MSAN
Summer 2017
Natural Language Processing

Day 02 Agenda

Notes

Basic Information Retrieval

Goal

The goal of this lecture is to explain the various parts to an Information Retrieval system.

The general problem that Information Retrieval attempts to solve is the retrieval of (all) documents in a collection which are related to a query. This is, of course, the way many companies, including Yahoo and Google, have monetized. The term “Information Retrieval” is attributed to Calvin Mooers 1948-1950.

A slightly more formal definition would be:
Inputs:

- A set of documents, D, each described by a document ID

- A set of search terms, S in conjunctive form

Outputs:

- A set of document IDs, $I \subset D$, such that $S \in I$

As a motivating toy example, we let's:

- Use two documents: ir-doc01.txt, ir-doc05.txt
- Use the file names as document IDs
- Use search terms: *moon* and *korea*

Concepts & Terminology

You already have some concepts and terminology from above. This list is (partially) repeated from above, with a few added.

- Document: a searchable file with a unique ID. Typical documents are in HTML, PDF, text, PDF, Word Document. An Information Retrieval system will usually support some document types and will be built to extract the supported types.
- Collection / Corpus: a set of documents.
- Token: a word instance in a corpus or in a search term. There are 5 tokens in "the grass in the park" — regardless of whether this is in a corpus or in a search term.
- Type: a lexical entry in a corpus or in a search term. There are 4 tokens in "the grass in the park" — specifically, there are two instances of "the" in this example.
- [Search] Term: a set of tokens used for searching the documents. The search term is assumed to be expressed in conjunctive form. If there are two tokens in the search term, s_1 and s_2, we require that both terms appear in any documents being returned by the Information Retrieval system.

It may be interesting to note that there is a conjunctive normal form for expressing boolean clauses. Conjunctive normal form may be useful in some Information Retrieval systems, and it may be supported by some; however, we not require this.

Approach: grep

In our first attempt, we can use the Unix utility `grep`. This utility has an option ("-l") for printing the name of the file matching the search term. This will be our first attempt. Restating (and slightly simplifying) the problem:

- We have two documents: ir-doc01.txt, ir-doc05.txt
- We have a search term: *korea*
- We want the file names containing the search terms

Assuming the documents are in the same local directory with nothing else in the same directory, the following Unix command should get us close to something which would work:

```
grep -l korea *
```

This fails because the `grep` utility is case sensitive. Let's try again, using the "-i" option, which ignores the case of the thing being searched for:

```
grep -l -i korea *
```

This works, giving us the file name (ir-doc05.txt) we want.

In general, the `grep` utility has a number of options which will help us with Information Retrieval tasks. Like many Unix utilities, we can use `man grep` to read a list of options and how those options can be used. One of the options, "-v" allows us to require a term be absent — useful.

Note that we can combine options using only one "-" symbol, like:

```
grep -li korea *
```

Let's do this from now on since it saves us at least 16 electrons.

Scaling up slightly: recall that our original search was for "*moon* and *korea*". Unix utilities allow us to do this in a few ways. Perhaps the easiest way is to place all the search terms in a string, specifying that any number of characters can be placed between them. We can express this with a "." — short for "any [single] character — and "*" — any number of repeats. For slightly annoying reasons, we have to place this between single quotes. Putting this all together, our modified command looks like this:

```
grep -li 'moon.*korea' *
```

… and, of course, this fails. Why? Let's look inside our document. It clearly contains our search terms, specifically:

```
[...] Korean election winner Moon [...]
```

… but the order is the opposite of what we specified in the `grep` command. To be clear, we want to find documents which contain all of the search terms in any order. In other words, this is an incorrect non-result. We can re-work the command to get instances of *moon* followed by *korea* or *korea* followed by *moon*. Here's one (of many) ways:

```
grep -li 'moon.*korea\|korea.*moon' *
```

If there's any magic in the above, it is in the "|" command — a boolean "or" — which is "escaped" by the "\" character.

File under "added material:" there are different flavours of the `grep` command. In one of these, `egrep`, we do not need to use the "\" character to escape the "|" character. In short, we can do the following:

```
egrep -li 'moon.*korea|korea.*moon' *
```

… but all that really does is save us one character while failing to solve several much larger problems. Two of the most glaring are below.

Problem 1: If we have two search terms, we need a grep with two "or"-ed conditions — one for each possible ordering of "moon" and "korea." How many conditions do we need for three search terms? Four? Five? This approach fails to scale.

Problem 2: We made the strong assumption that the files we are searching are not just local but also in the same directory. We could combine the `grep` commands with other Unix utilities like `find`, which allows us to look at a number of directories to get around the “only local directories,” and we could use other Unix utilities (“mount”?) to extend the local filesystem, but instead, we’ll look at a different approach.

