuring the properties of a text, such as its difficulty. They are also generally in ascending order of difficulty: it is easier to classify words than entire documents. We start with simple information retrieval techniques and we end in the proper field of natural language processing.

We think it is the most useful way to provide the information you need: you need to do X, we directly show the methods and tools you can use.

Table of Contents

The following table of contents shows the whole content of this guide.

Classifying Words 5
Grouping Similar Words 5
Stemming 5
Splitting Words 7
Classifying Documents 9
Text Metrics 9
Reading Time 9
Calculating the Readability of a Text 10
Identifying a Language 11
Understanding Documents 14
Generation of Summaries 14
SumBasic 15
Graph-based Methods: TextRank 15
Latent Semantic Analysis 17
Other Methods and Libraries 18
Other Uses 18
Parsing Documents 19
You Need Data 19
What You Can Do 20
The Best Libraries Available 22
Summary 26

Classifying Words

With the expression classifying words, we intend to include techniques and libraries that group words together. Some people consider these techniques more part of information retrieval than natural language processing. We think it depends on the intent of the developers. Sure, they are used in information retrieval, but they are also fundamental to make advanced natural language processing algorithms work well.

Grouping Similar Words

We are going to talk about two methods that can group together similar words. They are very used in information retrieval (i.e, to make easier finding the right stuff), but they are also necessary to connect similar words for any purpose. In information retrieval these methods are used to find the documents, with the words we care about, from a pool of documents. That is useful because if a user search for documents containing the word friend he is probably equally interested in documents containing friends and possibly friended and friendship. The same principle can be useful in the context of creating a summary for an article. If an article talk about friends it is also about friend.

So, to be clear, in this section we are not going to talk about methods to group semantically connected words, such as identifying all pets or all English towns.

The two methods are: stemming and division of words into group of characters. The algorithms for the first ones are language dependent, while the ones for the second ones are not. We are going to examine each of them in separate paragraphs.

Stemming

Stemming is the process of finding the stem, or root, of a word. In this context, the stem is not necessarily the morphological root according to linguists. So, it is not the form of a word that you would find, say, in a vocabulary. For example, an algorithm may produce the stem consol for the word consoling. While in a vocabulary, as a root, you would find console.

A typical application of stemming is grouping together all instances of words with the same stem for usage in a search library. So, if a user search for documents containing friend he can also find ones with friends or friended.

Porter Stemming Algorithm

Let's talk about an algorithm that remove suffixes to find the stem: the effective and widely used Porter Stemming Algorithm. The algorithm was originally created by Martin Porter for English. There are also Porter based/inspired algorithms for other languages: such as French or Russian. You can find all of them at the website of Snowball. Snowball is a simple language to describe stemming algorithms, but the algorithms are also described in plain English.

A complete description of the algorithm is beyond the scope of this guide. However, its foundation is easy to grasp. Fundamentally the algorithm divides a word in regions and then replace or remove certain suffixes, if they are completely contained in said region. So, for example, the Porter2 (i.e., the updated version) algorithm, state that:

*R*1 is the region after the first non-vowel following a vowel, or the end of the word if there is no such non-vowel

And then, there is a rule that says that you should replace *-tional* with *-tion*, if it is found inside R1.

For example:

- the word confrontational has as R1 region -frontational
- -tional is completely contained in its R1
- so confrontational becomes confrontation

The Porter Stemmer is purely algorithmic, it does not rely on an external database or computed rules (i.e., rules created according to a training database). This is a great advantage, because it makes it predictable and easy to implement. The disadvantage is that it cannot handle exceptional cases and known mistakes cannot be easily solved. For example, the algorithm creates the same stem for university and universal.

A Porter Stemmer is not perfect, but it is simple, effective and easy to implement. For a language like English, a stemmer can be realized by any competent developer. So, there are many out there for all notable programming languages and we are not going to list them here.

Typical Issues with Other Languages

Most languages that are somewhat close to English, like German or even Romance languages, are generally easy to stem. Actually, the creation of the algorithm itself is complex and requires a great knowledge of the language. However, once somebody has done the hard work of creating an algorithm, implementing one is easy.

In stemming there are many problems with two kinds of languages you will usually encounter. The first kind is agglutinative languages. Setting aside the linguistic meaning of the expression, the issue is that agglutinative languages pile up prefixes and suffixes on the root of a word.

In particular Turkish is problematic because is both an agglutinative language and a concatenative one. Which mean basically that in Turkish a word can represent a whole English sentence. This makes hard to develop a stemming algorithm for Turkish, but it also makes it less useful. That is because if you stem a Turkish word you might end up with one stem for each sentence, so you lose a lot of information.

The second kind of issue is related to language with no clearly defined words. Chinese is the prime example as a language that has no alphabet, but only symbols that represent concepts. So, stemming has no meaning for Chinese. Even determining the boundaries of concepts is hard. The problem of dividing a text in its component words

is called word segmentation. With English documents, you can find the boundaries of words just by looking at whitespace or punctuation. There are no such things in a Chinese text.

