

Natural Language Systems

Todd Davies

December 22, 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

2 Structural analysis **3**

2.1	Tries	3
2.2	Spelling rules	4

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 how 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:

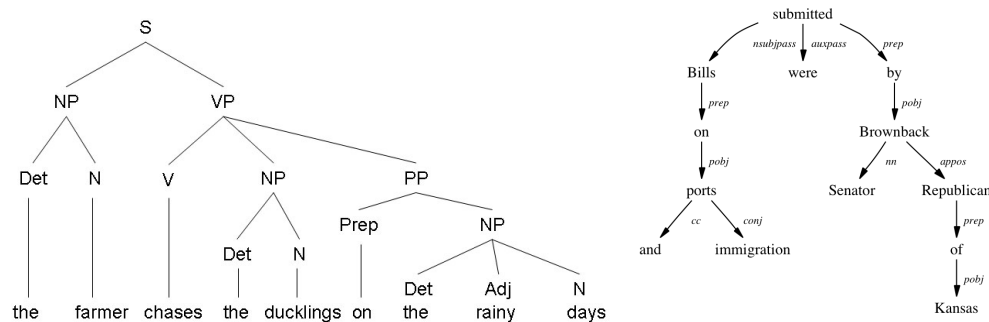


Figure 1: The left image is a phase structure tree, and the right image is a dependency tree.

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:

- 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 possible 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

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 handy datastructures for technical interviews, you should read up on them and implement one!

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!).

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:

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

For example, if we had a rule like:

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

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

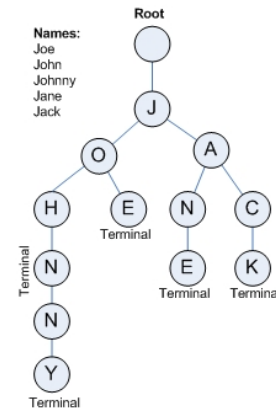


Figure 2: A trie storing some names.

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