Approach: Boolean Retrieval Model

While using the `grep` utility had its shortcomings, it was effective for a small list of search terms on a tiny, local corpus. Another possible method for Information Retrieval systems, called the ***Boolean Retrieval Model***, is inspired by this but overcomes the two problems we identified.

There are a few requirements for the boolean retrieval model. The most important is a dictionary (ala python dictionary) which maps from terms found in the corpus to individual documents. Let’s inspect our documents and construct a dictionary. We’ll do this by hand and the move to automated ways to perform the construction.

Our *ir-doc01.txt* document contains the following:

```
My daughter wants to drive my colourful car, but at the age of
2, she's too young.
```

Therefore, we want our dictionary to look like the following:

```
My : [ ir-doc01.txt ]
daughter : [ ir-doc01.txt ]
wants : [ ir-doc01.txt ]
to : [ ir-doc01.txt ]
```

… etc. The example above is not specifically using Python syntax, but the target could be a set or a list. There are advantages to both. A set, for example, gives us low overhead. A list allows us to establish some priorities — some documents could be more important than others. For the remainder of this exercise, let’s use a set.

Continuing our exercise with the *ir-doc02.txt* document, which contains the following:

```
South Korean election winner Moon Jae-in could work with U.S.
on
North.
```

… we want to add each type from the *ir-doc02.txt* document to our dictionary with that file as the target, like:

```
Jae-in : [ ir-doc02.txt ]
Korean : [ ir-doc02.txt ]
My : [ ir-doc01.txt ]
North : [ ir-doc02.txt ]
```

```
South : [ ir-doc02.txt ]
U.S. : [ ir-doc02.txt ]
daughter : [ ir-doc01.txt ]
election : [ ir-doc02.txt ]
wants : [ ir-doc01.txt ]
to : [ ir-doc01.txt ]
```

… etc. As it turns out, there is no overlap between the tokens in our two documents, so we would end up with a dictionary in which the keys point to single values. But that doesn't matter.

Let's write some code to build this dictionary. Let's simplify the problem a little. Specifically, let's only consider one search term (eg. "*election*"). This will allow us to build a rapid proof of concept to which we can add sophistication later. Given the simple example, let's make some design choices.

Figure 1 shows what is called a class diagram. It is a tool for quickly determining the important parts of an object oriented class. Each class in a class diagram has three sections. From top to bottom, these are: the name of the class, the data contained in the class and the methods (functions) on the class. We tend not to show constructors (the __init__ method) or destructors in the class diagrams. What we do show is whether the data or methods should be public — indicated with a "+" symbol — private ("-") or protected ("#"). This has implications in terms of where data is allowed to be changed or how methods can be called. In python, it doesn't matter too much.

From the class diagram, we can see the the "ir_system" class should have a (protected) dictionary called "__system_dictionary__" and two (public) methods, read_file() and get_documents_matching().

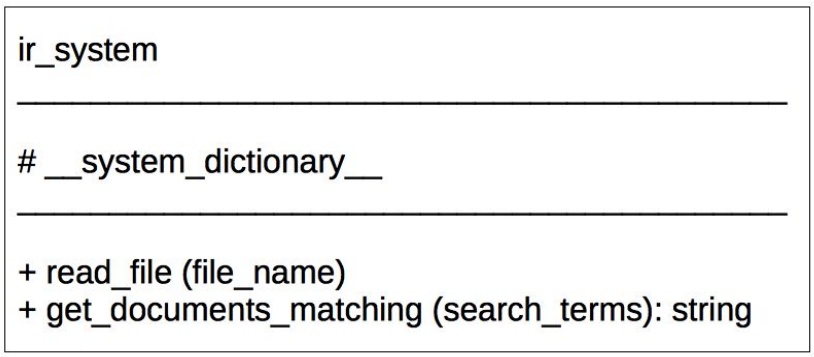

Figure 1: ir_system UML class diagram

I've written the "main" in a script called ir.py. (Warning: it is a proof-of-concept only.) Get it from github at https://github.com/dbrizan/MSAN-NLP/blob/master/ir.py if you like. If we need to save time, we can also get the ir_system.py. That said, it's just 50 lines of code, including comments, so we should try implementing it without looking.

Test this implementation with search terms like “daughter” and “election” … and also with terms not in our corpus, like “Narita” or whatever else you like. Does this approach work?

Recall that our exemplar use case is the return of document IDs containing our search terms: *moon* and *korea* from some arbitrary document collection, though we’re using two local files. In no particular order, here are some useful questions to ask:

- Can the boolean retrieval model support remote documents?
- Since speed is a consideration in all systems: can the boolean retrieval model support this in a reasonable amount of time — i.e. quickly?
- Can this boolean retrieval model support our use case? Specifically, can the boolean retrieval model support searching multiple search terms without machinations?

Guided by these questions, let’s think through the implications of the boolean retrieval model.

