Word Level Analysis

2

OBJECTIVES

After reading this chapter, the student will be able to understand:

- Morphology analysis
- Stemming and Lemmatization
- Regular expression
- Finite Automata
- Finite-State Morphological Parsing
- Combining FST Lexicon and Rules
- Lexicon free FST Porter stemmer
- N –Grams, N-gram language model, N-gram for spelling correction.

1. Morphology analysis

What are words?

Words are the fundamental building block of language. Every human language, spoken, signed, or written, is composed of words. Every area of speech and language processing, from speech recognition to machine translation to information retrieval on the web, requires extensive knowledge about words. Psycholinguistic models of human language processing and models from generative linguistic are also heavily based on lexical knowledge.

Words are Orthographic tokens separated by white space. In some languages the distinction between words and sentences is less clear.

Chinese, Japanese: no white space between words

nowhitespace → no white space/no whites pace/now hit esp ace

Turkish: words could represent a complete "sentence"

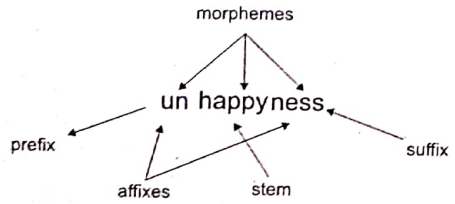
Eg: uygarlastiramadiklarimizdanmissinizcasina

"(behaving) as if you are among those whom we could not civilize"

Morphology deals with the syntax of complex words and parts of words, also called morphemes, as well as with the semantics of their lexical meanings. Understanding how words are formed and what semantic properties they convey through their forms enables human beings to easily recognize individual words and their meanings in discourse. Morphology also looks at parts of speech, intent and stress, and the ways context can change a word's pronunciation and meaning.

Morphology is the study of the structure and formation of words. Its most important unit is the morpheme, which is defined as the "minimal unit of meaning"

Consider a word like: "unhappiness". This has three parts:



Morphemes : un happy ness

Prefix : un

Suffix : ness

Affixes : happy

Stem : happy

There are three morphemes, each carrying a certain amount of meaning. UN means "not", while ness means "being in a state or condition". Happy is a free morpheme because it can appear on its own (as a "word" in its own right). Bound morphemes have to be attached to a free morpheme, and so cannot be words in their own right. Thus, you can't have sentences in English such as "Jason feels very un ness today".

Bound Morphemes: These are lexical items incorporated into a word as a dependent part. They cannot stand alone, but must be connected to another morphemes. Bound morphemes operate in the connection processes by means of derivation, inflection, and compounding. Free morphemes, on the other hand, are autonomous, can occur on their own and are thus also words at the same time. Technically, bound morphemes and free morphemes are said to differ in terms of their distribution or freedom of occurrence. As a rule, lexemes consist of at least one free morpheme

Morphology handles the formation of words by using morphemes base form (stem,lemma), e.g., believe affixes (suffixes, prefixes, infixes), e.g., un-, -able, -ly

Morphological parsing is the task of recognizing the morphemes inside a word e.g., hands, foxes, children and its important for many tasks like machine translation, information retrieval, etc. and useful in parsing, text simplification, etc

Survey of English Morphology

Morphology is the study of the way words are built up from smaller meaning- bearing units, morphemes. A morpheme is often defined as the minimal meaning-bearing unit in a language. So for example the word fox consists of a single morpheme (the morpheme fox) while the word cats consists of two: the morpheme cat and the morpheme -s. As this example suggests, it is often useful to distinguish two broad classes of morphemes: stems and affixes. The exact details of the distinction vary from language to language, but intuitively, the stem is the 'main' morpheme of the word, supplying the main meaning, while the affixes add 'additional' meanings of various kinds. Affixes are further divided into prefixes, suffixes, infixes, and circum- fixes. Prefixes precede the stem, suffixes follow the stem, circumfixes do both, and infixes are inserted inside the stem. For example, the word eats is composed of a stem eat and the suffix -s. The word unbuckle is composed of a stem buckle and the prefix un-. English doesn't have any good examples of circumfixes,

but many other languages do. In German, for example, the past participle of some verbs formed by adding ge- to the beginning of the stem and -t to the end; so the past participle of the verb sagen (to say) is gesagt (said). Infixes, in which a morpheme is inserted in the middle of a word, occur very commonly for example in the Philipine language Tagalog. For example, the affix um, which marks the agent of an action, is infixed to the Tagalog stem hingi 'borrow' to produce humingi.

Prefixes and suffixes are often called concatenative morphology since a word is composed of a number of morphemes concatenated together. A number of languages have extensive non-concatenative morphology, in which morphemes are combined in more complex ways. The Tagalog in- fixation example above is one example of non-concatenative morphology, since two morphemes (hingi and um) are intermingled. Another kind of non-concatenative morphology is called templatic morphology or root-and-pattern morphology. This is very common in Arabic, Hebrew, and other Semitic languages. In Hebrew, for example, a verb is constructed us- ing two components: a root, consisting usually of three consonants (CCC) and carrying the basic meaning, and a template, which gives the ordering of consonants and vowels and specifies more semantic information about the resulting verb, such as the semantic voice (e.g. active, passive, middle). For example the Hebrew tri-consonantal root lmd, meaning 'learn' or 'study', can be combined with the active voice CaCaC template to produce the word lamad, 'he studied', or the intensive CiCeC template to produce the word limed, 'he taught', or the intensive passive template CuCaC to produce the word lumad, 'he was taught'.

A word can have more than one affix. For example, the word rewrites have the prefix re-, the stem write, and the suffix -s. The word unbelievably has a stem (believe) plus three affixes (un-, -able, and -ly). While English doesn't tend to stack more than 4 or 5 affixes, languages like Turkish can have words with 9 or 10 affixes, as we saw above. Languages that tend to string affixes together like Turkish does are called agglutinative languages.

There are two broad (and partially overlapping) classes of ways to form words from morphemes: inflection and derivation. Inflection is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the same class as the original stem, and usually filling some syntac- tic function like agreement. For example, English has the inflectional mor- pheme -s for marking the plural on nouns, and the inflectional morphemes -ed for marking the past tense on verbs. Derivation is the combination of a

word stem with a grammatical morpheme, usually resulting in a word of a different class, often with a meaning hard to predict exactly. For example, the verb computerizes can take the derivational suffix -ation to produce the noun computerization.

Inflectional morphology & Derivational morphology

Morphemes are defined as smallest meaning-bearing units. Morphemes can be classified in various ways. One common classification is to separate those morphemes that mark the grammatical forms of words (-s, -ed, -ing and others) from those that form new lexemes conveying new meanings, e.g. un- and -ment. The former morphemes are inflectional morphemes and form a key part of grammar, the latter are derivational morphemes and play a role in word-formation, as we have seen. The following criteria help you to distinguish the two types:

- Effect: Inflectional morphemes encode grammatical categories and relations, thus marking word-forms, while derivational morphemes create new lexemes.
- Position: Derivational morphemes are closer to the stem than inflectional morphemes, cf. amendments (amend[stem] – ment[derivational] – s[inflectional]) and legalized (legal[stem] – ize[derivational] – ed[inflectional]).
- Productivity: Inflectional morphemes are highly productive, which means that they can be attached to the vast majority of the members of a given class (say, verbs, nouns or adjectives), whereas derivational morphemes tend to be more restricted with regard to their scope of application. For example, the past morpheme can in principle be attached to all verbs; suffixation by means of the adjective-forming derivational morphemes -able, however, is largely restricted to dynamic transitive verbs, which excludes formations such as *bleedable or *lieable.
- Class properties: Inflectional morphemes make up a closed and fairly stable class of items which can be listed exhaustively, while derivational morphemes tend to be much more numerable and more open to changes in their inventory.

Both inflectional and derivational morphemes must be attached to other morphemes; they cannot occur by themselves, in isolation, and are therefore known as bound morphemes

Inflected words are variations of already existing lexemes that were only changed in their grammatical shape. Therefore many of the Inflectional Morphemes are not listed in the dictionary. If you know the word surprise and look it up you will also find in the same entry the word surprise -s which simply expresses the plural.

Derivational Morphology on the other hand uses affixes to create new words out of already existing lexemes. Typical affixes are -ness, -ish, -ship and so on. These affixes do not change the grammatical form of a word such as inflectional affixes do, but instead often create a new meaning of the base or change the word class of the base. An Example would be the word light. The plural form light-s would be considered Inflectional Morphology, but if we consider de-light the prefix -de has changed the meaning of the word completely. We now do not think of light in the form of sunshine or lamps anymore but instead about a feeling. Also if we consider en-light the suffix -en has changed the word class of light from noun to verb.

INFLECTIONAL MORPHOLOGY

Inflection is a morphological process that adapts existing words so that they function effectively in sentences without changing the category of the base morphemes.

Inflection can be seen as the "realization of morphosyntactic features through morphological means" . But what exactly does that mean? It means that inflectional morphemes of a word do not have to be listed in a dictionary since we can guess their meaning from the root word. We know when we see the word what it connects to and most times can even guess the difference to its original. For example let us consider help-s, help-ed and help-er. According to what I have said about words listed in the dictionary, all of these variants might be inflectional morphemes, but then on the other hand does help-s really need an extra listing or can we guess from the root help and the suffix -s what it means? Does our natural feeling and instinct for language not tell us, that the suffix –s indicates the third person singular and that help-s therefore only is a variant from help (considering help as a verb and not a noun here)? Yes it does. As native speaker one instantly knows that –s, as also the past form indicator –ed only show a grammatical variant of the root lexeme help.

So why is help-er or even help-less-ness different? The answer is actually very simple. The suffixes in these last two words change the word class and therefore form a new lexeme. Help-er can now be the new root for additional suffixes such help-er-s which would then be an inflectional morpheme again, the root here being the smallest free morpheme after you remove all affixes.

After establishing this we still have a problem if we consider the word help as a noun and as a verb. How do we distinguish these two? The answer is context and the phenomenon is called a zero morphemes. Only threw context can we say if help is a verb or a noun.

To illustrate this consider the following two sentences:

1. I help my grandmother in her garden.
2. He is my grandmother`s help.

Here our general knowledge of words and their meaning shows us that in 1. help is used as a verb and expresses us working with our grandmother in order to support her. In 2. help is a noun and stands for the person that regularly supports my grandmother. This variation of a word without actually changing its form is called zero morphemes and cannot only distinguish verb and noun (which makes it a derivational morphemes) but also singular and plural, which makes it an inflectional morpheme. I will talk about this later in 2.2.: Inflection in nouns, though.

"We may define inflectional morphology as the branch of morphology that deals with paradigms. It is therefore concerned with two things: on the one hand, with the semantic oppositions among categories; and on the other, with the formal means, including inflections, that distinguish them." (Matthews, 1991).

