Text Preprocessing for Machine Learning & NLP

By Kavita Ganesan / Text Mining Concepts, Tips

Based on some recent conversations, I realized that text preprocessing is a severely overlooked topic. A few people I spoke to mentioned inconsistent results from their NLP applications only to realize that they were not preprocessing their text or were using the wrong kind of text preprocessing for their project.

With that in mind, I thought of shedding some light around what text preprocessing really is, the different techniques of text preprocessing and a way to estimate how much preprocessing you may need. For those interested, I've also made some text preprocessing code snippets in python for you to try. Now, let's get started!

Contents

What is text preprocessing?

To preprocess your text simply means to bring your text into a form that is ***predictable*** and ***analyzable***for your task. A task here is a combination of approach and domain. For example, extracting top keywords with tfidf (approach) from Tweets (domain) is an example of a *Task*.
Task = approach + domain

One task's ideal preprocessing, can become another task's worst nightmare. So take note, text preprocessing is not directly transferable from task to task.

Let's take a very simple example, let's say you are trying to discover commonly used words in a news dataset. If your pre-processing step involves removing stop words because some other task used it, then you are probably going to miss out on some of the common words as you have ALREADY eliminated it. So really, it's not a one-size-fits-all approach.

Types of text preprocessing techniques

There are different ways to preprocess your text. Here are some of the approaches that you should know about and I will try to highlight the importance of each.

Lowercasing

Lowercasing ALL your text data, although commonly overlooked, is one of the simplest and most effective form of text preprocessing. It is applicable to most text mining and NLP problems and can help in cases where your dataset is not very large and significantly helps with consistency of expected output.

Quite recently, one of my blog readers trained a word embedding model for similarity lookups. He found that different variation in input capitalization (e.g. 'Canada' vs. 'canada') gave him different types of output or no output at all. This was probably happening because the dataset had mixed-case occurrences of the word 'Canada' and there was insufficient evidence for the neural-network to effectively learn the weights for the less common version. This type of issue is bound to happen when your dataset is fairly small and lowercasing is a great way to deal with sparsity issues.
Here is an example of how lowercasing solves the sparsity issue, where the same words with different cases map to the same lowercase form:

Raw	Lowercased
Canada CanadA CANADA	canada
TOMCAT Tomcat toMcat	tomcat

Another example where lowercasing is very useful is for search. Imagine, you are looking for documents containing "usa". However, no results were showing up because "usa" was indexed as **"USA".** Now, who should we blame? The U.I. designer who set-up the interface or the engineer who set-up the search index?
While lowercasing should be standard practice, I've also had situations where preserving the capitalization was important. For example, in predicting the programming language of a source code file. The word System in Java is quite different from system in python. Lowercasing the two makes them identical, causing the classifier to lose important predictive features. While lowercasing *is generally* helpful, it may not be applicable for all tasks.

Stemming

Stemming is the process of reducing inflection in words (e.g. troubled, troubles) to their root form (e.g. trouble). The “root” in this case may not be a real root word, but just a canonical form of the original word.

Stemming uses a crude heuristic process that chops off the ends of words in the hope of correctly transforming words into its root form. So the words “trouble”, “troubled” and “troubles” might actually be converted to troubl instead of trouble because the ends were just chopped off (ughh, how crude!).
There are different algorithms for stemming. The most common algorithm, which is also known to be empirically effective for English, is Porters Algorithm. Here is an example of stemming in action with Porter Stemmer:

	original_word	stemmed_words
0	connect	connect
1	connected	connect
2	connection	connect
3	connections	connect
4	connects	connect

	original_word	stemmed_word
0	trouble	troubl
1	troubled	troubl
2	troubles	troubl
3	troublemsome	troublemsom

Stemming is useful for dealing with sparsity issues as well as standardizing vocabulary. I’ve had success with stemming in search applications in particular. The idea is that, if say you search for “deep learning classes”, you also want to surface documents that mention “deep learning ***class***” as well as “deep ***learn*** classes”, although the latter doesn’t sound right. But you get where we are going with this. You want to match all variations of a word to bring up the most relevant documents.
In most of my previous text classification work however, stemming only marginally helped improved classification accuracy as opposed to using better engineered features and text enrichment approaches such as using word embeddings.

