**APPENDIX B**

**STEMMING**

In [linguistic morphology](http://en.wikipedia.org/wiki/Linguistic_morphology), stemming is the process for reducing inflected (or sometimes derived) words to their [stem](http://en.wikipedia.org/wiki/Word_stem), base or [root](http://en.wikipedia.org/wiki/Root_%28linguistics%29) form – generally a written word form. The stem need not be identical to the [morphological root](http://en.wikipedia.org/wiki/Morphological_root) of the word; it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. [Algorithms](http://en.wikipedia.org/wiki/Algorithm) for stemming have been studied in [computer science](http://en.wikipedia.org/wiki/Computer_science) since 1968. Stemming programs are commonly referred to as stemming algorithms or stemmers.

A stemmer for [English](http://en.wikipedia.org/wiki/English_language), for example, should identify the [string](http://en.wikipedia.org/wiki/String_literal) "cats" (and possibly "catlike", "catty" etc.) as based on the root "cat", and "stemmer", "stemming", "stemmed" as based on "stem". A stemming algorithm reduces the words "fishing", "fished", "fish", and "fisher" to the root word, "fish".

The first ever published stemmer was written by [Julie Beth Lovins](http://en.wikipedia.org/wiki/Julie_Beth_Lovins) in 1968. A later stemmer was written by [Martin Porter](http://en.wikipedia.org/wiki/Martin_Porter) and was published in the July 1980 issue of the journal *Program*. This stemmer was very widely used and became the de-facto standard algorithm used for English stemming. Many implementations of the Porter stemming algorithm were written and freely distributed; however, many of these implementations contained subtle flaws. As a result, these stemmers did not match their potential. To eliminate this source of error, Martin Porter released an official [free-software implementation](http://tartarus.org/~martin/PorterStemmer/) of the algorithm around the year 2000. He extended this work over the next few years by building [Snowball](http://en.wikipedia.org/wiki/Snowball_programming_language), a framework for writing stemming algorithms, and implemented an improved English stemmer together with stemmers for several other languages.

**ALGORITHMS**

There are several types of stemming algorithms which differ in respect to performance and accuracy and how certain stemming obstacles are overcome.

**i. Brute-force algorithms**

[Brute force](http://en.wikipedia.org/wiki/Brute-force_search) stemmers employ a [lookup table](http://en.wikipedia.org/wiki/Lookup_table) which contains relations between root forms and inflected forms. To stem a word, the table is queried to find a matching inflection. If a matching inflection is found, the associated root form is returned. Brute force searches involve more operations during the search than should be necessary and also consume immense amounts of storage to host the list of relations (relative to the task). The algorithm is only accurate to the extent that the inflected form already exists in the table. Given the number of words in a given language, like English, it is unrealistic to expect that all word forms can be captured and manually recorded by human action alone. Manual training of the algorithm is overly time-intensive and the ratio between the effort and the increase is just marginally accurate.

Brute force algorithms do overcome some of the challenges faced by the other approaches. Not all inflected word forms in a given language "follow the rules" appropriately. While "running" might be easy to stem to "run" in a suffix stripping approach, the alternate inflection, "ran", is not. Suffix stripping algorithms cannot overcome this problem as it is short of increasing the number and complexity of the rules, but brute force algorithms only require storing a single extra relation between "run" and "ran".

Brute force algorithms are initially very difficult to design given the immense number of relations that must be initially stored to produce an acceptable level of accuracy (the number can span well into the millions). However, brute force algorithms are easy to improve in that decreasing the stemming error is only a matter of adding more relations to the table. Someone with only a minor experience in linguistics is capable of improving the algorithm, unlike the suffix stripping approaches which require a solid background in linguistics.

For technical accuracy, some programs may use suffix stripping to generate the lookup table given a text corpus, and then only consult the lookup table when stemming. This is not regarded as a brute force approach, although a lookup table is involved. To improve a brute force algorithm it may be built upon preliminary part-of-speech tagging. In this case the look up table includes suffixes and endings associated with a specific part of speech, which significantly reduces the number of over-stemming errors.