Splitting Words

An alternative method to group together similar words relies on splitting them. The foundation of this method is taking apart words into sequence of characters. These characters are called *k-grams*, but they are also known as *n-grams characters* (n-grams might also indicate groups of words). The sequence of characters is built in a sliding manner, advancing by one character at each step, starting and ending with a special symbol that indicates the boundaries of the word. For example, the 3-grams for happy are:

- $ha
- hap
- app
- ppy
- py$

With the symbol $ used to indicate the beginning and the end of the word.

The exact method used for search is beyond the scope of this article, but you need just an overview to understand it. In general terms:

1. you apply this process to the words in each document of your database
2. you apply the same process to the search term(s)
3. finally, you just have to compare the occurrences of the k-grams of the input with the one of the words in the documents

Usually you apply a statistical coefficient, like the Jaccard coefficient, to determine how much similar the words have to be to be grouped together (i.e., how many grams have to have in common). For example, by choosing a different coefficient you might group together cat and cats or divide cat and catty.

It is important to note a couple of things: the order of the k-grams and spelling mistakes. The order of the k-grams does not matter, in theory you could have completely different words that happens to have the same k-grams. In practice, this does not happen. This method is imprecise, which means that it can also protect from spelling mistakes of the user. For example, even if the user input locamotive instead of locomotive, it will probably still show the correct results. That is because 7 of 10 3-grams matches; exact matches would rank higher, but the words locamotive does not exist and so it probably has no matches.

Limits and Effectiveness

The great advantage of this technique is that it is not just purely algorithmic and very simple, but it also works with all languages. You do not need to build k-grams for English differently from the ones for French. You just take apart the words in the same

way. Although it is important to note that the effectiveness is in the details: you have to pick the right number of *k* to have the best results.

The ideal number depends on the average length of the word in the language: it should be lower or equal than that. Different languages might have different values, but in general you can get away with 4 or 5. You will not have the absolute best results with only one choice, but it will work.

The disadvantage is that it looks extremely stupid. Let's face it: it so simple that it should not work. But it actually does, it works well if not better than stemming (PDF). It is shamelessly effective, and it has many other uses. We are going to see one right now.

Generating Names

This is a great example of how k-grams can be used in natural language processing. The general case of generating fake words that looks like real words is hard and of limited use. You could create phrases for a fake language, but that is pretty much it. However, there are a few use case to create realistic fake names: for use in games or for any world building need. And it is possible to create them programmatically

There are several variantsof this simple method. The base method works roughly like that:

1. create a database of names of the same kind you want to generate (e.g., Roman names, Space Lizards Names, etc.)
2. divide the input names in k-grams (e.g., 3-grams of Mark -> $ma - mar - ark - rk$)
3. associate a probability to the k-grams: the more frequently they appear in the original database, the higher the chance they appear in the generated name)
4. generate the new names

A variant can be more elaborate to improve the quality of the generated names or their variability. For instance, you could combine a different number of k-grams for specific purposes (e.g., all names start with a 2-gram, but end in a 4-gram). This can be useful to force all names to end in a similar way or to fulfill a need for your story (e.g., if you start with a certain 2-gram you belong to a certain tribe or family).

You could also improve the soundness of the generated names, simply by looking at the probabilities of the sequences appearing in a certain order. For example, if you randomly start with ar the following syllable might be more likely th than or. You usually calculate these probabilities based on the original database of names, but you could manipulate them to achieve specific sounds.

This method is not perfect, but it generally works good enough. You can see a few simple libraries like langgen or VNameGenerator, which shows variations of said method and a few others.

Classifying Documents

In this section we include techniques and libraries that measure and analyze documents. For example, they can detect the language in which a document is written or measure how difficult it is to read it. From now on, we are strictly in the natural language processing territory: you do not search for documents that take a little time to read. However, you might need to understand how much time it takes to read a document so maybe you can save it for the weekend.

Text Metrics

There are two popular metrics of a text that can be easily implemented: reading time and difficulty of the text. These measurements are useful to inform the reader or to help the writer checking that the document respects certain criteria, such as being accessible to a young audience (i.e., low difficulty).

Reading Time

The simplest way of measuring the reading time of a text is to calculate the words in the document, and then divide them by a predefined *words per minute* (wpm) number. The words per minute figure represent the words read by an average reader in a minute. So, if a text has 1000 words and the *wpm* is set to 200, the text would take 5 minutes to read.

That is easy to understand and easy to implement. The catch is that you have to pick the correct wpm rate and that the rate varies according to each language. For example, English readers might read 230 wpm, but French readers might instead read 200 wpm. This is related to the length of the words and the natural flow of the language (i.e., a language could be more concise than another, for instance it might frequently omit subjects). These average figures are found by researcher, so you just need to find them somewhere.

The first issue is easily solved: for English most estimates put the correct wpm between 200 and 230. However, there is still the problem of dealing with different languages. This requires having the correct data for each language and to be able to understand the language in which a text is written.

To mitigate both problems you might opt to use a measurement of characters count in order to estimate the reading time. Basically, you remove the punctuation and spaces, then count the characters and divide the sum by 900-1000.

On linguistic grounds the measure makes less sense, since people do not read single characters. However, the difference between languages are less evident by counting characters. For example, an agglutinative language might have very long words, and thus fewer of them. So it ends up with a similar number of characters to a fusional language like English.

