

# CSE 431 NATURAL LANGUAGE PROCESSING

## STEMMING IN BANGLA

Stemming refers to root word origins. For example, Search, Searching, and Searches all have Search as the root stem. In most cases, morphological variants of words have similar semantic interpretations and can be considered as equivalent for the purpose of IR applications. For this reason, a number of so-called *stemming Algorithms*, or *stemmers*, have been developed, which attempt to reduce a word to its *stem* or root form. Thus, the key terms of a query or document are represented by stems rather than by the original words. This not only means that different variants of a term can be *conflated* to a single representative form – it also reduces the *dictionary size*, that is, the number of distinct terms needed for representing a set of documents. A smaller dictionary size results in a saving of storage space and processing time.

For IR purposes, it doesn't usually matter whether the stems generated are genuine words or not – thus, "computation" might be stemmed to "comput" – provided that (a) different words with the same 'base meaning' are conflated to the same form, and (b) words with distinct meanings are kept separate. An algorithm which attempts to convert a word to its linguistically correct root ("compute" in this case) is sometimes called a *lemmatiser*.

Before starting the project we search then net a lot and some stemming algorithms that are already implemented.

### **Different Stemming Algorithms:**

- [Paice/Husk Stemming Algorithm](#)
- [Porter Stemming Algorithm](#)
- [Lovins Stemming Algorithm](#)
- [Dawson Stemming Algorithm](#)
- [Krovetz Stemming Algorithm](#)

### [Paice/Husk Stemming Algorithm:](#)

The Paice/Husk Stemmer is a simple iterative Stemmer – that is to say, it removes the endings from a word in an indefinite number of

steps. The Stemmer uses a separate rule file, which is first read into an array or list. This file is divided into a series of sections, each section corresponding to a letter of the alphabet. The section for a given letter, say "e", contains the rules for all endings ending with "e", the sections being ordered alphabetically. An index can thus be built, leading from the last letter of the word to be stemmed to the first rule for that letter.

When a word is to be processed, the stemmer takes its last letter and uses the index to find the first rule for that letter. The rule is examined, and is accepted if:

- It specifies an ending which matches the last letters of the word.
- Any special conditions for that rule are satisfied (e.g, the so-called 'intact' condition, which ensures that the rule is only fired if no other rules have yet been applied to the word).
- Application of the rule would not result in a stem shorter than a specified length or without a vowel.

If a rule is accepted then it is applied to the word. If it is not accepted, the rule index is incremented by one and the next rule is tried. However, if the first letter of the next rule does not match with the last letter of the word, this implies that no ending can be removed, and so the process terminates.

When a rule is applied to a word, this usually means that the ending of the word is removed or replaced. For example, the rule

```
e1> { -e - }
```

means 'if the current word/stem ends with "e" then delete 1 letter and continue' (the curly brackets just contain a comment showing the rule in another form). So this is a simple 'e-removal' rule, which for example would convert "estate" to "estat". After applying this rule, the new final letter (now "t") would be taken and used to access a different section of the rule table. If, however, the final symbol had been "." instead of ">", the process would have terminated, and "estat" would have been returned at once.

Suppose now that the rule had said:

```
e1i> { -e -i }
```

In this case, the "e" would have been removed and then replaced by the letter "i" – giving, in the present case, "estati".

Once a rule has been found to match, it is not applied at once, but must first be checked to confirm that it would leave an acceptable stem. For example, it would not be sensible to apply the 'e-removal'

rule to the word "me", since the remaining stem would be too short - and would not even contain a vowel!

#### Porter Stemming Algorithm :

The Porter Stemmer is a conflation Stemmer developed by Martin Porter at the University of Cambridge in 1980. The Stemmer is based on the idea that the suffixes in the English language (approximately 1200) are mostly made up of a combination of smaller and simpler suffixes. This Stemmer is a linear step Stemmer. Specifically it has five steps applying rules within each step. Within each step, if a suffix rule matched to a word, then the conditions attached to that rule are tested on what would be the resulting stem, if that suffix was removed, in the way defined by the rule. For example such a condition may be, the number of vowel characters, which are followed by a consonant character in the stem (Measure), must be greater than one for the rule to be applied.

Once a Rule passes its conditions and is accepted the rule fires and the suffix is removed and control moves to the next step. If the rule is not accepted then the next rule in the step is tested, until either a rule from that step fires and control passes to the next step or there are no more rules in that step whence control moves to the next step. This process continues for all five steps, the resultant stem being returned by the Stemmer after control has been passed from step five.

#### Lovins Stemming Algorithm :

The Lovins Stemmer is a single pass, context-sensitive, longest-match Stemmer developed by Julie Beth Lovins of Massachusetts Institute of Technology in 1968. This early stemmer was targeted at both the IR and Computational Linguistics areas of stemming.

This stemmer, though innovative for its time, has the problematic task of trying to please two masters (IR and Linguistics) and cannot excel at either. The approach does not excel with linguistics, as it is not complex enough to stem many suffixes due to their not being present in the rule list. This is interesting as Lovins' rule list was derived by, processing and studying a word sample. Perhaps if this process was repeated with a much larger sample a more satisfactory rule list could be derived. There are also known to be problems regarding the reformation of words. This process uses the recoding rules to reform the stems into words to ensure they match stems of other similar meaning words. The main problem with this process is that it has been found to be highly unreliable and frequently fails to form words from the stems, or match the stems of like meaning words. The Stemmer does not excel from the IR viewpoint either, as its large rule set, and

its recoding stage, affect its speed of execution. As discussed above, it has also been found to be unreliable.