Question 1: Can the boolean retrieval model support remote documents?
Short answer: Yes.
Longer answer: The target document ID can be in any format. It can include a server name, a protocol for retrieving the document, a path to the document, a user ID and password, etc. Among other things, it can be a URL.

Question 2: Can the boolean retrieval model support [retrieval] in a reasonable amount of time?
Short answer: Most likely.
Longer answer: Simplifying to one token being searched, in order for the boolean retrieval model to find the search term, it uses the search term as the key to a dictionary and returns the values which are indexed by that key. Where “n” is the number of keys in the dictionary, the “get” method has a worst case running time of O(n).
Recall that a dictionary is really a hash table: an array in which the indices are converted from strings (in this example) to array positions, called buckets. The running time of the “get” method on a hash table depends on a number of factors, primarily the number of collisions. A collision is defined as two keys hashing to the same bucket. The number of collisions can be predicted from the *load factor*, defined as the number of items in the dictionary, *n*, divided by the number of buckets for those items, *k*.
Many languages, including python, keep the load factor low by automatically resizing the dictionary and amortising the cost of this over many operations. In this case, the (amortised) average running time for the “get” method of a dictionary is much closer to O(1). Assuming the “get” method stays close to O(1), we should be happy with this running time.

Question 3: Can this boolean retrieval model support our use case?
Short answer: Maybe… with modifications.
Longer answer: It depends on how we construct the keys for the dictionary. One of the things we may have noticed is that our keys contained a number of mixed case entries: *Korean* and *Moon*, for example. In our example, the search terms appeared in lowercase. In our first attempt approach, we ignored case using the -i option to `grep`. We need to do something about in order to fix this problem.

Along the same lines, we can find the search term “North.” (with the punctuation), but we get no valid results from the search term without the punctuation. It seems reasonable to fix both these problems by tokenizing[1], case standardisation, etc. Here are some points for improvement:

- We can convert both the document contents and the search terms (to lowercase?). Conventionally, Information Retrieval system perform the case conversion in a way that is invisible to the user: if the user searches for “*Korea*” (mixed case), the Information Retrieval system claims to use the mixed case version but really does not. It’s U.I. “sugar,” not something core to the functioning of the system, but something to keep in mind.
- We can handle punctuation in a way that makes sense. Some punctuation symbols like (`.,?;:`) should probably just be ignored in the document and in the search terms, though there is a potential problem with this: a type or search term like “U.S.” (which appears in ir-doc05.txt) would become “U.S” … or worse: “U” and “S”. Other punctuation (eg. the apostrophe in “she’s”) should be used to split tokens for further processing.
- We can create a set of words which are unlikely to be useful either in a dictionary or in search terms. These are known as “stop words.” Generally, these words are “function words” in English, such as articles (“a”, “the), prepositions (“of”, “by”) or pronouns (“them”, “mine”). We could automatically determine these words since they should have the lowest tf-idf scores. Alternately, these words are generally closed class words (in English), meaning these words belong to a grammar class for which all the words belonging to the class are known. (This is in contrast to “open class” words like nouns, verbs, adjectives and adverbs. As we invent new things and concepts, we can add new words to the open class.)
- Using the same example as we started with, we can create a mechanism for retrieving documents with the word “*Korean*” even when the search term includes “*Korea.*” This is probably the most important thing we can do to improve our Information Retrieval system. For this, perhaps we can leverage some linguistic knowledge.

Considerations

There are a few things we did not discuss: evaluation, spelling correction and web crawling. Let’s address those quickly before we move on to the linguistic theory we need to build a really excellent Information Retrieval system.

Generally, Information Retrieval systems are evaluated with F_1 (also called F1, F-score or F-measure), the harmonic mean of precision and recall. By now, we all know about these terms, so let’s move on.

[1] A note on the difference between American spelling — as appears here — and the British / Colonial / world spelling, which the author normally uses: the convention from programming languages is to use the American “tokenize,” and not “tokenise.” This deviation to American spelling is intentional and meant to support the conversion to a programming language.

It should come as no surprise that users who create documents and users who generate search terms both create spelling errors. Spelling correction is a non-trivial problem, but we touched on it briefly in our previous meeting. As a trivial treatment, we can at least detect spelling errors by "tokens not in our lexicon" and mitigate once we've detected a spelling error.

Lastly, if our documents come from the web, many web sites create a robots.txt to guide an Information Retrieval system during the collection period. This file may list a set of directories or items available to be retrieved, a set forbidden from being retrieved, a user ID and password to be used in retrieval, etc. If we are writing a spider or web crawler, we should become familiar with that file and its format.

Cool. On to linguistics!

Morphology

Morphology is the study of how words are formed. Using morphology, we divide words into parts, often called roots and stems. Generally speaking, stems are additions which make changes to the word in question.