Lemmatization

Lemmatization on the surface is very similar to stemming, where the goal is to remove inflections and map a word to its root form. The only difference is that, lemmatization tries to do it the proper way. It doesn't just chop things off, it actually transforms words to the actual root. For example, the word "better" would map to "good". It may use a dictionary such as WordNet for mappings or some special rule-based approaches. Here is an example of lemmatization in action using a WordNet-based approach:

	original_word	lemmatized_word
0	trouble	trouble
1	troubling	trouble
2	troubled	trouble
3	troubles	trouble

	original_word	lemmatized_word
0	goose	goose
1	geese	goose

In my experience, lemmatization provides no significant benefit over stemming for search and text classification purposes. In fact, depending on the algorithm you choose, it could be much slower compared to using a very basic stemmer and you may have to know the part-of-speech of the word in question in order to get a correct lemma. This paper finds that lemmatization has no significant impact on accuracy for text classification with neural architectures.
I would personally use lemmatization sparingly. The additional overhead may or may not be worth it. But you could always try it to see the impact it has on your performance metric.

Stop-word removal

Stop words are a set of commonly used words in a language. Examples of stop words in English are "a", "the", "is", "are" and etc. The intuition behind using stop words is that, by removing low information words from text, we can focus on the important words instead. For example, in the context of a search system, if your search query is *"what is text preprocessing?"*, you want the search system to focus on surfacing documents that talk about text preprocessing over documents that talk about what is. This can be done by preventing all words from your stop word list from being analyzed. Stop words are commonly applied in search systems, text classification applications, topic modeling, topic extraction and others.
In my experience, stop word removal while effective in search and topic extraction systems, showed to be non-critical in classification systems. However, it does help reduce the number of features in consideration which helps keep your models decently sized.

Here is an example of stop word removal in action. All stop words are replaced with a dummy character,
W:

```
original sentence =  this is a text full of content and we need to clean it up
sentence with stop words removed=  W W W text full W content W W W W clean W W
```

Stop word lists can come from pre-established sets or you can create a custom one for your domain. Some libraries (e.g. sklearn) allow you to remove words that appeared in X% of your documents, which can also give you a stop word removal effect.

Normalization

A highly overlooked preprocessing step is text normalization. Text normalization is the process of transforming text into a canonical (standard) form. For example, the word “goooood” and “gud” can be transformed to “good”, its canonical form. Another example is mapping of near identical words such as “stopwords”, “stop-words” and “stop words” to just “stopwords”.

Text normalization is important for noisy texts such as social media comments, text messages and comments to blog posts where abbreviations, misspellings and use of out-of-vocabulary words (oov) are prevalent. This paper showed that by using a text normalization strategy for Tweets, they were able to improve sentiment classification accuracy by ~4%.
Here’s an example of words before and after normalization:

Raw	Normalized
2moro 2mrrw 2morrow 2mrw tomrw	tomorrow
b4	before
otw	on the way
:) :-) ;-)	smile

Notice how the variations, map to the same canonical form.

In my experience, text normalization has even been effective for analyzing highly unstructured clinical texts where physicians take notes in non-standard ways. I’ve also found it useful for topic extraction where near synonyms and spelling differences are common (e.g. topic modelling, topic modeling, topic-modeling, topic-modelling).

Unfortunately, unlike stemming and lemmatization, there isn't a standard way to normalize texts. It typically depends on the task. For example, the way you would normalize clinical texts would arguably be different from how your normalize sms text messages.

Some common approaches to text normalization include dictionary mappings (easiest), statistical machine translation (SMT) and spelling-correction based approaches. This interesting article compares the use of a dictionary based approach and a SMT approach for normalizing text messages. Interestingly, I'm also seeing more and more papers related to text normalization in the research world.

Noise Removal

Noise removal is about removing characters digits and pieces of text that can interfere with your text analysis. Noise removal is one of the most essential text preprocessing steps. It is also highly domain dependent. For example, in Tweets, noise could be all special characters except hashtags as it signifies concepts that can characterize a Tweet. The problem with noise is that it can produce results that are inconsistent in your downstream tasks. Let's take the example below:

	raw_word	stemmed_word
0	..trouble..	..trouble..
1	trouble<	trouble<
2	trouble!	trouble!
3	<a>trouble	<a>trouble
4	1.trouble	1.troubl