* **The production technique**

Some programs attempt to automatically generate the table of root and inflected forms. A production algorithm attempts to infer the probable inflections for a given word. For example, if the word is "run", then the algorithm might automatically generate the forms "running" and "runs". In the traditional sense of the concept of stemming, this algorithm is its reverse process. Rather than try and remove suffixes, the goal of a production algorithm is to generate suffixes. Later, a brute force algorithm can simply query the automatically generated table of word relations to find the root form of a word.

There are many types of heuristic, experimental techniques for identifying inflected forms of words. Some algorithms work phonetically, looking at the final syllables in a word. Some are rather brute force, using rules that seem a lot like normalization rules, by inspecting the last few characters. Others are similar to the process of lemmatization, which takes advantage of the additional knowledge about the part of speech of the given word to limit what types of suffixes are considered when generating inflections for the word.

Distinguishing this technique from the other approaches is not very important. The other approaches share much of the same logic, but are simply working in the opposite direction. Some stemming approaches may employ both a production technique to generate inflections, and a suffix stripping approach to identify root forms together.

One drawback of the production technique is that there is no guarantee the inflection is real. For example, a technique might join together "run" and "ly" to create the word "runly" where "runly" is not a real word. Granted, it is highly unlikely that "runly" will ever be used as input to a stemming algorithm since it is not a real word, so this error may not affect the result and therefore the accuracy of a stemming algorithm. However, such errors do populate the table of relations, and each additional entry can increase the time required to perform the lookup of an inflection to find its corresponding root form.

**ii.** **Suffix-stripping algorithms**

Suffix stripping algorithms do not rely on a lookup table that consists of inflected forms and root form relations. Instead, a typically smaller list of “rules” is stored which provide a path for the algorithm, given an input word form, to find its root form. Some examples of the rules include:

* if the word ends in 'ed', remove the 'ed'
* if the word ends in 'ing', remove the 'ing'
* if the word ends in 'ly', remove the 'ly'

Suffix stripping approaches enjoy the benefit of being much simpler to maintain than brute force algorithms, assuming the maintainer is sufficiently knowledgeable in the challenges of linguistics and morphology and encoding suffix stripping rules. Suffix stripping algorithms are sometimes regarded as crude given the poor performance when dealing with exceptional relations (like 'ran' and 'run'). The solutions produced by suffix stripping algorithms are limited to those [lexical categories](http://en.wikipedia.org/wiki/Lexical_category) which have well known suffixes with few exceptions. This, however, is a problem, as not all parts of speech have such a well formulated set of rules. Lemmatization attempts to improve upon this challenge.

**Additional algorithm criteria**

Suffix stripping algorithms may differ in results for a variety of reasons. One such reason is whether the algorithm constrains that the output word must be a real word in the given language or not. Some approaches do not require the word to actually exist in the language lexicon (the set of all words in the language). Alternatively, some suffix stripping approaches maintain a database (a large list) of all known morphological word roots that exist as real words. These approaches check the list for the existence of the term prior to making a decision. Typically, if the term does not exist, alternate action is taken. This alternate action may involve several other criteria. The non-existence of an output term may serve to cause the algorithm to try alternate suffix stripping rules. It can be the case that two or more suffix stripping rules apply to the same input term, where there becomes an ambiguity in which rule to apply. The algorithm may assign a priority to one rule or another or the algorithm may reject one rule application because it results in a non-existent term whereas the other overlapping rule does not. For example, given the English term *friendlies*, the algorithm may identify the *ies* suffix and apply the appropriate rule and achieve the result of *friendl*. *friendl* is likely not found in the lexicon, and therefore the rule is rejected.

