Most web search engines today index every single word on a page because they have abundant storage and processing power. At the same, they can cater for the most unexpected queries (e.g., searching for exact phrases that contain stopwords). However, in general, it is not advisable to index every word but only those that have the highest values. How are values defined?

1 Selecting and Processing Index Terms

1.1 Stopword Removal and Stemming

To facilitate search, words are extracted from documents to create an index. In traditional IR systems, words typically go through the *stopword removal* and *stemming* processes before they are are inserted into the index.

1.1.1 Stopword removing

Stopword removal removes words that are considered *non-content bearing.* Since it is not easy to decide which words are non-content bearing, we have to resort to heuristic methods. For example, any combination of the following heuristics can be used to decide if a word is a stopword or not:

- Every word that has less than 2 characters
- Articles and prepositions
- Most frequent *n* terms in the collection.

Stopword removal has a significant impact on index size since a stopword typically has a very high number of occurrences in the collection. Thus, removing them from the index means the elimination of long postings lists from the index. If done right, stopword removal has little impact on precision and recall. Suppose if we do not index articles and prepositions, then the query "The President of United States" also matches "a president in United States", which is not perfect but acceptable for most users.

Stopword removal enhances efficiency but does not hurt effectiveness.

Since stopword removal enhances retrieval efficiency without hurting effectiveness, most early-generation web search engines implemented it until Alta Vista pioneered truly full-index indexing by including all words in its index. A well-known used example often quoted to support the need of indexing every word is the famous quote "To be or Not to Be", which consists of stopwords only, and as such would not be indexed and hence not searchable after stopword removal.

This famous example aside, there are subject domains where the simple rules stated above fail. For example, in a chemistry collection, words like "I" and "He" are symbols of chemical elements (respectively, for "Iodine" and "Helium"). In fact, all of the common chemical symbols consist of one or two letters. Removing them from the index would be a disaster.

Most, if not all, web search engines nowadays do not perform stopword removal during indexing, because the contents on the web are very diversified and the specific needs of users vary widely. Instead, they typically remove stopwords from a query when the query contains enough non-stopwords. For example, "of" in the query "Department of Computer Science" will be removed but the single-word query "of" or the famous quote "to be or not to be" will be accepted and processed as is.

Google, as of early 2008, removes stopwords and informs the users of the removal by displaying a message at the top of the result window. If the user wants to retain the stopword, he/she can enclosed the stopword with double quotes. However, as of end of 2008, Google appears to search all stopwords.

1.1.2 Stemming

Stemming converts various spellings and grammatical forms of the same word into their *root*, or *stem*. For example, "recognize" and "recognise" are, respectively, American and British spellings of the same word, and "computes", "computer", "computing", and "computation", etc., are various grammatical forms of the word "compute". In most cases, various spellings of the same word, be it cultural or grammatical variations, should be considered the same in search. That is, "recognize" should match "recognise", and "compute" should match "computes".

As in stopword removal, stemming reduces the size of the index, although this is not the main reason for using stemming. Stemming significantly improves recall and is considered a standard requirement (unlike stopword removal) in most search engine applications. Imagine how bad it is if a search on "computer" would not match "computers", or a search on "compute" would not match "computation".

As is the case with stopword removal, most web search engines do not use stemming, because the increase of recall is not a major issue for web search engines since even if the search engine gives you only pages that contain the word "computer", but not "computers", when you search for "computer", your are going to get *more than enough* results. Then, why take the risk of making mistake in stemming, that is, mapping words with different meanings into the same stem, causing incorrect results to be returned?

1.1.3 Consistency between searching and indexing

Typically, the same stopword removal and stemming methods have to be applied to both the search side and the index side. Figures 1(a) and 1(b) show whether consistent application of stopword removal or stemming is or not in different scenarios.

As can be seen from Fig. 1(a), once stopword removal is applied to the documents, query processing must apply it as well. Since if not, the stopwords that were not removed from the query will result in no matches because they cannot be found in the index. On the other hand, if all words are retained in

		Stopword Removed from Queries?	
		Yes	No
Stopword Removed from Documents?	Yes	√	×
	No	√	√

(a)

		Stemming on Queries Applied?	
		Yes	No
Stemming on Documents Applied?	Yes	√	×
	No	×	√

(b)

Figure 1: Consistency requirements for (a) stopword removal, and (b) stemming.

the index, the query processor can retain or remove stopwords from the queries without causing any problem. See the examples in Sec 1.1.1.