This works better because the differences in speed of reading characters in each language is smaller as a percentage of the total speed. Imagine for example that the

typical reader of English can read 200 wpm and 950 cpm, while the typical reader of French can read 250 wpm and 1000 cpm. The absolute difference is the same, but it is less relevant for reading characters. Of course, this is still less than ideal, but it is a simple solution.

Neither of the measure consider the difficulty of the text. That texts that are difficult to read take more time to read, even with the same number of words or characters. So, depending on your need, you might want to combine this measurement with one for the difficulty of the text.

Calculating the Readability of a Text

Usually the calculation of the readability of a text is linked to grades of education (i.e., years of schooling). So, an easy text might be one that can be read by 4th graders, while a harder one might need a 10th grade education. That is both a byproduct of the fact that the algorithms were created for educational purposes and because education is a useful anchor for ranking difficulty. Saying that a text is difficult in absolute terms is somewhat meaningless, saying that it is difficult for 7th-graders makes more sense.

There are several formulas, but they are generally all based on the number of words and sentences in addition to either syllables or the number of difficult words. Let's see two of the most famous: Flesch-Kincaid and Dale–Chall.

None of these formulas is perfect, but both have been scientifically tested. The only caveat is that they should be used only for checking and not as a guideline. They work if you write normally. If you try to edit a text to lower the score, the results might be incorrect and unnatural. For example, *if you use short words just to make a text seem easy, it looks bad*.

Flesch-Kincaid Readability Formula

There are two variants of this formula: *Flesch reading ease* and *Flesch–Kincaid grade level*. They are equivalent, but one output a score (the higher it is, the easier is the text) and the other a corresponding US grade level. We are going to show the first one.

$$\text{Flesch reading ease} = 206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

The readability is generally between 100 to 20. A result of 100 indicates a document that can be easily understood by a 11-years old student, while a result of 30 or less indicates a document suited for university graduates. You can find a more complete explanation and ranking table in How to Write Plain English.

The different parameters can be obtained easily. Only calculating the total number of syllables requires a significant amount of work. The good news is that it is actually doable and there is a reliable algorithm for it. The bad news is that the author of the TeX hyphenation algorithm (Frank Liang) wrote his PhD thesis about his hyphenation algorithm. You can find an implementation and an accessible explanation of the

algorithm in Hyphenopoly.js. The two problems are equivalent, since you can only divide a word in two parts between two syllables.

An alternative is to use a hack: instead of calculating syllables, count the vowels. This hack has been reported to work for English, but it is not applicable to other languages. Although, if you use it, you lose the scientific validity of the formula and you just get a somewhat accurate number.

The general structure of the formula has been applied to other languages (e.g., Flesch-Vacca for Italian), but each language have different coefficients.

Dale–Chall Readability Formula

This formula relies also on the number of words and sentences, but instead of syllables it uses the number of difficult words present in the text.

A difficult word is defined as one that do not belong to a list of 3000 simple words, that 80% of fourth graders understand.

$$\text{Dale-Chall} = 0.1579\left(\frac{\text{difficult words}}{\text{words}} \times 100\right) + 0.0496\left(\frac{\text{words}}{\text{sentences}}\right)$$

Thus, the formula is easy to use and calculate. The only inconvenience is that you have to maintain a database of these 3000 words. We are not aware of the formula having been adapted to languages other than English.

The formula generally output a score between 4 and 10. Less than 5 indicates a text suitable for 4th graders, a result of 9 or more indicates a text for college students. You can find a complete table of results at The New Dale-Chall Readability Formula.

It is natural to think that you could modify the formula, to calculate the difficulty in understanding a specialized text. That is to say you could define difficult words as words belonging to technical terminology. For example, you could calculate how difficult would be to understand a text for an average person, according to how much computer jargon it contains. Words like *parser* or *compiler* could be difficult, while *computer* or *mouse* could be considered easy. In theory this might work: however you would have to calculate the correct coefficients yourself. So, you would have to actually recruit people to conduct a proper research.

Identifying a Language

When you need to work with a natural language document, the need to identifying a language comes up often. For starters, almost all the algorithms we have seen works only on a specific language and not all of them. Even overcoming this problem, it is useful to be able to understand in which language is written a certain document. For instance, so that you can show or hide a document to certain users based on the language it understands. Imagine that a user search for documents containing the word computer: if you simply return all documents that contain that word you will get results even in languages other than English. That is because the word has been adopted by other languages (e.g., Italian).

Reliable language identification can be achieved with statistical methods. We are going to talk about two methods: a vocabulary based one and one based on frequencies of groups of letters.

In theory you could also hack together ad-hoc methods based on language clues, such as the one listed in Wikipedia:Language recognition chart. And then come up with a ranking method to order the probability of each language. The advantage of this approach is that you would not need any database. However, it has not been tested scientifically.

For example, two features of Italian are: most words end with a vowel and the word è (i.e., is) is quite common. So, you could check for the presence and frequencies of these features in a certain document and use them to calculate the probability that said document were in Italian (e.g., at least 70% of the words ends in vowel -> +50% chance that the document is in Italian; the word è is present in at least 10% phrases -> +50% chance that the document is in Italian).

Words in a Vocabulary

A basic method consists in comparing words in the text with the ones included in a vocabulary. First, you get a vocabulary of each language you care about. Then, you compare the text with the words in each vocabulary and count the number of occurrences. The vocabulary which includes the highest number of words denotes the language of the text.