The Lovins Stemmer removes a maximum of one suffix from a word, due to its nature as single pass algorithm. It uses a list of about 250 different suffixes, and removes the longest suffix attached to the word, ensuring that the stem after the suffix has been removed is always at least 3 characters long. Then the ending of the stem may be reformed (e.g., by un-doubling a final consonant if applicable), by referring to a list of recoding transformations.

#### Dawson Stemming Algorithm :

The Dawson Stemmer was developed by J.L. Dawson of the Literary and Linguistics Computing Centre at Cambridge University. It is a complex linguistically targeted Stemmer that is strongly based upon the Lovins Stemmer, extending the suffix rule list to approximately 1200 suffixes. It keeps the longest match and single pass nature of Lovins, and replaces the recoding rules, which were found to be unreliable, using instead an extension of the partial matching procedure also defined within the Lovins Paper.

#### Krovetz Stemming Algorithm:

The Krovetz Stemmer was developed by Bob Krovetz, at the University of Massachusetts, in 1993. It is quite a 'light' stemmer, as it makes use of inflectional linguistic morphology.

The area of morphology (the internal structure of words) can be broken down into two subclasses, inflectional and derivational. Inflectional morphology describes predictable changes a word undergoes as a result of syntax (the plural and possessive form for nouns, and the past tense and progressive form for verbs are the most common in English). These changes have no effect on a word's 'part-of-speech' (a noun still remains a noun after pluralizations). In contrast, changes of derivational morphology may or may not affect a word's meaning (e.g.; '-ise', '-ship'). Although English is a relatively weak morphological language, languages such as Hungarian and Hebrew have stronger morphology where thousands of variants may exist for a given word. In such a case the retrieval performance of an IR system would be severely be impacted by a failure to deal with such variations.

The Krovetz Stemmer effectively and accurately removes inflectional suffixes in three steps, the conversion of a plural to its single form (e.g. '-ies', '-es', '-s'), the conversion of past to present tense (e.g. '-ed'), and the removal of '-ing'. The conversion process firstly removes the suffix, and then through a process of checking in a dictionary for any recoding (also being aware of exceptions to the normal recoding

rules), returns the stem to a word. The low level of strength with the English language due to nature of the Stemmer, causes issues with its usage within the field of IR, where an increased level of strength and index compression may be sought. For this reason, this Stemmer is frequently used in conjunction with other Stemmers, making use of the advantage of the accuracy of removal of suffixes by this Stemmer, which then adds the compression of another Stemmer, such as the Paice/Husk Stemmer or Porter Stemmer.

### **Dutch and Hindi stemmer:**

We also look at stemming on another languages like dutch and hindi. In both cases they just follow the porter stemming.

### **Bangla Stemmer:**

Stemming is the process of adding / deleting suffixes from a word. In bangla there is both prefixes and postfixes. Prefix are those affixes that are added in the beginning of a word. In bangla these are like Ao, pori etc. These are added to the starting of a word. The postfixes are those that are added at the end of a word. These are shown in bangla usually. There are situations when many postfixes can be added. For example the "eachilam". It is like e+a+chi +lum. All these 4 postfixes are added in the end of root. For recognition we follow two approach to solve this. In first approach we find the last suffix then the previous one until we come to an end. In another approach we consider the "eachilam" as an individual suffix. We put the "same categorized" large suffixes in the upper side. Then try to find the suffix in the given word. Then we cut the suffix and return.

For generation we add suffix with the root word. Here we have some problem. For example, in bangla "Tana" +"i" becomes "Tani" on the other hand "Tana" +"e" becomes "Tene". It is because of the phonology. We solve it by just coding implementation.

However not all the time the stemmer will give the correct answer. There are situation when the stemmer may overstem. This is like it may chop the word as well . In such kind of situation the answer will obviously not be correct. However in this bangla stemmer we tried our best not to do over stemming.

Understemming is another problem when are suffixes and it is not removed. This will also similarly give the wrong answer. In such kind of situation we again need to find the root. In our stemmer there is few situation when the understemming is happened. This is because we applied the brute force algorithm witch will find all the suffixes in a sequentially. However it is not guaranteed that all the time it will not understem.

We can use stemming in spell checker. All the word in a corpus in not in root form. So it a problem to check the spell of a word. So by a stemmer we can get the root of a word and check it's spell.

In future we have some work left to do. We will compare the performance of our both approach. In our project we work only for verb. We will work on noun and adjective also. We will try to combine our stemmer with spell checker.

## References:

1. The Porter Stemming Algorithm by [Martin Porter](#)
2. Automatic Language-Specific Stemming in Information Retrieval by John A. Goldsmith<sup>1</sup>, Derrick Higgins<sup>2</sup>, and Svetlana Soglasnova<sup>3</sup>
3. [www.comp.lancs.ac.uk/computing/research/stemming](http://www.comp.lancs.ac.uk/computing/research/stemming)