Recalling Emily Bender's tweet, one of the things which remains the same in a language is the way in which we form plurals. In English, we can generally add an "s" to the end of a singular noun to make it a plural. For example:

dog —> dogs

That is the general rule. What if the word ends in -y? Generally, we drop the "y" and add "ies":

butterfly —> butterflies

There are more rules. What if the word ends in -x, -s, -z? We add "es":

fax —> faxes

Sometimes the same is the case if the word ends in -sh:

wish —> wishes

Some English words have singular and plural forms which are orthographically identical — i.e. the singular and plural are written the same way:

sheep —> sheep | fish —> fish

… and finally, some words are just irregular:

goose —> geese | child —> children

Types of morphology

The specific type of morphology above is called ***inflectional morphology***. In inflectional morphology, the word in question does not change part of speech ("child" and "children" are

---

both nouns). We care about this for Information Retrieval systems because if someone searches for “cars for children,” we want results returned from documents containing all instances “car for child” — whether singular or plural.

English inflectional morphology is reasonably easy. In Romantic languages — French, Italian, Portuguese and Spanish, for example, nouns are marked to distinguish gender and number. The French word for *dog* is *chien*, . Here are is a little inflectional morphology:

chien: male dog singular
chienne: female dog singular
chiens: male dog plural — OR — dogs of mixed gender
chiennes: female dog plural

Other languages have morphological markers to distinguish pairs of things (i.e. singular, pairs, groups of more than two), etc.

English verbs also have a relatively light treatment. Commonly, we can do the following:

Change to past tense: they hammer —> they hammered
Change to continuous tense: sleep —> sleeping

There is one general exception to this in English: second person singular. When we conjugate a regular verb, like “to write”, we get 3rd person singular = infinitive +s. For example:

write —> she writes

but in all other cases {I, you, we, you, they} write. Why? It may be a long story having to do with the verb “to be.” What else can we do in English with verbs?

In Romance languages, verbs are often marked for person, number and tense. The Spanish word *comer* (to eat) has 5 - 6 present-tense forms (depending on the Spanish dialect): como, comes, come, comemos, coméis, comen. For reference, see http://www.spanishdict.com/conjugate/comer.

English adjectives may also be marked for morphology, specifically in terms of the comparative / superlative:

happy —> happier | happiest

Even though English has a relatively poor inflectional morphology system, it still presents us with a problem: we cannot store all the words and their inflections, if only because we don’t have the space and data structures to do so. More importantly, English changes over time — new word types are added to the open classes of the lexicon. The only way to keep up is with rules.

If we have time, let’s discuss some thought questions:

- Are there other things we can do with nouns, verbs, adjectives?
- How many morphemes are there in English? How many in other languages?
- Is the set of morphemes an open set or a closed set?

Although English inflectional morphology is poor, English ***derivational morphology*** is extremely rich. Using derivational morphology, we are able to change the class of a word. Here are some examples:

- +ise | +ize = verb (computer —> computerise | computerise)
- +ation | +ee | +er | +ness = noun (kill —> killer)
- +al | +able | +less = adjective (embrace —> embraceable)

A third form of morphology is ***cliticisation*** (***cliticization***) in which we are able to shorten a word. An example in English is the [‘s] in she’s (indicating “she is” or “she has”). We distinguish between *proclitics* which precede a word and *enclitics* which appear after a word.

The final type of morphology is ***compounding***, which allows us to combine words, possibly using a hyphen. An example in English is the word “greenhouse” (an indoor garden). Compounding often results in words for which the compound carries a meaning different from the combination of the individual words. (A greenhouse is not a green house.) We have few examples in English, but this is common in other languages. German is famous for this. Swedish has words like “Tisdagsklubben” (the Tuesday Club). Turkish has many hefty examples such as uygarlas¸tıramadıklarımızdanmıs¸sınızcasına.

Morphotactics

To get from a surface form (eg. “cats”) to a lexical form (eg. “cat +N + Pl), we need a lexicon, spelling rules and morphotactics (an ordering of morphs).

We want to be able to write a function which can take tokens (in English) and return their root forms. Generally, we form English morphs by appending *prefixes* (im+ potable —> impotable) and *suffixes* (break + able —> breakable). It’s relatively easy to create or deconstruct morphs with this assumption.

There are three other types of morphs, all lightly used or absent in English. Many are used in other languages:

- *Circumfixes* surround a word (eg. a+__+ing —> a-sailing?).
- *Infixes* get inserted into a word (eg. unbelievable + frakking —> un-frakking-believable). This is common in Tagalog, where “in” is used to indicate past tense… so bili (to buy) + in (past tense marker) —> binili (bought)
- Templative forms are perhaps the least common — perhaps absent — in English, but are common in Semitic languages (Arabic, Hebrew). For example: דוד may be the root for “love.”