Inflectional morphology is that it changes the word form, it determines the grammar and it does not form a new lexeme but rather a variant of a lexeme that does not need its own entry in the dictionary.

word stem + grammatical morphemes

cat + s only for nouns, verbs, and some adjectives

Nouns

plural:

regular: +s, +es irregular: mouse - mice; ox - oxen

many spelling rules: e.g. -y -> -ies like: butterfly - butterflies

possessive: +'s, +'

Verbs

main verbs (sleep, eat, walk)

modal verbs (can, will, should)

primary verbs (be, have, do)

VERB INFLECTIONAL SUFFIXES

1. The suffix –s functions in the Present Simple as the third person marking of the verb : to work – he work-s
2. The suffix –ed functions in the past simple as the past tense marker in regular verbs: to love – lov-ed

3. The suffixes –ed (regular verbs) and –en (for some regular verbs) function in the marking of the past partciple and, in general, in the marking of the perfect aspect:

   To study studied studied / To eat ate eaten

4. The suffix –ing functions in the marking of the present participle, the gerund and in the marking of the continuous aspect: To eat – eating / To study - studying

NOUN INFLECTIONAL SUFFIXES

1. The suffix –s functions in the marking of the plural of nouns: dog – dogs
2. The suffix –s functions as a possessive marker (saxon genitive): Laura – Laura's book.

ADJECTIVE INFLECTIONAL SUFFIXES

The suffix –er functions as comparative marker: quick – quicker

The suffix –est functions as superlative marker: quick - quickest

DERIVATIONAL MORPHOLOGY

Derivation is concerned with the way morphemes are connected to existing lexical forms as affixes. Derivational morphology is a type of word formation that creates new lexemes, either by changing syntactic category or by adding substantial new meaning (or both) to a free or bound base. Derivation may be contrasted with inflection on the one hand or with compounding on the other. The distinctions between derivation and inflection and between derivation and compounding, however, are not always clear-cut. New words may be derived by a variety of formal means including affixation, reduplication, internal modification of various sorts, subtraction, and conversion. Affixation is best attested cross-linguistically, especially prefixation and suffixation. Reduplication is also widely found, with various internal changes like ablaut and root and pattern derivation less common. Derived words may fit into a number of semantic categories. For nouns, event and result, personal and participant, collective and abstract noun are frequent. For verbs, causative and applicative categories are well-attested, as are relational and qualitative derivations for adjectives. Languages frequently also have ways of deriving negatives, relational words, and evaluative. Most languages have derivation of some sort, although there are languages that rely more heavily on compounding than on derivation to build their lexical stock. A number of topics have dominated the theoretical literature on derivation, including productivity (the extent to which new words can be created with a given affix or morphological process), the principles that determine the ordering of affixes, and the place of derivational morphology with respect to other components of the grammar. The

study of derivation has also been important in a number of psycholinguistic debates concerning the perception and production of language.

Derivational morphology is defined as morphology that creates new lexemes, either by changing the syntactic category (part of speech) of a base or by adding substantial, non-grammatical meaning or both. On the one hand, derivation may be distinguished from inflectional morphology, which typically does not change category but rather modifies lexemes to fit into various syntactic contexts; inflection typically expresses distinctions like number, case, tense, aspect, person, among others. On the other hand, derivation may be distinguished from compounding, which also creates new lexemes, but by combining two or more bases rather than by affixation, reduplication, subtraction, or internal modification of various sorts. Although the distinctions are generally useful, in practice applying them is not always easy.

We can distinguish affixes in two principal types:

1. Prefixes - attached at the beginning of a lexical item or base-morpheme – ex: un-, pre-, post-, dis, im-, etc.
2. Suffixes – attached at the end of a lexical item ex: -age, -ing, -ful, -able, -ness, -hood, -ly, etc.

EXAMPLES OF MORPHOLOGICAL DERIVATION

a.	b.
Lexical item (free morpheme): like (verb)	Lexical item: like
+ prefix (bound morpheme) dis-	+ suffix –able = likeable
= dislike (verb)	+ prefix un- =unlikeable
	+ suffix –ness = unlikeableness

Derivational affixes can cause semantic change:

1. Prefix pre- means before; post- means after; un- means not, re- means again.
2. Prefix = fixed before; Unhappy = not happy = sad; Retell = tell again.
3. Prefix de- added to a verb conveys a sense of subtraction; dis- and un- have a sense of negativity.
4. To decompose; to defame; to uncover; to discover.

Derivation Versus Inflection

The distinction between derivation and inflection is a functional one rather than a formal one, as Booij (2000, p. 360) has pointed out. Either derivation or inflection may be affected by formal means like affixation, reduplication, internal modification of bases, and other morphological processes. But derivation serves to create new lexemes while inflection prototypically serves to modify lexemes to fit different grammatical contexts. In the clearest cases, derivation changes category, for example taking a verb like *employ* and making it a noun (*employment, employer, employee*) or an adjective (*employable*), or taking a noun like *union* and making it a verb (*unionize*) or an adjective (*unionish, unionesque*). Derivation need not change category, however. For example, the creation of abstract nouns from concrete ones in English (*king ~ kingdom; child ~ childhood*) is a matter of derivation, as is the creation of names for trees (*poirier* 'pear tree') from the corresponding fruit (*poire* 'pear') in French, even though neither process changes category. Derivational prefixation in English tends not to change category, but it does add substantial new meaning, for example creating negatives (*unhappy, inconsequential*), various quantitative or relational forms (*tricycle, preschool, submarine*) or evaluative (*megastore, miniskirt*). Inflection typically adds grammatical information about number (singular, dual, plural), person (first, second, third), tense (past, future), aspect (perfective, imperfective, habitual), and case (nominative, accusative), among other grammatical categories that languages might mark.

Nevertheless, there are instances that are difficult to categorize, or that seem to fall somewhere between derivation and inflection. Many of the difficult cases hinge on determining the necessary and sufficient criteria in defining derivation. Certainly, category change is not necessary, as there are many obvious cases of derivation that do not involve category change. But further, Haspelmath (1996) has argued that there are cases of inflection (participles of various sorts, for example) that are category-changing, so that this criterion cannot even be said to be sufficient. Productivity has also been used as a criterion—inflection typically being completely productive, derivation being only sporadically productive—but some derivation is just as productive as inflection, for example, forming nouns from adjectives with *-ness* in English, and some inflection displays lexical gaps (Booij, 2000, p. 363 says, e.g., that the Dutch infinitive *bloemlezen* 'to make an anthology' does not correspond to any finite forms), so productivity is neither necessary nor sufficient for distinguishing inflection and derivation. Anderson (1982) makes the criterion of relevance to syntax the most important one for distinguishing inflection from derivation; inflection is invariably relevant to syntax, derivation not. But Booij (1996) has argued that even this criterion is problematic unless we are clear what we mean by relevance to syntax. Case inflections, for example, mark grammatical context, and are therefore clearly inflectional. Number-marking on verbs is arguably inflectional when it is triggered by the number of subject or object, but number on nouns or tense and aspect on verbs is a matter of semantic choice, independent of grammatical configuration. Booij therefore distinguishes what he calls contextual inflection, inflection triggered by distinctions elsewhere in a sentence, from inherent inflection, inflection that does not depend on the syntactic context, the latter being closer to derivation than the former. Some theorists (Bybee, 1985; Dressler, 1989) postulate a continuum from derivation to inflection, with no clear dividing line between them. Another position is that of Scalise (1984), who has argued that evaluative morphology is neither inflectional nor derivational but rather constitutes a third category of morphology.

2. Stemming and Lemmatization

In natural language processing, there may come a time when you want your program to recognize that the words "ask" and "asked" are just different tenses of the same verb. This is the idea of reducing different forms of a word to a core root. Words that are derived from one another can be mapped to a central word or symbol, especially if they have the same core meaning.

Maybe this is in an information retrieval setting and you want to boost your algorithm's recall. Or perhaps you are trying to analyze word usage in a corpus and wish to condense related words so that you don't have as much variability. Either way, this technique of text normalization may be useful to you.

This is where something like stemming or lemmatization comes in.

For grammatical reasons, documents are going to use different forms of a word, such as *organize*, *organizes*, and *organizing*. Additionally, there are families of derivationally related words with similar meanings, such as *democracy*, *democratic*, and *democratization*. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.

The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. For instance:

am, are, is ⇒ be

car, cars, car's, cars' ⇒ car

The result of this mapping of text will be something like:

the boy's cars are different colors ⇒

the boy car be differ color

However, the two words differ in their flavor.

Stemming usually refers to a crude heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, and often includes the removal of derivational affixes. Stemming is definitely the simpler of the two approaches. With stemming, words are reduced to their word stems. A word stem need not be the same root as a dictionary-based morphological root, it just is equal to or smaller form of the word.

Stemming algorithms are typically rule-based. You can view them as heuristic process that sort-of lops off the ends of words. A word is looked at and run through a series of conditionals that determine how to cut it down.

For example, we may have a suffix rule that, based on a list of known suffixes, cuts them off. In the English language, we have suffixes like "-ed" and "-ing" which may be useful to cut off in order to map the words "cook," "cooking," and "cooked" all to the same stem of "cook."

Over stemming comes from when too much of a word is cut off. This can result in nonsensical stems, where all the meaning of the word is lost or muddled. Or it can result in words being resolved to the same stems, even though they probably should not be.

Take the four words university, universal, universities, and the universe. A stemming algorithm that resolves these four words to the stem "univers" has over stemmed. While it might be nice to have universal and universe stemmed together and university and universities stemmed together, all four do not fit. A better resolution might have the first two resolve to "univers" and the latter two resolve to "universi." But enforcing rules that make that so might result in more issues arising.

Under stemming is the opposite issue. It comes from when we have several words that actually are forms of one another. It would be nice for them to all resolve to the same stem, but unfortunately, they do not. This can be seen if we have a stemming algorithm that stems the words data and datum to "dat" and "datu."

Stemming Algorithms Examples

Porter stemmer: This stemming algorithm is an older one. It's from the 1980s and its main concern is removing the common endings to words so that they can be resolved to a common form. It's not too complex and development on it is frozen. Typically, it's a nice starting basic stemmer, but it's not really advised to use it for any production/ complex application. Instead, it has its place in research as a nice, basic stemming algorithm that can guarantee reproducibility. It also is a very gentle stemming algorithm when compared to others.

Snowball stemmer: This algorithm is also known as the Porter2 stemming algorithm. It is almost universally accepted as better than the Porter stemmer, even being acknowledged as such by the individual who created the Porter stemmer. That being said, it is also more aggressive than the Porter stemmer. A lot of the things added to the Snowball stemmer were because of issues noticed with the Porter stemmer. There is about a 5% difference in the way that Snowball stems versus Porter.