For example, the following phrase: the cat was in the boudoir, would be identified as English because there are 5 English words (the, cat, was, in, the) and 1 French word (boudoir). In case you are wondering, a boudoir is essentially a female version of a man cave, but for sophisticated people.

Despite its simplicity this method works well for large documents, but not so much for Twitter-like texts: texts that are short and frequently contains errors. The author of the Whatlanguage library, which implements this method, suggest a minimum of 10 words for reliable identification.

Frequencies of Groups of Letters

The most currently used technique relies on building language models of the frequencies of groups of letters. We are again talking about k-grams, that have to be analyzed for each language. We are going to describe the procedure outlined by the original paper N-Gram-Based Text Categorization (PDF), although an implementation might adopt a slight variation of it. Two well-known libraries implementing this method are: the PHP TextCat library by Wikimedia and NTextCat.

First, we are going to build a model for a language and then we are going to use it for scoring a document.

Building a Language Model

To create such language models, you need to have a large set of documents for each language. You divide the words in these documents to find the most used k-grams. The paper suggests calculating the number of k-grams for k from 1 to 5.

Once you have the frequencies of k-grams, you order them in descending order to build a language model. For instance, a language model for English might have th in first place, ing in second and so on. In this final stage the actual number of occurrences it is not relevant, only their final ranking in frequency. That is to say the fact that th might appear 2648 or 5895 times depends only on the size of your set of documents and it is irrelevant for the success of the method. What it is relevant is just the relative frequency in your set of documents and thus the respective ranking.

Once you have built this language model, you can use it to identify the language in which a document is written. You have to apply the same procedure to build a document model of the text that you are trying to identify. So, you end up with a ranking of the frequencies of k-gram in the document.

Finally, you calculate the differences between the rankings in the document and the ones for each language model you have. You could calculate this distance metric with many statistical formulas, but the paper uses a simple out-of-place method.

Scoring Languages

The method consists of comparing the position in each language model. For each position, you add a number equivalent to the differences of ranks between each language model and the document model. For example, if the language model for English put th in first place, but the document model for the document that we are classifying put it in 6th place you add 5 to the English score. At the end of the process, the language with the lowest score should be the language of the text.

Obviously, you do not compare all positions of k-grams up to the last one, because the lower you go the more the position becomes somewhat arbitrary. It will depend on the particular set of documents you will have chosen to build a language model. The paper uses a limit of 300 positions, but it also says that it depends on the short length of the documents the scientists have chosen. So, you would have to find the best limit for yourself.

There is also the chance of wrongly classifying a text. This is more probable if a text is written in a language for which there is no model. In that case the algorithm might wrongly classify the text as one belonging to a language close to the real one. That could happen because the method finds a language that has a good enough score for the document. The real language would have scored better, but it is not available, so the good enough score wins.

For instance, if you have a model for Italian, but do not have one for Spanish, your software might classify a Spanish text as an Italian one because Italian is similar enough to Spanish and the closest you have to it.

This problem can be mitigated by using a proper threshold. That is to say a match is found only if the input has a low score (compared to its length) for a language. Of course, now you have the problem of finding such proper threshold.

Limitations

This method works better with short texts, but it is not perfect. The suggested limit is based on the implementation and the quality of text and model. For example, NTextCat recommends a limit of 5 words. Generally speaking this method would not work reliably for Twitter messages or similar short and frequently incorrect texts. However, that would be true even for a human expert.

Understanding Documents

This section contains more advanced libraries of natural language processing, the ones used to understand documents and their content. We use the concept *understand* somewhat loosely: we talk about how a computer can extract or manage the content of a document beyond simple manipulation of words and characters.

We are going to see how you can:

- generate summary of a document (i.e., an algorithmic answer to the question what is this article about?)
- sentiment analysis (i.e., does this document contain a positive or negative opinion?)
- parsing a document written in a natural language
- translate a document in another language

For the methods listed in the previous sections you could build a library yourself with a reasonable effort. From now on, it will get harder. That is because they might require vast amount of annotated data (e.g., a vocabulary having each word with the corresponding part of speech) or rely on complex machine learning methods. So, we will mostly suggest using libraries.

This is an area with many open problems and active research, so you could find most libraries in Python, a language adopted by the research community. Though you could find the occasional research-ready library in another language.

A final introductory note is that statistics and machine learning are the current kings of natural language processing. So there is probably somebody trying to use TensorFlow to accomplish each of these tasks (e.g., deep news summarization). You might try that too, if you take in account a considerable amount of time for research.

Generation of Summaries

The creation of a summary, or a headline, to correctly represent the meaning of a document it is achievable with several methods. Some of them rely on information retrieval techniques, while others are more advanced. The theory is also divided in two strategies: extracting sentences or parts thereof from the original text, generating abstract summaries.

The second strategy it is still an open area of research, so we will concentrate on the first one.