One improvement upon basic suffix stripping is the use of suffix substitution. Similar to a stripping rule, a substitution rule replaces a suffix with an alternate suffix. For example, there could exist a rule that replaces *ies* with *y*. How this affects the algorithm varies on the algorithm's design. To illustrate, the algorithm may identify that both the *ies* suffix stripping rule applies as well as the suffix substitution rule. Since the stripping rule results in a non-existent term in the lexicon, but the substitution rule does not, the substitution rule is applied instead. In this example,  *friendlies* becomes *friendly* instead of *friendl*. Diving further into the details, a common technique is to apply rules in a cyclical fashion (recursively, as computer scientists would say). After applying the suffix substitution rule in this example scenario, a second pass is made to identify matching rules on the term *friendly*, where the *ly* stripping rule is likely identified and accepted. In summary, *friendlies* becomes (via substitution) *friendly* which becomes (via stripping) *friend*.

This example also helps illustrate the difference between a rule based approach and a brute force approach. In a brute force approach, the algorithm would search for *friendlies* in the set of hundreds of thousands of inflected word forms and ideally find the corresponding root form *friend*. In the rule based approach, the three rules mentioned above would be applied in succession to converge on the same solution. Chances are that the rule based approach was faster.

**iii. Lemmatisation algorithms**

A more complex approach to the problem of determining a stem of a word is [lemmatisation](http://en.wikipedia.org/wiki/Lemmatisation). This process involves first determining the [part of speech](http://en.wikipedia.org/wiki/Part_of_speech) of a word, and applying different normalization rules for each part of speech. The part of speech is first detected prior to attempting to find the root since for some languages, the stemming rules change depending on a word's part of speech.

This approach is highly conditional upon obtaining the correct lexical category (part of speech). While there is overlap between the normalization rules for certain categories, identifying the wrong category or being unable to produce the right category limits the added benefit of this approach over suffix stripping algorithms. The basic idea is that, if we are able to grasp more information about the word to be stemmed, then we are able to more accurately apply normalization rules (which are, more or less, suffix stripping rules).

**iv.** **Stochastic algorithms**

[Stochastic](http://en.wikipedia.org/wiki/Stochastic) algorithms involve using probability to identify the root form of a word. Stochastic algorithms are trained (they "learn") on a table of root form to inflected form relations to develop a probabilistic model. This model is typically expressed in the form of complex linguistic rules, similar in nature to those in suffix stripping or lemmatisation. Stemming is performed by inputting an inflected form to the trained model and having the model produce the root form according to its internal rule-set, which again is similar to suffix stripping and lemmatisation, except that the decisions involved in applying the most appropriate rule, or whether or not to stem the word and just return the same word, or whether to apply two different rules sequentially, are applied on the grounds that the output word will have the highest probability of being correct (which is to say, the smallest probability of being incorrect, which is how it is typically measured).

Some lemmatisation algorithms are stochastic in that, given a word which may belong to multiple parts of speech, a probability is assigned to each possible part. This may take into account the surrounding words, called the context, or not. Context-free grammars do not take into account any additional information. In either case, after assigning the probabilities to each possible part of speech, the most likely part of speech is chosen, and from there the appropriate normalization rules are applied to the input word to produce the normalized (root) form.

 **v.**  **Hybrid approaches**

Hybrid approaches use two or more of the approaches described above in unison. A simple example is a [suffix tree](http://en.wikipedia.org/w/index.php?title=Probabilistic_suffix_tree&action=edit&redlink=1) algorithm which first consults a lookup table using brute force. However, instead of trying to store the entire set of relations between words in a given language, the lookup table is kept small and is only used to store a minute amount of "frequent exceptions" like "ran => run". If the word is not in the exception list, apply suffix stripping or lemmatisation and output the result.