Finite State Automata

A Finite State Automaton (FSA) can recognise generate plurals given nouns. An FSA is a computational tool which we can use to recognise language phenomena. We (literally) think of it as a tape of input and a machine processing that table An FSA is defined by (Σ, S, s_0, δ, F), where:

- Σ = The input alphabet. In our case, it will be the letters a-z and A-Z, numbers 0-9, punctuation, etc.
- S = A set of states.
- s_0 = The initial state of the automaton.
- δ = A set of transitions from one state to another.
- F = One or more set of *accepting* states, in which the FSA has successfully recognised a language phenomenon at the end of the input.

Figure 2 (from Wikipedia) shows an FSA for recognition of the word “nice.” Rather than representing an FSA as the above quintiple, we can represent it is a directed graph as shown in the figure. This has the advantage of being easily consumable.

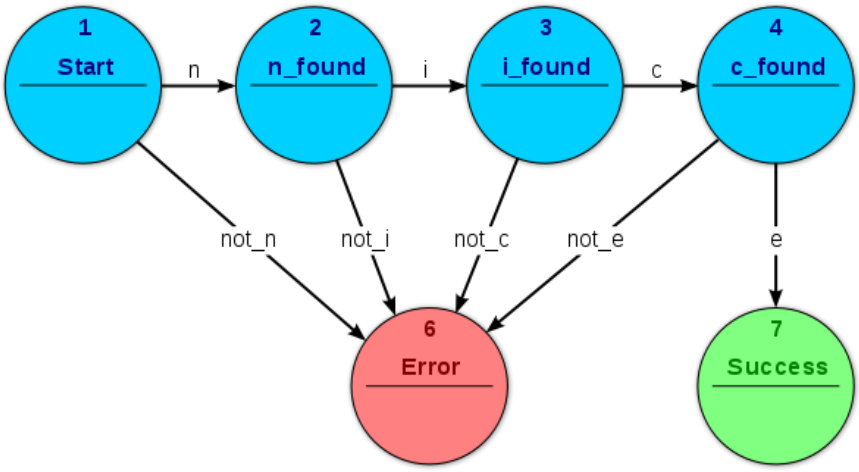

Figure 2: Finite State Automaton for the word “nice”. By en:User:Thowa, redrawn by User:Stannered - en:Image:Fsm parsing word nice.jpg, Public Domain, https://commons.wikimedia.org/w/index.php?curid=3400936

Let’s create an FSA for English country names as an exercise. (This is a totally contrived exercise but it’s useful for the things we’ll build.) Let’s restrict our vocabulary of nouns to the following: Dominica, Dominican Republic, Fiji, Finland, France, Jamaica, Japan, Jordan, Oman, Qatar.

We can write this FSA by examining our input, example by example and letter by letter. As the example of “Dominica” vs. “Dominican Republic” shows, it can be useful to sort the input and consider smaller cases before larger ones.

Onto a less contrived exercise: assuming we have a lexicon of English verbs and an indication of whether the verb in question is a “regular” verb or an irregular verb (in which case, we also have the irregular forms). Table 1 contains an example of the data we may assume in the lexicon. The only regular verb is “impeach,” and there are several examples of regular verbs and their forms, including:

- The past tense (for “*go*” — eg. “My niece ***went*** to the store alone for the first time today.”)
- The past participle (for “*sing*” — eg. “I ***sang*** to the choir”), sometimes defined as a *verbal* because it acts as an adjective or a noun
- The progressive (for “*impeach*” — eg. “I doubt ***impeaching*** Trump will solve the country’s problems”), though this form sometimes creates noun forms called “gerunds”
- The 3rd person singular (for “*cut*” — eg. “The knife ***cuts*** poorly because it is so dull.”)

Verb stem	Regular?	Past	Past part	Progressive (continuous)	3rd person singular
impeach	yes				
cut	no	cut		cutting	cuts
go	no	went	gone	going	goes
speak	no	spoke	spoken	speaking	speaks
sing	no	sang	sung	singing	singing

Table 1: Examples of regular verbs and irregular verbs with morphology

Let’s create an FSA to recognise verbs given a lexicon such as is shown in Table 1.

Finite-State Transducers

A finite-state transducer (FST) is an FSA with two tapes: one for input and one for output. We use it to map from one set of strings to another — i.e. from surface form (“cats”) to a lexical form (“cat +N +Pl)… or the other way around. FSTs share part of their formal definition with FSAs, but because of the additional tape, additional definitions are required. Specifically, an FST is defined by (Q, Σ, Γ, S, I, δ, F), where:

- Q = The set of states.
- Σ = the input alphabet. In our case, it will be the letters a-z and A-Z, numbers 0-9, punctuation, etc.