Lancaster stemmer: Just for fun, the Lancaster stemming algorithm is another algorithm that you can use. This one is the most aggressive stemming algorithm of the bunch. However, if you use the stemmer in NLTK, you can add your own custom rules to this algorithm very easily. It's a good choice for that. One complaint around this stemming algorithm though is that it sometimes is overly aggressive and can really transform words into strange stems. Just make sure it does what you want it to before you go with this option!

Lemmatization usually refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the *lemma* . lemmatization is a more calculated process. It involves resolving words to their dictionary form. In fact, a lemma of a word is its dictionary or canonical form!

Because lemmatization is more nuanced in this respect, it requires a little more to actually make work. For lemmatization to resolve a word to its lemma, it needs to know its part of speech. That requires extra computational linguistics power such as a part of speech tagger. This allows it to do better resolutions (like resolving is and are to "be").

Another thing to note about lemmatization is that it's often times harder to create a lemmatizer in a new language than it is a stemming algorithm. Because lemmatizers

require a lot more knowledge about the structure of a language, it's a much more intensive process than just trying to set up a heuristic stemming algorithm.

If confronted with the token *saw*, stemming might return just *s*, whereas lemmatization would attempt to return either *see* or *saw* depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma. Linguistic processing for stemming or lemmatization is often done by an additional plug-in component to the indexing process, and a number of such components exist, both commercial and open-source.

The most common algorithm for stemming English, and one that has repeatedly been shown to be empirically very effective, is *Porter's algorithm* (Porter, 1980). The entire algorithm is too long and intricate to present here, but we will indicate its general nature. Porter's algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix. In the first phase, this convention is used with the following rule group:

(F)	Rule			Example		
	SSES	→	SS	caresses	→	caress
	IES	→	I	ponies	→	poni
	SS	→	SS	caress	→	caress
	S	→		cats	→	cat

Many of the later rules use a concept of the *measure* of a word, which loosely checks the number of syllables to see whether a word is long enough that it is reasonable to regard the matching portion of a rule as a suffix rather than as part of the stem of a word. For example, the rule:

(m > 1) EMENT →

would map replacement to replac, but not cement to c.

Rather than using a stemmer, you can use a *lemmatizer* , a tool from Natural Language Processing which does full morphological analysis to accurately identify the lemma for each word. Doing full morphological analysis produces at most very modest benefits for retrieval. It is hard to say more, because either form of normalization tends not to improve English information retrieval performance in aggregate - at least not by very much. While it helps a lot for some queries, it equally hurts performance a lot for others. Stemming

increases recall while harming precision. As an example of what can go wrong, note that the Porter stemmer stems all of the following words:

operate operating operates operation operative operatives operational to oper.

However, since *operate* in its various forms is a common verb, we would expect to lose considerable precision on queries such as the following with Porter stemming:

operational and research

operating and system

operative and dentistry

For a case like this, moving to using a lemmatizer would not completely fix the problem because particular inflectional forms are used in particular collocations: a sentence with the words *operate* and *system* is not a good match for the query operating and system. Getting better value from term normalization depends more on pragmatic issues of word use than on formal issues of linguistic morphology.

The situation is different for languages with much more morphology (such as Spanish, German, Hindi and Finnish).

3. Regular expression

Regular expression (RE), a language for specifying text search strings or search pattern. The regular expression languages used for searching texts in UNIX (vi, Perl, Emacs, grep) and Microsoft. Usually search patterns are used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation. It is a technique developed in theoretical computer science and formal language theory.

Words are almost identical, and many RE features exist in the various Web search engines. Besides this practical use, the regular expression is an important theoretical tool throughout computer science and linguistics. A regular expression is a formula in a special language that is used for specifying simple classes of strings. A string is a sequence of symbols; for the purpose of most text-based search techniques, a string is a sequence of alphanumeric characters (letters, numbers, spaces, tabs, and punctuation). For these purposes a space is just a character like any other, and we represent it with the symbol ␣. Formally, a regular expression is an algebraic notation for characterizing a set of strings. Thus, they can be used to specify search strings as well as to define a language in a formal way.

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. You will usually find the name abbreviated to "regex" or "regexp". Regular expressions (regex or regexp) are extremely useful in extracting information from any text by searching for one or more matches of a specific search pattern (i.e. a specific sequence of ASCII or unicode characters).

```
Natural Language Processing, or [redacted] for short, is broadly defined as the
automatic manipulation of natural language, like speech and text, by
software.Statistical [redacted] aims to do statistical inference for the field of
natural language.
```

```
/\b(\w*NLP\w*)\b/g                                                    2 matches
```

Figure 1: shows matching of the string NLP in the given text on the site https://www.regextester.com/

Basic Regular Expression Patterns

Anchors — ^ and $

^The matches any string that **starts with The -> Try it!**

end$ matches a string that **ends with end**

^The end$ **exact string match** (starts and ends with **The end**)

roar matches any string that **has the text roar in it**

Quantifiers — * + ? and {}

abc* matches a string that has **ab followed by zero or more c**

abc+ matches a string that has **ab followed by one or more c**

abc? matches a string that has **ab followed by zero or one c**

abc{2} matches a string that has **ab followed by 2 c**

abc{2,} matches a string that has **ab followed by 2 or more c**

abc{2,5} matches a string that has **ab followed by 2 up to 5 c**

a(bc)* matches a string that has **a followed by zero or more copies of the sequence bc**

a(bc){2,5} matches a string that has **a followed by 2 up to 5 copies of the sequence bc**

OR operator — | or []

a(b|c) matches a string that has **a followed by b or c**

a[bc] same as previous

Character classes — \d \w \s and .

\d matches a **single character** that is a **digit**

\w matches a **word character** (alphanumeric character plus underscore)

\s matches a **whitespace character** (includes tabs and line breaks)

. matches **any character**

\d, \w and \s also present their negations with \D, \W and \S respectively.

For example, \D will perform the inverse match with respect to that obtained with \d.

\D matches a **single non-digit character**

In order to be taken literally, you must escape the characters ^.[$()|*+?{\with a backslash \ as they have special meaning.

\$\d matches a string that has a **$ before one digit**

We can match also non-printable characters like tabs \t, new-lines \n, carriage returns \r

Flags

We are learning how to construct a regex but forgetting a fundamental concept: **flags**.

A regex usually comes within this form /abc/, where the search pattern is delimited by two slash characters /. At the end we can specify a flag with these values (we can also combine them each other):

- **g** (global) does not return after the first match, restarting the subsequent searches from the end of the previous match
- **m** (multi-line) when enabled ^ and $ will match the start and end of a line, instead of the whole string
- **i** (insensitive) makes the whole expression case-insensitive (for instance **/aBc/i** would match **AbC**)

Grouping and capturing — ()

a(bc) parentheses create a **capturing group with value bc**

a(?:bc)* using **?:** we **disable the capturing group**

a(?<foo>bc) using **?<foo>** we put a name to the group

This operator is very useful when we need to extract information from strings or data using your preferred programming language. Any multiple occurrences captured by

several groups will be exposed in the form of a classical array: we will access their values specifying using an index on the result of the match.

If we choose to put a name to the groups (using (?<foo>...)) we will be able to retrieve the group values using the match result like a dictionary where the keys will be the name of each group.

Bracket expressions—[]

[abc] matches a string that has **either an a or a b or a c** -> is the **same as a|b|c**

[a-c] matches a string that has **either an a or a b or a c** -> is the **same as a|b|c**

[a-fA-F0-9] a string that represents **a single hexadecimal digit, case insensitively**

[0-9]% a string that has a character **from 0 to 9 before a % sign**

[^a-zA-Z] a string that has **not a letter from a to z or from A to Z.** In this case the **^** is used as **negation of the expression**

Greedy and Lazy match

The quantifiers (* + {}) are greedy operators, so they expand the match as far as they can through the provided text.

For example, <.+> matches simple div in This is a simple div test. In order to catch only the div tag we can use a ? to make it lazy:

<.+?> matches **any character one or more** times included **inside <** and **>, expanding as needed**

Boundaries — \b and \B

\babc\b performs a **"whole words only" search**

\b represents an anchor like caret (it is similar to $ and ^) matching positions where one side is a word character (like \w) and the other side is not a word character (for instance it may be the beginning of the string or a space character).

It comes with its negation, \B. This matches all positions where \b doesn't match and could be if we want to find a search pattern fully surrounded by word characters.

\Babc\B matches only if the pattern is **fully surrounded by word** characters

Look-ahead and Look-behind — (?=) and (?<=)

d(?=r) matches a **d** only if is **followed by r, but r will not be** part of the overall regex **match**

(?<=r)d matches a **d** only if is **preceded by an r, but r will not be** part of the overall regex **match**

4. Finite Automata

The regular expression is more than just a convenient metalanguage for text searching. First, a regular expression is one way of describing a finite-state automaton (FSA). Finite-state automata are the theoretical foundation of a good deal of the computational work we will describe in this book, Any FSA regular expression can be implemented as a finite-state automaton (except regular expressions that use the memory feature; more on this later). Symmetrically, any finite-state automaton can be described with a regular expression. Second, a regular expression is one way of characterizing a particular kind of formal language called a regular language. Both regular expressions and finite-state automata can be used to describe regular languages. The relation among these three theoretical constructions is sketched out in the figure below.

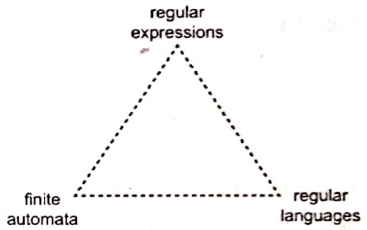


Figure 2: The relationship between finite automata, regular expressions,and regular languages

A formal language is completely determined by the 'words in the dictionary', rather than by any grammatical rules

A (formal) ***language*** L over alphabet Σ is just a set of strings in Σ*.

Thus any subset $L \subseteq \Sigma^*$ determines a language over Σ

The ***languaage determined by a regular expression*** r over Σ is

$$L(r) \stackrel{\text{def}}{=} \{v \in \Sigma^* | v \text{ matches } r\}.$$

Two regular expressions *r* and *s* (over the same alphabet) are ***equivalent*** iff *L*(r) and *L*(*s*) are equal sets (i.e. have exactly the same members.)

A finite automaton has a finite set of states with which it accepts or rejects strings.