**vi.** **Affix stemmers**

In [linguistics](http://en.wikipedia.org/wiki/Linguistics), the term [affix](http://en.wikipedia.org/wiki/Affix) refers to either a [prefix](http://en.wikipedia.org/wiki/Prefix) or a [suffix](http://en.wikipedia.org/wiki/Suffix). In addition to dealing with suffixes, several approaches also attempt to remove common prefixes. For example, given the word *indefinitely*, identify that the leading "in" is a prefix that can be removed. Many of the same approaches mentioned earlier apply, but go by the name affix stripping.

**vii.** **Matching algorithms**

Some algorithms use a stem database (for example a set of documents that contain stem words). These stems, as mentioned above, are not necessarily valid words themselves (but rather common sub-strings, as the "brows" in "browse" and in "browsing"). In order to stem a word the algorithm tries to match it with stems from the database, applying various constraints, such as on the relative length of the candidate stem within the word (so that, for example, the short prefix "be", which is the stem of such words as "be", "been" and "being", would not be considered as the stem of the word "beside").

**Error Metrics**

There are two error measurements in stemming algorithms, **overstemming** and **understemming**. Overstemming is an error where two separate inflected words are stemmed to the same root, but should not have been - a [false positive](http://en.wikipedia.org/wiki/False_positive). Understemming is an error where two separate inflected words should be stemmed to the same root, but are not - a [false negative](http://en.wikipedia.org/wiki/False_negative). Stemming algorithms attempt to minimize each type of error, although reducing one type can lead to increasing the other.

**Applications:**

1. **Information retrieval**

Stemmers are common elements in [query systems](http://en.wikipedia.org/wiki/Information_Retrieval) such as [Web](http://en.wikipedia.org/wiki/World_Wide_Web) [search engines](http://en.wikipedia.org/wiki/Search_engine), since a user who runs a query on "daffodils" would probably also be interested in documents that contain the word "daffodil" (without the s). The effectiveness of stemming for English query systems was soon found to be rather limited, however, and this has led early [Information retrieval](http://en.wikipedia.org/wiki/Information_retrieval) researchers to deem stemming irrelevant in general.

1. **Domain Analysis**

Stemming is used to determine domain vocabularies in [domain analysis](http://en.wikipedia.org/wiki/Domain_analysis).

1. **Use in commercial products**

Many commercial companies have been using stemming since at least the 1980s and have produced algorithmic and lexical stemmers in many languages. The Snowball stemmers have been compared with commercial lexical stemmers with varying results. [Google search](http://en.wikipedia.org/wiki/Google_search) adopted word stemming in 2003. Previously a search for "fish" would not have returned "fishing". Other software search algorithms vary in their use of word stemming. Programs that simply search for substrings obviously will find "fish" in "fishing" but when searching for "fishes" will not find occurrences of the word "fish".

**PORTERS STEMMING ALGORITHM**

Removing suffixes by automatic means is an operation which is especially useful in the field of information retrieval. In a typical IR environment, one has a collection of documents, each described by the words in the document title and possibly by words in the document abstract. Ignoring the issue of precisely where the words originate, we can say that a document is represented by a vector of words, or terms. Terms with a common stem will usually have similar meanings, for example: connect, connected, connecting, connection, connections.

Frequently, the performance of an IR system will be improved if term groups such as this are conflated into a single term. This may be done by removal of the various suffixes -ED, -ING, -ION, IONS to leave the single term CONNECT. In addition, the suffix stripping process will reduce the total number of terms in the IR system, and hence reduce the size and complexity of the data in the system, which is always advantageous.

Many strategies for suffix stripping have been reported in the literature. The nature of the task will vary considerably depending on whether a stem dictionary is being used, whether a suffix list is being used, and of course on the purpose for which the suffix stripping is being done. Assuming that one is not making use of a stem dictionary, and that the purpose of the task is to improve IR performance, the suffix stripping program will usually be given an explicit list of suffixes, and, with each suffix, the criterion under which it may be removed from a word to leave a valid stem. This is the approach adopted here. The main merits of the present program are that it is small (less than 400 lines of BCPL), fast (it will process a vocabulary of 10,000 different words in about 8.1 seconds on the IBM 370/165 at Cambridge University), and reasonably simple. At any rate, it is simple enough to be described in full as an algorithm in this paper. (The present version in BCPL is freely available from the author. BCPL is itself available on a wide range of different computers, but anyone wishing to use the program should have little difficulty in coding it up in other programming languages.) Given the speed of the program, it would be quite realistic to apply it to every word in a large file of continuous text, although for historical reasons we have found it convenient to apply it only to relatively small vocabulary lists derived from continuous text files.