SumBasic

SumBasic is a method that relies on the probability of individual words being present in a sentence to determine the most representative sentence:

1. First, you have to account the number of times a word appears in the whole document. With that you calculate the probability of each word appearing in the document. I.e., if the word appears 5 times and the document has 525 words, its probability is 5/525.
2. You calculate a weight for each sentence that is the average of the probabilities of all the words in the sentence. I.e., if a sentence contains three words with probability 3/525, 5/525 and 10/525, the weight would be 6/525.
3. Finally, you score the sentences by multiplying the highest probability word of each sentence with its weight. I.e., a sentence with a weight of 0.1 and whose best word had the probability of 0.5 would score 0.1 * 0. 5 = 0.05, while another one with weight 0.2 and a word with probability 0.4 would score 0.2 * 0.4 = 0.08.

Having found the best sentence, you recalculate the probabilities for each word in the chosen sentence. You recalculate the probabilities as if the chosen sentence was removed from the document. The idea is that the included sentence already contains a part of the whole meaning of the document. So that part become less important and this helps avoiding excessive repetition. You repeat the process until you reach the needed summary length.

This technique is quite simple. It does not require to have a database of documents to build a general probability of a word appearing in any document. You just need to calculate the probabilities in each input document. However, for this to work you have to exclude what are called *stopwords*. These are common words present in most documents, such as the or is. Otherwise you might include meaningless sentences that include lots of them. You could also perform stemming before applying this algorithm to improve the results.

It was first described in The Impact of Frequency on Summarization (PDF); there is an implementation available as a Python library.

The approach based on frequencies is an old and popular one, because it is generally effective and simple to implement. SumBasic is good enough that is frequently used as a baseline in the literature. However, there are even simpler methods. For example, Open Text Summarizer is a 2003 library that uses an even simpler approach. Basically you count the frequency of each word, then you exclude the common English words (e.g., the, is) and finally you calculate the score of a sentence according to the frequencies of the word it contains.

Graph-based Methods: TextRank

There are more complex methods of calculating the relevance of the individual sentences. A couple of them take inspiration from PageRank: they are called LexRank

and TextRank. They both rely on the relationship between different sentences to obtain a more sophisticate measurement of the importance of sentences, but they differ in the way they calculate similarity of sentences.

PageRank measures the importance of a document according to the importance of other documents that links to it. The importance of each document, and thus each link, is computed recursively until a balance is reached.

TextRank works on the same principle: the relationship between elements can be used to understand the importance of each individual element. TextRank actually uses a more complex formula than the original PageRank algorithm, because a link can be only present or not, while textual connections might be partially present. For instance, you might calculate that two sentences containing different words with the same stem (e.g., cat and cats both have cat as their stem) are only partially related.

The original paper describes a generic approach, rather than a specific method. In fact, it also describes two applications: keyword extraction and summarization. The key differences are:

- the units you choose as a foundation of the relationship
- the way you calculate the connection and its strength

For instance, you might choose as units n-grams of words or whole phrases. N-grams of words are sequences of n words, computed the same way you do k-gram for characters. So, for the phrase dogs are better than cats, there are these 3-grams:

- dogs are better
- are better than
- better than cats

Phrases might create weighted links according to how similar they are. Or they might simply create links according to the position they are (i.e., a phrase might link to the previous and following one). The method works the same.

TextRank for Sentence Extraction

TextRank for extracting phrases uses as a unit whole sentences, and as a similarity measure the number of words in common between them. So, if two phrases contain the words tornado, data and center they are more similar than if they contain only two common words. The similarity is normalized based on the length of the phrases. To avoid the issue of having longer phrases having higher similarity than shorter ones.

The words used for the similarity measure could be stemmed. Stopwords are usually excluded by the calculation. A further improvement could be to also exclude verbs, although that might be complicated if you do not already have a way to identify the parts of speech.

LexRank differs mainly because as a similarity measure it uses a standard TF-IDF (Term Frequency - Inverse Document Frequency). Basically with TF-IDF the value of individual words is first weighted according to how frequently they appear in all

documents and in each specific document. For example, if you are summarizing articles for a car magazine, there will be a lot of occurrences of the word car in every document. So, the word car would be of little relevance for each document. However, the word explosion would appear in few documents (hopefully), so it will matter more in each document it appears.

The paper TextRank: Bringing Order into Texts (PDF) describe the approach. ExplainToMe contains a Python implementation of TextRank.

Latent Semantic Analysis

The methods we have seen so far have a weakness: they do not take into account semantics. This weakness is evident when you consider that there are words that have similar meanings (i.e., synonyms) and that most words can have different meaning depending on the context (i.e., polysemy). *Latent Semantic Analysis* attempt to overcome these issues.

The expression *Latent Semantic Analysis* describes a technique more than a specific method. A technique that could be useful whenever you need to represent the meaning of words. It can be used for summarization, but also for search purposes, to find words like the query of the user. For instance, if the user search for happiness a search library using LSA could also return results for joy.

A Simple Description

The specific mathematical formulas are a bit complex and involve matrices and operations on them. However, the founding idea is quite simple: words with similar meaning will appear in similar parts of a text. So you start with a normal TF-IDF matrix. Such matrix contains nothing else than the frequencies of individual words, both inside a specific document and in all the documents evaluated.

The problem is that we want to find a relation between words that do not necessarily appear together. For example, imagine that different documents contain phrases containing the words joy and happiness together other words cookie or chocolate. The words do not appear in the same sentence, but they appear in the same document. One document contains a certain number of such phrases: a dog creates happiness and dogs bring joy to children. For this document, LSA should be able to find a connection between joy and happiness through their mutual connection with dog.