Notice that all the raw words above have some surrounding noise in them. If you stem these words, you can see that the stemmed result does not look very pretty. None of them have a correct stem. However, with some cleaning as applied in this notebook, the results now look much better:

	raw_word	cleaned_word	stemmed_word
0	..trouble..	trouble	troubl
1	trouble<	trouble	troubl
2	trouble!	trouble	troubl
3	<a>trouble	trouble	troubl
4	1.trouble	trouble	troubl

Noise removal is one of the first things you should be looking into when it comes to Text Mining and NLP. There are various ways to remove noise. This includes *punctuation removal*, *special character removal*, *numbers removal, html formatting removal, domain specific keyword removal (e.g. 'RT' for retweet), source code removal, header removal* and more. It all depends on which domain you are working in and what entails noise for your task. The code snippet in my notebook shows how to do some basic noise removal.

Text Enrichment / Augmentation

Text enrichment involves augmenting your original text data with information that you did not previously have. Text enrichment provides more semantics to your original text, thereby improving its predictive power and the depth of analysis you can perform on your data.

In an information retrieval example, expanding a user's query to improve the matching of keywords is a form of augmentation. A query like text mining could become text document mining analysis. While this doesn't make sense to a human, it can help fetch documents that are more relevant.
You can get really creative with how you enrich your text. You can use **part-of-speech tagging** to get more granular information about the words in your text. For example, in a document classification problem, the appearance of the word **book** as a **noun** could result in a different classification than **book**as a **verb** as one is used in the context of reading and the other is used in the context of reserving something. This article talks about how Chinese text classification is improved with a combination of nouns and verbs as input features.
With the availability of large amounts texts however, people have started using embeddings to enrich the meaning of words, phrases and sentences for classification, search, summarization and text generation in general. This is especially true in deep learning based NLP approaches where a word level embedding layer is quite common. You can either start with pre-established embeddings or create your own and use it in downstream tasks.
Other ways to enrich your text data include phrase extraction, where you recognize compound words as one (aka chunking), expansion with synonyms and dependency parsing.

Do you need all the text preprocessing types?

Not really, but you do have to do some of it for sure if you want good, consistent results. To give you an idea of what the bare minimum should be, I've broken it down to ***Must Do***, ***Should Do*** and ***Task Dependent***. Everything that falls under task dependent can be quantitatively or qualitatively tested before deciding you actually need it. Remember, less is more and you want to keep your approach as elegant as possible. The more overhead you add, the more layers you will have to peel back when you run into issues.

Must Do:

- Noise removal
- Lowercasing (can be task dependent in some cases)

Should Do:

- Simple normalization – (e.g. standardize near identical words)

Task Dependent:

1. Advanced normalization (e.g. addressing out-of-vocabulary words)
2. Stop-word removal
3. Stemming / lemmatization
4. Text enrichment / augmentation

So, for any task, the minimum you should do is try to lowercase your text and remove noise. What entails noise depends on your domain (see section on Noise Removal). You can also do some basic normalization steps for more consistency and then systematically add other layers as you see fit.

General Rule of Thumb

Not all tasks need the same level of preprocessing. For some tasks, you can get away with the minimum. However, for others, the dataset is so noisy that, if you don't preprocess enough, it's going to be garbage-in-garbage-out.

Here's a general rule of thumb. This will not always hold true, but works for most cases. If you have a lot of well written texts to work with in a fairly general domain, then preprocessing is not extremely critical; you can get away with the bare minimum (e.g. training a word embedding model using all of Wikipedia texts or Reuters news articles). However, if you are working in a very narrow domain (e.g. Tweets about health foods) and data is sparse and noisy, you could benefit from more preprocessing layers, although each layer you add (e.g. stop word removal, stemming, normalization) needs to be quantitatively or qualitatively verified as a meaningful layer.

Here's a table that summarizes how much preprocessing you should be performing on your text data:

Level of text preprocessing needed

	Domain Specific / Noisy Texts	General / Well Written Texts
Lots of data	- Moderate pre-processing - Text enrichment could be helpful	- Light pre-processing - Text enrichment could be helpful, but not critical
Sparse data	- Heavy pre-processing - Text enrichment is important	- Moderate pre-processing - Text enrichment could be helpful

By: Kavita Ganesan

I hope the ideas here would steer you towards the right preprocessing steps for your projects. Remember, *less is more*. A friend of mine once mentioned to me how he made a large e-commerce search system more efficient and less buggy just by throwing out layers of *unneeded* preprocessing.