Natural Language Systems

Todd Davies

January 18, 2016

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 translating 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:

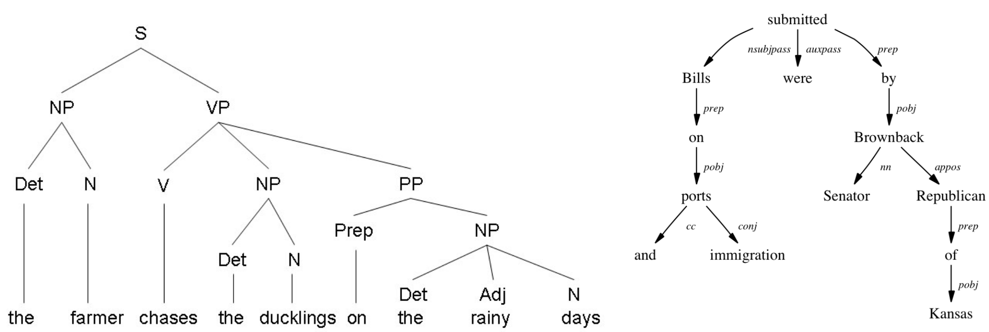


Figure 1: The left image is a phase structure tree, and the right image is a dependency tree.

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 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

Tries are very handy datastructures for technical interviews, you should read up on them and 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!).

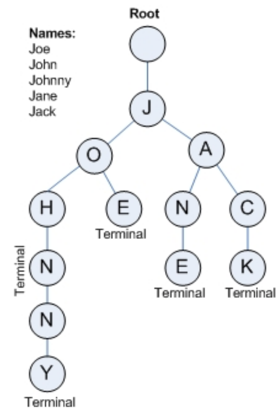


Figure 2: A trie storing some names.

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:

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 1, 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 a 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
```

$$\dfrac{\begin{array}{c}\text{Todd}\\ np\end{array} \quad \dfrac{\begin{array}{c}\text{writes}\\ (s\backslash np)/np\end{array} \quad \begin{array}{c}\text{notes}\\ np\end{array}}{s\backslash np}}{s}$$

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).

2.5 Unknown words

Now we can read in a sentence, parse each word and extract the relations between the words to produce a meaningful parse tree, great! But... what if we don't have a word in the input sentence in our dictionary? We won't be able to fit it into our parse tree since it won't have any meaning to us. Looking at the morphemes doesn't tell us too much; having 'ing' at the end means that a word is probably a verb and is in the present tense, but doesn't tell us what's actually going on.

There are two different classes of words; *open* and *closed*. Open classes are verbs, adjectives, nouns etc where there are lots of them and you can easily add more (new nouns are created all the time). Closed classes are things like prepositions ('in', 'on'), and auxiliaries ('be', 'have', 'would') where you rarely if ever get new words being added.

So, if we cannot recognise a word using the morphological rules we used before, then we should **back off** to using a more robust but less accurate strategy. Lets define exactly what robust and accurate mean:

Robust
: This is when a system always gives an answer, even if the answer might be wrong. It's sometimes a good idea have a robust system, since then you can always take *some* action.

Accurate
: An accurate system always gets the answer 'right' when asked a question.

A common strategy in Natural Language Processing is to use a system that is highly accurate but less robust, but fall back on a less accurate but more robust system when the first doesn't give an answer.

So, we need to build a robust word recogniser for unknown words. To do this, we need to know what word it is, and what part of speech tag it has (noun, verb etc).

2.5.1 Stemming

If you were to come across the word 'blodge', you probably wouldn't know what it means (although there are, as always a number of definitions in the Urban Dictionary). If a word like 'blodge' were to appear in the middle of a sentence, you could probably still extract some information from it:

- I went to blodge → Blodge is a place.
- I went to blodged it → Blodging is a thing you can do. The blodging happened in the past.
- The car was blodge → You can describe something as having the property/quality 'blodge'.

The **Porter Stemmer** is a set of rules that tell you what bits of a word you can remove to get the stem of the word that is common to all forms of the word. If we run the porter stemmer on 'blodge', 'blodging' and 'blodged', then we get the stem as 'blodg'. Now we can at least identify all instances of 'blodg*' as the same word.

2.6 Part of speech tagging

When we look up a word in a dictionary, we get something like this:

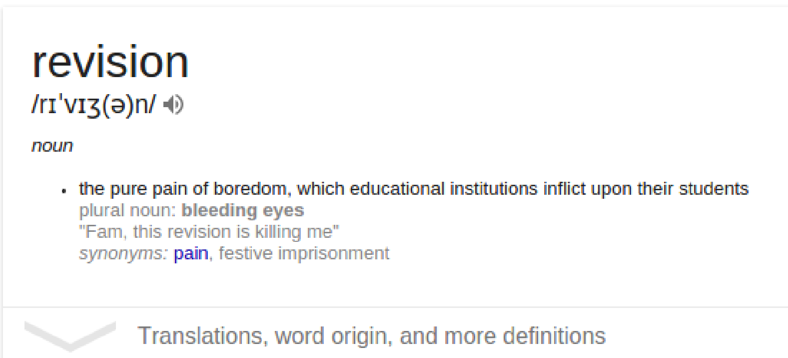


Figure 3: A dictionary definition of the word 'revision'.

Not only does a dictionary give us the definition of a word, but it also give us its part of speech tag. In the case of Figure 3 it is a noun. We need the POS tag in order to work out how words are related to each other, so we can put them in their proper place in a parse tree.

There are a number of ways to do part of speech tagging (this is what I'm doing for my third year project, and it's a rabbit hole that you don't want to go down if you're trying to revise for exams). Here are the ones you need to know for this course:

One thing to be aware of with using corpuses, is that they are often ordered. If you use the first N million words for training and the next M million for testing, then you could end up with disastrous results, because different areas of the corpus will be about different genres, and will contain different words.

Dictionary look up

The easiest way to do POS tagging (but also the worst), is to get a big massive dictionary, and find the part of speech tag for each word. Then, when you need to tag a word, then you just look it up in your big list.

However, you need lots and lots and lots of words to do this, and getting the tags for all of the words is a big task. Furthermore, if we're trying to handle unknown words, then by definition, we've not seen them before and they won't be in our dictionary.

Obviously, the bigger your list of words, the fewer words you'll need to handle. The British National Corpus has 100 million words in, which is in the right ballpark, but it is poorly tagged (the error rate is above 1% in parts). Other corpuses are available, but are either smaller, or you have to pay for them. In order to use the BNC (or any corpus) as a dictionary, you count how many times each word is tagged with each part of speech tag, and then assign the most common tag.

There are two main advantages to this method; first of all, it's really simple, and computationally easy. Using a sensible hash table, lookup is $O(1)$ and training is $O(n)$. It's so fast that the bottleneck is usually the IO operations of reading (and decoding) the corpus. The second advantage, is that you get alternative tags for words. A word might be in the corpus 50 times as a verb, but 20 times as a noun, and you can use this information in your processing.

As a **backoff** technique, you can also keep track of the POS probabilities for the last n letters of the word, so if the word isn't in your dictionary, you can use the last n letters of it to determine what tag to assign.

Transition probabilities and HMM's

HMM's use probabilities and context to determine what tag a word should get. To train a HMM, we collect statistics on what part of speech tags *follow and precede* other ones. For example, the probability of a noun being followed by another noun might be quite high, because of names (e.g. Todd Davies), and adjectives usually lead onto nouns, or other adjectives (e.g. hungry bored Todd).

These are called **bigram** probabilities (because you have information about following and preceding tags). A HMM based POS tagger works as follows:

- For each word, use a corpus to calculate how likely it is to belong to each class (just like we did for the dictionary look up approach). These are the emission probabilities.
- For each tag that we have found that might correspond to a specific word, look at the tag we assigned to the previous (and possibly next) word(s) and use the transition probabilities to choose the most likely tag.

We want to find the most likely sequence of tags for the whole sentence, which is why the hidden markov model works well. However, training can be slow and its hard to make use of the backward transition probabilities.

I think this is more important for the POS tagging part of the course than HMM's are.

Alternately, you can use no HMM at all, by using a function to evaluate the best next tag:

$$
\begin{aligned}
&F(dict(Word, tag_i),\\
&\quad \sum_k (P(tag(prevWord) = tag_k) \times P(tag_k \rightarrow tag_i)),\\
&\quad \sum_k (dict(nextWord, tag_k) \times P(tag_i \leftarrow tag_k)))
\end{aligned}
$$

You can define F to be anything, but in the course notes:

$$F(dictProbability, forward, backward) = \sqrt{dictProbability} \times (forward + backward)$$

Transformation Based Learning

This is the subject of my third year project, when I've finished it, I might remember to update these notes with a link to the code/my dissertation.

Doing statistical tagging is pretty tedious, and requires a lot of text. Is there another way of doing POS tagging? Maybe one that could require less data?

This is very possible, as was demonstrated in Eric Brill's PhD thesis in 1995. The idea is to have a *base tagger*, which is your first guess at what the tag is (this is any POS tagger, like the ones above for example), and then have rules that can correct mistakes in the output.

This presents its own problems though. For example, how do you generate rules? The solution is this:

1. Tag a small part of a corpus by hand, so the tags are 100% (or very close to it) accurate.
2. Then you get come up with a set of rule *templates*. Templates (in this course) are of the form `#name(entity1, entity2, entity3): oldTag > newTag if condition;` A set of templates looks something like this:
   - `#t0(T1,T2): T1 > T2 if tag[0] = T1`
     Change tag T1 to tag T2 in every instance.

- `#t1(T1,T2,T3):  T1 > T2 if tag[-1] = T3`
  Change tag T1 to tag T2 if the previous tag was T3.
- `#w0(T1,T2,W1):  T1 > T2 if word[0] = W1`
  Change tag T1 to tag T2 if the current word is equal to W1.

3. Tag the corpus with the base tagger (something like a dictionary lookup tagger, or transition probability tagger) plus any rules you've generated so far, and then find the errors that it made. For every error, **instantiate a rule based on the context of the error**.
4. Now you have lots of rules, you score each one by how many times it would have made a good change, and how many times it would have made a bad edit if you ran it on the whole corpus.
5. Select the top rule, and run steps 3-4 again until you have enough rules.

I'm not sure if I like the format of the rule templates here, since they require you to specify that you're turning one specific tag into another. An example of an instantiated rule is:

```
#t0(UN,NN,UN): UN > NN if tag[0]=UN;
```

This says turn every 'UN' into a 'NN'. What if we wanted to turn any tag into a 'NN' if it was preceded by a 'UN' though? We couldn't do this with this specific rule set, unless you had wildcards or something, e.g.:

```
#t1(*,NN,UN): * > NN if tag[-1]=UN;
```

When you're assigning a score to each rule, you should give a gross score and a net score. The gross score is how many problems the rule would fix, but the net score is how many problems it would fix minus the number of correct taggings it wrongly breaks. You might end up with something like this:

Rule ranking	Gross	Net
1	500	300
2	450	240
3	412	234
4	375	121
5	297	100
⋮	⋮	⋮

A brill tagger isn't really a part of speech tagger on its own, but it does go a long way to making existing POS taggers better. Depending on the characteristics of the base tagger, you can get anywhere from half a percentage extra accuracy, to a full ten percent extra. Remember though, if the base tagger is already 98% accurate, half a percent is a big improvement!

For the exam, you might need to know exact definitions of *precision*, *recall* and *f-measure*. They are:

Precision: When the tagger emits a POS tag, how likely is it to be right?

Recall: How many of the right things does it say?

F-measure: $F = \frac{2 \times p \times r}{p+r}$

Confusion matrices No matter how good our POS tagging algorithms are, they're always going to get *something* wrong. Confusion matrices are a way to spot common errors that our taggers make. It is a table showing the correct tags along one side, and the tags the POS tagger came up with along the other. Something like in Table 1.

Apart from using them to gloat about your awesome POS tagger to your natural language systems friends, confusion matrices are useful for improving the accuracy of your tagger. If you

	NN	VB	PUN	...
NN	80	4	0	...
VB	12	74	1	...
PUN	0	0	12	...
⋮	⋮	⋮	⋮	⋱

Table 1: A sample confusion matrix. Along the top is the number of 'correct' tags, along the side is the number of tags we predicted.

have multiple taggers, each with a different confusion matrix, then you can use the values in the matrix to determine which one tagger to believe on any word.

If Tagger A says a word is a noun, but Tagger B another says its a determiner, then what do you do? You can look at the confusion matrix, and see that 93% of the time, the Tagger A gets nouns right, but 98% of the time Tagger B gets determiners right, then you should go with Tagger B. It's kind of like if you herded a load of lecturers into a room and started asking them CS questions. If you asked a question about complexity theory, you're more likely to get a good answer from somebody who teaches an advanced algorithms course than you are from somebody who teaches a UX course.

NLP stands for Natural Language Processing, not Neuro-Linguistic Programming...

Special cases Sometimes, a POS tag is very ambiguous; the kind of thing that would have linguists arguing for ages trying to determine which was right. In these cases, we don't want to assign the wrong tag and mis-lead whatever system is using our tags further down the NLP pipeline. As a solution, we could define some special cases (the word 'that' is hard to tag, for example), and if we're not sure about a tag, make the tag equal to the word. This technique is called **underspecification**.

2.7 Regular expressions

Now we can find the part of speech tag for any word (or at least give a good guess for it), we can start to assign structure to the sentence. Given the words of a sentence and their part of speech tags, using a grammar to parse sentences can be done in $O(n^3)$ time **so long as lecixal ambiguity and out- of- place words are ignored**. The trouble with this, is that both these things really matter, and an algorithms that doesn't work, isn't very much use at all.

So, when our grammar does break down in those cases, we need to be able to *back off* to another more robust technique. This could be regular expressions. Say a noun phrase is made of an optional determiner, zero or more adjectives and then a noun:

```
NP = DET? ADJ* NOUN
```

A verb phrase is made of a verb and either one or two noun phrases, while a sentence is made of a noun phrase and a verb phrase:

```
VP = (VERB PREP? NP) | (VERB PREP? NP NP)
s = NP VP
```

Given a string that was tagged (correctly), you could apply the regular expression to the string to get the sentences:

The reader wishes for a long break.

DET NOUN VERB PREP DET ADJ NOUN

NP NP

VP

s

If we tag a sentence from the BNC, and run it through a simple regex, then we can efficiently and effectively parse the sentence. First we need to go from the BNC tags (they have different types of nouns, adjectives etc, looking like 'NN4' for a type 4 noun) to more sensible tags:

```
'noun': 'NN.'
'adj': 'AJ.'
'det0': '(AT|D.).'
...
```

Then we need to recognise phrases:

```
'nmod': 'adj|noun'
'np0': '((det0? nmod* noun)|name|pron)'
...
```

Then you end up with something like:

<NP><DET>The</DET><NOUN>reader</NOUN></NP><VP><VERB>wishes</VERB>
<PREP>for</PREP><NP><DET>a</DET><ADJ>long</ADJ><NOUN>break</NOUN>
<NP></VP><PUN>.</PUN>

This is good, and reasonable fast when your regex is small, but unfortunately, the regex has to get quite large for the output to be a correct parsing of the sentence, and then the regex parsing is quite slow. Furthermore, the output looks horrible, and since regex matching is an opaque thing, you can't debug it very well!

Regexes also don't give you alternate options for parsing. They give you one answer (i.e. 'the string you gave matches this structure'), or no answers at all.

Ultimately, you can recognise a sentence and its structure using regexes, but it's hard to make them parse a sentence just how you like it. Regexes aren't the exactly easy to come up with, and in order to properly define a sentence structure we need recursion (a noun phrase may contains another noun phrase). Unfortunately, regexes don't know about recursion, so we need to cascade them:

```
'a': 'c b'
'b': 'a d'
...
```

2.8 Supertagging

Make me a pull request to fill in this section, do it now!

`http://github.com/Todd-Davies/third-year-notes`

2.9 Deterministic dependency parsing

Using a regex to parse the structure of a sentence isn't widely done. In practice, the way that gets the best results is to use a non-deterministic algorithm. However, (as you may know all to well from `COMP36111`) non- deterministic algorithms don't run very quickly at all on deterministic computing machines, and will often exhibit an exponential runtime.

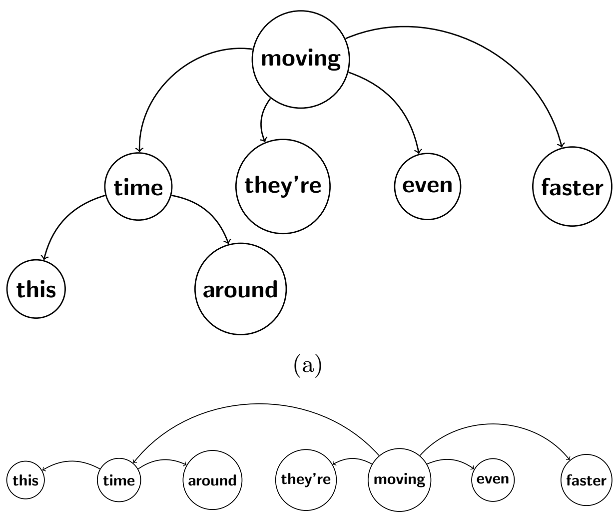
We therefore want to find a deterministic algorithm to extract a sentence structure. When we're thinking about this, we want to optimise for three things:

- Accuracy - get the correct answer
- Robustness - at least *give* an answer
- Speed - try to have a polynomial time complexity

A good approach is to try and make a system that has a good accuracy, then compromise until the other two criteria are satisfied too.

To parse a sentence into a structure, we can recognise the following points:

- Each word has exactly one parent (except the root word)
- You can draw arrows between words in a sentence, and none of the arrows will cross (see Figure 4b for an explanation).



(a)

(b)

Figure 4: (a)A dependency tree for the sentence 'This time around they're moving even faster'. (b) An alternative representation of the dependency tree. Note how none of the lines cross.

2.9.1 MALT

We can come up with a linear time algorithm for comping up with dependency relations (i.e. the arrows on Figure 4b). It uses three data structures:

A **list** of the input words (we treat it like a queue).

A **stack** of words that we've looked at, but haven't yet assigned a parent

A **set** of dependency relations for the output.

The algorithm can do three things:

Shift: Moves the word onto the stack

Left arc: Sets the top item on the stack as a dependent of the head of the input list, and removes it from the stack. This item is now gone.

Right arc: Sets the head of the input list as a dependent of the top item on the stack, move the head of the input list onto the stack.

I can't find what MALT stands for, but it seems to have a webpage: `http://maltparser.org/`

Since *shift* removes an item from the input list, the *arc* operations both remove an item from consideration altogether and at most one dependency is produced at every step, we can see that the algorithm runs in both linear time and space.

So, we have a mechanism for producing dependencies, but we don't actually have any logic to drive it yet; we need some rules to determine what actions we should take for each input item.

There are some rules that constitute 'low hanging fruit' here. First of all, we always need to perform exactly one operation on each iteration of the algorithm (since otherwise the state doesn't change). Secondly, sometimes there's only one thing we *can* do. For example, if there's nothing on the stack, then you can only do a *shift*.

Two rules that do this are:

```
// Always left arc with the dependency 'mod' if we don't know what else to do
// And the stack isn't empty
{ input: [?,...], stack: [?,...] } ==> leftArc(mod)
// Always shift if there is nothing on the stack
{ input: [?,...], stack: [] } ==> shift(mod)
```

In fact, for any (valid) sequence of operations we perform on the input, we will *always* end up with a dependency tree, so our algorithm is **robust**. The accuracy of our algorithm depends on the rules we give it; though we only have a choice of three operations at each iteration of the algorithm, if we do something wrong, then that error will propagate through the tree and we might end up getting it very wrong. We should make it so that the rules are applied in order (going down a list) and put the two rules shown above at the bottom (since they're the ones that will always apply and make our algorithm robust). We need to come up with good rules to put before them though...

In order to handle special cases, we could make left shift arc look anywhere in the stack, and right arc look anywhere in the queue. This defeats the object of the queue and stack (since they're FIFO and LIFO datastructures respectively), but never mind.

Of course, we could employ a linguist to come up with rules, or (as lazy and stingy computer scientists) we could try and learn a good set of rules. Of course, if we have some example sentences that we have the correct dependency relation trees for, we can see what MALT operations would have worked to get the sentence into the form given in the tree. To turn this information into rules, we can literally just encode it in the rule syntax:

```
{input: [old, man, ate, it], stack; [the], relations:[]} ==> shift
```

Obviously, this is very, very specific to our training sentence ('the old man ate it'), and we've overfitted our Machine Learning algorithm to our training data. However, we can generalise the rule, by replacing all the words with their part of speech tags:

```
{input: [ADJ, NOUN, VERB, PRONOUN], stack; [DET], relations:[]} ==> shift
```

Which is obviously less general; it would match a sentence like 'the small fish swam off'. However, we could generalise further by only looking at part of the input list and stack:

```
{input: [ADJ, NOUN, ...], stack; [DET, ...], relations:[]} ==> shift
```

Having more general rules means that we will probably have fewer rules, and we will have to fall back on our two 'dumb' robust rules less often, which are likely to make a mistake. However, if we make rules too general then we're probably not looking at enough information and are likely to make bad decisions (its analogous to how stereotyping people can make you misinformed/make bad decisions). We need to find the sweet spot for generality though; as we saw, if we have very specific rules, then we need lots of them to cover all the cases!

2.10 Pesky formats

Up to now, I've been taking it for granted that we have examples of parse trees, lists of tagged words etc all ready to use. If you've ever tried to write some software that does natural language processing, you might have realised that this is a very naive assumption.

Getting hold of corpora is a non-trivial task lots of the time. Not only do you have to pay for some of them (i.e. Penn Treebank, where you get 10% free and you have to stump up for the rest), but its very rare that they exist in the format you want.

For my third year project, I'm using the British National Corpus, which is encoded in XML (in fact, I've often found parsing the XML to be the largest bottleneck in my project, performance wise).

Unfortunately, its not always as easy as 'parse this XML'. For example, the Penn Treebank derives its name partly from the *headed phrase structure trees* that it contains. These are basically parse trees in a slightly different format than how we want them. An example headed phrase structure tree is given in Figure 5.

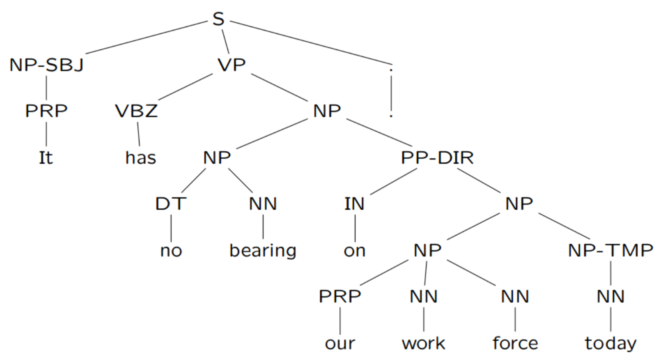


Figure 5: A headed phrase structure tree

In order to convert headed phrase structure trees into dependency trees, we can use the following algorithm:

1. Pick the most important child of each node
   - If it's a word, return a leaf node containing the word.
   - Else, recurse and convert it to a dependency tree
2. Recurse on all the other children, and make them subtrees of the important child we handled first.

This produces dependency trees that look like what's in Figure 6.

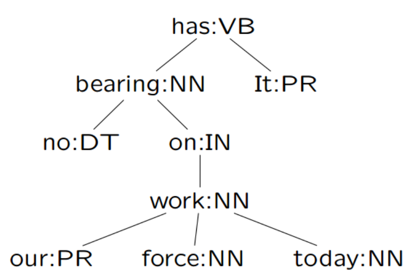


Figure 6: The dependency tree that was derived from the headed phrase structure tree in Figure 5.

The last thing we need to think about, is how to pick the most important child of each node (step 1 of the algorithm). The way to do this is by using a **Head Percolation Table (HPT)**. This lists each type of node ('S', 'VP' etc) and lists the order of importance whereby each child node should be ranked. See Table 2 for an example.

Root type	Most important	Second most important	...
S	VP	S	...
NP	NN	NNP	...
⋮	⋮	⋮	⋱

Table 2: A sample Head Percolation Table. Its important that the HPC be as accurate as possible, since otherwise the resulting dependency tree will be silly, and it'll make any MALT rules we learn from them silly.

2.11 N-Fold Cross Validation

N-Fold Cross Validation is a method used to prevent overfitting of Machine Learning models. Essentially, you split your data into N parts, and train your model N times. Then, you do N iterations of the training (training on $N-1$ parts each time), and then test on the remaining part. You can then average the results and thus obtain a reasonably accurate estimate for the error rate of your model. Figure 7 explains this.

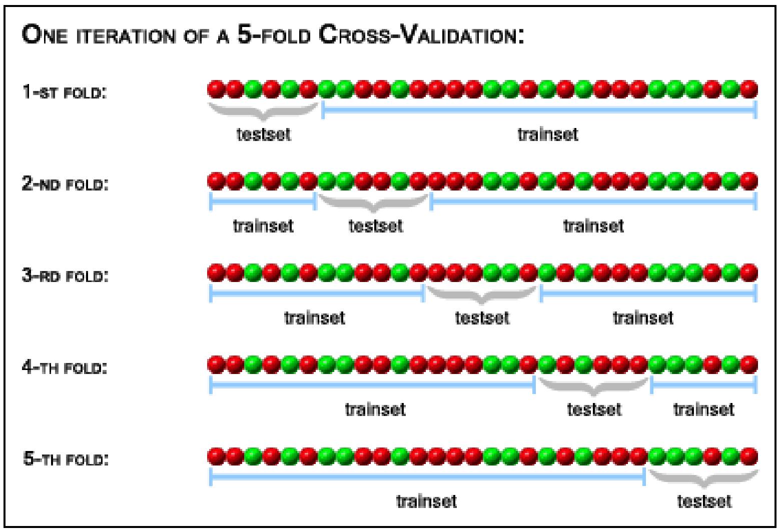


Figure 7: A visual explanation of N-Fold Cross Validation.

3 Semantics and Inference

Given any (normal) piece of text, a reasonably well informed and intelligent human can figure out, with a variable degree of certainty what the text means. This section is about conferring that skill to computers.

If you had to say what the blanked out word in the following sentence was; "The ******* saluted crisply to the general as he walked past, his uniform buttons gleaming", you'd probably guess 'soldier'. But why? Well, we can infer from the context that it's probably a soldier, since not many other things salute to a general, or have gleaming buttons. Furthermore, you might have unconsciously realised that it's a noun that fits in the gap (well, it's definitely not something like an adjective), and gotten clues from the length of the missing word by how many stars there are.

How could we get a computer to solve this problem? Well, one way might be to see how many of the words in the sentence are associated with soldier; I bet 'general, 'salute, 'uniform and possibly 'crisply' are all commonly used whenever 'soldier' is used.

To get information about which words commonly occur with which other words, we can simply go through every word in our dictionary and do the following:

1. Use a search engine to find documents about our word.
2. For each document, remove all instances of our search term.
3. Count the number of occurrences of each word, to make a **Vector Space Model**, something like this:

```
{'the': 20, 'or': 17, 'of': 16, 'to': 13, 'in': 9, 'are': 8,
 'soldiers': 7, 'from': 7, 'as': 7, 'a': 7, 'Army': 6, 'military': 5,
 'an': 5, 'word': 5, 'and': 5, 'is': 5, 'In': 5, 'term': 4, 'referred': 4,
 'US': 4, 'their': 4, 'Marine': 4, 'other': 4, 'meaning': 4, 'officer': 3,
 'United': 3, 'called': 3, 'service': 3, 'such': 3, 'The': 3, 'States': 3,
 'years': 2, 'soudeour': 2, 'personnel': 2, 'countries': 2, 'because': 2,
 'often': 2, 'employment': 2, 'Latin': 2, 'combat': 2, 'warfighter': 2,
 'by': 2, 'Infantry': 2, 'warfighters': 2, 'occupations': 2, 'French': 2,
 'one': 2, 'armed': 2, 'use': 2, 'type': 2, 'artillery': 2, 'serve': 2...}
```

This is from an example script I wrote to pull content from the Wikipedia and extract a vector space model. It's on Github if you want a look... Why don't you submit a pull request while your at it!

It's also a good idea to stem the words before continuing. For example, 'soldier' and 'soldiers' are for all intents and purposes, the same word here.

This information still isn't very useful; we would like a way to compare the similarity of two vector spaces (i.e. a candidate word, and a document we're trying to classify).

If you took `COMP24111`, and you're not half asleep from boredom, then you might remember a certain method of classification called *nearest neighbour.* Given a set of n-dimensional vectors and an n-dimensional vector you want to classify, nearest neighbour will find the distances between all the vectors in the set, and your other vector, and try to assign it a class based on the classes of 'near' vectors.

This *isn't* what we're going to do here, but it's pretty similar. For each vector, we can find the cosine of the angle between them:

$$\frac{\sum(d_1 \times \cdots \times d_n)}{\sqrt{\sum(d_0^2) \times \cdots \times \sum(d_n^2)}}$$

The cosine of two vectors is a rough approximation of their similarity. However, there is an issue; a large document (e.g. an explanation of the Church-Turing thesis) will have higher values in its vector space than the text off the back of a chocolate bar, since the former has far more words.

3.1 TF-IDF

We don't want document size to affect our similarity measures much, since you would want a sign posting the way to Manchester to have a high similarity score to a book about Manchester, but not a high similarity score with a leaflet about an environmental protest.

To normalise the vector spaces, we can divide each number in the vector by:

$$\sqrt{\sum_{i=1}^{n} s_i^2}$$

Where (I think, please correct me if I'm wrong), s_i is the value associated with the ith word in the vector space. This reduces the values in the vector space proportional to the size of the vector space.

Another thing that makes vector spaces less effective, is the large amount of noise we get from common words such as 'and', 'the' and 'a'. We want these words to have less weighting when it comes to determining whether documents are similar, and possibly give a higher word to more uncommon words such as 'marine' or 'howitzer'.

For each word in our dictionary, we could see how many documents it is included in; this is the **document frequency**. We could also see how often it occurs in a single document; the **term frequency**. We can use these two values to determine how much we care about each dimension in the vector space, specifically, we can multiply the vector space value by the term frequency and then the inverse of the document frequency:

$$\text{vector space value} = \text{vector space value} \times \text{term frequency} \times \frac{1}{\text{document frequency}}$$

This is called the **TF-IDF** (term frequency - inverse document frequency) score.

3.2 Clustering

Doing the above (document $\rightarrow$ vector space $\rightarrow$ TD-IDF) gets pretty good results. In the course notes, an example is given where five bacon themed websites are correctly marked as being more similar than four cricket themed ones.

The example doesn't get perfect results; it could be improved. To do so, we need to think back to the Machine Learning course again, and remember **K-means clustering**. This is where, given the vector spaces, we try and group documents into n similar clusters. This is what you do:

1. This is where you find the n least similar documents ($n =$ the amount of clusters you want to find), and set them as seeds. The logic is that these are not similar, and therefore won't be in the same clusters.
2. For each document, put it in the cluster that has the seed most similar to it.
3. For each group, find take the weighted average of them all, and make that the new seed. This seed is now called the **centroid**.
4. If some documents switched clusters, then go back to step 2 and repeat.

This gets better results than TF-IDF on its own, but still isn't perfect. This is probably because it only looks at words in isolation. However, as we know from our Part of Speech tagging exploits, the same word can have different meanings depending on the **context** that it appears in. For example, homonyms can mess up our analysis; we'd like to differentiate between 'lead' as a metal as a verb ("he lead her outside") and as a noun ("fetch the dog's lead").

What if we could cluster the vector space of a word. If there are lots of different clusters, then it's probably an ambiguous word...

To do this, we might want to look at the words surrounding a target word, or possibly think about *syntactically related* words; if two verbs take the same set of objects, then they're probably similar (e.g. 'she kissed him' $\approx$ 'she snogged him').

We can extract information from verb phrase-noun phrase interactions automatically using regular expressions. For example, the regex `noun (MV:verb) det? adj? (OBJ:noun)` will match "he kissed the beautiful lady", and assign `MV` to 'kissed' and `OBJ` to 'lady'.

Since regex matching is fast, we can extract these relationships quickly from the input corpus.

To see how similar different verbs are, we can do the above for both verbs, and then intersect what words they are related to. The larger the intersection, the more related the verbs are. The course notes does this for 'read' and 'write'. There are lots of nouns including 'book', 'letter', 'newspaper', 'paper', 'music' etc that are related to both words.

However, while obviously 'read' and 'write' are related, they're (in a sense) antonyms too. See, we are able to cluster words this way, but we can't tell *how* the words are related. Are they synonyms (kiss/snog), antonyms (eat/regurgitate) etc?

Furthermore, the analysis is quite dumb, and is easily confused by sentences that aren't 'simple' or require a context to understand properly. Again, the course notes give a good example; you might devour a book, and an intergalactic monster might devour a world, which presents issues if you're trying to cluster 'devour' with 'eat', which is obviously related.

3.2.1 Choosing cluster points

So, although we've decided that our measures for detecting whether words are similar are rather dumb, we know they do work. So, how do we start using them to cluster similar words together? As mentioned above, to do K-means clustering, we need to both decide on how many seeds to create (which will give us the same number of clusters) and which elements to set as the initial seeds.

We're going to cheat regarding the number of seeds we'll create, since it's hard to do automatically. For choosing the seeds though, there is a nice way of doing things:

1. For 0 to n
   (a) Pick m random seeds
   (b) Calculate their *density*, which is the sum of their pairwise cosines.
2. Choose the set of seeds that had the lowest density.

This works because if the sum of pairwise cosines is low, that means the items were dissimilar, which is likely to give us seeds that properly represent the clusters that they will go on to form.

In practice, the choice of seeds is important, but not vital; it is possible for the initial seed to move out of the cluster it initially created. Obviously better seeds will make the algorithm converge faster and make it less likely that silly clusters will result.

A good idea is to run the algorithm many times with steadily increasing seed sizes, and select the number of seeds that gives the maximum average density in the clusters (except from when the number of clusters is close to the number of words, since then the average density will be 1!).

3.3 Hand Coded Relations

The trouble we're having now is that we know when concepts are related, but not how they're related. For example, we can tell that two sentences are related (e.g. 'I drove the car' and 'I drove a fiat'), but not how they're related; is driving a fiat the opposite of driving a car, is it better than driving a car?