The connection is build based on the frequency the words appear together or with related words in the whole set of documents. This allows to build connection even in a sentence or document where they do not appear together. So if joy and happiness appears frequently with dog, LSA would associate the specific document with the words (joy, happiness) and dog.

Basically, this technique will transform the original matrix from one linking each term with its frequency, into one with a (weighted) combination of terms linked to each document.

The issue is that there are a lot of words, and combinations thereof, so you need to make a lot of calculation and simplifications. And that is where the complex math is needed.

Once you have this matrix, the world is your oyster. That is to say you could use this measurement of meaning in any number of ways. For instance, you could find the most relevant phrase and then find the phrases with are most close to it, using a graph-based method.

Text Summarization and Singular Value Decomposition describes one way to find the best sentences. The Python library sumy offers an implementation.

Other Methods and Libraries

The creation of summaries is a fertile area of research with many valid methods already devised. In fact, much more than the ones we have described here. They vary for the approaches and the objective they are designed for. For example, some are created specifically to provide an answer to a question of the user, others to summarize multiple documents, etc.

You can read a brief taxonomy of other methods in Automatic Text Summarization (PDF). The Python library sumy, that we have already mentioned, implements several methods, though not necessarily the ones mentioned in the paper.

Classifier4J (Java), NClassifier (C#) and Summarize (Python) implements a Bayes classifier in an algorithm described as such:

> *In order to summarize a document this algorithm first determines the frequencies of the words in the document. It then splits the document into a series of sentences. Then it creates a summary by including the first sentence that includes each of the most frequent words. Finally summary's sentences are reordered to reflect that of those in the original document.*
>
> - Summarize.py

These projects that implements a Bayes classifier are all dead, but they are useful to understand how the method could be implemented.

DataTeaser and PyTeaser (both in Python, but originally DataTeaser was in Scala) use a custom approach that combines several simple measurements to create a summary of an article.

Other Uses

You can apply the same techniques employed to create summaries to different tasks. That is particularly true for the more advanced and semantic based ones. Notice that the creation of only one summary for many documents is also a different task. That is because you have to take in account different documents lengths and avoid the repetitions, among other things.

A natural application is the identification of similar documents. If you can devise a method to identify the most meaningful sentences of one document, you can also compare the meaning of two documents.

Another objective with common techniques is information retrieval. In short, if a user search for one word, say car, you could use some of these techniques to find documents containing also automobile.

Finally, there is *topic modelling*, which consists in finding the topics of a collection of documents. In simple terms, it means grouping together words with similar themes. It uses more complex statistical methods that the one used for the creation of summaries. The current state of art is based upon a method called Latent Dirichlet allocation.

Gensim is a very popular and production-ready library, that have many such applications. Naturally is written in Python.

Mallet is a Java library mainly designed for topic modelling.

Parsing Documents

Most computer languages are easy to parse. This is not true for natural languages. There are approaches that give good results, but ultimately this is still an open area of research. Fundamentally the issue is that the parsing a sentence (i.e., analyzing its syntax) and its meaning are interconnected in a natural language. A subject or a verb, a noun or an adverb are all words and most words that can be subject can also be object.

In practical terms, this means that there are no ready to use libraries that are good for every use you can think of. We present some libraries that can be used for restricted tasks, such as recognizing parts of speech, that can also be of use for improving other methods, like the ones for creation of summaries.

There is also the frustrating fact is that a lot of software is made by academic researchers. Which means that it could be easily abandoned for another approach or lacking documentation. You cannot really use a work-in-progress, badly maintained software for anything production project. Especially if you care about a language other than English, you might find yourself seeing a good working demo, which was written ten years ago, by somebody with no contact information and without any open source code available.

You Need Data

To achieve any kind of result with parsing, or generally extracting information from a natural language document, you need a lot of data to train the algorithms. This group of data is called a *corpus*. For use in a system that uses statistical or machine learning techniques, you might just need a lot of real world data possibly divided in the proper groups (e.g., Wikipedia articles divided by category).

However, if you are using a smart system, you might need this corpus of data to be manually constructed or annotated (e.g., the word dog is a noun that has these X possible meanings). A smart system is one that tries to imitate human understanding, or at a least that uses a process that can be followed by humans. For instance, a parser that relies on a grammar which uses rules such as Phrase → Subject Verb (read: a phrase is made of a subject and a verb), but also defines several classes of verbs that humans would not normally use (e.g., verbs related to motion).

In these cases, the corpus often uses a custom format and is built for specific needs. For example, this system that can answer geographical questions about United States uses information stored in a Prolog format. The natural consequence is that even what is generally available information, such as dictionary data, can be incompatible between different programs.

On the other hand, there are also good databases that are so valuable that many programs are built around them. WordNet is an example of such database. It is a lexical database that links groups of words with similar meaning (i.e., synonyms) with their associated definition. It works thus as both a dictionary and a thesaurus. The original version is for English, but it has inspired similar databases for other languages.

