Natural Language Systems

Todd Davies

December 23, 2015

Introduction

Enabling computers to use 'natural language' (the kind of language that people use to communicate with one another) is becoming more and more important. It allows people to communicate with them without having to use strange artificial languages and awkward devices like keyboards and mice; and it allows the computer to access the enormous amount of material that is stored as natural language text on the web.

This course provides an introduction to this area, mixing theory (if you don't understand the theory of how language works you cannot possibly write programs that understand it) with practice (if you haven't written or played with tools that embody the theory, you can't get a concrete handle on what the theory means).

Aims

The course unit aims to teach the techniques required to extend the theoretical principles of computational linguistics to applications in a number of critical areas.

- To demonstrate how the essential components of practical NLP systems are built and modified.
- To introduce the principal applications of NLP, including information retrieval & extraction, spoken language access to software services, and machine translation.
- To explain the major challenges in processing large-scale, real-world natural language.
- To explain the principles underlying speech recognition and synthesis, and to explore the power of 'black box' tools for these tasks.
- To give students an understanding of the issues involved in evaluating NLP systems.

Contents

1 Introduction

We want computers to be able to interact with us, just like we interact with them. This involves having them understand written text and voiced speech, as well as being able to synthesise speech and text themselves. This includes things like translation text and searching for key words in text.

A computer or a suite of programs that can do all of this is the goal for Natural Language Systems. The catch is, that language is hard and complicated, and to make computers do the things we want them to, we need to know how3 language works, and express this as an executable program.

Language is the representation of ideas, and the linkage of different ideas together in such a way as to create new ideas. In order to understand any one sentence (a sentence usually corresponds to one idea, event or action), we have to understand what each symbol in the language means in isolation, and understand how they're connected, and what the connections to do change the meaning of the ideas.

Many factors affect the meaning of a sentence, but the connection between words is always hierarchical, and we can represent sentences as trees:

A parse tree is all well and good, but to a computer, this is only slightly more useful than the original text. Though we have extracted some information out of the text, we still just have a hierarchy of words, but we want a hierarchy of ideas.

Having ideas instead of words allows us to infer more than what the text literally says:

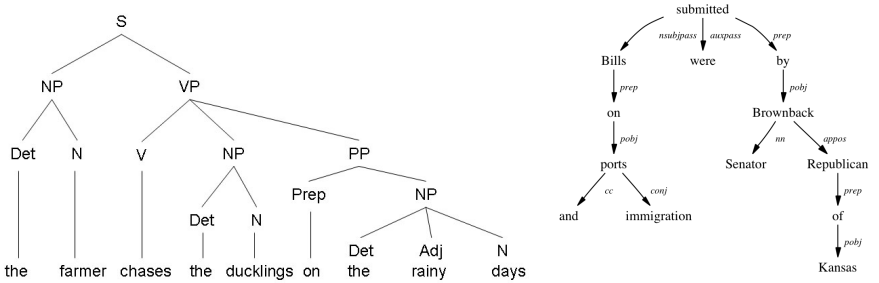


Figure 1: The left image is a phase structure tree, and the right image is a dependency tree.

- I'm fixing my motorbike → This person possesses a motorbike, and it is currently broken.
- The cake smells good → There is cake somewhere. Somebody is close enough to smell it.

But how can we do that?

2 Structural analysis

It is possibly to try and find out the meaning of a word simply by looking at what letters it is made up of. One way to do this is to split a word into **morphemes**, which are the most basic meaning-carrying components of a word, and try to associate a meaning with each. For example *undone* could be split into *un* and *done*, and meaning associated with each.

2.1 Tries

Note:

Tries are very handy datastructures for technical interviews, you should read up on them an implement one!

In order to examine the syntactic and semantic properties of the words, we need to represent them in the computer. A common way to do this is with a *trie*:

Tries are very memory efficient, since they if multiple words share the same prefix, then the prefix is only stored once in memory. Tries have a lookup time of $O(m)$, where m is the length of the word, which is quite good, and is better than a hash table in terms of speed in some cases. If you're stupid enough to represent your dictionary as a list of words, then you can do a binary search if its ordered (worst case $O(log(n) * m)$ comparisons (the m comes from having to possibly compare each character in the word)), or a linear search if it isn't ordered ($O(m * n)$ in the worst case!).

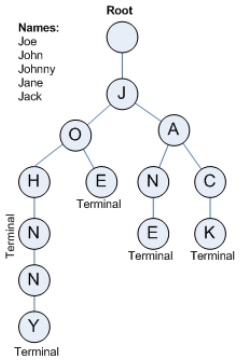


Figure 2: A trie storing some names.

2.2 Spelling rules

We want to understand why combining *big* and *est* produces *biggest* with an extra *g*. Why isn't it *bigest*? The reason why we want to understand this, is so we can go from a word that we're processing in text, and pick it apart into its components so we can better understand it.

That is to say, we're going from *biggest* to *big* + *est*.