Finite State Automata (FSA) can be:

Deterministic

On each input there is one and only one state to which the automaton can transition from its current state

Nondeterministic

An automaton can be in several states at once

Deterministic finite state automaton

1. A finite set of states, often denoted Q
2. A finite set of input symbols, often denoted Σ
3. A transition function that takes as arguments a state and an input symbol and returns a state. The transition function is commonly denoted δ If q is a state and a is a symbol, then δ(q, a) is a state p (and in the graph that represents the automaton there is an arc from q to p labeled a)
4. A start state, one of the states in Q
5. A set of final or accepting states F ($F \subseteq Q$)

A DFA is a tuple $A = (Q, \Sigma, \delta, q_0, F)$

Other notations for DFAs

Transition diagrams

- Each state is a node
- For each state $q \in Q$ and each symbol $a \in \Sigma$, let $\delta(q, a) = p$
- Then the transition diagram has an arc from q to p, labeled a
- There is an arrow to the start state q_0
- Nodes corresponding to final states are marked with doubled circle

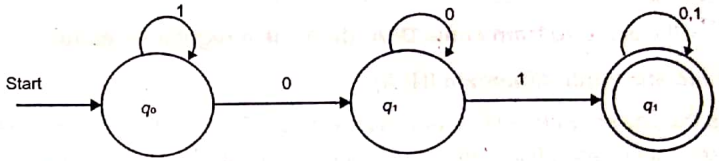
Transition tables

- Tabular representation of a function
- The rows correspond to the states and the columns to the input s
- The entry for the row corresponding to state q and the column corresponding to input a is the state δ(q, a)

$A = (\{ q_0, q_1, q_2 \}, \{ 0, 1 \}, \delta, q_0, \{ q_1 \})$

where the transition function δ is given by the table

		0	1
→	q_0	q_2	q_0
★	q_1	q_1	q_1
	q_2	q_2	q_1



The DFA define a language: the set of all strings that result in a sequence of state transitions from the start state to an accepting state.

Extended transition function

Describes what happens when we start in any state and follow any sequence of inputs

If δ is our transition function, then the extended transition function is denoted by δ

The extended transition function is a function that takes a state q and a string w and returns a state p (the state that the automaton reaches when starting in state q and processing the sequence of inputs w)

Formal definition of the extended transition function

Definition by induction on the length of the input string

Basis: $\delta(q, \epsilon) = q$

If we are in a state q and read no inputs, then we are still in state q Induction: Suppose w is a string of the form xa; that is a is the last symbol of w, and x is the string consisting of all but the last symbol

Then: $\delta(q, w) = \delta(\delta(q,x), a)$

To compute $\delta(q, w)$, first compute $\delta(q,x)$, the state that the automation is in after procesing all but the last symbol of *w*

Suppose this state is *p*, *i.e.*, $\delta(q,x) = p$

Then $\delta(q, w)$ is what we get by making a transition from state *p* on input *a* - the last symbol of *w*

Design a DFA to accept the language

L = {w | w has both an even number of 0 and an even number of 1}

The Language of a DFA

The language of a DFA A = (Q, Σ, δ, *q*o, *F*), denoted *L*(*A*) is defined by

$$L(A) = \{w \mid \delta(q_o, w) \text{ is in } F\}$$

The language of A is the set of strings w that take the start state *q*o to the one of the accepting states

If *L* is a *L(A)* from some DFA, then *L* is a regular language

Nondeterministic Finite Automata (NFA)

A NFA has the power to be in several states at once. This ability is often expressed as an ability to "guess" something about its input . Each NFA accepts a language that is also accepted by some DFA. NFA are often more succinct and easier than DFAs . We can always convert an NFA to a DFA, but the latter may have exponentially more states than the NFA (a rare case) .The difference between the DFA and the NFA is the type of transition function δ

For a NFA

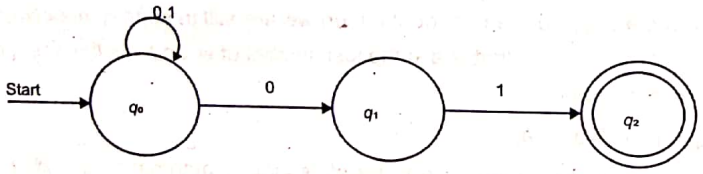
δ is a function that takes a state and input symbol as arguments

(like the DFA transition function), but returns a set of zero or more states

(rather than returning exactly one state, as the DFA must)

Example: An NFA accepting strings that end in 01

Nondeterministic automaton that accepts all and only the strings of 0s and 1s that end in 01



NFA: Formal definition

A nondeterministic finite automaton (NFA) is a tuple A = (Q, Σ, δ, q_0, F) where:

1. Q is a finite set of states
2. Σ is a finite set of input symbols
3. $q_0 \in Q$ is the start state
4. F (F ⊆ Q) is the set of final or accepting states
5. δ, the transition function is a function that takes a state in Q and an input symbol in Δ as arguments and returns a subset of Q

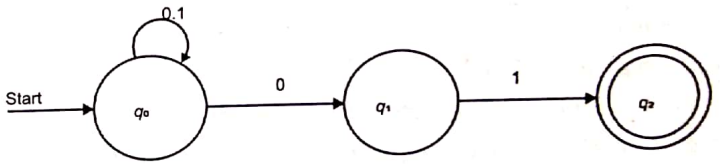
The only difference between a NFA and a DFA is in the type of value that δ returns

Example: An NFA accepting strings that end in 01

A = ({*q*0, *q*1, *q*2}, {0,1}, δ, *q*0, {*q*2})

where the transition function δ is given by the table

		0	1
→	*q*0	{*q*0, *q*1}	{*q*0}
	*q*1	θ	{*q*2}
★	*q*2	θ	θ



The Extended Transition Function

Basics: : δ^(q, ϙ) = {q}

Without reading any input symbols, we are only in the state we began in

Induction

Suppose w is a string of the form xa; that is a is the last symbol of w, and x is the string consisting of all but the last symbol

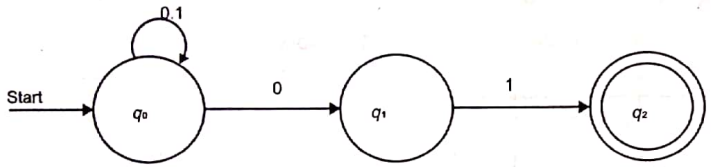
Also suppose that $\delta, (q, x) = \{P1, P2, ..., Pk,\}$

Let $\bigcup_{i=1}^{k} \delta(Pi, a) = \{r_1, r_2, ...r_m\}$

Then: $\delta(q, w) = \{r_1, r_2, ...r_m\}$

We compute $\delta\hat{}(q, w)$ by first computing $\delta\hat{}(q, x)$ and by then following any transition from any of these states that is labeled a

Example: An NFA accepting strings that end in 01



Processing $w = 00101$

1. $\delta(q0, \epsilon) = \{q0\}$
2. $\delta(q0, 0) = \delta(q0, 0) = \{q0, q1\}$
3. $\delta(q0, 00) = \delta(q0, 0) \cup \delta(q1, 0) = \{q0, q1\} \cup \theta = \{q0, q1\}$
4. $\delta(q0, 001) = \delta(q0, 1) \cup \delta(q1, 1) = \{q0\} \cup \{q2\}\ \{q0, q2\}$
5. $\delta(q0, 0010) = \delta(q0, 0) \cup \delta(q2, 0) = \{q0, q1\} \cup \theta = \{q0, q1\}$
6. $\delta(q0, 00101) = \delta(q0, 1) \cup \delta(q1, 1) = \{q0\} \cup \{q2\} = \{q0, q2\}$

The Language of a NFA

The language of a NFA A = (Q, Σ, δ, q_0, F), denoted L(A) is defined by

$L(A) = \{w \mid \delta(q0, w) \cap F \neq \theta\}$

The language of A is the set of strings $w \in \Sigma *$ such that $\delta\hat{}(q_0, w)$ contains at least one accepting state.

The fact that choosing using the input symbols of w lead to a non-accepting state, or do not lead to any state at all, does not prevent w from being accepted by a NFA as a whole.

Equivalence of Deterministic and Nondeterministic Finite Automata

Every language that can be described by some NFA can also be described by some DFA. The DFA in practice has about as many states as the NFA, although it has more transitions. In the worst case, the smallest DFA can have 2^n (for a smallest NFA with n state)

5. Finite-State Morphological Parsing

Consider a simple example: parsing just the productive nominal plural (-s) and the verbal progressive (-ing). Our goal will be to take input forms like those in the first column below and produce output forms like those in the second column.

Input	Morphological Parsed Output
cats	cat +N +PL
cat	cat +N +SG
cities	city +N +PL
geese	goose +N +PL
goose	(goose +N +SG) or (goose +V)
gooses	goose +V 3SG
merging	merge +V +PRES – PART
caught	(catch +V +PAST – PART) or (catch +V +PAST)

The second column contains the stem of each word as well as assorted morphological features. These features specify additional information about the stem. For example the feature +N means that the word is a noun; +SG means it is singular, +PL that it is plural. consider +SG to be a primitive unit that means 'singular'. Note that some of the input forms (like caught or goose) will be ambiguous between different morphological parses.

In order to build a morphological parser, we'll need at least the following:

1. a lexicon: The list of stems and affixes, together with basic information about them (whether a stem is a Noun stem or a Verb stem, etc).
2. morphotactics: the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. For example, the rule that the English plural morpheme follows the noun rather than preceding it.

3. orthographic rules: these spelling rules are used to model the changes that occur in a word, usually when two morphemes combine (for example the y ! ie spelling rule discussed above that changes city + -s to cities rather than cities).

6. Building a Finite-State Lexicon

A lexicon is a repository for words. The simplest possible lexicon would consist of an explicit list of every word of the language (every word, i.e. including abbreviations ('AAA') and proper names ('Jane' or 'Beijing') as follows: a, AAA, AA, Aachen, aardvark, aardwolf, aba, abaca, aback

Since it will often be inconvenient or impossible, for the various rea- sons we discussed above, to list every word in the language, computational lexicons are usually structured with a list of each of the stems and affixes of the language together with a representation of the morphotactics that tells us how they can fit together. There are many ways to model morphotactics; one of the most common is the finite-state automaton. A very simple finite-state model for English nominal inflection might look like Figure 3.

The FSA in Figure 3 assumes that the lexicon includes regular nouns (reg-noun) that take the regular -s plural (e.g. cat, dog, fox, aardvark). These are the vast majority of English nouns since for now we will ignore the fact that the plural of words like fox have an inserted e: foxes. The lexicon also includes irregular noun forms that don't take -s, both singular irreg-sg-noun (goose, mouse) and plural irreg-pl-noun (geese, mice).

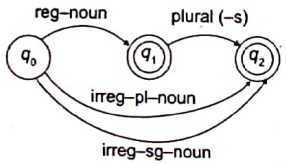


Figure 3: A finite-state automaton for English nominal inflection.

reg-noun	irreg-pi-noun	irreg-sg-noun	plural
fox	geese	goose	-s
cat	sheep	sheep	
dog	mice	mouse	
aardvark			

A similar model for English verbal inflection might look like Figure 4.