What You Can Do

We have presented some of the practical challenges to build your own library to understand text. And we have not even mentioned all the issues related to ambiguity of human languages. So differently from what we did for past sections we are just going to explain what you can do. We are not going to explain the algorithms used to realized them, both because there is no space and also without the necessary data they would be worthless. Instead in the next paragraph we are just going to introduce the most used libraries that you can use to achieve what you need.

Named-entity Recognition

Named-entity recognition basically means finding the entities mentioned in the document. For example, in the phrase John Smith is going to Italy, it should identify John Smith and Italy as entities. It should also be able to correctly keep track of them in different documents.

Sentiment Analysis

Sentiment analysis classifies the sentiment represented by a phrase. In the most basic terms, it means understanding if a phrase indicates a positive or negative statement. A naive Bayes classifier can suffice for this level of understanding. It works in a similar way a spam filter works: it divides the messages into two categories (i.e., spam and non-spam) relying on the probability of each word being present in any of the two categories.

An alternative is to manually associate an emotional ranking to a word. For example, a value between -10/-5 and 0 for catastrophic and one between 0 and 5/10 for nice.

If you need a subtler evaluation you need to resort to machine learning techniques.

Parts of Speech Tagging

Parts of Speech Tagging (usually abbreviated as POS-tagging) indicates the identification and labelling of the different parts of speech (e.g., what is a noun, verb, adjective, etc.). While is an integral part of parsing, it can also be used to simplify other tasks. For instance, it can be used in the creation of summaries to simplify the sentences chosen for the summary (e.g., removing subordinates' clauses).

Lemmatizer

A lemmatizer return the lemma for a given word and a part of speech tag. Basically, it gives the corresponding dictionary form of a word. In some ways it can be considered an advanced form of a stemmer. It can also be used for similar purposes, namely it can ensure that all different forms of a word are correctly linked to the same concepts.

For instance, it can transform all instances of cats in cat, for search purposes. However, it can also distinguish between the cases of run as in the verb to run and run as in the noun synonym of a jog.

Chunking

Parts of speech tagging can be considered equivalent to lexing in natural languages. Chunking, also known as shallow parsing, is a step above parts of speech tagging, but one below the final parsing. It connects parts of speech in higher units of meaning, for example complements. Imagine the phrase John always wins our matches of Russian roulette:

- a POS-tagger identifies that Russian is an adjective and roulette a noun
- a chunker groups together (of) Russian roulette as a complement or two related parts of speech

The chunker might work to produce units that are going to be used by a parser. It can also work independently, for example to help in named-entity recognition.

Parsing

The end result is the same as for computer languages: a parse tree. Though the process is quite different, and it might start with a probabilistic grammar or even with no grammar at all. It also usually continues with a lot of probabilities and statistical methods.

The following is a parse tree created by the Stanford Parser (we are going to see it later) for the phrase My dog likes hunting cats and people. Groups of letters such as NP indicates parts of speech or complements.

```
(ROOT
  (S
    (NP (PRP$ My) (NN dog))
    (VP (VBZ likes)
      (NP (NN hunting) (NNS cats)
        (CC and)
```

```
(NNS people)))))
```

Translation

The current best methods for automatic machine translation rely on machine learning. The good news is that this means you just need a great number of documents in the languages you care about, without any annotation. Typical sources of such texts are Wikipedia and the official documentation of the European Union (which requires documents to be translated in all the official languages of the Union).

As anybody that have tried Google Translate or Bing Translator can attest, the results are generally good enough for understanding, but still often a bit off. They cannot substitute a human translator.

The Best Libraries Available

The following libraries can be used for multiple purposes, so we are going to divide this section by the title of the libraries. Most of them are in Python or Java.

Apache OpenNLP

> *The Apache OpenNLP library is a machine learning based toolkit for the processing of natural language text. It supports the most common NLP tasks, such as tokenization, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, parsing, and coreference resolution. These tasks are usually required to build more advanced text processing services. OpenNLP also included maximum entropy and perceptron-based machine learning.*

Apache OpenNLP is a Java library with an excellent documentation that can fulfill most of the tasks we have just discussed, except for sentiment analysis and translation. The developers provide language models for a few languages in addition to English, the most notable are German, Spanish and Portuguese.

The Classical Language Toolkit

> *The Classical Language Toolkit (CLTK) offers natural language processing (NLP) support for the languages of Ancient, Classical, and Medieval Eurasia. Greek and Latin functionality are currently most complete.*

As the name implies the major feature of the Classical Language Toolkit is the support for classical (ancient) languages, such as Greek and Latin. It has basic NLP tools, such as a lemmatizer, but also indispensable tools to work with ancient languages, such as transliteration support, and peculiar things like Clausulae Analysis. It has a good documentation and it is your only choice for ancient languages.

FreeLing

> *FreeLing is a C++ library providing language analysis functionalities (morphological analysis, named entity detection, PoS-tagging, parsing,*

> *Word Sense Disambiguation, Semantic Role Labelling, etc.) for a variety of languages (English, Spanish, Portuguese, Italian, French, German, Russian, Catalan, Galician, Croatian, Slovene, among others).*

It is a library with a good documentation and even a demo. It supports many languages usually excluded by other tools, but it is released the Affero GPL which is probably the least user-friendly license ever conceived.