Stemming is more restricted, since it actually changes the spelling of a word. When stemming is applied to the documents, it must be applied to the queries as well. As with stopword removal, it is also advisable not to perform stemming on the document words (if computational cost is not a concern) so that the query processor does not have to be *forced* to stem the queries. Instead, a query term can be *expanded* into its spelling variations (e.g., by looking up a dictionary). For example, the word "computer" can be expanded into "computer OR computers OR ..." so that the search engine will return all documents matching any of the variations. Again, this gives the query processor the flexibility to search a word's spelling variations or not. For example, it is not necessary to expand the query terms, if it is estimated that the query has enough high quality results, or when the expansion of a query term does not make sense considering the context of the whole query (e.g., returning results on "computers science" given the query "computer science" may not make sense).

In conclusion, once you have committed to performing stopword removal or stemming on documents, you must do the same on the queries. If a word was treated as a stopword and not indexed but later on found to be a useful word to include in the index, the whole set of documents have to be re-indexed. Likewise, if a word is found to be stemmed incorrectly, the whole index has to be rebuilt to fix the problem. This advocates the indexing of every word in a document and without stemming to avoid the need for re-indexing should a mistake or a change of application needs is found later.

1.2 Stemming Methods

Stemming can be done by table lookup. That is, a table mapping every word into its spelling variations is maintained. Obviously, this method is simple but the maintenance of the table is expensive. In the rest of this subsection, we will study the two classes of algorithmic methods:

- *Linguistic methods* which define linguistic rules to transform a word into its stem. The Porter's algorithm is a well-known implementation based on linguistic method.
- *Statistical methods* that are based on corpus analysis. The successor variety method will be described.

1.2.1 Linguistic methods

Linguistic methods exploit knowledge about the language of the documents to define linguistic rules to transform a word into its stem. In English, perhaps in other western languages as well, spelling variations of the same word usually share the same prefix[1] but vary only in the suffixes. Therefore, we can identify the common suffixes and transform them properly so that they can be put back to the prefix to form a good stem. Of course, it is also possible to identify linguistic prefixes and remove them, e.g., "indefinite" can be transformed to "definite" by prefix removal. This leads to the term *affix removal algorithms* referring to methods that identify and transform both the prefix and suffix of a word to obtain the stem of the word. Since prefix transformation is rare in English, most of the examples given below apply to suffixes.

Linguistic transformation rules can be *context free* or *context sensitive*. A context-free rule has the form $X \longrightarrow Y$, which means that when a suffix matches X, it will be replaced with Y. A context-sensitive rule has the form $(A)X \longrightarrow Y$, which will be applied only if the suffix of a word matches X *and* the prefix satisfies the *context* specified by (A) (see Sec. 1.2.1 for examples of contexts).

Porter's Algorithm

It is clear that to obtain good stemming results, the rules must be carefully specified. The Porter's algorithm is a well-known linguistic stemming method. The rules are compact, leading to high efficiency, and yet the quality of the stemmed results is comparable to algorithms that are much more complex.

The Porter's algorithm identifies 60 common suffixes and groups them into five steps. In each step, a set of context-free and context-sensitive rules is applied to a word to either remove its suffix or transform it into another form

[1]In English, a prefix has a special meaning. For example, "in" in "indefinite" and "pre" in "predefine" are considered prefixes, whereas "inde" and "pred" are not. However, for string processing, a prefix does not have to have a special meaning in the language. A prefix is just a string that appears as the head of a string. In this chapter, we use the term *linguistic prefix* to denote the meaningful prefixes in a language and the term *prefix* to denote meaningless prefixes in strings.

Rule Type	Rules	Remarks
Context-free	$sses \longrightarrow ss$	
	$ies \longrightarrow i$	
	$s \longrightarrow NULL$	
Context-sensitive	$(*v*)$:$ed \longrightarrow NULL$,	$(*v*)$ requires the prefix before
	$ing \longrightarrow NULL$	"ed" to contain at least one vowel
	Examples:	
	$plastered \longrightarrow plaster$	
	$tied \longrightarrow ti$	
	$bled \longrightarrow bled$	Prefix "bl" does not have a vowel
	$bed \longrightarrow bed$	
	$motoring \longrightarrow motor$	
	$sing \longrightarrow sing$	Prefix "s" does not have a vowel

Figure 2: Examples of context-free and context-sensitive rules in Porter's algorithm.

for processing in the next step. Examples of context-free and context-sensitive rules are shown in Figure 2. It is interesting to note that a simple rule $(*v*) : ed$, where $(*v*)$ is a context specifying that the prefix before "ed" must contain at least one vowel can avoid many wrong stemming should the rule $ed \longrightarrow NULL$ be applied without the context requirement. As an exercise, try to remove "able" from the "table" or "s" from "gas".