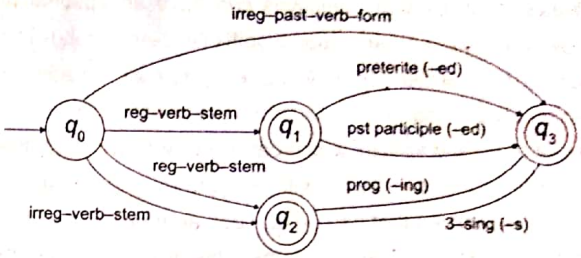


Figure 4: A finite-state automaton for English verbal inflection

This lexicon has three stem classes (reg-verb-stem, irreg-verb-stem, and irreg-past-verb-form), plus 4 more affix classes (-ed past, -ed participle, -ing participle, and 3rd singular -s):

reg-verb-stem	irreg-verb-stem	irreg-past-verb	past	past-part	pres-part	3sg
walk	cut	caught	-ed	-ed	-ing	-s
fry	speak	ate				
talk	sing	eaten				
impeach	sang					
	cut					
	spoken					

English derivational morphology is significantly more complex than English inflectional morphology, and so automata for modelling English derivation tend to be quite complex. Some models of English derivation, in fact, are based on the more complex context-free grammars

As a preliminary example, though, of the kind of analysis it would require, we present a small part of the morphotactics of English adjectives, taken from Antworth (1990). Antworth offers the following data on English adjectives:

big, bigger, biggest
cool, cooler, coolest, coolly
red, redder, reddest
clear, clearer, clearest, clearly, unclear, unclearly
happy, happier, happiest, happily
unhappy, unhappier, unhappiest, unhappily
real, unreal, really

An initial hypothesis might be that adjectives can have an optional pre- fix (un-), an obligatory root (big, cool, etc) and an optional suffix (-er, -est, or -ly). This might suggest the FSA in Figure 5. Alas, while this FSA will recognize all the adjectives in the table above, it will also recognize ungrammatical forms like unbig, redly, and realest. We need to set up classes of roots and specify which can occur with which suffixes. So adj-root1 would include adjectives that can occur with un- and -ly (clear, happy, and real) while adj-root2 will include adjectives that can't (big, cool, and red). Antworth (1990) presents Figure 6 as a partial solution to these problems. This gives an idea of the complexity to be expected from English derivation.

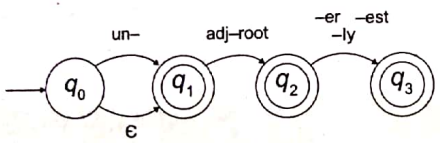


Figure 5: An FSA for a fragment of English adjective morphology: Antworth's Proposal #1.

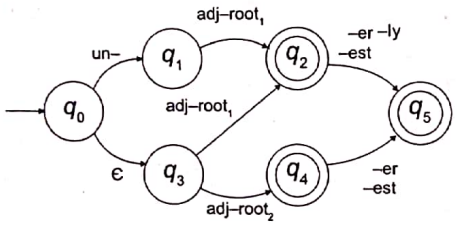


Figure 6: An FSA for a fragment of English adjective morphology: Antworth's Proposal #2

This gives an idea of the complexity to be expected from English derivation. For a further example, we give in Figure 7 another fragment of an FSA for English nominal and verbal derivational morphology, based on Sproat (1993), Bauer (1983), and Porter (1980). This FSA models a number of derivational facts, such as the well-known generalization that any verb ending in -ize can be followed by the nominalizing suffix -ation (Bauer, 1983; Sproat, 1993)). Thus since there is a word fossilize, we can predict the word fossilization by following states q0, q1, and q2. Similarly, adjectives ending in -al or -able at q5 (equal, formal, realizable) can take the suffix -ity, or sometimes the suffix -ness to state q6 (naturalness, casualness).

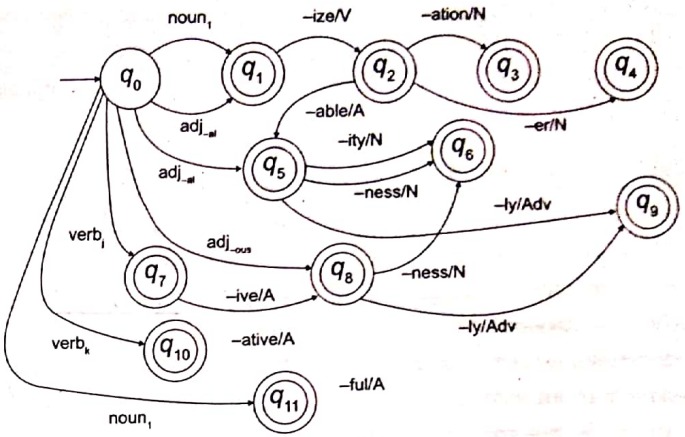


Figure 7: An FSA for another fragment of English derivational morphol-ogy.

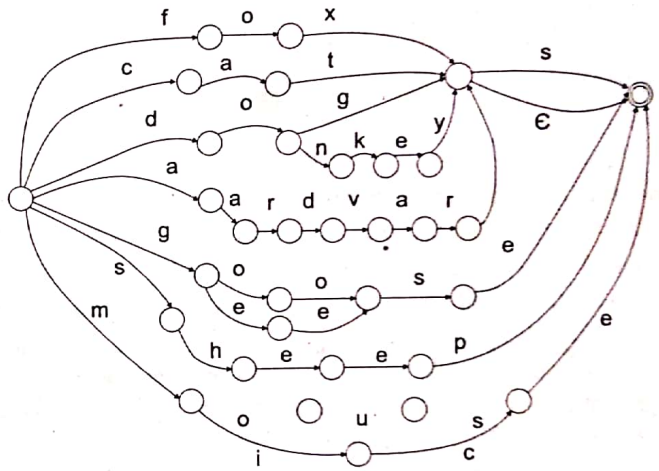


Figure 8: Compiled FSA for a few English nouns with their inflection. Note that this automaton will incorrectly accept the input foxs.

Figure 8 shows the noun-recognition FSA produced by expanding the Nominal Inflection FSA of Figure 9 with sample regular and irregular nouns for each class. We can use Figure 8 to recognize strings like aardvarks by simply starting at the initial state, and comparing the input letter by letter with each word on each outgoing arc, etc.,

Finite-State Transducers

We've now seen that FSAs can represent the morphotactic structure of a lexicon, and can be used for word recognition. A transducer maps between one representation and another; a finite-state transducer or FST is a type of finite automaton which maps between two sets of symbols. We can visualize an FST as a two-tape automaton which recognizes or generates pairs of strings. Intuitively, we can do this by labeling each arc in the finite-state machine with two symbol strings, one from each tape. Fig. 9 shows an example of an FST where each arc is labeled by an input and output string, separated by a colon.

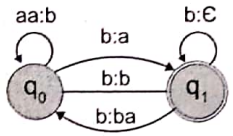


Figure 9: A finite-state transducer

The FST thus has a more general function than an FSA; where an FSA defines a formal language by defining a set of strings, an FST defines a relation between sets of strings. Another way of looking at an FST is as a machine that reads one string and generates another. Here's a summary of this four-fold way of thinking about transducers:

- FST as recognizer: a transducer that takes a pair of strings as input and outputs accept if the string-pair is in the string-pair language, and reject if it is not.
- FST as generator: a machine that outputs pairs of strings of the language. Thus the output is a yes or no, and a pair of output strings.
- FST as translator: a machine that reads a string and outputs another string.
- FST as set relater: a machine that computes relations between sets.

All of these have applications in speech and language processing. For morphological parsing (and for many other NLP applications), we will apply the FST as translator metaphor, taking as input a string of letters and producing as output a string of morphemes.

An FST can be formally defined in a number of ways; we will rely on the following definition, based on what is called the Mealy machine MEALY MACHINE extension to a simple FSA:

Q	a finite set of states $q0, q1, ..., qn-1$
Σ	a finite set corresponding to the input alphabet
Δ	a finite set corresponding to the output alphabet
$q_0 \in Q$	the start state
$F \subseteq Q$	the set of final states
$\delta(q, w)$	the transition function or transition matrix between states; Given a state $q \in Q$ and a string $w \in \Sigma^*$, $\delta(q, w)$ returns a set of new states $Q' \in Q$. δ is thus a function from $Q \times \Sigma^*$ to 2^Q (because there are 2^Q possible subsets of Q). δ returns a set of states rather than a single state because a given input may be ambiguous in which state it maps to.
$\sigma(q, w)$	the output function giving the set of possible output strings for each state and input. Given a state $q \in Q$ and a string $w \in \Sigma^*$, $\sigma(q, w)$ gives a set of output strings, each a string $o \in \Delta^*$. σ is thus a function from $Q \times \Sigma^*$ to 2^{Δ^*}.

Where FSAs are isomorphic to regular languages, FSTs are isomorphic to regular relations. Regular relations are sets of pairs of strings, a natural extension of the regular languages, which are sets of strings. Like FSAs and regular languages, FSTs and regular relations are closed under union, although in general they are not closed under difference, complementation and intersection (although some useful subclasses of FSTs are closed under these operations; in general, FSTs that are not augmented with the ε are more likely to have such closure properties). Besides union, FSTs have two additional closure properties that turn out to be extremely useful: inversion: The inversion of a transducer T (T^{-1}) simply switches the input and output labels. Thus, if T maps from the input alphabet I to the output alphabet O, T^{-1} maps from O to I.

composition: If T1 is a transducer from I1 to O1 and T2 a transducer from O1 to O2, then T1 ∘ T2 maps from I1 to O2.

Inversion is useful because it makes it easy to convert a FST-as-parser into an FST-as-generator. Composition is useful because it allows us to take two transducers that run in series and replace them with one more complex transducer. Composition works as in algebra; applying T1 ∘ T2 to an input sequence S is identical to applying T1 to S and then T2 to the result; thus T1 ∘ T2(S) = T2(T1(S)).

Fig. 10, for example, shows the composition of [a:b]+ with [b:c]+ to produce [a:c]+.

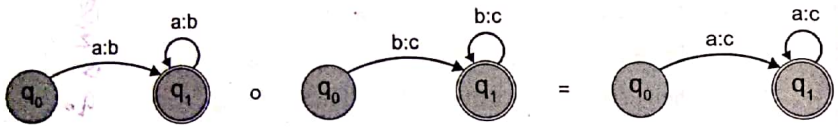


Figure 10: The composition of [a:b]+ with [b:c]+ to produce [a:c]+.

The projection of an FST is the FSA that is produced by extracting only one side of the relation. We can refer to the projection to the left or upper side of the relation as the upper or first projection and the projection to the lower or right side of the relation as the lower or second projection.

