Natural Language Systems: COMP34412

Allan Ramsay
School of Computer Science,
University of Manchester,
Manchester M13 9PL, UK

Contents

How does language work? 11
Architecture of a NL system 21
COMP24412 23
COMP34412 24
Course outline (draft) 25
What makes different sounds different? 33
Chords 35
Sounds & Fourier Transforms

Fourier analysis

Speech recognition

Spectrogram (Fourier analysis): AAA vs OOO

Formants

Spectrogram (Fourier analysis): CAT vs PAT

Spectrogram (quantised): CAT vs PAT

Spectrogram (rate of change): CAT vs PAT

Mel Frequency Cepstral Coefficients

CHECKPOINT

Hidden Markov Models

Viterbi algorithm

Features, networks, network composition

What kinds of observations are we going to use?

CHECKPOINT
Measuring accuracy 283
HMM-based tagging 286
Transformation-based learning (TBL) 304
Templates (see tbl.py for the full set) 306
Special cases 313
CHECKPOINT 318
Syntax & Parsing 319
Regular expressions as NL grammars 320
Regular expressions 324
Checkpoint 335
**Deterministic dependency parsing** *(Nivre et al. 2007; Nivre 2003)* 336
Learning from a treebank 358
Headed phrase structure trees ≡ dependency trees 359
Learning a set of parsing rules 367
Long-distance dependency revisited 380
Maximum spanning trees

‘Edmond’s algorithm’

CHECKPOINT

What can you do with it?

Better models

CHECKPOINT

Hand-coded lexical relations

WordNet

WordNet for pairwise entailment

WordNet for similarity measurements

CHECKPOINT

Textual entailment

Montague semantics

Bag of words

String-edit distance
Recommended reading:

These notes, which you can get to at http://syllabus.cs.manchester.ac.uk/ugt/2017/COMP34412/COMP34412.pdf:

I set my exams by trawling through my own notes, so everything I want you to know will be in here. You may find sets of notes in other places: **don’t use them**. In particular, don’t use last year’s notes, because the course was entirely different last year. And don’t read too far ahead, because they will evolve as we go.

‘Natural Language Processing with Python’, Bird, Klein & Loper: hard copy published by O’Reilly, but also available online with supporting software at http://www.nltk.org/. I will be using some of their software

‘SPEECH and LANGUAGE PROCESSING: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition’, Jurafsky & Martin: for anything that I don’t explain properly. Not actually the best book for semantics, but apart from that . . .

Bits of code live in /opt/info/courses/COMP34412/PROGRAMS.

Office hour: mail me, catch me after a lecture.

Allan Ramsay, I INTRO
I INTRO
How do programs that let you talk to your computer work?

How do programs that let you ask your computer questions whose answers are provided somewhere on the web work?

How do programs that translate documents that are written in some foreign language work?
To do any of these things, you have to know how language works.

And you have to be able to express your knowledge as a program.

But language is very complicated.
There are symbols that denote ‘ideas’.

A symbol can be any detectable object. Pictures, shapes, sounds, gestures.

Ideas can be fairly simple (‘green’ = ‘reflects light with a wavelength between 495 and 570 nm’) or complicated (‘I only borrowed your bike’ = ‘I did borrow your bike, but I didn’t do the more objectionable thing that you thought I’d done to it’).
But unconnected isolated ideas don’t do you much good (animal signalling systems). Language lets you arrange these ideas in ways that connect them together:

(1) a. John loves Mary.
   b. Mary loves John.
Speech is linear: one word followed by another

(2) I know he said he wanted it
Text doesn’t have to be. Mouse’s tale. Layout, HTML, XML

But speech is primary, and text is largely linear (some NLP systems pay a bit of attention to markup, but I’m not going to in this course)
Fury said to a mouse, That he met in the house,
Let us both go to law: I will prosecute YOU. --Come,
I'll take no denial; We must have a trial: For really this morning I've nothing to do."
Said the mouse to the cur, "Such a trial, dear Sir,
With no jury or judge, would be wasting our breath."
"I'll be judge, I'll be jury," Said cunning old Fury: "I'll try the whole cause, and condemn you to death."
Connections between ideas are hierarchical. One word is more important than another. Sentences describe events (states and actions): the most important word tells you what kind of event is being described, the other elements tell you what entities were involved and provide additional information about the event.
(3) a. Sometimes I feel like a nice cup of tea
   b. Sometimes I feel like a motherless child

(4) a. I wish the rivers was whisky and I was a diving duck
   b. I wished the rain would stop and I was pleased when it did
So the rules that relate the way the words are organised to the ideas they express have to turn a sequence into a tree.

(phrase structure tree, dependency tree, ???)
But I don’t want a parse tree

I want a set of ‘related ideas’

And I don’t expect my hearer to just assemble a set of related ideas

I expect them to enrich the bare picture that I have conveyed, and I expect them to reason about what I want

(5) Noone has done the washing-up \vdash \text{The washing-up hasn’t been done.}

(6) I went to a restaurant last night. The waiter was wearing a funny hat.

(7) I really need a cup of tea.
So what happens in language is that arrangements of words encode relations between ideas, which the participants think about.

And what happens in a computer is that operations are applied to datastructures!
Architecture of a NL system

input utterance

SPEECH RECOGNISER
input text

PARSER
structural analysis

DECODER
canonical form

INFERENCE ENGINE
anchored form
discourse model (minutes)

INFERENCE ENGINE
updated minutes

GENERATOR
output text

updated minutes

SPEECH SYNTHESISER
output utterance

background knowledge
There’s a lot to be done here, and we don’t know how to do it all, let alone how to stick it all together.

The machinery that you’d like & the theories behind that machinery (COMP24412): doing linguistics by writing computer programs

The compromises you have to make, given the best current technology (COMP34412)
Allan Ramsay, I INTRO -23- How does language work?
Introduction: applications, architectures, limitations.

**Speech recognition & generation (3 lectures):**
Characteristics of speech; the vocal tract; representing speech signals; formant vs diphone based synthesis
Recognition: acoustic features, language models, underspecification
Synthesis: concatenative models, HMM models

Coursework 1: speech synthesis & recognition exercise
Structural analysis (3 lectures)

- Words: lexical lookup, structure of words (morphology vs stemming), part of speech tagging (hidden Markov models, transformation-based learning)

- Grammar: parse trees vs dependency trees, parsing vs chunking, classical parsing, robust algorithms

Coursework 2: parsing exercise
Information retrieval & extraction (2 lectures):
Word senses and word sense disambiguation, slot-and-filler semantics, lexical relations, textual entailment,

Machine translation (2 lectures):

• Transfer-based MT: the transfer pyramid, transfer rules, interlingua.

• Statistical MT:
III Speech
A human being is like a very complicated woodwind instrument.

<table>
<thead>
<tr>
<th>woodwind instrument</th>
<th>human being</th>
</tr>
</thead>
<tbody>
<tr>
<td>blow air over</td>
<td>blow air over</td>
</tr>
<tr>
<td>vibrating reed</td>
<td>vibrating strings</td>
</tr>
<tr>
<td>column of air of variable</td>
<td>column of air of</td>
</tr>
<tr>
<td>length</td>
<td>variable length</td>
</tr>
</tbody>
</table>

Figure 1: The human vocal system is a woodwind instrument.
Figure 2: Crumhorn (very simple woodwind instrument) (home.earthlink.net/ cornetto45/art/crum2.htm)
Figure 3: Vocal tract (from www.indiana.edu/hlw/PhonUnits/vowels.html)
If you change the shape of the vocal tract, you’ll change the nature of the sound that it produces. Vocal cords for resonance, vocal tract for harmonics, various ways of closing it.

Different combinations make perceptually different sounds.

If you could recognise the differences between sounds then I could make a sound and you could work out which one it was.

As far as speech processing is concerned, the hardest part is speech recognition. Speech synthesis isn’t trivial, but speech recognition is **hard**.
‘Vowels’: some sounds are made by blowing air over the vocal cords and letting it resonate in the vocal tract. Because of the shape of the vocal tract, you tend to get several resonances. The mixture of these is perceptually different.

It’s like when you play a chord on a guitar. A minor chord sounds sadder, or sweeter, or something than the corresponding major chord. Some chords are just horrible. You get a different experience from different relations between mixtures of notes.
The sound that comes out of your mouth when you produce a vowel is like a chord produced by a group of wind instruments. Somehow you can experience the relationship between them (which may or may not involve separating them into individual notes, in the same way that when you add blue to red you see yellow) (not purple: that’s when you remove the red and blue from white)
You don’t hear three separate notes: you hear a wash of sounds, but you can hear a general ‘quality’ – that the minor chord is soft, or sweet, or sad, or something like that.

\(^1\) (pictures produced using Praat: Google it, stick it on your machine)
How do I change the shape of the vocal tract to get different chords?

I can move my tongue around: try saying ‘aaaaaa’ and ‘eeeeeeeee’ with your finger in your mouth.

I can change the shape of my lips: look in the mirror while saying ‘ooooooo’ and ‘eeee’
‘**Nasals**’: vowels are made by blowing air over your vocal cords while they are tense and letting it resonate in your oral cavity before it comes out through your lips.

Other sounds (/m/, /n/) also involve blowing air over your vocal cords while they are tense: but when you produce /m/, the exit from your vocal tract is closed by putting your lips together; and when you produce /n/ it is closed by putting your tongue behind your upper teeth.
So how does the air escape, and where does it resonate?

There’s a route from your airways into a space behind your nose (the nasal cavity). It’s usually closed when you’re talking (but presumably it’s open when you breathe in). If you open it while blowing air over your tensed vocal cords, the air can resonate in there (try saying ‘mmmmmmmm’ while holding your nose).
Some sounds (‘obstruents’) are made by closing the exit from the vocal tract. Or aren’t: if you close it, no sound can escape, so you don’t make sounds by closing it.

But when you close it, pressure continues to build up inside. And then when you do re-open it, the air comes rushing out.

As it comes out it passes through quite a small space, so there’s lots of turbulence, which you can hear.

How fast it comes out, what shape the space it comes through is, what sounds precede or follow it affect what it sounds like.
Some sounds (‘sibilants’, ‘liquids’, ‘trills’) are made by nearly closing the exit from the vocal tract.

Air does continue to come out, but again it passes through quite a small space, so there’s lots of turbulence, which you can hear.

What shape the space it comes through is, what sounds precede or follow it affect what it sounds like.
So that’s what you do when you speak. You push air through your vocal tract: as you do so, you tense or relax your vocal cords, which makes it vibrate or not; you change the shape of your vocal tract, which changes the mixture of harmonics; and from time to time you close or nearly close the exit from the vocal tract, using combinations of the route through to the nasal cavity, your tongue, the roof of your mouth, your teeth and your lips.
Very clever. Requires a great deal of muscular coordination. Not surprising that babies take a long time to learn to do it.

Some transitions are easier than others: easy to get from /m/ to /p/ (keep your lips closed, close your velar flap, relax your vocal cords, open your mouth). Hard to get from /n/ to /p/ (move your tongue from behind your teeth, close your lips, close your velar flap, relax your vocal cords, open your mouth). Which is why English speakers say ‘impossible’ rather than ‘inpossible’. Remember that for later.
Sounds & Fourier Transforms

Sounds are made out of vibrations. Two different sounds will be made out of different sets of vibrations, and it is hard to directly compare them.

‘aaaaa’, ‘aaaaaa’ and ‘oooo’

You can see that the first two are similar and the last one is different. How can we compute this?
Fourier analysis

Start by thinking about steady repetitive sounds.

The easiest way to make a steady note is playing a sine wave. You can add harmonics (waves whose frequency is an integer multiple of the original) to get different versions of the same sound.

```python
>>> s1 = square1(g=11, f=10); s1.play(); s1.plot(save=True)  # g=11 means
```

‘Square’ wave with 1, 2 and 5 parameters
Not too surprising that if you add harmonics to a sine wave you’ll get something repetitive: the first harmonic fits inside the original, so it has the same effect on each of the main cycles; but then so does the second harmonic, and then the third, and . . .

More unexpected is that any repetitive signal can be decomposed into a collection of harmonics. Even better, there is a way of doing this fast.
The ‘fast Fourier transform’ (‘FFT’) computes the harmonics that make up the signal: works beautifully if you take a window which is indeed an exact set of cycles.

```python
>>> s = combination([(1, 20), (2,12), (3, 6), (15, 2)], f=5)
>>> s.plot(N=3*441, show=True, save=True)
>>> histogram(lastnonzero(abs(pylab.rfft(s.signal[:441])),t=10), show=True, save="hist-%s.eps"%(s.name))
```

![Figure 4: FFT for a simple signal](image)
Even better, the inverse function exactly recreates the original signal.

```
>>> l = [(1, 20), (2,12), (3, 6), (15, 2)]
>>> thereAndBackAgain(l, f=3)
```

Figure 5: From signals to Fourier transforms and back again
Similar signals will have similar FTs

```python
>>> l0 = [(1, 20), (2, 12), (3, 6), (15, 2)]
>>> l1 = [(1.1, 20), (2.3, 10), (3, 7), (16.5, 2)]
>>> l2 = [(1.1, 20), (5, 6), (16, 2)]
>>> x = multisignals([l0, l1, l2])
```

![Fourier transforms of similar signals](image)

Figure 6: Fourier transforms of similar signals
Speech recognition

Start by revisiting ‘ooo’ and ‘aaa’

‘ooo’ looks quite like a nice simple sine wave. ‘aaa’ looks like it might have harmonics added to it.
Spectrogram (Fourier analysis): AAA vs OOO

Strong dark bands show you how the energy is distributed at different frequencies.
Formants

The main bands are strong enough for us to plot them. The ratio between them tells us a lot about the nature of the sound.

The ratios between the frequencies of the main three such ‘formants’ is enough to distinguish between different kinds of vowels. What about other sounds?
‘cat’ starts sharply, then stays steady. Maybe you can see the formants. ‘pat’ ramps up from silence to full intensity, gets louder than ‘cat’, no formants visible.
You could divide the FT up into chunks: everything from 0 to 50, everything from 50 to 100, ...

That would let you cope with the fact that unless you cut the signal at exactly the right place then you get some noise around the peaks.

And it would eliminate the differences between signals that weren’t really very different.
'cat' has long bands at about mid-frequency, 'pat' has narrow quite intense columns near the start.
But sounds don’t occur ‘at an instant’. The effect of closing the airway is to change the sound, and the differences between different consonants arise largely because they change the sound in different ways.

So measure the parameters the frame by frame and record them and their rate of change and the rate of change of their rate of change (speed and acceleration: also known as ‘deltas’).
Spectrogram (rate of change): CAT vs PAT

‘cat’ has shorter period when the intensity is changing?
What we’ve just seen suggests the kind of thing we should be looking for. It also suggests that it might be hard to spot, so people generally do a round of preprocessing steps\(^2\), some of which are based on observations about human hearing.
We’re sort of interested in the signal itself and sort of interested in the way in which it’s changing.

If you took a sequence of points $p_1, p_2, p_3, \ldots$ then $p_1 - p_2, p_2 - p_3, p_3 - p_4, \ldots$ would be a reasonable approximation to the rate of change of the original sequence.
But we’re sort of interested in the signal and sort of interested in the way it’s changing. So take $p_1 + 0.95 \times (p_1 - p_2), p_2 + 0.95 \times (p_2 - p_3), \ldots$. Why 0.95? Because it works.