Context-free rules are convenient because they can apply to a large number of words, but they don't always produce the right stems. Thus, a combination of context-free and context-sensitive rules must be used. For example,

Context-free:	$ies \longrightarrow i$	
Context-sensitive:	$*v*ki \longrightarrow ky$	
Examples:	$skies \longrightarrow ski$	"ski" is not turned into "sky"
Context-free	$ful \longrightarrow NULL$	
Context-sensitive	$*v*ti \longrightarrow ty$	
Examples:	*beautiful* $\longrightarrow$ *beauti*	
	$\longrightarrow$ beauty	

Matching multiple rules

When a word matches more than one rule, the longest matching rule will be applied. For example, give two rules, $ability \longrightarrow NULL$ and $ty \longrightarrow ti$, both of which can be applied to *computability*, the longer rule should apply, i.e., *computability* $\longrightarrow$ *computa*.

Rules can be applied iteratively, e.g., in the following transformation:

$$willingness \longrightarrow willing \longrightarrow will$$

By specifying the right context, we could prevent the rule $ing \longrightarrow NULL$ be repeatedly applied to produce an incorrect stem:

$$singing \longrightarrow sing \longrightarrow s$$

1.2.2 Statistical Methods

The creation of linguistic rules require a lot of linguistic knowledge about the language. It is not only time consuming to create but even more so to maintain. Change of one rule may have unpredictable rippling effects on other rules. This partly explains the fact that there were very few extensions made to Porter's algorithm ever since it was published in 1980.

Linguistic methods can be considered as *reduction methods*, since they try to identify suffixes and reduce them. This is why linguistic knowledge is required. For example, how do you know that "tion" is a reducible suffix but not "on"?

Statistical methods aim at deriving the stem of a word by studying analyzing a large corpus. Statistical methods are based on the understanding of a language's *generative mechanism*, that is, how people create spelling variants from a stem.

Statistical stemming is based on the generative principle of a language.

Take English as an example, spelling variants are created by either appending different suffixes directly to a stem or after changing the last one or two characters of the stem. Consider the stem "compute", which should have existed in the English vocabulary for a long time. As the language develops, we need to create a word for devices that can computer. Thus, we create the word "computer" by adding an "r" to "computer". Then, as the field of computer science advances, the need for "computability" and "computable" arise. We created them again based on the word "compute" but this time we drop the last letter "e" before the suffixes are appended to it. Likewise, we can observe that spelling variants like "computed" and "computes" simply add one more character to the end of "compute", whereas spelling variants like "computing" and "computation" are formed by removing the last "e" from "compute".

If the generative mechanism is like what we describe above, then a heuristic to find a stem would be to identify a *stable prefix* that has a lot of different suffixes. This leads to the *successor variety method.*

1.2.3 Successor variety method

The Successor Variety method defines the *successor variety* (SV) of a prefix as the number of different characters found at the character position following the prefix. When the SV of a prefix is low, it means very few words are formed out of the prefix and the prefix is not likely a stem. On the other hand, when the SV is high, it means that many words are formed based on the prefix, and the prefix is likely a stem.

Formally, given a corpus, the SV of a string is defined as the number of different characters that are found to follow the string in the corpus. Figure 3 shows a set of words extracted from a corpus and the SVs of the prefixes of the word "computer".

To identify the stem of a word, we find the SVs of all of the prefixes of the word and identify the prefix with the highest SV. There are several ways to find

Corpus
`compare`ϕ
`computation`ϕ
`computational`ϕ
`compute`ϕ
`computer`ϕ
`computing`ϕ

Prefix	SV	Distinct Letters
c	1	o
co	1	m
com	1	p
comp	2	a,u
compu	1	t
comput	3	a,e,i
compute	2	r,ϕ
computer	1	ϕ

H_α
0
0
0
0.65
0
1.52
1
0

Figure 3: Computing the successor variety values of "computer".

prefixes with high SVs. The obvious way to take the absolute SV values and apply a threshold to it (hence the *threshold method*). When the threshold is set to be $SV \geq 2$, then "comp", "comput" and "compute" are considered to have high SVs and are candidates as the stem of the word.

It is clear that the threshold method is suitable since the SV values vary widely for different words. A more robust rule is to identify the prefixes that have higher SVs than their right and left neighbors. This is called the *peak and plateau* method. In the example, the peak occurs at "comput", whereas "comp" can be considered as the second choice.

1.2.4 Entropy-based SV

We note that the absolute SV are not reliable because it counts the number of different characters following a prefix but not *how many times* the characters appear. This creates unfairness as shown in Figure 4.

The two prefixes "probab" and "probabl" both have SV equal 2 and as such are tie. However, looking closer, we find that "probab" is followed by one "i" and two "l"s, where as "probabl" is followed by one "e" and one "y". Intuitively, "probab" has *less variation* than "probabl", because the former occurs three times but only has two different succeeding characters whereas the latter occurs only twice with the same number of different succeeding characters. From an information theory point of view, the characters after "probab" is more predictable, and as such less varied, than that of "probabl".