In any suffix stripping program for IR work, two points must be borne in mind. Firstly, the suffixes are being removed simply to improve IR performance, and not as a linguistic exercise. This means that it would not be at all obvious under what circumstances a suffix should be removed, even if we could exactly determine the suffixes of a word by automatic means.

Perhaps the best criterion for removing suffixes from two words W1 and W2 to produce a single stem S, is to say that we do so if there appears to be no difference between the two statements `a document is about W1' and `a document is about W2'. So if W1=`CONNECTION' and W2=`CONNECTIONS' it seems very reasonable to conflate them to a single stem. But if W1=`RELATE' and W2=`RELATIVITY' it seems perhaps unreasonable, especially if the document collection is concerned with theoretical physics. (It should perhaps be added that RELATE and RELATIVITY \are\ conflated together in the algorithm described here.) Between these two extremes there is a continuum of different cases, and given two terms W1 and W2, there will be some variation in opinion as to whether they should be conflated, just as there is with deciding the relevance of some document to a query. The evaluation of the worth of a suffix stripping system is correspondingly difficult.

The second point is that with the approach adopted here, i.e. the use of a suffix list with various rules, the success rate for the suffix stripping will be significantly less than 100% irrespective of how the process is evaluated. For example, if SAND and SANDER get conflated, so most probably will WAND and WANDER. The error here is that the -ER of WANDER has been treated as a suffix when in fact it is part of the stem. Equally, a suffix may completely alter the meaning of a word, in which case its removal is unhelpful. PROBE and PROBATE for example, have quite distinct meanings in modern English. (In fact these would not be conflated in our present algorithm.) There comes a stage in the development of a suffix stripping program where the addition of more rules to increase the performance in one area of the vocabulary causes an equal degradation of performance elsewhere. Unless this phenomenon is noticed in time, it is very easy for the program to become much more complex than is really necessary. It is also easy to give undue emphasis to cases which appear to be important, but which turn out to be rather rare. For example, cases in which the root of a word changes with the addition of a suffix, as in DECEIVE/DECEPTION, RESUME/RESUMPTION, INDEX/INDICES occur much more rarely in real vocabularies than one might at first suppose.

**THE ALGORITHM**

To present the suffix stripping algorithm in its entirety we will need a few definitions.

A consonant in a word is a letter other than A, E, I, O or U, and other than Y preceded by a consonant. (The fact that the term `consonant' is defined to some extent in terms of itself does not make it ambiguous.) So in TOY the consonants are T and Y, and in SYZYGY they are S, Z and G. If a letter is not a consonant it is a vowel.

A consonant will be denoted by c, a vowel by v. A list ccc... of length greater than 0 will be denoted by C, and a list vvv... of length greater than 0 will be denoted by V. Any word, or part of a word, therefore has one of the four forms:

CVCV ... C

CVCV ... V

VCVC ... C

VCVC ... V

These may all be represented by the single form

[C]VCVC ... [V]

where the square brackets denote arbitrary presence of their contents.

Using (VC){m} to denote VC repeated m times, this may again be written as

[C](VC){m}[V].

m will be called the measure of any word or word part when represented in

this form. The case m = 0 covers the null word. Here are some examples:

m=0 TR, EE, TREE, Y, BY.

m=1 TROUBLE, OATS, TREES, IVY.

m=2 TROUBLES, PRIVATE, OATEN, ORRERY.