(not sure why the scaling is different)
To do the next bit we have to split the signal into ‘frames’ – chunks that are long enough that you can actually calculate the FFT but short enough that the sound doesn’t change very much during them. Signals are sampled at quite high rates (44.1K observations per second), frames are more like 100 per second. I’ve done these diagrams at 40 frames/second, so a frame has 1102 observations.

Frames are set to overlap – the first frame covers the observations 0-1102, the second covers observations 400-1502, the third covers 800-1902, . . . If they don’t overlap then you’ll get discontinuities in your observations, if they overlap too much you’ll have too much data and everything will be very slow
We actually take the square of the FFT. Why?

The crucial bit of the ear is like a spring. The energy required to stretch a spring is proportional to the square of the amount you want to stretch it by. So taking the square of the FFT measures the energy in the signal at each frequency.
The human ear actually compensates for this: if one signal contains $P$ times as much energy as another then you will hear it as being $\log(P)$ as loud.

So we now take the log of what we just had.
Next bit is also based on human perception (entirely reasonable: speech perception is done by the human ear – it’s not that the human ear is fabulously tuned to doing the ‘right’ thing with sounds, but that speech is tuned to what the human ear is good at)

People are more sensitive to differences in pitch at low frequencies than at high ones: you can easily hear the difference between a signal at 100Hz and one at 105Hz, much less easy to hear the difference between 500Hz and 505Hz.

What you’re sensitive to is the ratios between the pitches.
Frequency chunks: split the range of frequencies into overlapping portions, fix it so that points in the spectrogram are allocated into the two chunks where they appear.

The area under the downslope of segment \( i \) is the same as the area under the upslope of segment \( i + 1 \). A bit of signal near the centre of segment \( i \) will be assigned mainly to block \( i \), a bit half way between segment \( i \) and segment \( i + 1 \) will be equally allocated to the two segments.
Frequency chunks: split the range of frequencies into overlapping portions, fix it so that points in the spectrogram are allocated into the two chunks where they appear.

The area under the downslope of segment $i$ is the same as the area under the upslope of segment $i + 1$. A bit of signal near the centre of segment $i$ will be assigned mainly to block $i$, a bit half way between segment $i$ and segment $i + 1$ will be equally allocated to the two segments.
Frequency chunks: split the range of frequencies into overlapping portions, fix it so that points in the spectrogram are allocated into the two chunks where they appear.

The area under the downslope of segment $i$ is the same as the area under the upslope of segment $i + 1$. A bit of signal near the centre of segment $i$ will be assigned mainly to block $i$, a bit half way between segment $i$ and segment $i + 1$ will be equally allocated to the two segments.

Figure 9: MFCC quantisation (drawn by MFCC.MFCC(9, 10))
So we now allocate the fairly fine-grained $\log(\text{power(spectrogram)})$ to these bins: call these Mel frequency coefficients.

Different sounds produce different pictures: perceptually similar sounds produce similar pictures?
Next step is **not** based on any observations about human hearing. Fourier transform: any cyclic behaviour can be **exactly** reproduced by a sum of sine waves of harmonics of the fundamental frequency. Cosine transform: any function can be **approximated** by a sum of sine waves. The low frequency elements of this give you the general shape, the high frequency elements give you the local variation\(^3\).

Do the cosine transform to Fig. 10, take the low frequency parts: these are the ‘Mel frequency cepstral coefficients’ (MFCCs)

Figure 11: MFCCs
We now have a set of 12 numbers such that similar sounds have similar sets of MFCCs, so they can be used as features for machine learning: given a set of MFCCs, tell me what sound they represent.  

One final complication: we are interested not just in the values but also in the way they are changing and maybe also their acceleration. So calculate these deltas, but be wary – if you have too many features then your learning algorithm will overtrain.

\[\text{there is actually a 13th, but we've had quite enough detail already}\]
We still have three problems

- It’s still just very messy. The information that we can extract from a signal is just not clear (recordings do contain all the relevant information, since humans can interpret them. But getting it out is hard).

- Sounds are quite substantially changed by the surrounding context. Your vocal tract goes through different stages getting from ‘t’ to ‘a’ and getting from ‘r’ to ‘a’, and the parameters during the transition will be different.

- Sounds can last different amounts of time. Deciding where one ends and the next one starts is hard.
• What you can do to change the sounds that you produce

• Characteristics of different sounds: formants for vowels, shape of the intensity envelope & pitch profile for consonants

• Fourier transform to get energy distributed over frequencies

• Representation as a vector of parameters

• Using MEL frequency bins for quantisation (MFCs): not the final bit where we get the approximation (MFCCs).
Most people use ‘Hidden Markov Models’ for the next bit.

Markov model: probabilistic model of what will happen next, given where you are now.
Figure 12: Things I might do in the kitchen

- get milk (0.6)
- boil kettle (0.9)
- make tea (0.4)
- make coffee (0.6)
- pour in cat’s bowl (0.1)
- eat cake (0.8)
- get cake (0.4)
- get milk (0.2)
- boil kettle (1.0)
- make tea (0.8)
- eat cake (1.0)
- make coffee (0.2)
- eat cake (1.0)
If you see me go into the kitchen and get the milk out of the fridge, you know that there’s a $0.9 \times 0.4 = 0.36$ chance that I will make myself a cup of tea.

If you know I’m in the kitchen, you know there’s a $(0.6 \times 0.9 \times 0.4) + (0.4 \times 0.2 \times 1.0 \times 0.8) = 0.42$ chance that I will make myself a cup of tea.

If you’ve got a Markov model, you can work out how likely a given sequence of events is, and you can work out what is the most likely sequence of events.
But suppose you are in the kitchen with me but you’re blindfolded.

You can hear me moving around, but you’re not sure what I’m doing. You do, however, know what I’m likely to be doing if it sounds like I’m getting milk out of the fridge, and indeed you know how likely I am to be doing it.
P(get cake | sounds like he’s getting cake) = 0.8
P(get milk | sounds like he’s getting cake) = 0.2
P(get cake | sounds like he’s getting milk) = 0.8
P(get milk | sounds like he’s getting milk) = 0.1
P(boil kettle | sounds like he’s getting milk) = 0.1
P(boil kettle | sounds like he’s boiling the kettle) = 0.8
P(put milk in cat’s bowl | sounds like he’s boiling the kettle) = 0.2
P(boil kettle | sounds like he’s putting milk in the cat’s bowl) = 0.1
P(put milk in cat’s bowl | sounds like he’s putting milk in the cat’s bowl) = 0.9
P(make tea | sounds like he’s making tea) = 0.5
P(make coffee | sounds like he’s making tea) = 0.5
P(make tea | sounds like he’s making coffee) = 0.5
P(make coffee | sounds like he’s making coffee) = 0.5
My actual actions are hidden. What you’ve got is observations and probabilities linking those observations to the underlying model, so you can still make sensible guesses.

If you make a series of observations you may be able to do better than if you just make one.
Suppose you think you hear me get cake. If that’s all you’ve got to go on, then your best assumption is that I got cake.

But if you think you hear me get cake, and then boil the kettle, and then make tea, that might make you change your mind. That’s what hidden Markov models are about.
Find the best route through the network: you’ve got ‘emission probabilities’ (how likely I am to be opening the fridge given that there was a faint click), and ‘transition probabilities’ (how likely is it that my next step after opening the fridge will be to close it again).

You want to find the most likely route through the network.
Make an observation: how likely is it that just given the observation I am in each of the possible states (how likely given that the speech signal looks like $S_i$) are my articulators to be in each of the states $a_1, a_2, \ldots$, i.e. given the sound in Fig 13, how likely is it that my tongue is raised close to the roof of my mouth and my lips are rounded, i.e. I’m in the state of saying /e/?

Figure 13: Speech signal: what was I doing?

(not very: that was the signal for /a/).
If the last state I was in was /x/, how likely is it that I'm now currently in state /e/?

I must have been in one of the states $S_{i-1}^{i-1}$; and I must have gone down the transition to one of the states $S_i^j$. What’s the most likely thing to have happened?

For each of the current states $S_i^j$, the likelihood that I got to it from $S_{i-1}^{i-1}$ is

$$p(S_i^j | O_i) \times P(S_{i-1}^{i-1}) \times P(S_{i-1}^{i-1} \rightarrow S_i^j)$$
Example with speech.

Not realistic: assumes that each state is a single word, and that we can divide the speech signal into individual words and hence derive emission probabilities linking segments of the signal to words/states (do Praat example with ‘the cat sat on the mat’)

But not unrealistic, and realistic is too hard to cram into a slide.
Initial state

a (0.00) → cat (0.00) → runs (0.00)
the (0.00) → dog (0.00) → sleeps (0.00) → walks (0.00)

0.85 0.05
0.27 0.66
0.15 0.58
0.73 0.14
0.20 0.37
0.60 0.66
0.40 0.08
0.21 0.34
0.79 0.57

Allan Ramsay, III Speech -84- Viterbi algorithm
Normalise after filling in this level

Allan Ramsay, III Speech

Viterbi algorithm
Transition from 'a' to 'cat': $p=0.27$ (0.40×0.85×0.79)
Transition from 'a' to 'dog': $p=0.01 \ (0.40 \times 0.15 \times 0.21)$
Transition from 'the' to 'cat': \( p = 0.13 \times 0.60 \times 0.27 \times 0.79 \)

Diagram:

- **a (0.40)** → **cat (0.27)** with a transition probability of 0.85
- **the (0.60)** → **cat (0.27)** with a transition probability of 0.27
- **the (0.60)** → **dog (0.01)** with a transition probability of 0.15
- **cat (0.27)** → **runs (0.00)** with a transition probability of 0.05
- **cat (0.27)** → **sleeps (0.00)** with a transition probability of 0.58
- **cat (0.27)** → **walks (0.00)** with a transition probability of 0.14
- **cat (0.27)** → **dog (0.01)** with a transition probability of 0.37
- **dog (0.01)** → **walks (0.00)** with a transition probability of 0.20
- **dog (0.01)** → **sleeps (0.00)** with a transition probability of 0.34
- **dog (0.01)** → **runs (0.00)** with a transition probability of 0.34
- **dog (0.01)** → **the (0.60)** with a transition probability of 0.08
- **dog (0.01)** → **a (0.40)** with a transition probability of 0.57
Transition from 'the' to 'dog': $p = 0.09 \times 0.60 \times 0.73 \times 0.21$
Normalise after filling in this level

```
Allan Ramsay, III Speech  -90-  Viterbi algorithm
```
Transition from 'cat' to 'runs': $p=0.02 \ (0.74 \times 0.05 \times 0.57)$
Transition from 'cat' to 'sleeps': $p=0.04 \ (0.74 \times 0.58 \times 0.08)$

```
<table>
<thead>
<tr>
<th></th>
<th>a (0.40)</th>
<th>the (0.60)</th>
<th>cat (0.74)</th>
<th>dog (0.26)</th>
<th>runs (0.02)</th>
<th>sleeps (0.04)</th>
<th>walks (0.00)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.85</td>
<td>0.27 0.15</td>
<td>0.73</td>
<td>0.02</td>
<td>0.05</td>
<td>0.66</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>0.21 0.79</td>
<td>0.37 0.20</td>
<td>0.34 0.08</td>
<td>0.57</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Allan Ramsay, III Speech

Viterbi algorithm
Transition from 'cat' to 'walks': $p = 0.09$ ($0.74 \times 0.37 \times 0.34$)

a (0.40) \quad \rightarrow \quad 0.85 \rightarrow \quad \text{cat} (0.74) \quad \rightarrow \quad 0.05 \rightarrow \quad \text{runs} (0.02)

\text{the} (0.60) \quad \rightarrow \quad 0.73 \rightarrow \quad \text{dog} (0.26) \quad \rightarrow \quad 0.14 \rightarrow \quad \text{sleeps} (0.04)

\rightarrow \quad 0.05 \rightarrow \quad \text{walks} (0.09)

Allan Ramsay, III Speech
Transition from 'dog' to 'runs': $p=0.10 \ (0.26 \times 0.66 \times 0.57)$
Transition from 'dog' to 'sleeps': $p=0.00 \ (0.26 \times 0.14 \times 0.08)$
Transition from 'dog' to 'walks': \( p = 0.02 \) (0.26×0.20×0.34)

Allan Ramsay, III Speech

Viterbi algorithm
Normalise after filling in this level

Viterbi algorithm
The programs for running these simulations are in the course repository. They’re not hugely efficient, and they won’t cope with cases where there are loops (which we will need). But they do draw nice pictures.

There’s a very elegant, and very efficient, Python implementation at https://github.com/phvu/misc/tree/master/viterbi. Elegant ≈ short ≈ hard to follow. But if you need one, it’s worth a look.
What are states?

- Words?

- Phonemes?

- Parts of phonemes
Record people reading material, link the sounds they make to a phonetic transcription

(you want a phonetic transcription because you’re trying to recognise words from the sounds that make them up, not from the letters that make them up

... ACCOSTED  AHO K AA1 S T AHO D
   ACCOSTING  AHO K AA1 S T IH0 NG
   ACCOUNT  AHO K AW1 N T
...

The British English Pronouncing Dictionary (BEEP), CMU Pronouncing Dictionary (find them with Google). There are probably loads of others. Then it’s easy to get a phonetic transcription: just copy the phonemes.)
HMMs are driven by observations and transition probabilities. What kinds of observations are used, and how are emission and transition probabilities derived and used?

Most of what follows is HTK specific. There’s enough detail here as it is, trying to consider more general cases would just be overwhelming. But most other systems will do similar things, so considering just this one system will carry lessons about others.
What kinds of observations are we going to use?

How fine-grained should our vectors be?

- Two cepstral blocks?

Every element of the spectrum will fall into one of two bins. We won’t be able to distinguish between sounds.
• 1000 cepstral blocks?

No, we’re not going to be able to make the generalisations we need. 100hz will be in a different block from 101hz.

• The HTK book suggests 12 pieces. So do most other people. So that’s what we use.
How frequently should we sample the signal?

- As frequently as possible: but you can’t make sensible estimates about frequency if the sample is too small. Suppose you wanted to know how frequently waves were breaking on a beach. You’d probably want to wait for at least two consecutive waves to arrive. So if they were only arriving every 6 seconds, you wouldn’t take measurements every second.

- But we haven’t really got regular patterns, so it’s not about estimating the regularity and using that as a guide. It’s about what works.

- Sample at 44K hz preserves all the distinctions that the human ear can respond to. So there’s no need to go more fine-grained than that. If your recording setup is noisy (noisy environment, poor quality microphone, . . . ), it’s actually better to sample less frequently.
So the sound files get converted to something like the one below

<table>
<thead>
<tr>
<th>Source: EXPT/train1.mfc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample Bytes: 52</td>
</tr>
<tr>
<td>Num Comps: 13</td>
</tr>
<tr>
<td>Num Samples: 125</td>
</tr>
<tr>
<td>Sample Kind: MFCC_K_0</td>
</tr>
<tr>
<td>Sample Period: 10000.0 us</td>
</tr>
<tr>
<td>File Format: HTK</td>
</tr>
</tbody>
</table>

--- Observation Structure ---

<table>
<thead>
<tr>
<th>Samples: 0-&gt;-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x: MFCC-1 MFCC-2 MFCC-3 MFCC-4 MFCC-5 MFCC-6 MFCC-7 MFCC-8 MFCC-9 MFCC-10 MFCC-11 MFCC-12 C0</td>
</tr>
</tbody>
</table>

...
How can I use these observations as probabilities?

Let’s imagine that I’ve counted how tall all the men on Venus are and how tall all the women on Venus are, and I’ve found that the average height of Venusian men is 1.80 and the average height of Venusian women is 1.60.

I’m on Venus, I see someone in the distance, I reckon they are about 1.78, I guess that they are male. How sure am I of this? Don’t know.
How can I use my observations as probabilities?

Let’s imagine that I’ve counted how tall all the men on Mars are and how tall all the women on Mars are, and I’ve found that the average height of Martian men is 1.80 and the average height of Martian women is 1.60.

I’m on Mars, I see someone in the distance, I reckon they are about 1.78, I guess that they are male. How sure am I of this? Don’t know.
Try again.

Imagine that I’ve counted how tall all the men on Venus are and how tall all the women on Venus are, and I’ve found that typical heights for men are between 1.40 and 2.20, and typical heights for women are between 1.20 and 2.00.

Imagine that I’ve counted how tall all the men on Mars are and how tall all the women on Mars are, and I’ve found that typical heights for men are between 1.78 and 1.82, and typical heights for women are between 1.58 and 1.62.

I’m on Venus/Mars, I see someone in the distance, I reckon they are about 1.78, I guess that they are male. How sure am I of this? On Mars I’m pretty confident.
Distributions of things like heights very often have distributions like the ones below. So often, in fact, that these are called ‘normal distributions’.

Figure 14: Simple Gaussian/normal distribution: mean=180, $\sigma=30$
The area to the left of the blue curve at 160 tells you how likely a Martian man is to be 1.60 (not very), and how likely a Venusian man is to be 1.60 (fairly).

![Graphs showing distributions for Martian men and Venusian men and women.](image)

Figure 15: Martian men and women are quite distinct; Venusians are all sorts of sizes.

It doesn’t directly tell you how likely a Martian who is 1.60 is to be male, but a bit of tinkering will tell you how much more likely they are to be female than male.
But although you can’t tell much about Venusian men and women from their heights, Venusian men are all very fat. So while their weights vary, a short Venusian man weighs much more than a short Venusian woman.

You can make good use of such ‘Gaussian mixture models’. And that’s what our MFCC vectors will do for us. I have to count the values of each feature for each snapshot of each phoneme; and then I can make sensible guesses from a complete snapshot to the name of a phoneme.
What do the transition tables look like?

Markov model: network where you can get from one node to another, with know transitions from one state to the next, with known probabilities for getting from one state to another.

Easiest way to represent this is as a matrix. Any network of N nodes can be represented as an $N \times N$ matrix, where a non-zero entry at point $i, j$ means that you can get from state $i$ to state $j$. 
We’re thinking about phones: a phone could perhaps be split into 3 major pieces—what does it sound like to start with, what does it sound like in the middle, what does it sound like at the end? The start and end will be affected by the adjacent sounds, the middle will be more characteristic of the sound itself.
Problem: we don’t know how long we’re going to stay in a given phone. So we can’t do it the way I did earlier, where each state is followed by a set of different states. We have to be able to stay in a single state: that doesn’t make any sense in a transition network: the best we can do is go from state\(_i\) back to state\(_i\).

That doesn’t contradict anything about this being a Markov model, and it replicates the notion of being in a state for a period of time because we know that each transition takes a fixed period of time.

It means that we can’t use anything about how long we’re likely to stay in a state: as far as the model is concerned, you’re just as likely to go on saying /u/ if you’ve already been saying it for 1 sec as if you’ve only been saying it for 0.01 sec.

What kinds of observations are we going to use?
So for each phoneme we make a skeleton Markov model: don’t know anything, so they’re all the same.

<BeginHMM>
<NumStates> 5

<State> 2
<Mean> 25
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
<Variance> 25
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

<State> 3
<Mean> 25
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
<Variance> 25
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0

<State> 4
...

<TransP> 5
0.0 1.0 0.0 0.0 0.0
0.0 0.6 0.4 0.0 0.0
0.0 0.0 0.6 0.4 0.0
0.0 0.0 0.0 0.7 0.3
0.0 0.0 0.0 0.0 0.0

<EndHMM>
First bit is the emission probabilities for each state given the observations. We don’t know anything about them, so ‘flat start’ them all as 0.

Second bit is the transition probabilities. The only places you can go are to where you are now (down the diagonal) or to the next state (go one to the right). We don’t know anything about them so we flat start them with some set of values.
What have I got at this point?

Lots of recordings, with phonetic transcriptions.
There is one of these for each phoneme. identical to these.

"h "SH"
<BEGINHMM>
<NUMSTATES> 5
<STATE> 2
<MEAN> 25
3.085929e-09 2.503057e-10 -8.678441e-09 -1.520049e-08 -1.985113e-09 -4.497278e-09 ...
<VARIANCE> 25
3.606373e+01 3.494007e+01 4.427650e+01 4.339913e+01 3.902340e+01 6.111320e+01 ...
<GCONST> 1.000144e+02
<STATE> 3
<MEAN> 25
3.085929e-09 2.503057e-10 -8.678441e-09 -1.520049e-08 -1.985113e-09 -4.497278e-09 ...
<VARIANCE> 25
3.606373e+01 3.494007e+01 4.427650e+01 4.339913e+01 3.902340e+01 6.111320e+01 ...
<GCONST> 1.000144e+02
... 
<TRANSP> 5
0.000000e+00 1.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
0.000000e+00 6.000000e-01 4.000000e-01 0.000000e+00 0.000000e+00
0.000000e+00 0.000000e+00 6.000000e-01 4.000000e-01 0.000000e+00
0.000000e+00 0.000000e+00 0.000000e+00 7.000000e-01 3.000000e-01
0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
<ENDHMM>
So that’s one HMM per phoneme.

How did we get that from the training data?

Given a sequence of phonemes, I can make a single transition network out of each of the individual networks: suppose we have the following networks for /c/, /a/, /t/:
Allan Ramsay, III Speech  

What kinds of observations are we going to use?
Allan Ramsay, III Speech

What kinds of observations are we going to use?
Now assume that we know that someone has said ‘cat’, and that the phonetic transcription of this is /c/ /a/ /t/. (this is exactly what we have in our training data, in phone0.mlf)

Then we know that the only place you can go from the last state in the /c/ network is the first state in the /a/ network, and the only place you can go from the last state in the /a/ network is the first state in the /t/ network.

So we can build a new network for this utterance.
So now we have a transition network, with initial probabilities, for the sequence of phonemes /c/ /a/ /t/.
Allan Ramsay, III Speech -124- What kinds of observations are we going to use?
(note that the matrix representation is very sparse: the only non-zero values are on the diagonal and next to it. So sensible to use sparse matrix representations – saves space, miles quicker. The one at https://github.com/phvu/misc/tree/master/viterbi uses ordinary matrices, so could be made even neater (≡ even harder to follow) by using sparse ones)
If we knew how long we were in each state, we could estimate the state→state probabilities and how likely we were to be in a given state given a set of observations (the emission probabilities).

(if we make a transition every $n$ seconds, then if we stay in that state for $N$ seconds then we had $N/n$ chances to leave it before we did, so a decent estimate of the chance of leaving it at after any given state is $n/(2 \times N)$.

At each moment that we’re in a given state, we know what the observations were. So we can count how many times we made each observation and how many times when we made that observation we were in a given state. So that one’s dead easy)

But we don’t.
If we knew the state→state probabilities, we could estimate how long we were in each state and how likely we were to be in a given state given a set of observations. But we don’t.

If we knew the emission probabilities, we could estimate how long we were in each state and hence the state→state probabilities. But we don’t.

We have to somehow estimate all of these at the same time.
Given an initial set of transition and emission probabilities, we can ‘reestimate’ them, using Baum-Welch reestimation/forward-backward algorithm.

Very very roughly: if your current set of emission probabilities were correct, you could come up with a good set of transition probabilities. So assume that the emission probabilities are correct, and work out what you think the transition probabilities are.

But if your current set of transition probabilities were correct, you could come up with a good set of emission probabilities. You’ve just worked out what you think are a good set of transition probabilities, so assume that they’re actually correct and work out what the emission probabilities should be.

Do it again till bored.
That’s what we do with the HTK: at each stage, we make a transition network for each utterance out of the phonemes that we know make up that utterance, learn the transition and emission probabilities for that utterance.

It’s easy enough to decompose the compound networks that we made for sequences of phonemes back into the individual phonemes. So now we have several HMMs for each phoneme, which we turn into a single HMM (presumably by taking average values: the HTK book doesn’t make any of this very clear)

Successive stages involve making slightly richer models: introduce space for a silence between words, try to get statistics about sequences of three sounds, ...
Then when we want to do recognition, we have a collection of HMMs for individual phonemes.

But we’re not going to get individual phonemes. We’re going to get sequences of phonemes. So we need to build a complex network like the ones we built for each element of the training data.

How can we do that? We don’t know what is going to be said (if we did, we wouldn’t have to do speech recognition on it!)

If we write a grammar, we can use that to build the compound network. So that’s what we do.
What kinds of observations are we going to use?
This generates the following network:
But we don’t want words, we want phonemes. So the grammar actually looks like

\[
\begin{align*}
\text{DET} &= a \mid \text{the}; \\
\text{NOUN} &= \text{cat} \mid \text{dog}; \\
\text{VERB} &= \text{sleeps} \mid \text{runs}; \\
\text{NP} &= \text{DET NOUN}; \\
\text{SENTENCE} &= \text{NP VERB}; \\
\end{align*}
\]

\[
\begin{align*}
a &= \text{uh}; \\
\text{the} &= \text{th uh}; \\
\text{cat} &= \text{c a t}; \\
\text{dog} &= \text{d o g}; \\
\text{sleeps} &= \text{s l iy p s}; \\
\text{runs} &= \text{r u n s};
\end{align*}
\]
which in turn generates
(well, actually

...uh cats slipp...
That’s what the HTK does.

HTK was developed at Cambridge. They sold it Microsoft, and for a few years you couldn’t get a copy, and then Microsoft made it publically available again.

What did they do in the meantime?

Emission probabilities don’t have to be Gaussian mixtures. You can get them from anywhere that you like. For instance, you could use a (deep) neural network (Hinton et al. 2006).

But you need to have frames aligned to the speech signal before you start. So train a standard HTK model, use that to align the signal with the phoneme labels. And then train your neural network on that. **Very hard to train. Very slow to train.** And that really is the state-of-the-art.
• Representation of sampled speech signal as states in a Markov model with loops

• Viterbi algorithm

• Construction of compound Markov model for a known utterance by composing models for individual phonemes

• Reestimation of parameters from initial guess

• Construction of compound Markov model for possible future utterances by composing individual models by using a grammar
Human speech is made up of a mixture of chords (produced by generating a set of individual notes) and white noise, interrupted by pauses and bursts of pure white noise.

We can generate notes. We can generate white noise. And we can mix them up.

So you could produce a mixture of chords and white noise, and you could interrupt it with silences and more noise.

‘Formant synthesisers’ of this kind are dreadful (but more recent variations on this are much better Tokuda et al. (2013))
Human speech is produced by saying one word after another.

Record lots of words, say them one after another (try it with Praat).

synthesise.py (command line interface), edit recordings, Praat script
Comprehensible. Not good on boundary effects between words. Need to record a very large number of words (average vocabulary is 60K words, need to have all inflected forms which means recording every noun twice, every verb three times: Arabic—record every verb 56 times!).

Very difficult to say words in isolation with absolutely no emphasis, and with the same pitch and speed and intensity.

Widely used in airports, railway stations, ...
Human speech is produced by producing one phoneme after another.

Record lots of phonemes, say them one after another.

Comprehensible? Horrible, at best. Terrible boundary effects (demo again)

There are only about 50 phonemes for any given language, so at least you don’t have to record all that much stuff.
Why does it sound horrible?

Partly because it’s not seamless. Obviously the way I’ve just done it has clicks and silences between the sounds, but it wouldn’t be too hard to squeeze the sounds more closely together.

More problematically, you can’t just say ‘c’ or ‘t’ in isolation. Physically impossible. They’re interruptions in the airflow, so if the air isn’t flowing they can’t happen.
Record lots of pairs of phonemes (‘diphones’), say them one after another.

Perfectly comprehensible. Generally a bit monotone, because words have stress patterns, but phones/diphones/syllables recorded in isolation don’t.

Have to record somewhat less than $|\text{phones}|^2$ diphones (because plenty of pairs just don’t occur: there are no English words with the sequences ‘gx’, or ‘mg’, or . . . )
Using forced alignment to find diphones

Manually chopping a speech file into diphones is hard

- It’s difficult to cut at the right points
- And it’s tedious

Train a recogniser.

- Recognisers are driven by a grammar: if the grammar is exactly the input text, then the recogniser can hardly help getting it right.
- But you can ask a recogniser about timing, and you can get it to output phonemes as well as words
- A diphone goes from the midpoint of one phone to the midpoint of the next
That’s the coursework!
Finding the best sequence of units

If you’ve got a small set of units, you may find that you’re stringing together things that don’t sound very nice next to each other

• mismatches of pitch or intensity
• /ka/ from ‘cat’ might not fit nicely with the /an/ from ‘man’ if I’m trying to say ‘can’.
If you’ve got lots of units, you have to find the best sounding sequence.

‘Dijkstra’s algorithm’

Make an array of units: columns are alternative realisations of the current phoneme.

for each unit $U^n_i$ in this column
  for each unit $U^{n+1}_j$ in the next column
    calculate the mismatch between them: add that to the cost of getting to $U^n_i$

    if that’s better than the best current path to $U^{n+1}_j$, replace that path with the path to $U^n_i$ plus $U^{n+1}_j$
Neat enough algorithm: complexity is \(|\text{number of columns} \times |\text{number of diphones/column}|^2\) – not amazing, but OK.

But obviously you need a good way of calculating the cost of joining two diphones.

- were they extracted from the same word?
- do they have the same pitch at the point where they meet? (see below for calculating pitch: FFT doesn’t give you it)
- do they have the same intensity at the point where they meet?
- do they have the same MFCCs at the point where they meet!?!!?
Allan Ramsay, III Speech

Finding the best sequence of units
Allan Ramsay, III Speech

Finding the best sequence of units
Allan Ramsay, III Speech -151- Finding the best sequence of units
Finding the best sequence of units
Allan Ramsay, III Speech

-153-

Finding the best sequence of units
Finding the best sequence of units
Allan Ramsay, III Speech

-155- Finding the best sequence of units
Finding the best sequence of units
Allan Ramsay, III Speech

Finding the best sequence of units
Allan Ramsay, III Speech

-158-

Finding the best sequence of units
Allan Ramsay, III Speech

Finding the best sequence of units
Allan Ramsay, III Speech

Finding the best sequence of units
Allan Ramsay III Speech -161- Finding the best sequence of units
Finding the best sequence of units
Allan Ramsay, III Speech

Finding the best sequence of units
Allan Ramsay, III Speech

-164-

Finding the best sequence of units
Finding the best sequence of units
Allan Ramsay, III Speech

-166-

Finding the best sequence of units
Allan Ramsay, III Speech

Finding the best sequence of units
1. Big difference between MFCCs at the boundary → low score
2. Small difference between MFCCs at the boundary → low score
3. Diphones came from the same recording → low score
Each recording has a duration and a pitch.

But to get natural sounding speech, we need to be able control the pitch and duration as we stitch them together.

- Obviously enough, for global pitch contour: falling at the end for statements, rising at the end for questions

- But also for local stress: ‘entering’, ‘inferring’
Easy enough to control the pitch: slow it down, speed it up.

Do that digitally: insert a copy of every $N^{th}$ frame (for every $N_{th}$ frame insert the average of the $N - 1_{th}$, $N^{th}$ and $N + 1_{th}$ frames), delete every $N^{th}$ frame.
But that makes it longer/shorter at the same time.

To fix it, delete every $N^{th}$ chunk, insert a copy of every $N^{th}$ chunk (for every $N_{th}$ chunk insert the average of the $N - 1_{th}$, $N^{th}$ and $N + 1_{th}$ chunks), delete every $N^{th}$ chunk.

If chunks are quite big, then they’ll contain lots of repetitions of the wave form, so they’ll have the same pitch and sound quality. But if they’re quite big they might contain a change from one sound to the next, and repeating that will sound weird.

If chunks are quite small, you won’t be able to hear that they are being repeated, but you might not get complete cycles.
Go through it trying to find the pitch at each point, e.g. by doing ‘auto-correlation’:

```
400 682 800 682 400 117 0 117 399 682 800 682 400 117 0
400 682 800 682 400 117 0 117 399 682 800 682 400 117
282 118 118 282 283 117 117 282 283 118 118 282 283 117
400 682 800 682 400 117 0 117 399 682 800 682 400 117
400 0 400 565 400 0 399 565 401 0 400 565 400
400 682 800 682 400 117 0 117 399 682 800 682 400 117
282 282 683 682 283 282 682 683 283 282 683 682 400 682 800 682 400 117 0 117 399 682 800 682 400 117
0 565 800 565 1 565 800 565 1 565 800
400 682 800 682 400 117 0 117 399 682 800 682 400 117
283 682 683 283 282 683 682 283 282 683 682 400 682 800 682 400 117 0 117 399
400 565 401 0 400 565 400 0 399
400 682 800 682 400 117 0 117
283 283 118 118 282 283 117 117
400 682 800 682 400 117 0
1 0 0 0 0 0 0 0
400 682 800 682 400 117
282 118 118 282 283 117
```
Go through it trying to find the pitch at each point, e.g. by doing ‘auto-correlation’:

400 682 800 682 400 117  0 117 399 682 800 682 400 117  0
400 682 800 682 400 117  0 117 399 682 800 682 400 117
282 118 118 282 283 117 117 282 283 118 118 282 283 117
400 682 800 682 400 117  0 117 399 682 800 682 400
400 0 400 565 400 0 399 565 401 0 400 565 400
400 682 800 682 400 117  0 117 399 682 800 682
282 282 683 682 283 282 682 683 283 282 683 682
400 682 800 682 400 117  0 117 399 682 800
0 565 800 565 1 565 800 565 1 565 800
400 682 800 682 400 117  0 117 399 682
283 682 683 283 282 683 682 283 282 683 682
400 682 800 682 400 117  0 117 399
400 565 401 0 400 565 400 0 399
400 682 800 682 400 117  0 117
283 283 118 118 282 283 117 117
400 682 800 682 400 117  0
1 0 0 0 0 0 0 0
400 682 800 682 400 117
282 118 118 282 283 117
Go through it trying to find the pitch at each point, e.g. by doing ‘auto-correlation’:

<table>
<thead>
<tr>
<th>400</th>
<th>682</th>
<th>800</th>
<th>682</th>
<th>400</th>
<th>117</th>
<th>0</th>
<th>117</th>
<th>399</th>
<th>682</th>
<th>800</th>
<th>682</th>
<th>400</th>
<th>117</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td>117</td>
<td>399</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>282</td>
<td>118</td>
<td>118</td>
<td>282</td>
<td>283</td>
<td>117</td>
<td>117</td>
<td>282</td>
<td>283</td>
<td>118</td>
<td>118</td>
<td>282</td>
<td>283</td>
<td>117</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td>117</td>
<td>399</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>0</td>
<td>400</td>
<td>565</td>
<td>400</td>
<td>0</td>
<td>399</td>
<td>565</td>
<td>401</td>
<td>0</td>
<td>400</td>
<td>565</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td>117</td>
<td>399</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>282</td>
<td>282</td>
<td>683</td>
<td>682</td>
<td>283</td>
<td>282</td>
<td>682</td>
<td>683</td>
<td>283</td>
<td>282</td>
<td>683</td>
<td>682</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td>117</td>
<td>399</td>
<td>682</td>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>565</td>
<td>800</td>
<td>565</td>
<td>1</td>
<td>565</td>
<td>800</td>
<td>565</td>
<td>1</td>
<td>565</td>
<td>800</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td>117</td>
<td>399</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>283</td>
<td>682</td>
<td>683</td>
<td>283</td>
<td>282</td>
<td>683</td>
<td>682</td>
<td>283</td>
<td>282</td>
<td>682</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td>117</td>
<td>399</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>565</td>
<td>401</td>
<td>0</td>
<td>400</td>
<td>565</td>
<td>400</td>
<td>0</td>
<td>399</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>283</td>
<td>283</td>
<td>118</td>
<td>118</td>
<td>282</td>
<td>283</td>
<td>117</td>
<td>117</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>682</td>
<td>800</td>
<td>682</td>
<td>400</td>
<td>117</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>282</td>
<td>118</td>
<td>118</td>
<td>282</td>
<td>283</td>
<td>117</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This is the ‘**PSOLA**’ algorithm ([Charpentier and Stella 1986](#)).

(not sure why it get the length of the second ‘the’ wrong)
• Formant vs concatenative vs diphone vs multiphone synthesis

• Naturalness, comprehensibility

• Using Dijkstra’s algorithm to find the smoothest path through a set of potential units

• Raising and lowering pitch: using autocorrelation to find chunks that can be inserted or deleted without messing things up
II Morphosyntax
(8) a. grasmaaier, tondeuse, Rasenmähmaschine, falciatrice, cortacéspedes
   (lawnmower)

b. He debirled it. (reasonable English word)

c. He drenstfurged it. (less plausible: looks more like an import from German)

d. He degirtted it. (this isn’t the past tense of ‘degirt’, but ‘degirttt’ doesn’t look like English)

e. He chought it. (past tense of some German/Scandinavian import).
Reconstructions of serious crimes can help to jog people’s memories.

He’s a complete unreconstructed Stalinist.

* It’s just an unreconstruction.

One of my best friends is watching old movies.

One of my favourite activities is watching old movies.

The woman who you said that you expected me to meet is waiting at the door.
(12) a. I believe Betty is a fool.

    b. I believe that Betty is a fool.

    c. Betty, I believe, is a fool.

    d. Betty is, I believe, a fool.

    e. * Betty, I believe that, is a fool.

(13) I saw the man with a big nose in the park with a telescope.
I am aware of the problems that arise in defining the word, ... Nevertheless, it seems to be widely agreed among linguists these days that most, or perhaps even all, languages make use of units which could reasonably be called words. Hudson (1984), my emphasis
There seem to be basic meaning bearing items, which are called ‘morphemes’ (very wary of talking about ‘words’, because it’s a very fuzzy word. But I can think about the smallest things that seem to carry meaning, which I might split into roots and affixes: the root carries the basic meaning, the affixes add to it or change it: ‘un-re-con-struct-ed’, ‘in-struct-ion’, ‘structural’, . . . . No such word as ‘struct’ (except in C)).

We need to store them in some way that will make them easy to retrieve.
For analysis, we presumably expect to retrieve the syntactic and semantic information from an examination of the surface form. Most people use something like the representation in Fig. 17.
The obvious advantage of this representation is that it saves you lookup time. At each point, you are led directly to the next possible node, so that there is a minimum of search (and hence of backtracking).

Some sums: suppose that you have a 20000 word dictionary, where the average word length is 6 characters, with the following words at the end: zaibatsu, zander, zeal, zebra, zenith, zeolite, zero, zest, zidovudine, zigzag, zinc, zip, zither, zloty, zodiac, zombie, zone, zoo, zoology, zoom, zoot
Then to look up ‘zoom’ in a straight alphabetic list you’re going to do something between 20000 and 120000 comparisons. To look it up in the current representation you’re going to do $26+3+4+2$ comparisons. Well worth it, and it gets better later.

In abstract terms, the lookup time is $o(N \times I)$ for the simple linear list representation and $o(I)$ for the branching tree representation, where $N$ is the number of words in the dictionary and $I$ is the maximum length of a word.
Set $PYTHONPATH$ to /opt/info/courses/COMP34411/PROGRAMS (if you're using bash, put export PYTHONPATH=/opt/info/courses/COMP34411/PROGRAMS:$PYTHONPATH in your .bash_profile file.

```
import lextrie

t = lextrie.TRIE()