Morphological Parsing with Finite-State Transducers

Let's now turn to the task of morphological parsing. Given the input cats, for instance, we'd like to output cat +N +Pl, telling us that cat is a plural noun. Given the Spanish input bebo ('I drink'), we'd like beber +V +PInd +1P +Sg, telling us that bebo is the present indicative first-person singular form of the Spanish verb beber, 'to drink'. In the finite-state morphology paradigm that we will use, we represent a word as a correspondence between a lexical level, which represents a concatenation of mor- Surface phemes making up a word, and the surface level, which represents the concatenation of letters which make up the actual spelling of the word. Fig. 11 shows these two levels for (English) cats.

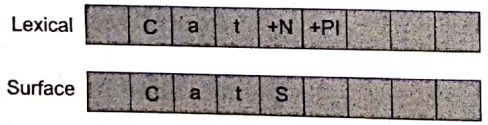


Figure 11: Schematic examples of the lexical and surface tapes;

The actual transducers will involve intermediate tapes as well.

For finite-state morphology it's convenient to view an FST as having two tapes. The upper or lexical tape, is composed from characters from one alphabet Σ. The lower or surface tape, is composed of characters from another alphabet Δ. In the two-level morphology of

Koskenniemi (1983), we allow each arc only to have a single symbol from each alphabet. We can then combine the two symbol alphabets Σ and Δ to create a new alphabet, Σ', which makes the relationship to FSAs quite clear. Σ' is a finite alphabet of complex symbols. Each complex symbol is composed of an input output pair i : o; one symbol i from the input alphabet Σ, and one symbol o from an output alphabet Δ, thus $\Sigma' \subseteq \Sigma \times \Delta$. Σ and Δ may each also include the epsilon symbol ε. Thus, where an FSA accepts a language stated over a finite alphabet of single symbols,

such as the alphabet of our sheep language:

(3.2) Σ = {b,a,!}

an FST defined this way accepts a language stated over pairs of symbols, as in:

(3.3) Σ'= {a : a, b : b, ! :!, a :!, a : ε, ε :!}

In two-level morphology, the pairs of symbols in Σ' Feasible pair are also called feasible pairs. Thus each symbol a : b in the transducer alphabet Σ'expresses how the symbol a from one tape is mapped to the symbol b on the other tape. For example a : ǫ means that an a on the upper tape will correspond to nothing on the lower tape. Just as for an FSA, we can write regular expressions in the complex alphabet Σ'. Since it's most common for symbols to map to themselves, in two-level morphology we call pairs like a : a default pairs, and just refer to them by the single letter a. We are now ready to build an FST morphological parser out of our earlier morphotactic FSAs and lexica by adding an extra "lexical" tape and the appropriate morphological features. Fig. 12 shows an augmentation of Fig. 13 with the nominal morphological features (+Sg and +Pl) that correspond to each morpheme. The symbol ^ indicates a morpheme boundary, while the symbol # indicates a word boundary. The morphological features map to the empty string ǫ or the boundary symbols since there is no segment corresponding to them on the output tape.

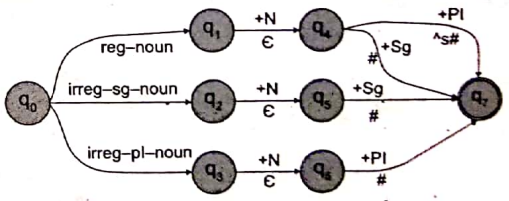


Figure 12: A schematic transducer for English nominal number inflection Tnum.

The sym- bols above each arc represent elements of the morphological parse in the lexical tape; the symbols below each arc represent the surface tape (or the intermediate

tape, to be described later), using the morpheme-boundary symbol ^ and word-boundary marker #. The labels on the arcs leaving q0 are schematic, and need to be expanded by individual words in the lexicon.

In order to use Fig. 11 as a morphological noun parser, it needs to be expanded with all the individual regular and irregular noun stems, replacing the labels reg-noun etc. In order to do this, we need to update the lexicon for this transducer, so that irregular plurals like geese will parse into the correct stem goose +N +Pl. We do this by allowing the lexicon to also have two levels. Since surface geese maps to lexical goose, the new lexical entry will be "g:g o:e o:e s:s e:e". Regular forms are simpler; the two-level entry for fox will now be "f:f o:o x:x", but by relying on the orthographic convention that f stands for f:f and so on, we can simply refer to it as fox and the form for geese as "g o:e o:e s e". Thus, the lexicon will look only slightly more complex:

reg-noun	irreg-pl-noun	irreg-sg-noun
fox	g o:e o:e s e	goose
cat	sheep	sheep
aardvark	m o:i u:ϵ s:c e	mouse

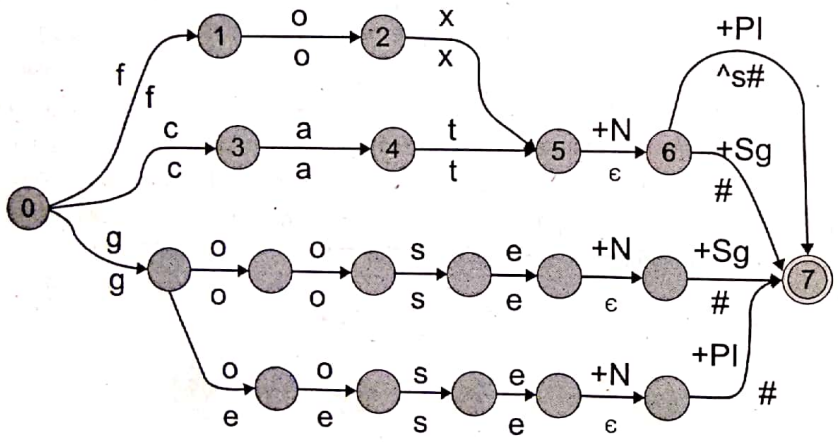


Figure 13: A fleshed-out English nominal inflection FST Tlex, expanded from Tnum by re- placing the three arcs with individual word stems (only a few sample word stems are shown).

The resulting transducer, shown in Fig. 13, will map plural nouns into the stem plus the morphological marker +Pl, and singular nouns into the stem plus the morphological marker +Sg. Thus a surface cats will map to cat +N +Pl. This can be viewed in feasible-pair format as follows: c:c a:a t:t +N:ϵ +Pl:^s# Since the output symbols include the morpheme and word boundary markers ^ and #, the lower labels do not correspond exactly to the surface level. Hence we refer to tapes with these morpheme boundary markers in Fig. 14 as intermediate tapes; the next section will show how the boundary marker is removed.

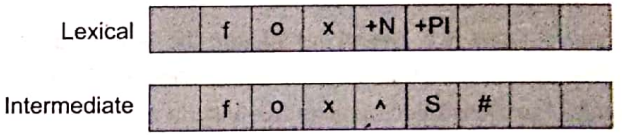


Figure 14: An example of the lexical and intermediate tapes.

Orthographic Rules and Finite-State Transducers

The method described in the previous section will successfully recognize words like aardvarks and mice. But just concatenating the morphemes won't work for cases where there is a spelling change; it would incorrectly reject an input like foxes and accept an input like foxs. We need to deal with the fact that English often requires spelling changes at morpheme boundaries by introducing spelling rules (or orthographic rules). Here will introduce a number of notations for writing such rules and shows how to implement the rules as transducers. Some of these spelling rules:

Name	Description of Rule	Example
Consonant doubling	I-letter consonant doubled before-ing/ed	beg/begging
E deletion	Silent e dropped before *-ing* and *-ed*	make/making
E Inserting	e added after *-s*, *-z*, *-x*, *-ch*, *-sh* before *-s*	watch/watches
Y replacement	*-y* changes to *-ie* before *-s*, *-i* before *-ed*	try/tries
K insertion	verbs ending with vowel + *-c* add *-k*	panic/panicked

We can think of these spelling changes as taking as input a simple concatenation of morphemes and producing as output a slightly-modified, (correctly-spelled) concatenation of morphemes. Figure 15 shows the three levels we are talking about:

lexical, intermediate, and surface. So for example we could write an E-insertion rule that performs the mapping from the intermediate to surface levels shown in Figure 15.

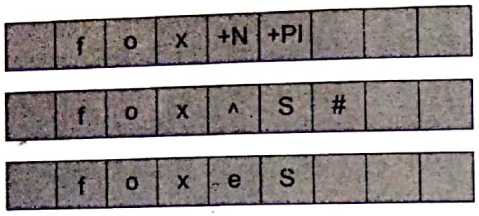


Figure 15: An example of the lexical, intermediate and surface tapes.

Between each pair of tapes is a 2-level transducer; the lexical transducer between the lexical and intermediate levels, and the E-insertion spelling rule between the intermediate and surface levels. The E-insertion spelling rule inserts an e on the surface tape when the intermediate tape has a morpheme boundary ^ followed by the morpheme

Such a rule might say something like "insert an e on the surface tape just when the lexical tape has a morpheme ending in x (or z, etc) and the next morpheme is -s.

$$\epsilon \rightarrow e / \left\{ \begin{matrix} x \\ s \\ z \end{matrix} \right\} \hat{} - s \#$$ (T..1)

This is the rule notation of Chomsky and Halle (1968); a rule of the form a -> b/ c__d means 'rewrite a as b when it occurs between c and d'. Since the symbol ε means an empty transition, replacing it means inserting something. The symbol ^ indicates a morpheme boundary. These boundaries are deleted by including the symbol ^:ε in the default pairs for the transducer; thus morpheme boundary markers are deleted on the surface level by default. (Recall that the colon is used to separate symbols on the intermediate and surface forms). The # symbol is a special symbol that marks a word boundary. Thus (T..1) means 'insert an e after a morpheme-final x, s, or z, and before the morpheme s'. Figure 15 shows an automaton that corresponds to this rule.

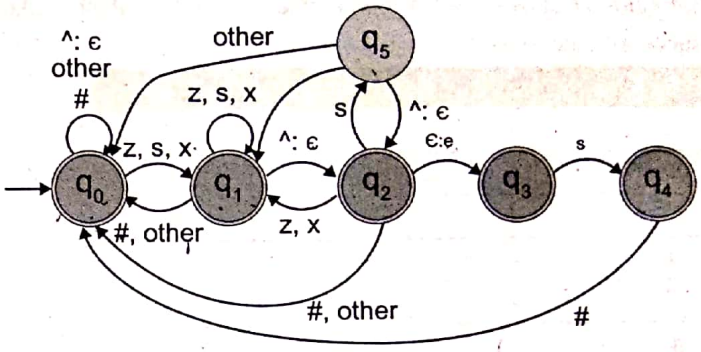


Figure 16: The transducer for the E-insertion rule of (T..1), extended from a similar transducer in Antworth

The idea in building a transducer for a particular rule is to express only the constraints necessary for that rule, allowing any other string of symbols to pass through unchanged. This rule is used to insure that we can only see the ε:e pair if we are in the proper context. So state q0, which models having seen only default pairs unrelated to the rule, is an accepting state, as is q_1, which models having seen a z, s, or x. q_2 models having seen the morpheme boundary after the z, s, or x, and again is an accepting state. State q_3 models having just seen the E-insertion; it is not an accepting state, since the insertion is only allowed if it is followed by the s morpheme and then the end-of-word symbol #.