The \rules\ for removing a suffix will be given in the form

(condition) S1 -> S2

This means that if a word ends with the suffix S1, and the stem before S1satisfies the given condition, S1 is replaced by S2. The condition is usually given in terms of m, e.g.

(m > 1) EMENT ->

Here S1 is `EMENT' and S2 is null. This would map REPLACEMENT to REPLAC,

since REPLAC is a word part for which m = 2.

The `condition' part may also contain the following:

\*S - the stem ends with S (and similarly for the other letters).

\*v\* - the stem contains a vowel.

\*d - the stem ends with a double consonant (e.g. -TT, -SS).

\*o - the stem ends cvc, where the second c is not W, X or Y (e.g.-WIL, -HOP).

And the condition part may also contain expressions with \and\, \or\ and \not\, so that (m>1 and (\*S or \*T)) tests for a stem with m>1 ending in S or T, while (\*d and not (\*L or \*S or \*Z)) tests for a stem ending with a double consonant other than L, S or Z. Elaborate conditions like this are required only rarely.

In a set of rules written beneath each other, only one is obeyed, and this will be the one with the longest matching S1 for the given word. For example, with

SSES -> SS

IES -> I

SS -> SS

S ->

(here the conditions are all null) CARESSES maps to CARESS since SSES is the longest match for S1. Equally CARESS maps to CARESS (S1=`SS') and CARES to CARE (S1=`S').

In the rules below, examples of their application, successful or otherwise, are given on the right in lower case. The algorithm now follows:

Step 1a

SSES -> SS caresses -> caress

IES -> I ponies -> poni

ties -> ti

SS -> SS caress -> caress

S -> cats -> cat

Step 1b

(m>0) EED -> EE feed -> feed

 agreed -> agree

(\*v\*) ED -> plastered -> plaster

 bled -> bled

(\*v\*) ING -> motoring -> motor

 sing -> sing

If the second or third of the rules in Step 1b is successful, the following is done:

AT -> ATE conflat(ed) -> conflate

BL -> BLE troubl(ed) -> trouble

IZ -> IZE siz(ed) -> size

(\*d and not (\*L or \*S or \*Z) -> single letter hopp(ing) -> hop

 tann(ed) -> tan

 fall(ing) -> fall

 hiss(ing) -> hiss

 fizz(ed) -> fizz

 (m=1 and \*o) -> E fail(ing) -> fail

 fil(ing) -> file

The rule to map to a single letter causes the removal of one of the double letter pair. The -E is put back on -AT, -BL and -IZ, so that the suffixes -ATE, -BLE and -IZE can be recognised later. This E may be removed in step 4.

Step 1c

(\*v\*) Y -> I happy -> happi

 sky -> sky

Step1 deals with plurals and past participles. The subsequent steps are much more straightforward.

Step 2

(m>0) ATIONAL -> ATE relational -> relate

(m>0) TIONAL -> TION conditional -> condition

 rational -> rational

(m>0) ENCI -> ENCE valenci -> valence

(m>0) ANCI -> ANCE hesitanci -> hesitance

(m>0) IZER -> IZE digitizer -> digitize

(m>0) ABLI -> ABLE conformabli -> conformable

(m>0) ALLI -> AL radicalli -> radical

(m>0) ENTLI -> ENT differentli -> different

(m>0) ELI -> E vileli - > vile

(m>0) OUSLI -> OUS analogousli -> analogous

(m>0) IZATION -> IZE vietnamization -> vietnamize

(m>0) ATION -> ATE predication -> predicate

(m>0) ATOR -> ATE operator -> operate

(m>0) ALISM -> AL feudalism -> feudal

(m>0) IVENESS -> IVE decisiveness -> decisive

(m>0) FULNESS -> FUL hopefulness -> hopeful

(m>0) OUSNESS -> OUS callousness -> callous

(m>0) ALITI -> AL formaliti -> formal

(m>0) IVITI -> IVE sensitiviti -> sensitive

(m>0) BILITI -> BLE sensibiliti -> sensible

The test for the string S1 can be made fast by doing a program switch on the penultimate letter of the word being tested. This gives a fairly even breakdown of the possible values of the string S1. It will be seen in fact that the S1-strings in step 2 are presented here in the alphabetical order of their penultimate letter. Similar techniques may be applied in the other steps.