# Read the output of show as though the letters were labels on arcs, not nodes
# t.show()

t.lookup('cat', rules=[], printing=True)
```
Morphographemetics, morphophonemetics (‘spelling rules’)

Why does ‘change+ing’ become ‘changing’, and how do we know that ‘hanging’ wasn’t ‘hange+ing’?

(Chomsky and Halle 1968) provide a very detailed and insightful account of the phonology of English, looking particularly at the way stress moves around, but with some useful things to say about how this is reflected in spelling.
We want **spelling** rules, not phonological rules. Spelling rules are horrible, because they are a mixture of ‘**phonology**’ (what things sound like) and ‘**realisation rules**’ (which letters correspond to which sounds: can’t find a proper name for it).

C & H were interested in why ‘*change+ing*’ becomes ‘*chang-ing*’. We’re interested in how to recognise that ‘*changing*’ is the written form of ‘*change+ing*’.

- Our rules go in the opposite direction from theirs
- Their rules are compulsory, ours are optional (have to be: ‘*hanging*’ ≠ ‘*hange+ing*’)

Allan Ramsay, II Morphosyntax
Because we're looking at a mixture of things, people tend to just do what works. Here's the start of a set of spelling rules. Note the direction in which they are applied: left-hand side is what is WRITTEN, right-hand side is the UNDERLYING form (the bit with a # is a comment to provide an example of what the rule does).
# cha[s][e]d ==> chase+ed, cha[s][i]ng
[] ==>[e] : [c0] _ [v0];

# kis[s][e][s] ==> kiss+s
[e] ==>[s]: [s] _ [s];

# [pu][tt][i]ng ==> put+ing, re[be][ll][e]d ==> rebel+ed
[c0, c0] ==>[c0]: [c1, v0] _ [v1];

...
...

(v0, v1, ...: vowels, c0, c1, ...: consonants. You might want extra classes--short vowels, hard consonants, ...)

Allan Ramsay, II Morphosyntax -191- The lexicon
from lextrie import *

# Supply the name for a file with spelling rules in it
rules = spelling.readSpellingRules('spellingRules.txt')
t = TRIE()
t.addAll([('chase', 'noun/num'), ('chase', 'verb/tns'), ('ed', 'tns')])
t.lookup('chased', rules=rules, printing=True)
Exercises for the reader

- Find words where my spelling rules will apply but they shouldn’t. Tighten them up so they don’t.

- Find words which need new rules. Add them.
See the discussion of ‘two-level morphology’ in (Jurafsky and Martin 2000; Koskiennemi 1985; Ritchie et al. 1992) for some thoughts on how to do this.

One of the key issues is how you combine the application of spelling rules and lexical lookup. You don’t really want to apply all the spelling rules before you start looking in the dictionary (typical Arabic verb has around 60 different forms: even with my neat representation of the dictionary I don’t want to multiply it by 60); but you won’t find what you’re looking for in the dictionary unless you apply the spelling rules.
• The principles of how and why you would represent your lexicon as a letter trie

• What spelling rules look like, and how to combine them with lexical lookup

• The matching algorithms used for applying spelling rules to strings and hence to dictionary entries
Morphology

Words can be broken into little pieces (otherwise what was all that stuff about spelling rules about?)

Why?

‘Inflectional morphology:’ the stem is incomplete, the affix supplies extra information:

- ‘eat’+‘ing’ = verb + pres part
- ‘box’+‘s’ = noun + plural
- ‘box’+?? = noun + sing
- ‘box’+‘s’ = verb + pres tense, 3rd sing (as in hitting people)
- ‘blancos’ = ‘blanc’+‘o’+‘s’ = adj + masc + plural
‘Derivational morphology:’ the stem says one thing, the affix turns it into something else

- ‘construct’ + ‘ation’ = verb + affix = noun
- ‘bad’ + ‘ly’ = adj + affix = adv?
(14) a. Dia ajar bapanya
   He taught father-him
   ‘He taught his father’

b. Guru itu memberi ajaran kepada masyarakatnya
   Teacher that gives education to his community
   ‘The teacher gives education to his community’

c. Dia belajar di bawah pokok
   He studies under the tree
d. Dia ialah golongan berajar di kampungnya
He is groups educated in village-him

e. Dia diajar oleh bapanya
He was taught by father-him.

f. Pelajar itu pandai
Student that (is) clever
‘That student is clever’

g. Pelajaran itu saya suka
Lesson that I like

h. Dia mengajar pelajar itu
He teaches student that
i. Guru itu **mengajarkan** pelajaran itu kepada pelajar itu  
Teacher that **teaches** lesson that to student that  
(ditransitive version of teach)

j. Pelajaran itu **diajarkan** oleh gurunya  
Lesson that **taught** (passive) by teacher-him  
‘The lesson was taught by his teacher’  
(passive of the ditransitive version)

k. Saya suka kaedah **pembelajaran** kawan saya  
I like method **learning** friend me  
‘I like my friend’s way of learning’

l. Saya suka kaedah **pengajaran** guru saya  
I like method **teaching** teacher me  
‘I like my teacher’s way of teaching’
What have we got to cope with?

How do we specify that some morpheme requires an affix, and how do we describe what it wants?

How do we control the order in which they get added? (‘unreconstructed’, ‘reconstructions’, *‘unreconstructions’).

How do we do the fine detail? Why ‘known’ rather than ‘knowed’?

How do we mesh it with our lookup process? Now I have to check that the spelling rules are obeyed and that there are appropriate dictionary entries and that they will combine appropriately. How do I get ‘impossibilities’?
We spend a lot of time combining small things into bigger ones: combining words into sentences, combining morphemes into words, ... 

One way of doing that it is by writing rules that tell you what kinds of things combine, and what they make:

\[ s \rightarrow \text{name, verb, name} \]

Then (assuming that I knew that ‘loves’ was a verb and ‘John’ and ‘Mary’ were names) I’d be able to make a sentence out of ‘John loves Mary’. I’ve got all the bits, and they’re in the right order, so they make a sentence.
There’s another way of thinking about it.

What does a sentence do? It tells you about some event, telling you what kind of thing happened and who or what was involved.

What does a verb do? It specifies the kind of event, but not who or what was involved. So it’s an incomplete description of an event, i.e. it’s an incomplete sentence.
So instead of saying that ‘loves’ is a verb, I’ll say that it’s an incomplete sentence, and I’ll specify the things which it needs and where to look for them.

loves = s[\overrightarrow{np}, \overleftarrow{np}]

loves = (s\backslash np)/np

(order of combination is reversed in the two notations: the first is better, the second is the one that gets used)

Then

John loves Mary
np (s\backslash np)/np np

----------------

s\backslash np

----------------

s
Descriptions of this kind are called ‘categorial descriptions’. Items that need something to make themselves complete are referred to as ‘unsaturated’, complete items are ‘saturated’: see Wood (1993) for a good introduction to categorial grammar.
But right now I’m interested in the way morphemes combine to make words. The same trick will work (Bauer 1983):

kick = N[NUM
kick = V[TNS

s = NUM
s = TNS

(to make it work, I need to allow for empty affixes: which is a bit irritating, but I’m always going to have do to something about the fact that ‘walk’ can be a complete word, or it can be a part of bigger item—‘walks’, ‘walking’, ‘walked’, ‘walker’)

Allan Ramsay, II Morphosyntax -206- Categorial descriptions
Variable length affix sequences

blanc = ADJ/NUM/GEN
o = GEN
s = NUM

blanc = ADJ/GEN
o = GEN/NUM
s = NUM
construct = V/TNS

ation = (N/NUM)\(V/TNS\)

re = (V/TNS)/(V/TNS)
If I want to mesh it really nicely with the lookup process, I need to do things as early as I can.

Left-to-right parsing with categorial grammar: extend my rule set to include (Ades and Steedman 1982)

Cancellation

\[ X/Z \rightarrow X/Y \ Y/Z \]
\[ X\Z \rightarrow Y\Z \ X\Y \]

Raising

\[ Y/(Y/X) \rightarrow X \]
\[ Y/(Y\X) \rightarrow X \]

(makes a kind of sense, by analogy with division)
Within categorial morphology I can make use of the cancellation rules: so processing ‘blancos’ I can do the following:

<table>
<thead>
<tr>
<th>blanc</th>
<th>o</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADJ/GEN</td>
<td>GEN/NUM</td>
<td>NUM</td>
</tr>
</tbody>
</table>

---------------------
|               |
| ADJ/NUM       |

---------------------
|               |
| ADJ           |
That gives me a nice handle on words that take a variable number of affixes:

‘escribiremos’ ⇒ ‘{{escrib,i},re},mos}’
‘escribimos’ ⇒ ‘{{escrib,i},”},mos}’
‘escribimos’ ⇒ ‘{{escrib,i},mos}’

escrib ==> V/TH
i ==> TH/TNS
i ==> TH[NUM (+tensed, preterite)]
re ==> TNS[NUM (future)]
’’ ==> TNS[NUM (present)]
mos ==> NUM
Have to be careful about which words take which affixes:

- ‘escrib’ takes ‘-i-’ as its normal theme vowel, and consequently it takes ‘-a-’ as the theme vowel in the subjunctive.
- ‘take’ has a German past, and consequently takes ‘-en’ as its past participle???
The rules are usually rather simple defaults, and people have developed special logics for dealing with them (Evans and Gazdar 1989). I’m not too worried about that: what does matter is that there are patterns.

English:

ed (past part) → ed (past tense)
ed (past tense) → s (pres tense)
s (pres tense) → ing (pres part)
ing (pres part) → ’’ (infinitive)
Where shall we put affixes?

In the lexicon: then when you get to a point where you’ve found the end of a word, go back into the trie with whatever’s left.

Empty affixes: put them in the lexicon (nice and uniform), or keep them in a table (probably more efficient)
trie.lookup(t, 'cars')
Unseen: cars, items found so far: []
Allan Ramsay, II Morphosyntax

Categorial descriptions
Unseen: s, items found so far: [noun>agr]
Allan Ramsay, II Morphosyntax

Categorial descriptions

Unseen: , items found so far: [noun>agr]
Unseen: , items found so far: [noun]
	noun>agr+agr -> noun,
trie.lookup(t, 'carts')
Unseen: carts, items found so far: []
Unseen: rts, items found so far: []
Unseen: ts, items found so far: [noun>agr]
Allan Ramsay, II Morphosyntax

Categorial descriptions
Unseen: s, items found so far: [noun>agr]
Unseen: , items found so far: [noun>agr]
Unseen: , items found so far: [noun]

noun>agr+agr -> noun,
trie.lookup(t, 'kisses')
Unseen: kisses, items found so far: []
Unseen: es, items found so far: []
Unseen: s, items found so far: [noun>agr, verb>tns]
Unseen: +s, items found so far: []
Spelling rule applied: [e] ==> [+] : [s, x0] -= [s]; unseen was es and is now +s (seen ssik)
Unseen: s, items found so far: [noun>agr, verb>tns]
noung\>agr+agr -> noun, verb>tns+tns -> verb,
Unseen: +es, items found so far: []
Spelling rule applied: [c1] => [+]: [c1, v0, c0] _ []]; unseen was ses and is now +es (seen sik)
trie.lookup(t, 'chasing')
Unseen: chasing, items found so far: []
Unseen: hasing, items found so far: []
Unseen: asing, items found so far: []
Unseen: ing, items found so far: []
Unseen: e+ing, items found so far: []
Spelling rule applied: [v0] =⇒ [e, +, v0] : [c0, v1] _ [d/n]; unseen was ing and is now e+ing (seen sahc)
Unseen: +ing, items found so far: []
Unseen: ing, items found so far: []

Allan Ramsay, II Morphosyntax
verb>tns+tns -> verb,
• Distinction between inflectional and derivational morphology

• ‘Unsaturated lexical items’, categorial morphology, cancellation rules for left→right processing
That’s what we want: a dictionary that contains stems and affixes, spelling rules that relate surface forms to stems, specification of how to put stems and affixes together.

But we’re always going to come across words that aren’t in the dictionary: we need to be able to do something sensible when we see an unknown word—we don’t want a program that just falls over when it sees a new word.
('Open' and 'closed' classes:

- nouns depict things, verbs depict events, adjectives add extra information. There are lots of them, and they’re quite easy to add to. They’re ‘open’.

- prepositions ('in', 'on', . . . : depict relations between things), auxiliaries ('be', 'have', 'would', . . . : add temporal information to verbs). They get their significance from what they do to other words, there’s a smallish number of them, you don’t get new ones. They’re ‘closed’)
So we need to be able to ‘back off’ to something more robust (but less accurate: if we had something which was robust AND accurate then that’s what we’d use in the first place).

This is a common strategy: use something you like when you can, back off to something that sort of works when you can’t.

Two parts to this here: what word was it (lookup), what part of speech was it (tagging)?
What word was it?

Suppose you see ‘splanging’? It could be a word, but it’s probably the present participle of some other word.

What other word? Could be ‘splang’, could be ‘splange’.

Suppose you then see ‘splanged’. This is probably the past tense/past participle of another word. What other word? Could be be ‘splang’, could be ‘splange’.
If we just remove ‘-ing’ and ‘-ed’ we’ll get ‘splang’. If the only forms we ever see are ‘splanging’ and ‘splanged’ then this will get us a reasonable outcome—we’ll spot that these are forms of the same word, which is useful; and we won’t get into any trouble by not knowing whether the underlying form was ‘splang’ or ‘splange’. 
But then suppose we later see ‘splanges’. This can only come from ‘splange’ + ‘-s’. And we won’t see that this is actually the same item as ‘splanging’ and ‘splanged’, because we’ve already decided that these come from ‘splang’.

Solution: when you see ‘e’ at the end of a word, strip it off as well.

So ‘splanges’, ‘splanged’, ‘splanging’ are all derived from ‘splang’. Even ‘splange’ is.
‘Porter stemmer’: set of rules about removing bits of a word in turn until you get to a stem which is common to all forms of the same word.

```python
>>> from nltk.stem.porter import *
>>> stemmer = PorterStemmer()
>>> stemmer.stem("changing")
>>> stemmer.stem("derofier")
>>> stemmer.stem("derofy")
```

The ‘stem’ that you get from the Porter stemmer can sometimes surprise you: try ‘derofier’ and ‘derofy’. The aim is to get the same thing for all forms of a word, not to work out what the word itself is.

Backoff: use dictionary and spelling rules, and if that doesn’t work back off to stemmer.
The other thing we got out of the dictionary was the ‘part of speech tag’ for the word.

We need this if we want to work out the relationships between the words that make up a sentence.

(15) John loves Mary.

How do we know that this is an action involving two people: because ‘loves’ is a verb and ‘John’ & ‘Mary’ are names.
You need to know the part of speech of each word in a sentence in order to work out how they are related.
Simplest thing to do: get a big collection of words, assign a tag to each one, see how often each word occurs with each tag.

• get a big collection of words, assign a tag to each one, see how often each word occurs with each tag.

• you need a LOT of words in order to get useful information, and hand-tagging a lot of words is a big task.
There are several publicly available corpora, e.g.

- **BNC (British National Corpus)**: 100M words of English. Tagged, but not terribly accurately. Useful for looking at how the size of the data set affects things.

- **Universal dependencies**: (?) at //lindat.mff.cuni.cz/repository/xmlui/handle/11234/LRT-1478. It’s not the biggest that you can get (about 250K words), but it seems to be quite well tagged, and it has an accompanying ‘treebank’ which I quite like.

- **Penn treebank**: It’s big, but you have to pay for it. The NLTK contains an extract (about 90K words). It’s a treebank, but it’s a phrase structure treebank and I want a dependency treebank, and the conversion process is not 100% reliable.
import tag

# Read the corpus (takes a while: set it up in advance!)
# (use the corpus reader version rather than reading all
# the words into one great big list 100000000 words and
# working through that from the top)
>>> cd = tag.corpusdict(tag.BNC, splitAmbiguousTags="ignore")
>>> cd.basedict['extraterrestrial']
{'AJ0': 46, 'NPO': 2}
>>> cd.basedict['cavalier']
{'AJ0': 100, 'NN1': 109}
>>> cd.basedict['injuncted']
{'VVN': 2}
...
>>> dicttagger = tag.dicttagger(cd)
>>> dicttagger.tag('the cat sat on the mat')
[('the', ('AT0', 894993)), ('cat', ('NN1', 356)), ('sat', ('VVD', 1343)), ('on', ('PRP', 88874)),
 ('the', ('AT0', 894993)), ('mat', ('NN1', 65))]
• Dead simple.

• Gives you alternatives, with reasonable information about probabilities
You know the words you know. As the corpus gets bigger, the number of words you know gets bigger.
The number of new names seems almost linear. The number of new other words seems to be flattening out, but it is still increasing even after 100M.

Proper names are fairly easy to tag correctly in English (unknown word beginning with a capital letter). Harder in German (all nouns begin with capital letters), Arabic/Hebrew/Persian (there are no capital letters).
What happens to the number of words you don’t know?

Figure 18: Probability of unknown words: 100M words
Figure 19: Probability of unknown words: 1M words
Figure 20: Probability of unknown words: 100M words, ignore first 1M
• Spikes followed by downward drift correspond to new genres being included. Huge spike at the end: you never know what’s coming next.

• Distributions of names and other kinds of words roughly follow each other. Presumably new genres introduce new technical terms and new names.

No matter how many words you’ve seen, there’s a 0.6% chance that the next word is unknown (so in a sentence of length 20 there’s a \(1 - (1 - 0.006)^{20} = 11\%\) chance of finding a new word). This barely changes after you’ve seen 500K words.
• You need a big **accurately** tagged corpus. Where do you get one of those? Who confirms that it’s accurate?

• Doesn’t pay any attention to the local context:
  
  (16) She is the great love of his life.

• Can’t cope with unknown words (obviously)
  
  (17) ’Twas brillig, and the slithy toves did gyre and gimble in the wabe
Measuring accuracy

How many words does it get right?

How many words does it get wrong????

‘Precision’: when it says something, how likely is it to be right?

‘Recall’: how many of the right things does it say?

If you have some way of estimating how likely you are to be right, then you can improve your precision by lowering your recall. ‘F-measure’ is a way of balancing these two:

\[ F = \frac{2 \times p \times r}{p + r} \]
Figure 21: Precision, recall, F-measure vs. dictionary size
BACKOFF!!! Keep probabilities of last N letters for words of length > N (N=3 works quite well)

atagger.dict['ing']
{ 'VBG': 467, 'VDG': 170, 'AJ0': 889, 'AV0': 4, 'VVB': 40, 'NN2': 11, 'PNI': 826, 'PRP': 407, 'VVI': 82, 'VVG': 6229, 'CJS': 16, 'VHG': 187, 'NP0': 182, 'NN1': 2041, 'NN0': 12}

atagger.dict['hes']
{ 'NP0': 5, 'NN2': 286, 'VVZ': 67}
If ‘run’ is preceded by ‘a’, it’s probably a noun. If it’s preceded by ‘I’, it’s probably a verb.

If ‘that’ is followed by a noun, it’s probably a determiner (‘I saw that man’); if it’s followed by a pronoun it’s probably a complementiser (‘I saw that she had finished’).
Collect statistics on forward and backward ‘transitions’:

Forward transitions (IN is the PTB tag for prepositions, SA is sentence start, SZ is sentence end):

IN: NN 0.33, DT 0.32, JJ 0.11, CD 0.06, PR 0.06, VB 0.04, ...
JJ: NN 0.71, JJ 0.07, IN 0.06, ...
NN: NN 0.26, IN 0.17, VB 0.13, ...

Prepositions are usually followed by nouns, determiners, adjectives, i.e. the sorts of things that start NPs
Adjectives are usually followed by nouns, sometimes by other adjectives (‘big fat old man’)
Nouns can be followed by nouns (‘man hole cover’) or prepositions or verbs. May be surprising that the commonest next item is a noun
Backward transitions:

IN: NN 0.53, VB 0.19, ...
JJ: DT 0.28, IN 0.16, VB 0.14, JJ 0.07, ...
NN: NN 0.26, DT 0.19, JJ 0.16, IN 0.11, VB 0.06, ...

Prepositions are usually preceded by nouns (‘man in the park’) or verbs (‘slept in the park’): note that the fact that the preceding word is most often a noun does not mean that the PP headed by the preposition is probably attached to the noun (‘I saw the man with a telescope’: the word before ‘with’ is a noun, but ‘with a telescope’ is probably a modifier on ‘saw’.

Nouns are typically preceded by nouns, determiners or adjectives. No great surprises there, since nouns and adjectives were often followed by nouns.
These ‘bigram’ probabilities can be used in all sorts of ways. HMM-based approaches just use forward transitions, but other algorithms use both: note that we do not need to use the complicated algorithm for estimating emission and transition probabilities, because someone has kindly annotated a great big corpus for us.

**Emission probabilities**

```python
{'park': {'VB': 0.25, 'NN': 0.75},
 'work': {'VB': 0.645, 'NN': 0.354},
 'love': {'VB': 0.826, 'NN': 0.173},
 ...}
```

**Transition probabilities**