The other symbol is used in Figure 16 to safely pass through any parts of words that don't play a role in the E-insertion rule. other means 'any feasible pair that is not in this transducer'; it is thus a version of @:@ which is context-dependent in a transducer-by-transducer way. So for exam- ple when leaving state q_0, we go to q_1 on the z, s, or x symbols, rather than following the other arc and staying in q_0. The semantics of other depends on what symbols are on other arcs; since # is mentioned on some arcs, it is (by definition) not included in other, and thus, for example, is explicitly mentioned on the arc from q_2 to q_0. A transducer needs to correctly reject a string that applies the rule when it shouldn't. One possible bad string would have the correct environment for the E-insertion, but have no insertion. State q_5 is used to insure that the e is always inserted whenever the environment is appropriate; the transducer reaches q_5 only when it has seen an s after an appropriate morpheme boundary. If the machine is in state q_5 and the next symbol is #, the machine rejects the string (because there is no legal transition

on # from q_5). Figure 17 shows the transition table for the rule which makes the illegal transitions explicit with the '–' symbol.

State\Input	5:5	x:x	z:z	^:Є	Є:e	#	other
q_0:	1	1	1	0	-	0	0
q_1:	1	1	1	2	-	0	0
q_2:	5	1	1	0	3	0	0
q_3:	4	-	-	-	-	-	-
q_4:	-	-	-	-	-	0	-
q_5:	1	1	1	2	-	-	0

Figure 17: The state-transition table for E-insertion rule of Figure T.2.

7. Lexicon free FST Porter stemmer

While building a transducer from a lexicon plus rules is the standard algorithm for morphological parsing, there are simpler algorithms that don't require the large on-line lexicon demanded by this algorithm. These are used especially in Information Retrieval (IR) tasks in which a user needs some information, and is looking for relevant documents (perhaps on the web, perhaps in a digital library database). User gives the system a query with some important characteristics of documents she desires, and the IR system retrieves what it thinks are the relevant documents. One common type of query is Boolean combinations of relevant keywords or phrases, e.g. (marsupial OR kangaroo OR koala). The system then returns documents that have these words in them. Since a document with the word marsupials might not match the keyword marsupial, some IR systems first run a stemmer on the keywords and on the words in the document. Since morphological parsing in IR is only used to help form equivalence classes, the details of the suffixes are irrelevant; what matters is determining that two words have the same stem.

One of the most widely used such stemming algorithms is the simple and efficient Porter (1980) algorithm, which is based on a series of simple cascaded rewrite rules. Since cascaded rewrite rules are just the sort of thing that could be easily implemented as an FST, we think of the Porter algorithm as a lexicon-free FST stemmer . The algorithm contains rules like:

Rule: ATIONAL ! ATE (e.g. relational ! relate)

Rule: ING ! ε if stem contains vowel (e.g. motoring ! motor)

Do stemmers really improve the performance of information retrieval engines? One problem is that stemmers are not perfect. Following kinds of errors of omission and of commission in the Porter algorithm:

Errors of Commission

organization - organ

Doing - doe

generalization - generic

numerical - numerous

policy - police

university - universe

negligible - negligent

Errors of Omission

European - Europe

analysis - analyzes

matrices - matrix

noise - noisy

sparse - sparsity

explain - explanation

urgency - urgent

Porter Stemmer

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 S1 satisfies 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

Step 1 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 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

8. N –Grams

Imagine listening to someone as they speak and trying to guess the next word that they are going to say. For example, what word is likely to follow this sentence fragment?

I'd like to make a collect. . .

Probably the most likely word is call, although it's possible the next word could be telephone, or person-to-person or international. (Think of some others). Guessing the next word (or word prediction) is an essential subtask of speech recognition, hand-writing recognition, augmentative communication for the disabled, and spelling error detection. In such tasks, word-identification is difficult because the input is very noisy and ambiguous. Thus, looking at previous words can give us an important clue about what the next ones are going to be.

N-gram is simply a sequence of N words. For instance, let us take a look at the following examples.

1. San Francisco (is a 2-gram)
2. The Three Musketeers (is a 3-gram)
3. She stood up slowly (is a 4-gram)

Now which of these three N-grams have you seen quite frequently? Probably, "San Francisco" and "The Three Musketeers". On the other hand, you might not have seen "She stood up slowly" that frequently. Basically, "She stood up slowly" is an example of an N-gram that does not occur as often in sentences as Examples 1 and 2.

Now if we assign a probability to the occurrence of an N-gram or the probability of a word occurring next in a sequence of words, it can be very useful. Why?

First of all, it can help in deciding which N-grams can be chunked together to form single entities (like "San Francisco" chunked together as one word, "high school" being chunked as one word).

It can also help make next word predictions. Say you have the partial sentence "Please hand over your". Then it is more likely that the next word is going to be "test" or "assignment" or "paper" than the next word being "school".

We formalize this idea of word prediction with probabilistic models called N-gram models, which predict the next word from the previous N − 1 words. Such statistical models of word sequences are also called language models of LMs. Computing the probability of the next word will turn out to be closely related to computing the probability of a sequence of words. The following sequence, for example, has a non-zero probability of appearing in a text:

. . . all of a sudden, I notice three guys standing on the sidewalk...

while this same set of words in a different order has a much lower probability:

on guys all I of notice sidewalk three a sudden standing the

As we will see, estimators like N-grams that assign a conditional probability to possible next words can be used to assign a joint probability to an entire sentence. Whether estimating probabilities of next words or of whole sequences, the N-gram model is one of the most important tools in speech and language processing. N-grams are essential in any task in which we have to identify words in noisy, ambiguous input. In speech recognition, for example, the input speech sounds are very confusable and many words sound extremely similar. N-gram models are also essential in statistical machine translation.

It can also help to make spelling error corrections. For instance, the sentence "drink coffee" could be corrected to "drink coffee" if you knew that the word "coffee" had a high probability of occurrence after the word "drink" and also the overlap of letters between

"coffee" and "coffee" is high. In spelling correction, we need to find and correct spelling errors like the following that accidentally result in real English words:

They are leaving in about fifteen minutes to go to her house.

The design and construction of the system will take more than a year.

Since these errors have real words, we can't find them by just flagging words that aren't in the dictionary. But note that in about fifteen minutes is a much less probable sequence than in about fifteen minutes. A spellchecker can use a probability estimator both to detect these errors and to suggest higher-probability corrections. Word prediction is also important for augmentative communication systems that help the disabled. People who are unable to use speech or sign language to communicate, like the physicist Steven Hawking, can communicate by using simple body movements to select words from a menu that are spoken by the system. Word prediction can be used to suggest likely words for the menu. Besides these sample areas, N-grams are also crucial in NLP tasks like part-of- speech tagging, natural language generation, and word similarity, as well as in applications from authorship identification and sentiment extraction to predictive text input systems for cell phones.

Language model

Problem of Modelling Language

Formal languages, like programming languages, can be fully specified. All the reserved words can be defined and the valid ways that they can be used can be precisely defined. We cannot do this with natural language. Natural languages are not designed; they emerge, and therefore there is no formal specification. There may be formal rules for parts of the language, and heuristics, but natural language that does not confirm is often used. Natural languages involve vast numbers of terms that can be used in ways that introduce all kinds of ambiguities, yet can still be understood by other humans. Further, languages change, word usages change: it is a moving target. Nevertheless, linguists try to specify the language with formal grammars and structures. It can be done, but it is very difficult and the results can be fragile. An alternative approach to specifying the model of the language is to learn it from examples.

Statistical Language Modelling, or Language Modelling and LM for short, is the development of probabilistic models that are able to predict the next word in the sequence given the words that precede it. It is a probability distribution over sequences of words.

Given such a sequence, say of length m, it assigns a probability P(*w*1,...,*wm*) to the whole sequence.

The goal of probabilistic language modelling is to calculate the probability of a sentence of sequence of words:

P(W) = P(w1, w2, w3,..., wn)

and can be used to find the probability of the next word in the sequence:

P(w5 | w1, w2, w3, w4)

A model that computes either of these is called a Language Model

Method for Calculating Probabilities:

Conditional Probability:

let A and B be two events with P(B) =/= 0, the conditional probability of A given B is:

$$P(A|B) = \frac{P(A, B)}{P(B)}$$

$$P(x_1, x_2,....x_n) = P(x_1)P(x_2|x_1)...P(x_n|x_1,...,x_{n-1})$$

Chain Rule

The chain rule applied to compute the joined probability of words in a sequence is therefore:

$$P(w_1 w_2....w_n) = \prod_i P(w_i|w_1 w_2...w_{i-1})$$

For example,

$$P(\text{"its water is so transparent"}) =$$

$$P(its)*$$

$$P(water|\ its)*$$

$$P(is|\ its\ water)*$$

$$P(so|\ its\ water\ is)*$$

$$P(transparent|\ its\ water\ is\ so)$$

This is a lot to calculate, could we not simply estimate this by counting and dividing the results as shown in the following formula:

$$P(transparent|its\ water\ is\ so) = \frac{count(its\ water\ is\ so\ transparent)}{count(its\ water\ is\ so)}$$

In general, no! There are far to many possible sentences in this method that would need to be calculated and we would like have very sparse data making results unreliable.

Markov Property

A stochastic process has the Markov property if the conditional probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state, not on the sequence of events that preceded it. A process with this property is called a Markov process.

In other words, the probability of the next word can be estimated given only the previous k number of words.

For example, if k=1:

$P(transparent|\ its\ water\ is\ so) \approx P(transparent|\ so)$

or if k=2:

$P(transparent|\ its\ water\ is\ so) \approx P(transparent|\ is\ so)$

General equation for the Markov Assumption, k=i :

$P(w_i|\ w_1 w_2...w_{i-1}) \approx P(w_i|\ w_{i-k}...\ w_{i-1})$

The language model provides context to distinguish between words and phrases that sound similar. For example, in American English, the phrases "recognize speech" and "wreck a nice beach" sound similar, but mean different things.

Data sparsity is a major problem in building language models. Most possible word sequences are not observed in training. One solution is to make the assumption that the probability of a word only depends on the previous n words. This is known as an n-gram model or unigram model when n = 1. The unigram model is also known as the bag of words model.

Estimating the relative likelihood of different phrases is useful in many natural language processing applications, especially those that generate text as an output. Language modeling is used in speech recognition, machine translation, part-of-speech tagging, parsing, Optical Character Recognition, handwriting recognition, information retrieval and other applications.