Step 3

(m>0) ICATE -> IC triplicate -> triplic

(m>0) ATIVE -> formative -> form

(m>0) ALIZE -> AL formalize -> formal

(m>0) ICITI -> IC electriciti -> electric

(m>0) ICAL -> IC electrical -> electric

(m>0) FUL -> hopeful -> hope

(m>0) NESS -> goodness -> good

Step 4

(m>1) AL -> revival -> reviv

(m>1) ANCE -> allowance -> allow

(m>1) ENCE -> inference -> infer

(m>1) ER -> airliner -> airlin

(m>1) IC -> gyroscopic -> gyroscop

(m>1) ABLE -> adjustable -> adjust

(m>1) IBLE -> defensible -> defens

(m>1) ANT -> irritant -> irrit

(m>1) EMENT -> replacement -> replac

(m>1) MENT -> adjustment -> adjust

(m>1) ENT -> dependent -> depend

(m>1 and (\*S or \*T)) ION -> adoption -> adopt

(m>1) OU -> homologou -> homolog

(m>1) ISM -> communism -> commun

(m>1) ATE -> activate -> activ

(m>1) ITI -> angulariti -> angular

(m>1) OUS -> homologous -> homolog

(m>1) IVE -> effective -> effect

(m>1) IZE -> bowdlerize -> bowdler

The suffixes are now removed. All that remains is a little tidying up.

Step 5a

(m>1) E -> probate -> probat

 rate -> rate

(m=1 and not \*o) E -> cease -> ceas

Step 5b

(m > 1 and \*d and \*L) -> single letter controll -> control

 roll -> roll

The algorithm is careful not to remove a suffix when the stem is too short, the length of the stem being given by its measure, m. There is no linguistic basis for this approach. It was merely observed that m could be used quite effectively to help decide whether or not it was wise to take off a suffix. For example, in the following two lists:

 list A list B

 ------ ------

RELATE DERIVATE

PROBATE ACTIVATE

CONFLATE DEMONSTRATE

PIRATE NECESSITATE

PRELATE RENOVATE

-ATE is removed from the list B words, but not from the list A words. This means that the pairs DERIVATE/DERIVE, ACTIVATE/ACTIVE, DEMONSTRATE/DEMONS- TRABLE, NECESSITATE/NECESSITOUS, will conflate together. The fact that no attempt is made to identify prefixes can make the results look rather inconsistent. Thus PRELATE does not lose the -ATE, but ARCHPRELATE becomes ARCHPREL. In practice this does not matter too much, because the presence of the prefix decreases the probability of an erroneous conflation.

Complex suffixes are removed bit by bit in the different steps. Thus GENERALIZATIONS is stripped to GENERALIZATION (Step 1), then to GENERALIZE (Step 2), then to GENERAL (Step 3), and then to GENER (Step 4). OSCILLATORS is stripped to OSCILLATOR (Step 1), then to OSCILLATE (Step 2), then to OSCILL (Step 4), and then to OSCIL (Step 5). In a vocabulary of 10,000 words, the reduction in size of the stem was distributed among the steps as

follows:

Suffix stripping of a vocabulary of 10,000 words

------------------------------------------------

Number of words reduced in step 1: 3597

Number of words reduced in step 2: 766

Number of words reduced in step 3: 327

Number of words reduced in step 4: 2424

Number of words reduced in step 5: 1373

**Number of words not reduced: 3650**

The resulting vocabulary of stems contained 6370 distinct entries. Thus the suffix stripping process reduced the size of the vocabulary by about one third.