```python
{'VB': {'PR': 0.115, 'DT': 0.165, 'VB': 0.108, ...},
 'AT': {'NN': 1.0},
 'NN': {'VB': 0.11, 'IN': 0.180,'NN': 0.192, ...},
 ...}
```
time flies like an arrow
Probability of going from verb to noun is $0.40 \times 0.50 \times 0.50 = 0.10$ (better than previous value)
Probability of going from noun to noun is $0.60 \times 0.21 \times 0.50$ ($=0.06$)
(not better than previous value)
Probability of going from verb to verb is $0.40 \times 0.12 \times 0.50 = 0.03$ (better than previous value)

Allan Ramsay, II Morphosyntax -293- HMM-based tagging
Probability of going from noun to verb is $0.60 \times 0.50 \times 0.50 = 0.15$ (better than previous value)
Probability of going from verb to prep is $0.15 \times 0.06 \times 0.50 = 0.00$
(better than previous value)
Probability of going from noun to prep is $0.10 \times 0.07 \times 0.50 (=0.00)$ (not better than previous value)
Probability of going from verb to verb is $0.15 \times 0.12 \times 0.50 = 0.01$
(better than previous value)
Probability of going from noun to verb is $0.10 \times 0.50 \times 0.50 = 0.02$ (better than previous value)
Probability of going from verb to det is $0.02 \times 0.31 \times 1.00 = 0.01$ (better than previous value)
Probability of going from prep to det is $0.00 \times 0.30 \times 1.00 (=0.00)$ (not better than previous value)
Probability of going from det to noun is $0.01 \times 0.57 \times 0.70 = 0.00$ (better than previous value)
Probability of going from det to verb is $0.01 \times 0.05 \times 0.30 = 0.00$ (better than previous value)
Allan Ramsay, II Morphosyntax

HMM-based tagging

0.60 time
0.40 flies
0.50 like
0.50 an
0.50 arrow
Transformation-based learning (TBL)

Statistical tagging is pretty tedious (you need a LOT of text to have any worthwhile statistics).

Writing rules is pretty difficult.

Is there an easier way of getting rules?
• Write a base tagger (e.g. the HMM-based one above)

• Run it on a corpus and correct part of it by hand. Don’t have to correct a huge amount, and correcting is much quicker and less tedious than annotating from scratch.

• Learn corrective rules (Brill 1995; Lager 1999). Apply these to the output of the original
Retag the current word from T1 to T2 if its tag is T3
#t0(T1, T2, T3): T1 > T2 if tag[0]=T3;

Retag the current word from T1 to T2 if its tag is T3 and the
tag of the next word is T4
#t1(T1, T2, T3, T4): T1 > T2 if tag[0]=T3 and tag[1]=T4;

...
Try all possible instantiations of the patterns with things that you find in the corpus:

Top 5 candidate rules

**********************
Candidate rule #t0(UN, NN, UN): UN > NN if [tag[0]=UN];: gross score 371
Candidate rule #t3(UN, NN, UN, NN): UN > NN if [tag[0]=UN, tag[1, 2]=NN];: gross score 205
Candidate rule #t4(UN, NN, UN, NN): UN > NN if [tag[0]=UN, tag[-1, -2]=NN];: gross score 179
Candidate rule #t3(UN, NN, UN, VB): UN > NN if [tag[0]=UN, tag[1, 2]=VB];: gross score 132
Candidate rule #t0(UN, CD, UN): UN > CD if [tag[0]=UN];: gross score 123
...

Allan Ramsay, II Morphosyntax -307- Templates (see tbl.py for the full set)
The ‘**gross score**’ is how many problems each of these would fix.

- The first four are all attempts to fix the fact that most things that are tagged as unknown are actually nouns: 371 cases overall, of which 205 have a noun as one of the next two words, 179 have a noun as one of the previous two words, 132 have a verb as one of the next two. These will overlap, so you wouldn’t want to use all of them.

- A rule may introduce new errors as well as fixing old ones. The last one in this group suggests marking unknown items as CD. That would fix 123 problems, but it would also change all the ones that should be NN to CD.
Try each candidate rule to see what its ‘net score’ is, i.e. the numbers of things it fixes minus the number of new errors it introduces. You can quit looking when the gross effect is less than the best net effect yet seen.

Top 5 net scoring rules
************************
rule: #t0(UN, NN, UN): UN > NN if [tag[0]=UN];,
gross score 371, net score 121
rule: #t4(UN, NN, UN, NN): UN > NN if [tag[0]=UN, tag[-1, -2]=NN];,
gross score 179, net score 103
rule: #t3(UN, NN, UN, VB): UN > NN if [tag[0]=UN, tag[1, 2]=VB];,
gross score 132, net score 94
rule: #t3(UN, NN, UN, NN): UN > NN if [tag[0]=UN, tag[1, 2]=NN];,
gross score 205, net score 33
rule: #t0(UN, CD, UN): UN > CD if [tag[0]=UN];,
gross score 123, net score -375
...

Allan Ramsay, II Morphosyntax -309- Templates (see tbl.py for the full set)
Apply it, try again.

Top 5 candidate rules

***************

Candidate rule #t3(NN, JJ, NN, NN): NN > JJ if [tag[0]=NN, tag[1, 2]=NN];: gross score 175
Candidate rule #t0(NN, JJ, NN): NN > JJ if [tag[0]=NN];: gross score 158
Candidate rule #t0(NN, CD, NN): NN > CD if [tag[0]=NN];: gross score 136
Candidate rule #t0(NN, VB, NN): NN > VB if [tag[0]=NN];: gross score 122
Candidate rule #t4(VB, PO, VB, NN): VB > PO if [tag[0]=VB, tag[-1, -2]=NN];: gross score 122
...

None of these are about unknown items, because the rule we chose last changed all of those to NN.
Top 5 net scoring rules
************************
rule: #w0(VB, PO, 's): VB > PO if [word[0]='s'];, gross score 96, net score 93
rule: #w2(VB, PO, 's, NN): VB > PO if [word[0]='s, tag[-1]=NN];, gross score 94, net score 93
rule: #t2(VB, PO, VB, NN): VB > PO if [tag[0]=VB, tag[-1]=NN];, gross score 94, net score -303
rule: #t3(VB, PO, VB, NN): VB > PO if [tag[0]=VB, tag[1, 2]=NN];, gross score 104, net score -411
rule: #t4(VB, PO, VB, NN): VB > PO if [tag[0]=VB, tag[-1, -2]=NN];, gross score 122, net score -588

... These are all about relabelling '‘s’ as a possessive marker, because the original tagger thinks that this item is a verb, but in 93/96 cases it’s actually a possessive marker.
And so on until there’s no significant improvement: with the tagset and tagger given above trained on the NLTK version of the PTB we get an increase in accuracy from 0.89 to 0.95 (the first rule, setting unknown items to be NNs, gets us to 0.934, so the remainder isn’t huge, but it’s worth having).
And after all that, we still get mistakes.

Why are we tagging anyway? Because we want to parse, and we can't do that if we don't know what kinds of words we've got.

If we get it wrong, then we won't be able to get a sensible analysis.
Some words are particularly difficult to tag, and also particularly important.

(18) a. I know that.
   (pronoun)

b. I know that man.
   (determiner)

c. I know that that man is an idiot.
   (complementiser, determiner)

d. I know that man is an idiot
   (determiner)

e. I know that man is supposedly God’s highest creation
   (complementiser)

f. The man that I know is an idiot
   (relative pronoun: like ‘who’)
I know of no tagger that can get these right

If you get them wrong you can’t possibly get the right syntactic analysis

Assign ‘that’ the tag THAT. Then you’ll be right! And you won’t mislead the parser (you won’t help it all that much, but at least you won’t mislead it)

Small set of words where this is the best thing to do – ‘that’, ‘to’, ‘if’, . . .
And some words are in some way weird, and it’s not a good idea to assign them a general tag which will lead subsequent stages to do the wrong things with them.

‘in’, ‘on’, ‘at’ are all prepositions. Standard tag for prepositions is IN (because ‘in’ such an exemplary example).

These are words that are followed by an NP and which modify either a noun or a verb.
Most taggers assign the same tag to ‘ago’ and ‘of’.

But ‘ago’ **follows** the NP that it combines with – ‘*I saw him a year ago*’, not ‘*I saw him ago a year*’. So giving it the tag **IN** doesn’t seem like a very good idea.

And ‘of’ is very seldom preceded by a verb\(^5\). So just calling it an **IN** is less informative than giving it its own tag.

Again, there are just a few words like this, but noticing them can be very useful.

\(^5\)apart from ‘*I’m bored of that*’, which is ungrammatical
CHECKPOINT

- Open and closed classes
- Backoff as a strategy
- Stemming vs morphology
- Tagging: corpus-based, affix-based, HMM-based, transformation-based learning
- Underspecification: if you often get a word wrong, give it its own tag
We now have most of what we need for storing and retrieving words.

But why did we want to do that in the first place? Knowing that some word is a noun isn’t in itself very useful.

But it is crucial for finding the relations between words!

```python
>>> tagger.tag("she is the great love of his life")
[['she', 'PR'], ['is', 'VB'], ['the', 'DT'], ['great', 'JJ'], ['love', 'NN'], ['of', 'OF'], ['his', 'PX'], ['life', 'NN']]
```

```python
>>> tagger.tag("I love her with all my heart")
[['I', 'NN'], ['love', 'VB'], ['her', 'PR'], ['with', 'IN'], ['all', 'DT'], ['my', 'PX'], ['heart', 'NN']]
```

```python
>>> tagger.basetagger.dict['love']
{'VB': 0.7672413793103449, 'NN': 0.23275862068965517}
```

Both interpretations do have to be in the dictionary: ‘loves’ doesn’t occur even once as a noun in the UD treebank.

---

Allan Ramsay, II Morphosyntax -319- Special cases
How do wide-coverage linguistically-sound grammars behave?

(19) I enjoyed the main course. (comes out right, fast as you could want)

(20) I thought the dessert was disgusting. (comes out right, fast as you could want (0.034 sec ≈ 175 words/sec))

(21) I enjoyed the main course but I thought the dessert was disgusting.

(22) I enjoyed the main course but the dessert I thought was disgusting.
(23) I saw the man.

(24) I saw the man in the park

(25) ...

(26) I saw the man with a big nose in the park with a telescope which he had stolen from his friend. (First analysis in 1.43 sec, 104 analyses in total after 24.3 sec, correct one is number 35)
(27) I believe the man knows she is a fool.

(28) I believe the man who she was talking to knows she is a fool.

(29) I believe the man who you say she was talking about me to knows she is a fool. (First analysis in 2.3 sec, 4 analyses in total after 71 sec, correct one is number 2)

(30) I believe the man who you say she were talking about me to knows she is a fool. (No analyses. Long time).
They get slower and slower as the text gets longer. Complexity is hard to analyse: if you ignore lexical ambiguity and ‘out-of-place items’ it’s at least $n^3$ in the length of the sentence; but it’s lexical ambiguity and out-of-place items that make parsing hard, so you can’t really ignore them.

They’re fragile.

(31) a. I believe the man who you were talking about is a fool.
   b. I believe the man that you were talking about is a fool.
   c. I believe the man you were talking about is a fool.

They generate lots of interpretations, many of which can look very weird.
Is there something simpler we can back off to when our linguistic grammar breaks down?

Grammars describe sequences of items: so an NP can be made of an optional determiner, followed by some optional adjectives, followed by a noun.

\[ np = \text{det? adj* noun} \]

A VP is made out of a series of verbs followed by an appropriate set of items

\[ vp = (\text{verb NP}) \mid (\text{verb NP NP}) \]

An S is made out of an NP followed by a VP

\[ s = \text{NP VP} \]
So if you had a string which had been correctly pre-tagged, you could apply the regex for Ss to it to find the sentences.

<DT0>This</DT0>
<NN1>virus</NN1>
<VVZ>affects</VVZ>
<AT0>the</AT0>
<NN1>body</NN1>
<POS>’s</POS>
<NN1>defence</NN1>
<NN1>system</NN1>
...

Allan Ramsay, II Morphosyntax -325- Regular expressions
Dead easy to write. And regexes can be applied very efficiently. Can’t they?

# First few are to match BNC tags
tags = {# First few are to match BNC tags
    'noun': tag('NN.'),
    'adj': tag('AJ.'),
    'det0': tag('(AT|D.).'),
    ...

# and then ones for recognising phrases
    'nmod': 'adj1|noun',
    'np0': '((det0?nmod*noun)|name|pron)',
    ...
}
Try these on a couple of representative sentences

(32) This virus affects the body ’s defence system so that it can not fight infection.

(33) ACET volunteers work as part of a team and provide help in many different ways to ensure that people do n’t spend time in hospital unnecessarily.
This virus affects the body’s defence system so that it can not fight infection.

ACET volunteers work as part of a team and provide help in many different ways to ensure that people don’t spend time in hospital unnecessarily.

What have I missed? That ‘the body’s’ is part of the larger phrase ‘the body’s defence system’
This virus affects the body's defence system so that it can not fight infection.

ACET volunteers work as part of a team and provide help in many different ways to ensure that people don't spend time in hospital unnecessarily.

What have I missed this time? That 'of a team' is part of the larger phrase 'part of a team', and that 'in hospital' is part of 'time in hospital'.
This virus affects the body’s defence system so that it can not fight infection.

ACET volunteers work as part of a team and provide help in many different ways to ensure that people don’t spend time in hospital unnecessarily.

What have I missed this time? ...
Unreadable! The specification of 'np' above is

((((((det0?((adj1| noun))*noun)| name| pron)))possmarker))| det0| card)?((adj1| noun)*noun)| pron| name))((prepp)((((((det0?((adj1| noun))*noun)| name| pron)))possmarker))| det0| card)?((adj1| noun)*noun)| pron| name))((conj)(((det0?((adj1| noun))*noun)| name| pron)))*)))(conj((((((((((noun))*noun)| name| pron))))possmarker))| det0| card)?((adj1| noun)*noun)| pron| name))((prepp)((((((det0?((adj1| noun))*noun)| name| pron)))possmarker))| det0| card)?((adj1| noun)*noun)| pron| name))((conj)(((det0?((adj1| noun))*noun)| name| pron)))*))*)

Specification for 's' is just impossible. Hence undebuggable.
This is the HTK approach to grammar!

Not as lightning fast as we hoped. Proper regexes are linear time in the length of pattern, applying them at every point in a string is $|\text{length of pattern}| \times |\text{length of string}|$. But Python/PERL/Java implementations use extensions which are very convenient but which introduce choice points, so they do have to backtrack, and this can be slow.

Hard to get it reveal the structure. It's a recogniser, not a parser.
No recursion !!!!!

np  ==>  det, nn
det  ==>  np, pos
...
nn  ==>  nn, pp
pp  ==>  prep, np
np  ==>  det, nn
...
s  ==>  np, vp
vp  ==>  verb, s
No ‘movement’:

(40) a. He’s arriving this afternoon.
    b. Is he arriving this afternoon?

(41) a. I enjoyed the main course, but I thought the dessert was disgusting.
    b. I enjoyed the main course, but the dessert I thought was disgusting.
• Regexes are a very effective way of finding basic building blocks: basic noun phrases, auxiliary+verb sequences, . . .

• Large regexes are unreadable, and are not as fast you as expected

• They cannot handle recursion!

• They will not give you alternative options. They do backtracking internally, but they only produce one answer. So only use them where you trust them
Deterministic dependency parsing (Nivre et al. 2007; Nivre 2003)

Regexes aren’t very expressive. Could try to find a better way of writing decomposition rules.

If you apply the wrong rule you’ll get stuck. So standard practice is to write ‘non-deterministic’ parsing algorithms. But non-deterministic parsing algorithms can be very slow (see above)

And anything which doesn’t fit my grammar won’t get an analysis at all.
There are three things we’d like to optimise:

- **Accuracy**: if my program produces an analysis, I’d like it to be ‘right’

- **Robustness**: I’d like to get some kind of analysis even for texts that don’t fit the norms of the language

- **Speed**: obvious enough. Classical parsing algorithms have horrible complexity.

Standard approaches concentrate on accuracy, and then make compromises to improve the other criteria (e.g. using regexes improves speed). Is there anything else we could do?
‘Dependency grammar’: what I care about is relations between words.

- Every word except one has a parent
- No word has more than one parent

(between them these mean there are no cycles)

- Projectivity: you can draw the tree as below without any arcs crossing
(42) I could see the distant mountains with a powerful telescope

(43) I could see the old man with a big nose
Long distance dependency ≠ non-projectivity. But even fairly simple examples of long-distance dependency cause problems, and ones with crossed-arcs are even worse.
(44) a. analyse('I saw the man who she loves').

\[
\begin{align*}
\text{saw} \\
\text{man} & \text{I} \\
\text{object} & \text{agent} \\
\text{headlessEq} & \text{the} \\
\text{headlessMod} & \text{specifier} \\
\text{love, S} & \\
*\text{headlessEq} & \\
\text{who} & \text{she} \\
\text{object} & \text{agent} \\
\end{align*}
\]
b. ?- analyse('I ate the peach which she said he wanted.').