Corpus
`probability`ϕ
`probable`ϕ
`probably`ϕ

Prefix	SV
probab	2
probabl	2
probable	1

Figure 4: Successor variety ignores frequency information.

To introduce frequency information into SV, we can compute the *entropy* of a prefix α as follows. First, we compute the probability that character "j"

appears after α, denoted by $P_{\alpha j}$.

$$P_{\alpha j} = \frac{|D_{\alpha j}|}{|D_{\alpha}|} \tag{1}$$

where $|D_{\alpha j}|$ is the number of strings in which "j" appears right after α and $|D_{\alpha}|$ is the number of strings containing α as the prefix.

The entropy of α is the summation of the probabilities of all characters that might appear right after α.

$$H_{\alpha} = \sum_{j \in `a' \ldots `z'} -P_{\alpha j} \cdot \log_2 P_{\alpha j} \tag{2}$$

Applying the entropy formulation to the previous example, we obtain results that confirm our intuition that "probabl" is a better stem.

$$\begin{aligned} H_{\alpha}(\alpha = \text{``}probab\text{''}) &= -(0.33 * log_2 0.33 + 0.67 * log_2 0.67) &&= 0.91 \\ H_{\alpha}(\alpha = \text{``}probabl\text{''}) &= -(0.5 * log_2 0.5 + 0.5 * log_2 0.5) &&= 1 \end{aligned}$$

The entropy for the example in Fig. 3 is given in the H_{α} table. The following shows the computation of H_{α} for the prefixes with non-zero H_{α}:

$$\begin{aligned} H_{\alpha}(\alpha = \text{``}comp\text{''}) &= \tfrac{1}{6} log_2 \tfrac{1}{6} + \tfrac{5}{6} log_2 \tfrac{5}{6} &&= 0.65 \\ H_{\alpha}(\alpha = \text{``}comput\text{''}) &= \tfrac{2}{5} log_2 \tfrac{2}{5} + \tfrac{2}{5} log_2 \tfrac{2}{5} + \tfrac{1}{5} log_2 \tfrac{1}{5} &&= 1.52 \\ H_{\alpha}(\alpha = \text{``}compute\text{''}) &= \tfrac{1}{2} log_2 \tfrac{1}{2} + \tfrac{1}{2} log_2 \tfrac{1}{2} &&= 1 \end{aligned}$$

1.2.5 Picking the stem

Once the SVs of a word are computed, the remaining question is where to segment the word because there may be more than one choice. For the example shown if Fig 3, "comput" is the winner with "comp" a close second. When there is a tie, we can resort to heuristics to break the tie. For example,

- Pick the longer prefix
- Pick the prefix that is a complete word by itself (assuming a dictionary is available)
- Pick a prefix that appears as a word in the corpus (in the lack of a dictionary).

1.2.6 Shared bigram method

The shared bigram method is based on the observation that the spelling variants of a stem are very similar in spelling. From the search point of view, the function of stemming is to allow various spellings of the same stem to be matchable. Thus, we can derive a method that allow words to match based on how close their spellings are. The shared bigram method divides a word into *overlapping bigrams* and matches a word against another by finding out how many of their bigrams match (thus the name *shared bigram.* For example, the words "statistics" and "statistical" can be represented by the following sets of bigrams"

$$\begin{array}{lll} \text{statistics} & \Longrightarrow & S_1 = \{st, ta, at, ti, is, st, ti, ic, cs\} \\ \text{statistical} & \Longrightarrow & S_2 = \{st, ta, at, ti, is, st, ti, ic, ca, al\} \end{array}$$

We can observe that a word is transformed into a set of bigrams. Two sets of bigrams can be compared using any of the similarity measures that we learnt in the vector space model. For example, using the Dice

$$\begin{array}{ll} \text{Inner product:} & |S_1 \cap S_2| = 8 \\ \text{Dice:} & \frac{2\times|S_1\cap S_2|}{|S_1|+|S_2|} = \frac{2\times 6}{7+8} = 0.8 \\ \text{Jaccard:} & \frac{|S_1\cap S_2|}{|S_1\cup S_2|} = \frac{6}{11} = 0.55 \end{array}$$

Note that the bigrams must overlap so that much of the character sequence information of the word is preserved in the bigram set. For example, "abcd" matches "abed" better than "cdab". Furthermore, trigrams (sequences of three characters) can also be used. However, trigrams are more strict than bigrams in matching. For example, the trigrams in "abcd" would not match any of the trigrams of "abed", thus producing a zero similarity score, whereas bigrams would produce a non-zero similarity score.