- Γ = The output alphabet. In this case, it may consist of more than just the input alphabet; it may include items from our lexical forms like “+N” (as a single unit)
- *S* = A set of states.
- *I* = The set of possible initial states.
- δ = A set of transitions from one state to another.
- *F* = One or more set of *accepting* states, in which the FSA has successfully recognised a language phenomenon at the end of the input.

One advantage of the FST is that it accepts (or rejects) some input while providing some analysis about what it has accepted or rejected. Let’s set an overall goal of accepting or rejecting English nouns and providing an analysis of their forms, specifically: whether the input is a noun and whether it is plural. We will call this the *Lexical* form, and we’ll indicate — for example — “cats” (surface form) as “cat+N+Pl” (lexical form).

For this, we will define two special symbols: #, which indicates a word boundary, and ^, which indicates a morpheme boundary. One useful FSA/FST tool is a null symbol, often marked with an ε. This allows us to move from one state to another without receiving any input for doing so, so will be useful when processing the # and ^ symbols.

Practically speaking, we will almost never build an FST by hand. Typically, we write rules in “rewrite notation,” which may look like:

- a —> b / c__d (rewrite a as b when observed between c and d.)
- ε —> e / {x, s, z} ^__s # (Insert e when a word ends in x, s or z before inserting s at the end of the word.)

Specifically for dictionaries, we start from a closed vocabulary, compile the elements into a transducer and add exceptions, such as the ones we have been discussing. This allows us to make several optimisations to the lexical entries and to maintain the dictionary as a separate object, very much in line with our objectives of making changes as our language changes.

Stemming (NLTK)

In building an Information Retrieval system, we may find it useful to remove the surface form of morphemes and reduce the search terms to verb roots, singular nouns, etc. For example, if an IR system were presented with the search term “hacking cars,” it may want to search instead for “hack car.” Of course, this applies equally during construction of the dictionary in a boolean retrieval model.

Ideally, we want to use our knowledge of morphotactics to construct such a system. In practice, we can get to an approximation of this using a Stemming algorithm. In stemming, we remove the derivational and inflectional prefixes and suffixes of input words until we get to a root word.

MSAN
Summer 2017
Natural Language Processing

The idea for stemming is from a 1968 paper, "Development of a Stemming Algorithm" by Julie Beth Lovins.

We make several assumptions during stemming, Perhaps the most important two assumptions are as follows:

1) We use stemming without relying on a lexicon or lookup table because of the way in which a language may change.
2) We use stemming with the expectation that the derivational and inflectional rules are constant.

In other words, we expect our root words to change as new types are added to our lexicon. But no matter how many new words we add to our lexicon, we expect that constructing the present continuous form always involves adding "ing" or that making a singular noun plural always involves adding a "s" (or "es") to the singular form.

In English, as an many languages, it is relatively easy to construct the valid derivational and inflectional forms given a base form, a part-of-speech. The opposite problem is harder. As an example of our goals for stemming, the word "internationalisation" can be seen as built from one prefix ("inter") and several suffixes ("al", "ise", "ation") — inter+ nation +al +ise +ation — getting us to the root word of "nation." Unfortunately, this is often difficult to do for a few reasons. Firstly, English contains a number of irregular words. For example, it is difficult to construct the root of "brought" in absence of a lookup table. Secondly, a token which appears to be an inflection may not be. "Ceiling" is not the present continuous form of "ceil."

Our goal with stemming is to strip away non-root parts of a token until we arrive at a root word which is suitable for indexing a document or for performing a search through the constructed index. However, the purpose of a stemmer is not this; it is to reduce (or *rewrite*) all variant forms of a word consistently. This means that the results we see from a stemmer may not be proper (English) words at all. One example is the word "abatements." Applying what we know about English morphotactics, we could see this as "abate +ment +s"; however, a stemmer may rewrite this as abat.

There are several stemming algorithms. The most popular may be the collective variants of the Porter Stemmer, which only removes suffixes. (Perhaps Martin Porter did not want to worry about words like "anticipate" being reduced to … "cip"?) The algorithm is detailed at https://tartarus.org/martin/PorterStemmer/def.txt. Generally, it only operates on strings of length greater than 2 characters — so words like "as" and "is" do not get changed to "a" and "i." The Porter stemmer is defined by conditions and 5 categories of rewrite rules. An example of a rewrite rule is the following:

IES —> I

… by which a word like "ponies" would be rewritten to "poni." Conditions, commonly placed in parentheses before a rewrite rule, govern the cases when the rule may be applied. One such example is:

(*v*) ED —>

… meaning: “if there is a vowel in a word which ends in “ed,” remove the “ed” from the word. In this example, “plastered” may be rewritten as “plaster” but “bled” may not be rewritten. Rules are prioritised and applied successively. The ***measure*** of a rewritten word in the Porter stemmer is the number of consonant and vowel clusters. Many of the rules require that the target of the rewrite have a measure of a certain length.