```
  ate
  mood
  peach, object
  I
  agent
  headlessEq
  headlessMod
  said
  *headlessEq
  want,ed
  she
  event
  agent
  which
  he
  object
  agent
```

Allan Ramsay, II Morphosyntax

-342-

Regular expressions
latexconll(55).

I eat the peach which she say he want.
A linear time algorithm for assigning dependency relations:

- Three data structures:
  - an **input list** of words that you haven’t looked at yet
  - a **stack** of words that you’ve looked at but haven’t given a parent to
  - a collection of **dependency relations**. Every word except one is dependent on exactly one other word.
Three possible operations (this is one version of ‘MALT’ (Nivre 2006, 2010). There are numerous others. This one is the one they call ‘arc eager’):

‘**shift:**’ move a word from the input list to the stack
‘**leftArc:**’ set the top item on the stack as a dependent of the head of the input list and remove it from the stack.
‘**rightArc:**’ set the head of the input list as a dependent of the top item on the stack and replace it by the top item on the stack.
The algorithm: make some decision about which of these to do next (use an ‘oracle’). For a sentence of length N, you can only do the first one at most N times, and each of the others removes a word from consideration so they can only be done at most N times between them.

So it’s linear in the length of the sentence. You’re not going to get anything faster than that. You have to look at every word at least once!
You can always do one of them. Sometimes the only thing you can do is shift (if there’s nothing on the stack). Sometimes the only thing you should do is rightArc (if there’s one item in the queue and something on the stack).

So given a sentence, there will be a sequence of moves that produces a dependency tree (in fact there will be lots and lots of sequences, each of which will terminate having produced a dependency tree).

So it’s robust.
Accuracy??  I have to make the right choice at every stage. Why should that be easier here than when I’ve got a set of rules and I need to choose the right one at every stage?
Input: ['the', 'man', 'ate', 'it']
Stack: []
Relations: []

shift

Input: ['man', 'ate', 'it']
Stack: ['the']
Relations: []

leftArc(word('the', False), det, word('man', False))

Input: ['man', 'ate', 'it']
Stack: []
Relations: ['the <det< man']
shift

Input: ['ate', 'it']
Stack: ['man']
Relations: ['the <det< man']

leftArc(word(man, False), subj, word(ate, False))

Input: ['ate', 'it']
Stack: []
Relations: ['the <det< man', 'man <subj< ate']

shift

Input: ['it']
Stack: ['ate']
Relations: ['the <det< man', 'man <subj< ate']
rightArc(word(ate, False), obj, word(it, False))

Input: []
Stack: ['it', 'ate']
Relations: ['the <det< man’, ’man <subj< ate’, ’ate >obj> it’]
reduce

Input: []
Stack: ['ate']
Relations: ['the <det< man’, ’man <subj< ate’, ’ate >obj> it’]
Input: ['the', 'old', 'man', 'ate', 'it']
Stack: []
Relations: []

shift

Input: ['old', 'man', 'ate', 'it']
Stack: ['the']
Relations: []

shift

Input: ['man', 'ate', 'it']
Stack: ['old', 'the']
Relations: []
leftArc(word(old, False), mod, word(man, False))

Input: ['man', 'ate', 'it']
Stack: ['the']
Relations: ['old <mod< man']

leftArc(word(the, False), det, word(man, False))

Input: ['man', 'ate', 'it']
Stack: []
Relations: ['old <mod< man’, 'the <det< man']

shift

Input: ['ate', 'it']
Stack: ['man']
Relations: ['old <mod< man’, 'the <det< man']

...
I’m allowed to look at the whole of the state to decide what to do next (a state isn’t that complicated a thing)
So I could write a set of rules:

\{
\text{input: [DET, ADJ*, NN1, ...], stack: [NN1, ...]} \Rightarrow \text{shift}
\}
\{
\text{input: [DET, ...], stack: [NN?, ...]} \Rightarrow \text{leftArc(det)}
\}
\{
\text{input: [?, ...], stack: [?, ...]} \Rightarrow \text{leftArc(mod)}
\}
\{
\text{input: [?, ...], stack: []} \Rightarrow \text{shift}
\}

It’ll be robust & fast (it’s always going to be fast, the last two rules guarantee that it’s robust). If I write good rules it’ll be accurate.
But writing good rules is difficult. More difficult than writing good ordinary grammar rules, because the long term effects of a rule are hard to see.
I could learn a set of rules!
Given an input sentence and a tree, I can discover a sequence of actions that would obtain the tree from the sequence.

Given a tree, represented as a set of relations, do

1. if there is a relation with the head of the queue as hd and the top of the queue as dtr, try leftArc
2. if there is a relation with the top of the stack as hd and the head of the queue as dtr, try rightArc
3. otherwise try shift

This algorithm is not deterministic. See the coursework
And then I could turn that sequence into a rule:

\{\text{input:} ['old', 'man', 'ate', 'it'], \text{stack:} ['the'], \text{relations:} []\}

\Rightarrow \text{shift}

I could generalise this by replacing words by their parts of speech:

\{\text{input:} \text{[AJ0, NN1, VVD, PRN]}, \text{stack:} \text{[AT0]}, \text{relations:} []\}

\Rightarrow \text{shift}

I could decide that you don’t need to look more than two words to the right:

\{\text{input:} \text{[AJ0, NN1, ...]}, \text{stack:} \text{[AT0, ...]}, \text{relations:} []\}

\Rightarrow \text{shift}
Learning from a treebank

Find a set of trees that someone has kindly generated. The Penn Wall Street Journal Treebank (http://www.cis.upenn.edu/~treebank/home.html) is a well-known resource (2,499 stories from the Wall Street Journal), with two disadvantages:

- You have to pay for it. Quite a lot. Fortunately there’s a subset of about 10% (3914 sentences, 90211 words) in the NLTK, which is enough to do useful experiments with.

- It’s a set of phrase structure trees. We need dependency trees.
Headed phrase structure trees \(\equiv\) dependency trees

```
S
  NP-SBJ
    PRP
      It
  VP
    VBZ
      has
    NP
      DT
      no
      NN
      bearing
    PP-DIR
      IN
      on
    NP
      PRP
      our
      NN
      work
      NN
      force
      NN
      today
```
What's the most important thing in this tree?
What's the most important thing in this tree?
What's the most important thing in the VP?

[S
  NP-SBJ
    PRP
      It
    VP
      VBZ
        has
      NP
        DP
          DT
            no
          NP
            NN
              bearing
          PP-DIR
            IN
              on
            NP
              NP
                PRP
                  our
                NN
                  work
                NN
                  force
            NP-TMP
              NN
                today]
What’s the most important thing in the VP?
1. Find the most important daughter of the current tree
   (a) If it’s a word, return it
   (b) Otherwise convert it to a dependency tree
2. Convert all the other dtrs to dependency trees and make them subtrees of this one
Find the most important daughter: use a `head percolation table`.

HPT = {
"ADJP": ['JJ', 'ADJP', 'VBD', 'VBG', 'VBN', ...],
"NP": ['NN', 'NNP', 'NNPS', 'NNS', 'PRN', ...],
"PP": ['IN', 'TO', ...],
"S": ['VP', 'S', 'SBAR', 'SBARQ', 'SINV', ...],
}

It’s important to get the HPC table right. Any table will produce dependency trees, but if it’s not linguistically plausible then they won’t be consistent, and it will be impossible to learn from them.
It has bearing on our work today.
Use the forced parsing strategy outlined above (only do leftArcs and rightArcs if they produce relations that are in your target tree).

Re-run the successful action sequence and record what action you performed and the state when you performed it.

Use these to learn a set of rules (Nivre used nearest neighbour classification in this thesis. Other people have used support vector machines and DNNs (Andor et al. 2016). Examples below are done with ID3 (Quinlan 1986)—easy to implement, easy to read the inferred rules, **quick to train**, usually competitive with other algorithms (Jaf and Ramsay 2013). In WEKA (Witten and Frank 2005) it's called J45.)
<table>
<thead>
<tr>
<th>Q0</th>
<th>Q1</th>
<th>S0</th>
<th>S1</th>
<th>DIST</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR</td>
<td>VB</td>
<td>*</td>
<td>*</td>
<td>N</td>
<td>shift</td>
</tr>
<tr>
<td>VB</td>
<td>DT</td>
<td>PR</td>
<td>*</td>
<td>-1</td>
<td>leftArc</td>
</tr>
<tr>
<td>VB</td>
<td>DT</td>
<td>*</td>
<td>*</td>
<td>N</td>
<td>shift</td>
</tr>
<tr>
<td>DT</td>
<td>NN</td>
<td>VB</td>
<td>*</td>
<td>-1</td>
<td>shift</td>
</tr>
<tr>
<td>NN</td>
<td>IN</td>
<td>DT</td>
<td>VB</td>
<td>-1</td>
<td>leftArc</td>
</tr>
<tr>
<td>NN</td>
<td>IN</td>
<td>VB</td>
<td>*</td>
<td>-2</td>
<td>shift</td>
</tr>
<tr>
<td>IN</td>
<td>PR</td>
<td>NN</td>
<td>VB</td>
<td>-1</td>
<td>shift</td>
</tr>
<tr>
<td>PR</td>
<td>NN</td>
<td>IN</td>
<td>NN</td>
<td>-1</td>
<td>shift</td>
</tr>
<tr>
<td>NN</td>
<td>NN</td>
<td>PR</td>
<td>IN</td>
<td>-1</td>
<td>leftArc</td>
</tr>
<tr>
<td>NN</td>
<td>NN</td>
<td>IN</td>
<td>NN</td>
<td>-2</td>
<td>shift</td>
</tr>
<tr>
<td>NN</td>
<td>NN</td>
<td>IN</td>
<td>NN</td>
<td>-1</td>
<td>rightArc</td>
</tr>
<tr>
<td>NN</td>
<td>NN</td>
<td>IN</td>
<td>NN</td>
<td>-2</td>
<td>rightArc</td>
</tr>
<tr>
<td>NN</td>
<td>*</td>
<td>NN</td>
<td>IN</td>
<td>-2</td>
<td>rightArc</td>
</tr>
<tr>
<td>NN</td>
<td>*</td>
<td>IN</td>
<td>NN</td>
<td>-2</td>
<td>rightArc</td>
</tr>
<tr>
<td>IN</td>
<td>*</td>
<td>NN</td>
<td>VB</td>
<td>-1</td>
<td>rightArc</td>
</tr>
<tr>
<td>NN</td>
<td>*</td>
<td>VB</td>
<td>*</td>
<td>-2</td>
<td>rightArc</td>
</tr>
<tr>
<td>VB</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>N</td>
<td>shift</td>
</tr>
</tbody>
</table>
Q0, Q1 are the tags of the first and second items on the queue (if any), S0 and S1 are the tags of the first and second items on the stack (if any), DIST is the distance between the two items being combined, ACTION is the action performed in this situation.

The set of features you use to describe a situation is crucial: too few and the machine learning algorithm won’t be able to distinguish between different situations, too many and it won’t be able to spot common patterns.
Now give all this to your favourite machine learning algorithm. I like ID3 because it gives me a representation as a set of questions, which I can apply very quickly.

(F0, F1, F2, F3, F4 = Q1, Q2, S1, S2, DIST)
What’s the top item on the stack
-> TH  Is it tagged as "TH" (i.e. the word "that")?
  ?F0=??? Yes: then what’s the head of the queue
  -> FW  Is it tagged as "FW" (don’t know what that is)?
     {shift:1.000} Do a shift
  -> VB  Is it a verb?
  ?F1=??? What’s the second item on the queue?
     -> VB  Is it a verb?
       ?F3=??? What’s the second item on the stack?
         -> PR  Is it a pronoun?
           {leftArc:1.000} Make "that" a dtr of the verb at the head of the Q
         -> VB
           ?F4=???
             -> -1
               {leftArc:1.000}
             -> -2
               {shift:0.105, leftArc:0.895} (leftArc is the best idea, but it may be wrong)
               -> {shift:0.103, leftArc:0.897}
... ... ... ...
The classifier itself is pretty accurate (around 88% of situations are allocated the right class (= right action)).

But when we use it as an oracle for guiding MALT’s decisions, the ‘parsing accuracy’ on the UD treebank goes down to 73.7%, because once it’s made one mistake it will be in a situation which may be irrecoverable (e.g. something which itself should have daughters has been given a head). So the parsing accuracy goes down as sentences get longer.

73.7% isn’t amazing, but then it’s very challenging data. And it is robust (by definition) and FAST (11.5K words/second).

---

Since the lecture I’ve retrained it using a DNN (two hidden layers, 200 and 40 nodes, but varying this doesn’t make much difference): classifier accuracy is 87.8%, parsing accuracy 71.1%. Using different classifiers makes little difference – the task just is difficult.
(when you’re using a machine learning algorithm, you want to use as much data as possible for training. But you also want to use as much data as possible for testing, and you must keep them separate.

‘N-fold cross validation’: suppose you have $K$ data points. Take points 1 to $K/N$ out for testing, train on the remainder. Take points $K/N$ to $2K/N$ out for testing, train on the remainder. . . . . Take points $(N-1)K/N$ to $K$ out for testing, train on the remainder. This lets you use all the data points for testing, and all of them for training, without testing on something you have used for training)
Classifier accuracy has levelled off. Parsing accuracy may have been over-trained. No point in getting more data (use better features, use a better classifier, . . . )
Use better features?

- Part of speech tags for first three items on the queue and top two items on the stack. Fewer items – not looking at enough information, more items – too sparse to learn from.

- Are the words that are to be combined adjacent, or one, two, three or more words apart? Smaller threshold – not looking at enough information, larger threshold – too sparse to learn from.

- How many dtrs do they already have? (curiously this makes the classifier accuracy better but the parser accuracy worse).

- What are the actual words? (disastrous: to use actual words for anything, you need BNC-sized corpora.)
Trinity Industries Inc. said it reached a preliminary agreement to sell 500 railcar platforms to Trailer Train Co. of Chicago.
(46) Terms were n’t disclosed

```
  were:VB
  /    \\
Terms:NN n’t:RB disclosed:VB
  \\
:NN
```
Trinity said it plans to begin delivery in the first quarter of next year.

Note that ‘plans’ is ambiguous—if we tag ‘its plans’ we get [[‘its’, ‘DT’], [‘plans’, ‘NN’]].
(48) Sen. Kennedy said in a separate statement that he supports legislation to give the president line-item veto power, but that it would be a “reckless course of action” for President Bush to claim the authority without congressional approval.
(49) a. analyse('I saw the man who she loves').

\[ \text{I} \rightarrow \text{saw} \rightarrow \text{the} \rightarrow \text{man} \rightarrow \text{who} \rightarrow \text{she} \rightarrow \text{love} \]
b. | ?- analyse('I ate the peach which she said he wanted.').