The format of the rules we're using in the course is as follows:

> Note:
>
> You can use `cX` and `vX` where `X` is an integer, and `c/v` denotes a consonant or vowel inside the context brackets.

```
[from] ==> [to]: [prevContext] _ [nextContext];
```

For example, if we had a rule like:

```
[g] ==> []: [g] _ [e,s,t];
```

It would turn *biggest* into *big* + *est*.

2.3 Categorical descriptions

Even if we find the meaning of every word by splitting it info morphemes, we just end up with a collection of words that we know the construction of, but we're still no closer to understanding a sentence.

In order to find the relationships between words in a sentence, we should use the approach in Figure **??**, and try to fit the words into some tree structure.

One way of doing this, is to specify lots of different forms that a sentence can take. For example 'noun verb noun' might describe a sentence such as 'Todd writes notes'. Providing that we have some *prototype* for a sentence that fits the

words that we've been given, we can ascertain that we can in fact make a sentence out of these words.

There is a better approach though. Each word in a sentence changes its meaning in some way, and we can put words into buckets according to how the meaning of the sentence is changed; for example, a verb specifies what kind of event is happening. Having a verb on its own doesn't do us much good; we will know that *something* is happening, but not where, why, what, when etc. Verbs need other parts of sentence around them to make them work.

We can treat each word as part of a jigsaw, specifying what other words or phrases it needs in order to have meaning, and then fit the pieces together according to some schema.

We need to specify a schema with which we can specify what different words require:

```
<word> = <rules>
```

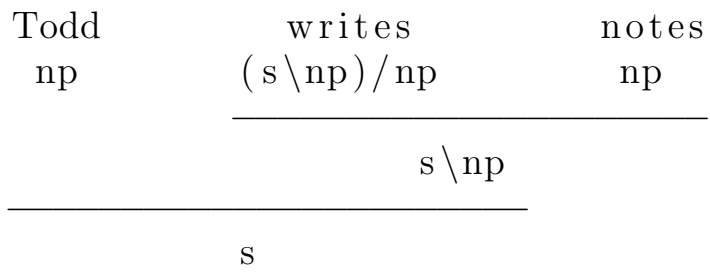
The rules are their own language, which has a few rules. Brackets are treated as they usually are in maths, and the only other special characters are forward and backward slashes. These indicate whether a word or phrase should be before or after the word.

```
writes = (s\np)/np
```

This indicates that the word 'writes' needs a noun phrase (`np`) to its right and then a noun phrase to its left to make a sentence (`s`).

With the right set of rules for each word, we can now parse sentences:

```
writes = (s\np)/np
Todd = np
notes = np
```



Here, the words 'Todd' and 'notes' are *saturated* since they don't require anything else to make them into complete 'items'. 'writes' is unsaturated, since it needs other stuff to make it into a complete item. Word rules of these kind are called **Categorical Descriptions**.

2.4 Morphology

Now we've figured out how to decompose a word into morphemes using spelling rules, and we can fit these words into a sentence. However, we also want to know the precise meaning for each word (this helps when arranging them in a sentence too). We can do this by looking at each morpheme.

There are two types of morphology that we're going to look at:

Inflectional morphology
: This is when the stem of the word is incomplete, and other morphemes provide more information to specify exactly what we mean:

- 'sing' + 'ing' = verb + present participle
- 'work' + 'ed' = verb + past participle
- 'work' + '' = noun + singular

Derivational morphology

This is where the meaning of the stem is significantly changed by other morphemes. For example, 'smelly' could be combined with 'er' to give 'smellier', or 'est' to give 'smelliest'. Obviously we need spelling rules to do this correctly!

We need a way to specify what words can have what suffixes/affixes. For example, 'conscript' can be combined with 'tion' to give 'conscription', but not 'ly' to give 'conscriptly'.

Furthermore, there are spelling rules concerned with adding bits onto words; as we saw before, 'smelly' becomes 'smellier', not 'smellyer'. We will come across this later though.

It turns out that composing morphemes is similar to composing words. We can use the same notation:

```
'conscript' = noun>agr
'conscript' = verb>tns
'tion' = (noun>agr)<(verb>tns)
'ing' = tns
'ed' = tns
's' = agr
'' = agr
```

These descriptions allow you to construct the following words:

- conscript (noun)
- conscripts (noun)
- conscription (noun)
- conscriptions (noun)
- conscripting (verb)
- conscripted (verb)

However, the rules are not perfect, and will also allow you to make:

- conscriptingtion (noun)
- conscriptedtion (noun)
- conscriptingtions (noun)
- conscriptedtions (noun)

We can also make rules cancel out. If we have a rule that is 'verb>tns' and another that is 'tns>agr' then we can make a 'verb>agr' from them.

To see a worked example of these rules, and how they're applied, look on slides $64 - 110$ in the course notes.

One thing that is important to note, is that we can process words from left to right, and don't have to back-track. This means that processing is in linear time, which is fantastic (though we need a big dictionary of words, which is rather less fantastic).