Moses

> *Moses is a statistical machine translation system that allows you to automatically train translation models for any language pair. All you need is a collection of translated texts (parallel corpus). Once you have a trained model, an efficient search algorithm quickly finds the highest probability translation among the exponential number of choices.*

The only thing to add is that the system is written in C++ and there is ample documentation.

NLTK

> *NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.*

Natural Language Toolkit (NLTK) is probably the most known NLP library for Python. The library can accomplish many tasks in different ways (i.e., using different algorithms). It even has a good documentation (if you include the freely available book).

Simply put: it is the standard library for NLP research. Though one issue that some people have is exactly that: it is designed for research and educational purposes. If there are ten ways to do something NLTK would allow you to choose among them all. The intended user is a person with a deep understanding of NLP

TextBlob is a library that builds upon NLTK (and Pattern) to simplify processing of textual data. The library also provides translation, but it does not implement it directly: it is simply an interface for Google Translate.

Pattern

Pattern is the most peculiar software in our group because it is a collection of Python libraries for web mining. It has support for data mining from services such as Google and Twitter (i.e., it provide functions to directly search from Google/Twitter) , an HTML parser and many other things. Among these things there is natural language processing for English and a few other languages, including German, Spanish, French and Italian.

Though English support is more advanced than the rest.

> *The pattern.en module contains a fast part-of-speech tagger for English (identifies nouns, adjectives, verbs, etc. in a sentence), sentiment analysis, tools for English verb conjugation and noun singularization & pluralization, and a WordNet interface.*

The rest of the libraries can only support POS-tagging.

Polyglot

Polyglot is a set of NLP libraries for many natural languages in Python. It looks great, although it has only little documentation.

It supports fewer languages for the more advanced tasks, such as POS tagging (16) or named entity recognition (40). However, for sentiment analysis and language identification can work with more than a hundred of them.

Sentiment and Sentiment

Sentiment is JavaScript (Node.js) library for sentiment analysis. The library relies on AFINN (a collection of English words with an associated emotional value) and a similar database for Emoji. These database associate to each word/Emoji a positive or negative value, to indicate a positive or negative sentiment. For example, the word joy has a score of 3, while sad has -2.

The code for the library itself is quite trivial, but it works, and it is easy to use.

```
var sentiment = require('sentiment');

var r1 = sentiment('Cats are stupid.');
console.dir(r1);      // Score: -2, Comparative: -0.666

var r2 = sentiment('Cats are totally amazing!');

console.dir(r2);      // Score: 4, Comparative: 1
```

Sentiment analysis using machine learning techniques.

That is the extent of the documentation for the Python library sentiment. Although there is also a paper and a demo. The paper mentions that:

We have explored different methods of improving the accuracy of a Naive Bayes classifier for sentiment analysis. We observed that a combination of methods like negation handling, word n-grams and feature selection by mutual information results in a significant improvement in accuracy.

Which means that it can be a good starting point to understand how to build your own sentiment analysis library.

SpaCy

Industrial-Strength Natural Language Processing in Python

The library spaCy claims to be a much more efficient, ready for the real world and easy to use library than NLTK. In practical terms it has two advantages over NLTK:

- better performance
- it does not give you the chance of choosing among the many algorithms the one you think is best, instead it chooses the best one for each task. While less choices might seem bad, it can actually be a good thing. That is if you have no idea what the algorithms do, and you have to learn them before making a decision.

In practical terms it is a library that supports most of the basic tasks we mentioned (i.e., things like named entity recognition and POS-tagging, but not translation or parsing) with a great code-first documentation.

Textacy is a library built on top of spaCY for higher-level NLP tasks. Basically, it simplifies some things including features for cleaning data or managing it better.

The Stanford Natural Language Processing Group Software

The Stanford NLP Group makes some of our Natural Language Processing software available to everyone! We provide statistical NLP, deep learning NLP, and rule-based NLP tools for major computational linguistics problems, which can be incorporated into applications with human language technology needs. These packages are widely used in industry, academia, and government.

The Stanford NLP group creates and support many great tools that cover all the purposes we have just mentioned. The only thing missing is sentiment analysis. The most notable software are CoreNLP and Parser. The parser can be seen in action in a web demo. CoreNLP is a combination of several tools, including the parser.

The tools are all in Java. The parser supports a few languages: English, Chinese, Arabic, Spanish, etc. The only downside is that the tools are licensed under the GPL. Commercial licensing is available for proprietary software.

Excluded Software

We think that the libraries we choose are the best ones for parsing, or processing, natural languages. However we excluded some other interesting software, which are usually mentioned, like CogCompNLP or GATE for several reasons:

- there might have little to no documentation
- it might have a purely educational or any non-standard license
- it might not be designed for developers, but for end-users

Summary

In this article we have seen many ways to deal with a document in a natural language to get the information you need from it. Most of them tried to find smart ways to bypass the complex task of parsing natural language. Despite being hard to parse natural languages it is still possible to do so, if you use the libraries available.

Essentially, when dealing with natural languages hacking a solution is the suggested way of doing things, since nobody can figure out how to do it properly.

Where it was possible we explained the algorithms that you can use. For the most advanced tasks this would have been impractical, so we just pointed at ready-to-use libraries. In any case, if you think we missed something, be it a subject or an important library, you can tell us.

The code for the library itself is quite trivial, but it works, and it is easy to use.