In Python, the Natural Language Toolkit has an implementation of Porter Stemmer. We believe it to be one of the more correct implementations. It may be used as follows:

```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()
stemmed_word = stemmer.stem('internationalisation')
```

… after which “stemmed_word” should contain “internation.” Let’s test the NLTK version using data from the Porter Stemmer site.

1) Download the following files from the Porter Stemmer site:
    a) Sample vocabulary: https://tartarus.org/martin/PorterStemmer/voc.txt
    b) Expected output: https://tartarus.org/martin/PorterStemmer/output.txt
2) Create a script to read the files above, generate stemmed words from the input using the Porter Stemmer, and check those words against the expected output.
3) Output any differences between the expected output and the actual output.
4) Perform an analysis on the output, specifically:
    a) What is the accuracy in this set?
    b) What items does it fail on?
    c) Is there a pattern to these failures?

A partial answer is provided in check_porter.py.

A similar stemming algorithm, the Snowball Stemmer, works in the same way, but there are slightly different rules, conditions and orderings, all of which give it higher performance. The Snowball Stemmer also has an implementation in NLTK, with the notable difference that it is instantiated with the (lowercase) name of the language being processed. The following languages are supported:

> danish dutch english finnish french german hungarian italian norwegian porter portuguese romanian russian spanish swedish

… with the “porter” language being the Porter Stemmer, which operates only on English. Other modules in NLTK include: ***isri*** (for Arabic), ***lancaster*** (another English-language stemmer, with richer rules so that — for example — “provision” gets rewritten as “provide”), ***regexp*** (regular expression-based English stemmer #recommend_avoid), ***rlsp*** (for Portuguese). Additional stemmers (the original Lovins stemmer, along with algorithms for Paice/Husk and others) have no implementation in NLTK but exist in other toolkits.

Aside: NLTK is a large toolkit, and we will see it throughout this course. It contains a number of corpora, modules for tagging words (verb, noun plural, etc.), generating syntax trees and

performing semantic processing. NLTK has the reputation of being somewhat old, relatively slow and sometimes inaccurate, as we see above. That said, it is a great starting point for the work we do because it has a rich set of modules for NLP efforts. For systems you write, it may be useful to start with NLTK as a proof of concept.

Generally, stemming allows us to retrieve get documents from vaguely-related search terms. However, one of the criticisms of stemmers is that stemming fails to generate a form for irregular verbs. So while “played” is rewritten as “play” — what we want and expect, words like “ran” and “begun” to not get rewritten in their infinitive form. A word like “saw” is particularly interesting (and difficult) because it may be the past tense of “see” or may be the noun, representing the tool used by carpenters. This negatively affects the performance of our IR system. It is often the case that the benefits outweigh the drawbacks. Instead of stemming, it may be preferable to apply ***lemmatization***[2] , which gives the base form of a word rather than an approximation.

NLTK contains one lemmatizer, WordNetLemmatizer. It is used similarly to the stemmers, but the results are often different because it is backed by the WordNet ontology (a kind of lexicon). Let’s explore a few examples:

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemma = lemmatizer('geese')
```

… which returns the form (“goose”) we care to see. By default, WordNetLemmatizer expects the input to be a noun. The caller may sometimes offer clues by providing a part-of-speech tag, as in the following:

```
lemma = lemmatizer('ran', pos='v')
```

… in which we specify that the word “ran” is a verb and expect the output to be “run.”

For text in a context, NLTK also contains a word tokenizer and part-of-speech tagger. We can add these to our Information Retrieval system to improve its accuracy. The word tokenizer handles punctuation (mostly) correctly, and the part-of-speech tagger can be used to get the correct lemma from WordNet. Here is an example of how to use them:

```
corpus = 'My daughter wants to drive ...'
from nltk import word_tokenize, pos_tag
text = word_tokenize(corpus) # text is a list of tokens
text_and_tags = pos_tag(text)
```

After this, the variable “text_and_tags” will contain tuples of input words (which may be punctuation) and their tags.Putting together these modules from NLTK: tokenisation, stemming or lemmatization (possible with part-of-speech tagging), along with the addition of stop words and conversion of text to standard case (often lowercase), we can create a basic and powerful Information Retrieval system.

[2] NLTK uses the American spelling. I use that here for consistency.

Advanced Information Retrieval

We deliberately skipped or simplified a number of items in our discussion of Information Retrieval systems. Many of these involve multiple words or the way in which we may infer one word from another. Let's briefly discuss these issues and some of their solutions.

Multi-word searches

One item we did not address completely is an algorithm for finding the intersection of documents returned by each word in a search term. For example, if we search for "Caesar Brutus," we may get two sets or lists of documents. Because we assume that the user performing this query has done so in conjunctive form, our system must return documents containing both Caesar and Brutus. This algorithm is relatively easy, but we want to ensure that it is performed as quickly as possible.

We expect document IDs to be returned from each of the words in a search term. For simplicity, let us assume that each document ID is a number, and that the document IDs are ordered (smallest-to-largest) upon return. Figure 3 shows an example.

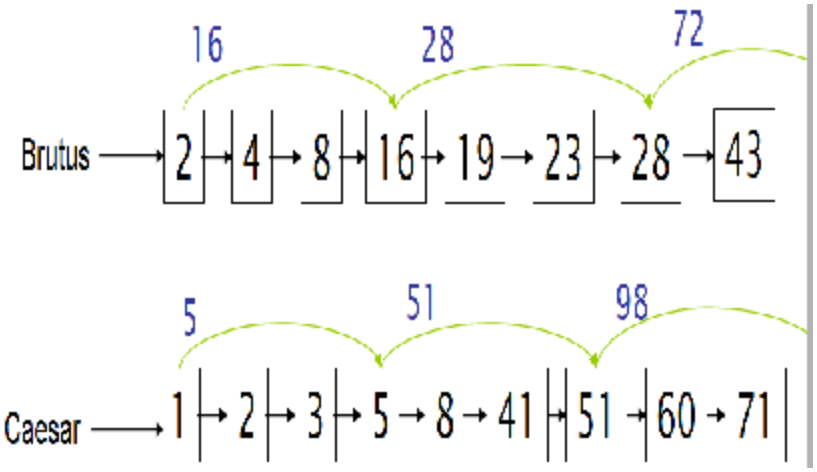

Figure 3: Example document ID with skip pointers, taken from Introduction to Information Retrieval

A naive algorithm for finding the intersection of two lists is similar to that for merging two lists via MergeSort. The obvious difference being that the goal is to find where document IDs from the two lists overlap. The algorithm for finding this intersection can be something like the following:

```
intersection = []
for doc_id1 in list1:
    for doc_id2 in list2:
        if doc_id1 == doc_id2:
```

```
intersection = doc_id1
```

If *n* and *m* are the sizes of the lists for words in the search term, what is the running time of this algorithm? Is this a good running time? Can we do any better given this pair of data structures — i.e. two lists?

This running time quickly becomes unbearably slow as we add words to a search term. There are many ways to speed this algorithm up. One method is to use skip pointers. Skip pointers allow the intersect algorithm to “fast forward” in either of the lists being intersected. Any record contains not only a document ID but also a link to a far-away document ID. Our algorithm may take advantage of these pointers to skip over document IDs. A good guideline is to include sqrt(document IDs) pointers for indexed item and to space the pointers evenly with respect to the list. Using skip pointers this way, we may see a running time of O(log(n) + m).

Multi-word Expressions

Often, we use Information Retrieval systems with multiple words in our search term. For example, we may want to find documents related to “Michael Jackson” and we may specifically care to exclude documents not about the singer — for example, one on “Saint Michael, who never travelled to Jackson, Mississippi.”

One solution is to create an index which includes these combinations of words, called biword or phrase indices. Multi-word expressions (MWEs), also known as collocations, are interesting for a number of reasons. Some MWEs are “frozen.” Although some variation is allowed, the expression “kick the bucket” has a fixed word ordering. Other MWEs have fewer constraints on their structure. This is especially true for light verb constructions (LVCs) in English, such as “take a walk” or “have some coffee.”

While some modules (eg. NLTK, gensim) have collocation-identifying modules, detecting good collocations is an area exploration. This may be a good project.

Proximity Searches

Sometimes, we care about MWEs which are not frozen, or the presence of a search term only when in proximity of another. For example, the search term is “korean politics,” the user may really want to retrieve documents in which “polit” (the stemmed version of politics) is within 2 - 3 tokens of “korea” (the stemmed version of “korean”). This may be supported by an Information Retrieval system, and it exists in some specialty systems. In order for an Information Retrieval system to support this functionality, the dictionary must include a numeric index for each of the tokens in the documents. This is known as a positional index. Supported search terms may

follow the pattern “korea near politics” and “korea /3 politics” (“korea within three tokens of politics”).

Semantic Equivalences

One additional item we did not address in constructing indices or search terms is semantic equivalence. In our example corpus, one documents mentions a “daughter.” In this case, if a user searches for a closely related term, such as “children,” we expect our information retrieval system to return documents containing the words “child” in addition to ones containing words such as “daughter” or “son.”

One solution is to use an ontology, which creates semantic equivalences between words like the above. WordNet is one popular tool which allows us to do this. The WordNet entry for “child” is at the URL http://wordnetweb.princeton.edu/perl/webwn?s=child and using the WordNet API, we are able to explore synsets — relationships among words and find many of these semantic equivalences.

WordNet is available in approximately 48 languages.