In speech recognition, sounds are matched with word sequences. Ambiguities are easier to resolve when evidence from the language model is integrated with a pronunciation model and an acoustic model.

Language models are used in information retrieval in the query likelihood model. There a separate language model is associated with each document in a collection. Documents are ranked based on the probability of the query Q in the document's language model P(Q | Md). Commonly, the unigram language model is used for this purpose.

N-gram language model

Statistical language models, in its essence, are the type of models that assign probabilities to the sequences of words. In this article, we'll understand the simplest model that assigns probabilities to sentences and sequences of words, the n-gram

You can think of an N-gram as the sequence of N words, by that notion, a 2-gram (or bigram) is a two-word sequence of words like "please turn", "turn your", or "your homework", and a 3-gram (or trigram) is a three-word sequence of words like "please turn your", or "turn your homework"

Let's start with equation P(w|h), the probability of word w, given some history, h. For example,

P(The|its water is so transparent that)

Here,

w = The

h = its water is so transparent that

And, one way to estimate the above probability function is through the relative frequency count approach, where you would take a substantially large corpus, count the number of times you see its water is so transparent that, and then count the number of times it is followed by the. In other words, you are answering the question:

Out of the times you saw the history h, how many times did the word w follow it

P(the | its water is so transparent that) = C(its water is so transparent that the)/C(its water is so transparent that)

Now, you can imagine it is not feasible to perform this over an entire corpus; especially if it is of a significant size.

This shortcoming and ways to decompose the probability function using the chain rule serves as the base intuition of the N-gram model. Here, you, instead of computing probability using the entire corpus, would approximate it by just a few historical words

The Bigram Model

As the name suggests, the bigram model approximates the probability of a word given all the previous words by using only the conditional probability of one preceding word. In other words, you approximate it with the probability: P(the | that)

And so, when you use a bigram model to predict the conditional probability of the next word, you are thus making the following approximation:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1})$$

This assumption that the probability of a word depends only on the previous word is also known as Markov assumption.

Markov models are the class of probabilistic models that assume that we can predict the probability of some future unit without looking too far in the past. The Markov assumption is the presumption that the future behavior of a dynamical system only depends on its recent history. In particular, in a kth-order Markov model, the next state only depends on the k most recent states, therefore an N-gram model is a (N−1)-order Markov model.

Maximum Likelihood Estimate

$$P(w_i|w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

$$P(w_i|w_{i-1}) = \frac{c(w_i|w_{i-1})}{c(w_{i-1})}$$

Example

$$P(w_i|w_{i-1}) = \frac{c(w_i|w_{i-1})}{c(w_{i-1})}$$

<s> I am Sam </s>
<s> I am Sam </s>
<s> I do not like green eggs and ham </s>

$P(I|\text{<s>}) = \frac{2}{3} = .67$ $\quad P(\text{Sam}|\text{<s>}) = \frac{1}{3} = .33$ $\quad P(\text{am}|\text{I}) = \frac{2}{3} = .67$

$P(\text{</s>}|\text{Sam}) = \frac{1}{2} = 0.5$ $\quad P(\text{Sam}|\text{am}) = \frac{1}{2} = .5$ $\quad P(\text{do}|\text{I}) = \frac{1}{3} = .33$

You can further generalize the bigram model to the trigram model which looks like two words into the past and can thus be further generalized to the N-gram model. Basically, an N-gram model predicts the occurrence of a word based on the occurrence of its N – 1 previous words. An N-gram model uses the previous N-1 words to predict the next one: P(wn | wn-N+1 wn-N+2… wn-1)

P(phone | Please turn off your cell)

Number of parameters required grows exponentially with the number of words of prior context.

Unigram: P(phone)

Bigram: P(phone | cell)

Trigram: P(phone | your cell)

Another example for the sentence boy is chasing the big dog

unigrams: P(dog)

bigrams: P(dog | big)

trigrams: P(dog | the big)

quadrigrams: P(dog | chasing the big)

Assume a language has T word types in its lexicon, how likely is word x to follow word y?

Simplest model of word probability: 1/T

Alternative 1: estimate likelihood of x occurring in new text based on its general frequency of occurrence estimated from a corpus (unigram probability)

popcorn is more likely to occur than unicorn

Alternative 2: condition the likelihood of x occurring in the context of previous words (bigrams, trigrams,...) Estimate the probability of each word given prior context.

mythical unicorn is more likely than mythical popcorn

So here we are answering the question – how far back in the history of a sequence of words should we go to predict the next word? For instance, a bigram model (N = 2) predicts the occurrence of a word given only its previous word (as N – 1 = 1 in this case). Similarly, a trigram model (N = 3) predicts the occurrence of a word based on its previous two words (as N – 1 = 2 in this case).

Let us see a way to assign a probability to a word occurring next in a sequence of words. First of all, we need a very large sample of English sentences (called **a corpus**).

For the purpose of our example, we'll consider a very small sample of sentences, but in reality, a corpus will be extremely large. Say our corpus contains the following sentences:

1. He said thank you.
2. He said bye as he walked through the door.
3. He went to San Diego.

4. San Diego has nice weather.
5. It is raining in San Francisco.

Let's assume a bigram model. So we are going to find the probability of a word based only on its previous word. In general, we can say that this probability is (the number of times the previous word 'wp' occurs before the word 'wn') / (the total number of times the previous word 'wp' occurs in the corpus) =

(Count (wp wn))/(Count (wp))

Let's work this out with an example.

To find the probability of the word "you" following the word "thank", we can write this as P (you | thank) which is a conditional probability.

This becomes equal to:

=(No. of times "Thank You" occurs) / (No. of times "Thank" occurs)

= 1/1

= 1

We can say with certainty that whenever "Thank" occurs, it will be followed by "You" (This is because we have trained on a set of only five sentences and "Thank" occurred only once in the context of "Thank You"). Let's see an example of a case when the preceding word occurs in different contexts.

Let's calculate the probability of the word "Diego" coming after "San". We want to find the P (Diego | San). This means that we are trying to find the probability that the next word will be "Diego" given the word "San". We can do this by:

=(No of times "San Diego" occurs) / (No. of times "San" occurs)

= 2/3

= 0.67

This is because in our corpus, one of the three preceding "San"s was followed by "Francisco". So, the P (Francisco | San) = 1 / 3.

In our corpus, only "Diego" and "Francisco" occur after "San" with the probabilities 2 / 3 and 1 / 3 respectively. So if we want to create a next word prediction software based on our corpus, and a user types in "San", we will give two options: "Diego" ranked most likely and "Francisco" ranked less likely.

Generally, the bigram model works well and it may not be necessary to use trigram models or higher N-gram models.

Challenges

There are, of course, challenges, as with every modeling approach, and estimation method. Let's look at the key ones affecting the N-gram model, as well as the use of Maximum Likelihood Estimation (MLE)

Sparse data

Not all N-grams found in the training data, need smoothing

Change of domain

Train on WSJ, attempt to identify Shakespeare – won't work

Sensitivity to the training corpus

The N-gram model, like many statistical models, is significantly dependent on the training corpus. As a result, the probabilities often encode particular facts about a given training corpus. Besides, the performance of the N-gram model varies with the change in the value of N.

Moreover, you may have a language task in which you know all the words that can occur, and hence we know the vocabulary size V in advance. The closed vocabulary assumption assumes there are no unknown words, which is unlikely in practical scenarios.

Smoothing

A notable problem with the MLE approach is sparse data. Meaning, any N-gram that appeared a sufficient number of times might have a reasonable estimate for its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it.

As a result of it, the N-gram matrix for any training corpus is bound to have a substantial number of cases of putative "zero probability N-grams"

N-gram for spelling correction

Let's define the job of a spell checker and an auto corrector. A word needs to be checked for spelling correctness and corrected if necessary, many a time in the context of the surrounding words. A spellchecker points to spelling errors and possibly suggests alternatives. An auto corrector usually goes a step further and automatically picks the most likely word. In case of the correct word already having been typed, the same is retained. So, in practice, an auto correct is a bit more aggressive than a spellchecker, but this is more of an implementation detail — tools allow you to configure the behaviour.

There is not much difference between the two in theory. So, the discussion in the rest of the blog post applies to both.

There are different types of spelling errors. Let's try to classify them a bit formally. Note that the below is for the purpose of illustration, and is not an rigorous linguistic approach to classifying spelling errors.

Non-word Errors: These are the most common type of errors. You either miss a few keystrokes or let your fingers hurtle a bit longer. E.g., typing langage when you meant language; or hurryu when you meant hurry

Real Word Errors: If you have fat fingers, sometimes instead of creating a non-word, you end up creating a real word, but one you didn't intend. E.g, typing buckled when you meant bucked. Or your fingers are a tad wonky, and you type in three when you meant there.

Cognitive Errors: The previous two types of errors result not from ignorance of a word or its correct spelling. Cognitive errors can occur due to those factors. The words piece and peace are homophones (sound the same). So you are not sure which one is which. Sometimes your damn sure about your spellings despite a few grammar nazis claim you're not.

Short forms/Slang/Lingo: These are possibly not even spelling errors. May be u r just being kewl. Or you are trying hard to fit in everything within a text message or a tweet and must commit a spelling sin. We mention them here for the sake of completeness.

Intentional Typos: Well, because you are clever. You type in teh and pwned and zomg carefully and frown if they get autocorrected. It could be a marketing trick, one that probably even backfired.

The word N-gram approach to spelling error detection and correction is to generate every possible misspelling of each word in a sentence either just by typographical modifications (letter insertion, deletion, substitution), or by including homophones as well, (and presumably including the correct spelling), and then choosing the spelling that gives the sentence the highest prior probability. That is, given a sentence $W = \{w_1 ; w_2; : : : ; w_k ; : : : ; w_n\}$, where wk has alternative spelling w'k , w"k , etc, we choose the spelling among these possible spellings that max- imizes P(W), using the N-gram grammar to compute P(W). A class-based N-gram can be used instead, which can find unlikely part-of-speech combinations, although it may not do as well at finding unlikely word combinations.

Expected Questions:

1. What is morphology. Why do we need to do Morphological Analysis? Discuss various application domains of Morphological Analysis.
2. Explain derivational & inflectional morphology in detail with suitable examples.
3. What are morphemes? What are different ways to create words from morphemes?
4. What is language model? Explain the use of Language model?
5. Write a note on N-Gram language Model.
6. What is the role of FSA in Morphological analysis? Explain FST in detail.

Syntax Analysis

3

OBJECTIVES

After reading this chapter, the student will be able to understand:

- Part-Of-Speech tagging (POS)- Tag set for English (Penn Treebank) ,
- Rule based POS tagging, Stochastic POS tagging,
- Issues –Multiple tags & words, Unknown words.
- Introduction to CFG,
- Sequence labelling: Hidden Markov Model (HMM), Maximum Entropy, and Conditional Random Field (CRF).