In order to solve this problem, we're going to need to create (or obtain some data that our algorithms can use to help distinguish between words). WordBanks have **Synsets**, which are collections of the different senses of words. This lets us get a list of potential meanings for a word in a sentence, like this (but in a different format obviously):

Run:

- On the run
- Run across a friend
- Run around in the field
- Run for president
- Many more...

Synset relationships (superset and opposite relations) are also given. This is most useful to us, since we can see what items other items are linked to. Obviously the relations aren't always good; sometimes you get ones that seem completely arbitrary, but you can use synset relations so do things like:

I melted the ice $\rightarrow$ The ice is no longer frozen

3.4 Word Sense Disambiguation

These are lots of different strategies for working out what the words in a sentence mean, as we'll see in the following subsections. However, note that they all have different strengths and weaknesses.

Most strategies are coarse (bag of words, string edit distance etc), having a high recall but poor precision.

3.4.1 Page Rank

We can use the **Page Rank** algorithm to decide on what interpretation of each word is correct. Page Rank is (was, I expect that its evolved quite a bit now) the algorithm that powers Google search. The idea is that we can think of synsets just like webpages. Synsets relate one word to a collection of related words, and web pages are related to other web pages via links.

Page rank works by using the *random surfer model*, which is the assumption that a web surfer will click random links on a web page to get to other web pages (each link has a $\frac{1}{n}$ chance of being clicked).

We can use a markov model to represent this; where the probability of the user being on web page x at time $t+1$ is equal to the sum of the probability of them being on a web page with a link to x at time t multiplied by the probability of them clicking the link to x:

$$prob(\text{Web Page}_x^{t+1}) = \sum(prob(\text{Web Page}_i^t) \times prob(S_i \rightarrow S_x))$$

If you construct a weighted graph representing the web pages and their connections, and simulate the time increasing, then eventually, you will find a convergence, so long as the following criteria hold:

- All web pages must have at least one working link (i.e. no inescapable webpages).
- No inescapable loops.

If one of these criteria is not met, then the model will converge to zero. This is an issue, since we can't always guarantee that our input web pages will conform to these rules. See the course notes (p309-314) for an example of the convergence.

We can also supply a damping parameter for some page, which is the chance that a surfer will visit it. Then, the chance of visiting other pages linked by a page is $\frac{1-damping}{n}$.

You can 'personalise' results by reserving some probability at each stage, dividing it by the pages you're biased towards (e.g. web pages of bands you like). This has *the effect of making those pages more important, as well as making the pages linked too by those pages more important too.*

3.4.2 Application of Page Rank to Synsets

As you might have guessed, the reason we looked at Page Rank, is that you can apply it to synsets just like you can between web pages. It works for synset/superset links and between synsets and the synsets of words that appear inside it.

Given a word we want to find more about, we can get its definition, and then get the synsets of the open class words in the definition. For example, if 'run' was defined as 'moving faster than a walk', we could get the synsets for 'move', 'fast' and 'walk' (this is called a *gloss*). If we ran Page Rank on a graph consisting of those words (+ 'run'), then we'd get the most prominent synsets for 'run'. If we personalise the Page Rank by making the synsets that appear in whatever text you're analysing more weighty, then we'll get better results.

Two big advantages of using synsets and page rank, is that they disambiguate all the words in a phrase in one pass, so they're not too slow. Furthermore, since we have relatively good word banks, somebody doesn't have to spend ages coming up with and inputting lots of different senses of different words.

3.4.3 Textual entailment

Though we can now extract simple lexical relations, we want to determine the consequences that arise from actions. Given the statement 'I ironed my clothes yesterday', we should be able to answer the question 'Did I iron my clothes yesterday'. We can use logic to determine these consequence relations.

One way to do this, is to make a system that takes in statements, turns them into logic and adds them into a database. There are examples in the course notes, but the syntax is a bit painful, so I won't include them here. The idea is that as you build up more statements in your database, you will be able to answer more queries using the database. You can use a theorem prover to try and answer questions formatted as logical propositions. There are some problems with this approach though; namely:

- It's hard to go from question to logic, and statement to logic. If you get it wrong, then you might carry your errors forward later.
- It's hard to prove the theorems.
- Collecting the background knowledge is very hard and takes lots of time.

3.4.4 Bag of words model

There are some simple rules for extracting consequences from sentences. For example, if you saw a fighter jet over Manchester, then you also saw a plane. Here are some rules regarding this:

Okay
Sentence S entails S' if every word in S' is in S.
E.g. I ate a banana yesterday $\rightarrow$ I ate a banana.

Better
Sentence S entails S' if every word in S' is subsumed by a word in S.
E.g. I threw a ball $\rightarrow$ I threw an object.

Halfway-decent
Sentence S entails S' if every word in S' is subsumed by a word in S, where the subsumption respects the order of the sentences.
E.g. I threw a ball $\rightarrow$ I threw an object.
Bob said hi to Betty $\neq$ Betty said hi to Bob.

Subsumption, noun: incorporating something under a more general category

We could also determine whether sentences are related by working out what it would take to transform S into S'.

3.4.5 String edit distance and dynamic time warping

Finding the number of operations that transform one string into another is called the *string-edit distance*, and algorithms that do this can be applied to lots of things, from spell correction to DNA sequencing!

We're going to do this by using an algorithm called **Dynamic Time Warping**. This entails:

1. Make a grid of size $n \times m$, where n and m are the lengths of your strings plus 1, and initialise it so that all cells are -1. The cell $(0, 0)$ is set to 0.
2. Make a score for each operation you can perform from *exchange*, *add* and *delete*.

3. Go through the grid cell by cell. For each cell that is adjacent from the cell you're currently in, give the cell a score equal to the current cell score added to the cost of the transformation required to get there.

This is dynamic programming, yay!

This isn't easy to explain in bullet points, so here's an example. Lets go from 'the' to 'that', when exchanging costs 3, and adding/deleting costs 2 (this is because we want an exchange to be better than an addition then a deletion):

	X	T	H	A	T
X	0	2	4	6	8
T	2	0	2	4	6
H	4	2	0	2	4
E	6	4	2	3	5

When explaining how dynamic time warping works, I've shown an example where you time-warp two words. However, we're going to use it for sentences to see how similar they are.

We can change the costs for the exchange/add/delete depending on the situation. For example, if we're finding the distance between sentences, and if one word is a subsumed by another, then maybe they could be exchanged for a score of 0. We could make deletion free for certain unimportant words, take part of speech tags into account (and make exchange cost less if they're the same). Doing this might let our algorithm give '*I ate a juicy salmon*' and '*I ate a fish*' a score of zero, which is good, since they're pretty similar!

You can also make a machine learning algorithm learn what weights to assign to each operation, and maybe supply a function that scores how similar two elements (words, letters part of speech tags etc) are, and incorporates that into the cost given for each step.

3.4.6 Subsumption on parse trees

We can make some logical rules that operate directly on parse trees. These will derive more parse trees based on the ones we have now. A simple example is that if a sentence is of the form:

```
X used to Y
```

Then:

```
X does not Y
```

We can incorporate words and their opposites into this too; if 'Tom Cruise and Nicole Kidman broke up' then 'Tom Cruise and Nicole Kidman are not going out'.

So, we want to match two parse trees, and derive parse trees from other parse trees. *Apparently*, a technique called **continuation programming** is a good way to solve this problem. This involves:

Through I've never tried this, I view this use of exceptions as badly thought out at the least, but sacrilegious, irreverent and heretic at worst. An *Either* type might work better?

- Making every function take an extra argument (usually a function pointer, called the continuation), which is what should be called if everything goes well.
- If the function doesn't work properly, then it returns.
- If the continuation gets called, then once it is finished, then it should throw an exception to bypass the stacked up return calls.

This is a simple (if messy) way of handling backtracking search, since you use the normal call/return stack to keep track of state (which is efficient, since function calls are generally quite efficient). Exceptions are only used when you find the right answer (which is good, since exceptions are usually slow).

If we apply continuation programming to our subsumption methods, then we can do pretty well. However, edge cases, particularly the polarity of certain words/phrases (e.g. didn't, haven't etc) can mess us up and cause the opposite of what is right to be returned.

A solution to this, is to keep the polarity of each node in the parse tree and have a table of words that switch or set the polarity.

3.4.7 Deriving rules from a corpus

We can get rules to do tree transformations from a large enough corpus. If you know that two sentences are the same, then you can derive a rule to transform one into the other. If its generic enough, then you can use the rule in other situations in the future.

News websites are a good way to do this; lots of different website have the same content, just written in a different way. The chance that there will be sentences with identical meaning, but differing words is quite high. To work out what sentences mean the same thing, we can use our TF-IDF scoring system.

This will often get very technical rules, and will not get easy ones such as 'if X is good for Y, then Y should do X'. These rules are pretty simple, and you can probably put them in yourself.

4 Speech

We want to do both *speech synthesis*, and *speech recognition*. Collectively, this is speech processing. Synthesis is not trivial, but its doable, while recognition is *hard*.

Vowels are the sounds made when you blow air over your vocal chords, and let them resonate. Both different resonation frequencies, and the shape of your vocal tract and mouth impact on what chord is produced, which is part of what makes a vowel sound like, a vowel. Consonants involve no vocal chord resonance.

Some sounds are *nasal*, ('mmm' and 'nnn'), and require you to expel air predominantly from your nose.

If a sound requires you to close your mouth, and obstruct the air flow, then it's an *obstruent* sound. When you open your mouth again, the pressure will have built up, the air will come out quickly, and you'll be able to hear the turbulence.

Other sounds are made by nearly closing the mouth, so air is forced through a small gap. This often produces a hissing noise.

Doing all this takes a lot of practice; there are a lot of muscles to move into very precise configurations! Some transitions between sounds are easier than other ones; going from 'nnn' to 'ppp' is much harder than going from 'mmm' to 'ppp', especially in the middle of a word. This affects how we speak.

If you're trying to understand sound from a computer's point of view, all you get is some waveform. You might be able to tell that two recordings of the same sound/phrase are similar, but not derive what was being said just from looking at a diagram of the waveform.

4.1 Analysis of waveforms using FFT

There's more on FFT in my `COMP28512` notes I think.

We can use the Fast Fourier Transform to find out what frequencies make up a waveform (the inverse FFT can also create a wave from the frequencies).

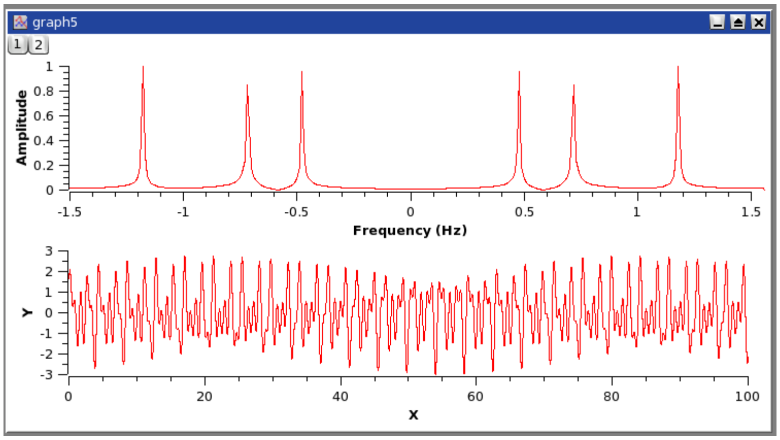


Figure 8: Here is the output from the FFT function (top) and the input wave (bottom). You can see that the frequencies that make up the wave have been extracted. Note that the top graph is a mirror image, this is always the case.

If we plot a graph of the waveform once its been through FFT, then we can start to see distinct differences between recordings of different sounds. In particular, they will have different ratios between the main frequencies (called *formants*), which will allow us to distinguish between different sounds.

It is important to look at the ratios, since then pitch is less relevant.

However, even when we've extracted the frequencies, we need to look at how they change over time. By looking at the rate of change of the frequencies and their intensity (the *deltas*), we can better see how a sound is formed.

4.2 MFCC quantisation

One problem with using deltas to identify sounds is at what resolution do we look at the deltas? If we pick the wrong boundaries or make them too big, then because of the *sampling theorem* (look it up), then we could miss all the important information.

Furthermore, a small change in pitch at low frequencies is more significant than one at high frequencies.

Mel Frequency Cepstral Coefficients is a way to solve this. It splits the frequency range into overlapping segments, so that all possible frequencies will be in two segments. See slides `466-468` for an example of this.

This is helpful, but its ultimately not that useful; besides being messy, we often need the surrounding context to interpret what a sound is (going from sound 'a' to sound 'b' will be different than going from 'c' to 'b').

4.3 Using Hidden Markov Models to recognise sound

A **Markov Model** is a probabilistic model of what's going to happen given the current state of affairs now.

So according to Figure 9, if you see me warming up in the gym, then there's a $0.55 \times 0.6 = 0.33$ chance that I'll do tricep extensions. You could also work out how likely a chain set of events is, simply by multiplying the probabilities of the events in the tree.

This is a hidden markov model now, as opposed to a markov model.

Suppose you're a Jedi; you can use the force to sense weights moving around, but you lost your eyesight in an epic battle years earlier. You want to work out what I'm doing in the gym from how the weights I'm pushing affect the force. You can come up with something like this table:

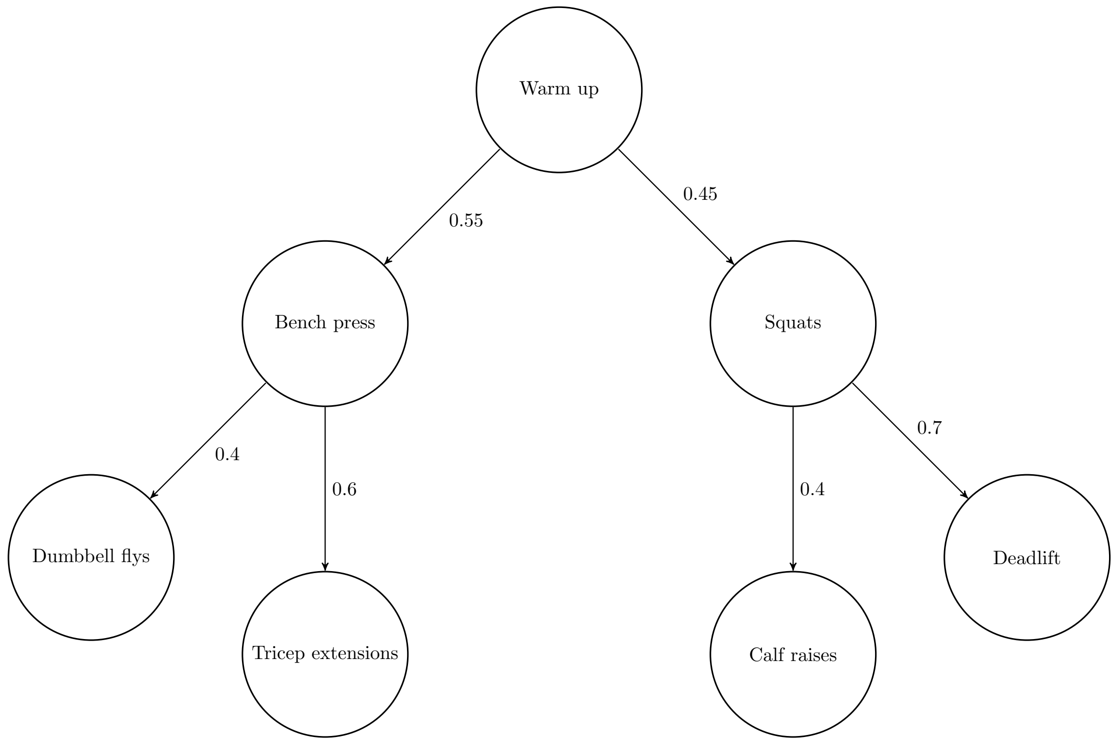


Figure 9: A Markov Model of what I'm likely to do when I go to the gym.

Activity	Probability if X weights sensed		
	Light	Medium	Heavy
Warm up	0.9	0.1	0.0
Bench press	0.2	0.5	0.3
Dumbbell flys	0.8	0.15	0.05
Tricep extensions	0.7	0.25	0.05
Squats	0.2	0.4	0.4
Calf raises	0.2	0.5	0.3
Deadlift	0.1	0.3	0.6

So even now you can't actually see what's going on, you might be able to get a pretty good idea. If you made a series of observations about what weights are moving, you could make an educated guess about where I am in the markov model!

Suppose you sense that I'm pushing light weights, then medium weights, and finally you sense that I'm pushing heavy weights. With this sequence of weights, its fairly likely that I warmed up, then did squats and then went on to deadlift. It's far more likely that I did that than if I did warming up, then bench press, then tricep extensions!

4.3.1 The Viterbi Algorithm

The Viterbi Algorithm tries to find the most likely route through a HMM given whatever events you think you have. You annotate the HMM with **emission probabilities**; 'I'm now at state X, how likely is it that I'm going to be moving to state Y given that Z happened' and **transition probabilities**; 'How likely is it that I'm going to stay in this state, or move to state X?'

Lets continue with the Jedi-gym example (this doesn't have self-loops like most HMM's do, but it doesn't matter too much). Given that I'm in the warm up state, and you're sensing me pushing medium weights, what's the probability of me moving to the other states?

$$
\begin{aligned}
\mathbb{P}(\text{Bench press}) &= \mathbb{P}(\text{Bench press}|\text{medium}) \times \mathbb{P}(\text{Warm up}) \times \mathbb{P}(\text{Warm up} \rightarrow \text{Bench press}) \\
&= 0.5 \times 1 \times 0.55 \\
&= 0.275
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{P}(\text{Squats}) &= \mathbb{P}(\text{Squats}|\text{medium}) \times \mathbb{P}(\text{Warm up}) \times \mathbb{P}(\text{Warm up} \rightarrow \text{Squats}) \\
&= 0.4 \times 1 \times 0.45 \\
&= 0.18\ldots
\end{aligned}
$$

In general, the probability of being in State X is:

$$\mathbb{P}(\text{X}) = \mathbb{P}(\text{X}|\text{Observations}) \times \mathbb{P}(\text{Current state}) \times \mathbb{P}(\text{Current state} \rightarrow \text{next state})$$

Note that (not being a Jedi), I picked some very arbitrary probabilities for the observations. In real life, you want better ones; maybe if you're doing part of speech tagging, you could use frequency counts or something to determine them automatically.

So, back to the Viterbi Algorithm; you just compute the probability of being at each node all along the HMM, and then pick the most likely path along it.

Unfortunately for us, this isn't that applicable to transcribing speech just yet; having a HMM where the nodes are equivalent to words doesn't work very well. Instead, phonemes or parts of phonemes are more commonly used (though words might be used occasionally). Going from words to phonemes is fairly easy; you can just use a phonetic dictionary like the British English Pronouncing Dictionary (BEEP).

4.3.2 Features of sounds

Before, we said that in order to construct a HMM, we need to work out transition probabilities and emission probabilities for each sound.

Maybe consult the `COMP28512` notes if you're struggling to understand some terminology here. Unfortunately, I don't have time to provide a more detailed explanation.

When we're sampling the sound, it's a good idea to quantise (put it into bins) into around 12 values. In terms of the sampling frequency, $44kHz$ is a good value to work with; it encapsulates everything that the human ear can hear, so going higher is pointless.

We can split each phoneme into three parts; a start, middle and end. Since phonemes are said in a sequence, the start and end parts tend to be fairly distorted by the previous/next phonemes, but the middle part is usually quite characteristic of what sound is intended to be said.

Since phonemes are of a variable length (people speak at different speeds, change the tempo of speech to give emphasis, etc), we need to have self-loops in the HMM to account for this, so a phoneme in the HMM can transition to itself if the sound is long.

What happens in reality, is that a HMM is created for each different phoneme, and when sound is being recognised, they are concatenated together for form the expected phrases (e.g. 'c,a,t' or 'c,a,r').

In actual fact, we use an algorithm called Baum-Welch re-estimation to work out the HMM transition and emission probabilities (which we thankfully don't need to know for this course).

4.3.3 Using a grammar for recognition

So, we've seen that we can make a HMM for each phoneme, and then compose HMM's together to make more HMM's that we can use to recognise speech. The trouble is, if we knew what phonemes to put together, then we would already know what was being said (since we would know what phonemes to use).

A solution is to write a grammar to define what we might be said. This does a great job of limiting the complexity of the recognition (since the number of possible answers is constrained by what is allowable in the grammar). We then build a compound HMM from the grammar.

For example, the grammar:

```
DET = a | the ;
NOUN = cat | dog ;
VERB = sleeps | runs ;
NP = DET NOUN ;
SENTENCE = NP VERB;
```

Generates a network like:

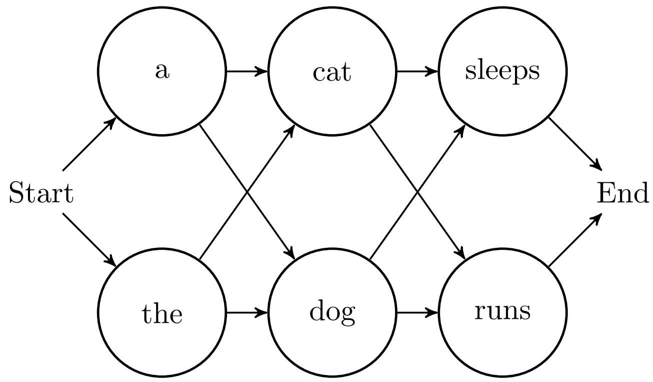


Figure 10: A hidden markov model generated by a grammar.

Each of the words is actually split up into phonemes, based of the phonetic descriptions of the word.

4.4 Synthesising sound

Human speech is made up of chords (vowels) and hissing (consonants). We can generate waves of specific frequencies (chords) and white noise pretty easily, so if we can work out what noises need to be produced for different phonemes, then we're good. However, these '*formant synthesizers*' don't work well in practice.

There are lots of different ways of synthesising that work better:

Whole word recording
: We could record lots of words, and then say them one after the other. This is certainly listenable, but you can really hear the boundaries between words. Furthermore, to make a useful grammar, you might need to record tens of thousands of words, and many different forms of the same words! Ohh, and remember that all the words need to be of the same pitch, speed, intensity etc. This is used a lot in railways and airports etc where the vocab is limited.

Phoneme recording
: Instead of recording whole words, we could just record phonemes. Since there are only a few tens of phonemes in a language (but they make up all words), this will be a lot easier. However, the pronunciation of phonemes depends on what phonemes they're next to, and so it's going to sound pretty bad when our recordings don't take that into account. Furthermore, it's actually impossible to say some phonemes in isolation, so you can't even record all the sounds!

Diphome recording
: A solution between the two ones above, is to record *diphones*, which are pairs of phonemes. This is far more comprehensible than listening to concatenated phonemes, and isn't too

far off full words, while only requiring a few thousand recordings. However, the result is monotone since words have stress patterns that aren't replicated. However, if we make separate recordings with different pitch and duration, then we can simulate the pitch rising at the end of questions, falling at the end of statements etc.

In fact, raising and lowering the pitch can be done automatically if you change the speed of the recording. You simply delete sound frames, or add new ones in (composed of the moving average of frames) according to how much you want to adjust the recording. If we don't want the duration to change when we chance the pitch, then we should delete some chunks, or insert copies of chunks to make it longer.

Making the join between two recordings that you've stitched together is hard, but luckily there's an algorithm to do it (the 'PSOLA' algorithm).

5 Machine Translation

We want to make a computer translate a document from one language into another. More precisely, we want to construct a new document from the old one that expresses the same message, but in a different language. We could generate a datastructure that represents the meaning of the source message and then use it to generate the document in the target language.

Unfortunately, I don't think I can explain this section particularly well, so you're going to have to refer to the course notes for this!