```
| ate  
mood  
| peach, object  
| headlessEq  
| headlessMod  
said  
| *headlessEq  
| want,ed  
| event  
| which  
| he  
| object  
| agent  
| she  
| agent  
| I  
| agent
```

Allan Ramsay, II Morphosyntax -381- Long-distance dependency revisited
It is **not possible** to get this tree using the standard operations. Try it.
There are different ways of dealing with this. Simplest change to the algorithm is to allow `leftArc` to combine the head of the queue with an item somewhere buried in the stack, and to allow `rightArc` to combine the top of the stack with something deep in the queue.

```python
s = malt.STATE('I am walking')
s.shift()
s.shift()
s.shift()
s.leftArc(subj, 1)  # Use item with one thing above it on the stack
s.rightArc()
```
More options

= more difficult to make the right choice
= more difficult to learn rules for making the right choice

Ideal: combine it with a grammar. Let the grammar penalise moves that violate its rules.
MALT gets worse as the sentence gets longer. It’s working its way from left to right, and each time it makes an attachment it changes the situation.

But its next decision is based on what the current situation looks like.

Once it’s made a mistake, it will have an incorrect view of the situation, so it will probably do the wrong thing.
What’s the difference between a graph and a tree?

A graph has loops in it and a tree doesn’t (and a tree has a node designated as its root).
Make a graph of every possible head:dtr pair, preferably with some kind of probability/confidence score.

Complexity: $N$ words $\times$ $R/N$ rules headed by each word $\times$ length of antecedent of the rule. Quite quick.
That would be a parsing algorithm, were it not for the fact that you might end up with a graph rather than a tree.

Making this graph and then making good decisions about which arcs to remove is a parsing algorithm (McDonald et al. 2005). No special moves needed to allow for long-distance dependencies.
Given a graph, find the cheapest incoming arc for each node.

Make them into a graph. Does it contain a loop? If not, it’s the cheapest spanning tree for this graph (it’s a tree, it’s got the cheapest set of arcs)

It contains a loop. Replace that loop by a single node (remember what you did). You’ve got a tree with one less loop. Everybody who led into someone in the loop now leads to this node. Decrease the cost of each of these arcs by the cheapest way out of them.

Do it to that tree. At the end, replace the invented nodes by the trees that they correspond to.
Allan Ramsay, II Morphosyntax
cycle

d \rightarrow b : 0.25

c \rightarrow d : 1.00

b \rightarrow c : 2.00
new links

\[ d \rightarrow j : 2.00 \Rightarrow X1 \rightarrow j : 2.00 \]
\[ a \rightarrow b : 2.00 \Rightarrow a \rightarrow X1 : 1.75 \]
\[ c \rightarrow j : 3.00 \Rightarrow X1 \rightarrow j : 3.00 \]
\[ c \rightarrow k : 2.00 \Rightarrow X1 \rightarrow k : 2.00 \]
\[ f \rightarrow c : 9.00 \Rightarrow f \rightarrow X1 : 7.00 \]

cycle

\[ d \rightarrow b : 0.25 \]
\[ c \rightarrow d : 1.00 \]
\[ b \rightarrow c : 2.00 \]
Edmond's algorithm
cycle

\[ x \rightarrow y : 1.00 \]
\[ z \rightarrow a : 0.50 \]
\[ y \rightarrow z : 1.00 \]
\[ a \rightarrow x : 1.00 \]
new links

\[ j \rightarrow a : 1.00 \implies j \rightarrow X2 : 0.50 \]
\[ i \rightarrow a : 5.00 \implies i \rightarrow X2 : 4.50 \]
\[ top \rightarrow a : 2.00 \implies top \rightarrow X2 : 1.50 \]
\[ a \rightarrow X1 : 1.75 \implies X2 \rightarrow X1 : 1.75 \]

cycle

\[ x \rightarrow y : 1.00 \]
\[ z \rightarrow a : 0.50 \]
\[ y \rightarrow z : 1.00 \]
\[ a \rightarrow x : 1.00 \]
Edmond's algorithm
Allan Ramsay, II Morphosyntax

‘Edmond's algorithm’
cycle

\[ X_1 \rightarrow j : 3.00 \]
\[ j \rightarrow X_2 : 0.50 \]
\[ X_1 \rightarrow j : 2.00 \]
\[ X_2 \rightarrow X_1 : 1.75 \]
new links

\[
i \rightarrow X_2 : 4.50 \implies i \rightarrow X_3 : 4.00
\]
\[
X_1 \rightarrow k : 2.00 \implies X_3 \rightarrow k : 2.00
\]
\[
top \rightarrow X_2 : 1.50 \implies top \rightarrow X_3 : 1.00
\]
\[
f \rightarrow X_1 : 7.00 \implies f \rightarrow X_3 : 5.25
\]

cycle

\[
X_1 \rightarrow j : 3.00
\]
\[
j \rightarrow X_2 : 0.50
\]
\[
X_1 \rightarrow j : 2.00
\]
\[
X_2 \rightarrow X_1 : 1.75
\]
Edmond's algorithm
put these links back

\( i \rightarrow X2 : 4.50 \Rightarrow i \rightarrow X3 : 4.00 \)
\( X1 \rightarrow k : 2.00 \Rightarrow X3 \rightarrow k : 2.00 \)
\( top \rightarrow X2 : 1.50 \Rightarrow top \rightarrow X3 : 1.00 \)
\( f \rightarrow X1 : 7.00 \Rightarrow f \rightarrow X3 : 5.25 \)

**cycle**

\( X1 \rightarrow j : 3.00 \)
\( j \rightarrow X2 : 0.50 \)
\( X1 \rightarrow j : 2.00 \)
\( X2 \rightarrow X1 : 1.75 \)
Allan Ramsay, II Morphosyntax

'Edmond's algorithm'
put these links back

\[ j \rightarrow a : 1.00 \Rightarrow j \rightarrow X2 : 0.50 \]
\[ i \rightarrow a : 5.00 \Rightarrow i \rightarrow X2 : 4.50 \]
\[ top \rightarrow a : 2.00 \Rightarrow top \rightarrow X2 : 1.50 \]

\[ a \rightarrow X1 : 1.75 \Rightarrow X2 \rightarrow X1 : 1.75 \]

**cycle**

\[ x \rightarrow y : 1.00 \]
\[ z \rightarrow a : 0.50 \]
\[ y \rightarrow z : 1.00 \]
\[ a \rightarrow x : 1.00 \]
Allan Ramsay, II Morphosyntax

‘Edmond's algorithm’
put these links back

\[ d \to j : 2.00 \implies X_1 \to j : 2.00 \]
\[ a \to b : 2.00 \implies a \to X_1 : 1.75 \]
\[ c \to j : 3.00 \implies X_1 \to j : 3.00 \]
\[ c \to k : 2.00 \implies X_1 \to k : 2.00 \]
\[ f \to c : 9.00 \implies f \to X_1 : 7.00 \]

**cycle**

\[ d \to b : 0.25 \]
\[ c \to d : 1.00 \]
\[ b \to c : 2.00 \]
Allan Ramsay, II Morphosyntax

Edmond's algorithm
Suppose you had a set of rules that looked like the following:

det:SPEC adj* noun:HD 1.0;
adj:MOD adj* noun:HD 1.0;
noun:SUBJ verb:HD 0.2;
noun:MOD noun:HD 0.3;
verb:HD noun:OBJ 1.0;
verb:HD det? adj* noun:OBJ 0.7;
verb:HD (.*)* noun:IOBJ 0.5;
verb:HD (.*)* verb:COMP 0.5;
verb:ARG (.*)* stop:HD 0.6;
If you applied these to a sequence like *det:a adj:big adj:fat noun:man verb:stole det:the noun:car stop:.* you'd end up with a dependency tree (choose the full stop to be the designated element): each element has one parent, it's a pretty sensible looking tree.

```
stop:-7
  \--- verb:stole-4
    |   \--- noun:car-6
    |       0.5
    |        \--- det:the-5
    |            1.0
    \--- noun:man-3
        1.0
          \--- adj:big-1
              0.5
              \--- det:a-0
                  1.0
                  \--- adj:fat-2
                      1.0
```

Allan Ramsay, II Morphosyntax

'Edmond's algorithm'

and with a slightly more complicated grammar:

det:SPEC adj* noun:HD 1.0;
adj:MOD adj* noun:HD 1.0;
noun:SUBJ verb:HD 0.2;
noun:MOD noun:HD 0.3;
verb:HD noun:OBJ 1.0;
**noun:HD verb:MOD 0.5**;
verb:HD det? adj* noun:OBJ 0.7;
verb:HD (.*)* noun:IOBJ 0.5;
verb:HD (.*)* verb:COMP 0.5;
verb:ARG (.*)* stop:HD 0.6;
Allan Ramsay, II Morphosyntax

'Edmond's algorithm'
Edmond's algorithm
cycle

\[ verb : knows \rightarrow noun : man \rightarrow 2 \rightarrow noun : 1 : 0.20 \]

\[ noun : man \rightarrow 1 \rightarrow verb : knows \rightarrow 2 : 0.50 \]
new links

noun : man − 1 → det : the − 0 : 1.00 ⇒ X9 → det : the − 0 : 1.00
verb : knows − 2 → verb : gave − 5 : 0.50 ⇒ X9 → verb : gave − 5 : 0.50
stop : . − 10 → verb : knows − 2 : 0.60 ⇒ stop : . − 10 → X9 : 0.10
verb : knows − 2 → noun : woman − 4 : 0.70 ⇒ X9 → noun : woman − 4 : 0.70
verb : knows − 2 → noun : present − 9 : 0.50 ⇒ X9 → noun : present − 9 : 0.50

cycle

verb : knows − 2 → noun : man − 1 : 0.20
noun : man − 1 → verb : knows − 2 : 0.50
cycle

verb: gave − 5 → noun: woman − 4 : 0.20
noun: woman − 4 → verb: gave − 5 : 0.50
Allan Ramsay, II Morphosyntax

Stop: -10

Verb: gave

Noun: woman
Det: a

Noun: girl
Det: the

Noun: present
Det: a

Det: the

'Edmond's algorithm'
new links

verb: gave − 5 → noun: present − 9 : 0.50 \(\Rightarrow\) X10 \(\Rightarrow\) noun: present − 9 : 0.50

X9 → verb: gave − 5 : 0.50 \(\Rightarrow\) X9 \(\Rightarrow\) X10 : 0.00
	noun: woman − 4 → det: a − 3 : 1.00 \(\Rightarrow\) X10 \(\Rightarrow\) det: a − 3 : 1.00

X9 → noun: woman − 4 : 0.70 \(\Rightarrow\) X9 \(\Rightarrow\) X10 : 0.50

verb: gave − 5 → noun: girl − 7 : 0.70 \(\Rightarrow\) X10 \(\Rightarrow\) noun: girl − 7 : 0.70

stop: . − 10 → verb: gave − 5 : 0.60 \(\Rightarrow\) stop: . − 10 \(\Rightarrow\) X10 : 0.10

cycle

verb: gave − 5 → noun: woman − 4 : 0.20

noun: woman − 4 → verb: gave − 5 : 0.50
Edmond's algorithm
put these links back

verb: gave − 5 → noun: present − 9 : 0.50 ⇒ X10 → noun: present − 9 : 0.50
X9 → verb: gave − 5 : 0.50 ⇒ X9 → X10 : 0.00

noun: woman − 4 → det: a − 3 : 1.00 ⇒ X10 → det: a − 3 : 1.00
X9 → noun: woman − 4 : 0.70 ⇒ X9 → X10 : 0.50

verb: gave − 5 → noun: girl − 7 : 0.70 ⇒ X10 → noun: girl − 7 : 0.70

stop: . − 10 → verb: gave − 5 : 0.60 ⇒ stop: . − 10 → X10 : 0.10

cycle

verb: gave − 5 → noun: woman − 4 : 0.20

noun: woman − 4 → verb: gave − 5 : 0.50
stop: -10

X9

verb: gave-5

noun: girl-7

noun: woman-4

noun: present-9

det: the-0

det: the-6

det: a-3

det: a-8

'Edmond's algorithm'
put these links back

noun : man − 1 → det : the − 0 : 1.00 ⇒ X9 → det : the − 0 : 1.00
verb : knows − 2 → verb : gave − 5 : 0.50 ⇒ X9 → verb : gave − 5 : 0.50
stop : . − 10 → verb : knows − 2 : 0.60 ⇒ stop : . − 10 → X9 : 0.10
verb : knows − 2 → noun : woman − 4 : 0.70 ⇒ X9 → noun : woman − 4 : 0.70
verb : knows − 2 → noun : present − 9 : 0.50 ⇒ X9 → noun : present − 9 : 0.50
cycle
verb : knows − 2 → noun : man − 1 : 0.20
noun : man − 1 → verb : knows − 2 : 0.50
Where would you get such rules from?

If you had a set of dependency trees, you could find all the dependency pairs, and then you could find the words in between.

So if you had the tree at the end of the last trace of this algorithm, you’d have pairs like \texttt{verb:gave-5 \rightarrow noun:present-9}, where the words in between are \texttt{det:the-6, noun:girl-7, det:a-8}.

Abstracting away the details, this would become the rule \texttt{verb:HD, det, noun, det, noun:DTR}. And then you could count how many times you saw that sequence in a tree where the verb was the head of the final noun and how many times you saw it in total, and that would give you the probabilities.
Handles long-distance dependencies without any extra machinery

Complexity is slightly worse than MALT (Edmond’s algorithm is linear in the number of arcs in the graph, but that can be more than the number of words, and making the graph can also be time-consuming unless you implement it very neatly), but the extra steps required to let MALT deal with long-distance dependencies makes MALT worse anyway.

Doesn’t have MALT’s problems with early mistakes having knock-on effects.

Simple-minded training is quick.
• Dependency grammar is good for
  – free word-order languages
  – extragrammatical sentences (e.g. speech)

• The trade-off is three ways: accuracy, speed, robustness. Nivre-style parsing is fast and robust, but it's hard to make it accurate

• A given tree will have a single derivation path, which can be found & used as input to a classifier

• Conversion of headed phrase-structure trees to dependency trees

• Long-distance dependencies cannot be handled with the three basic MALT operations. There are several solutions: looking deep into the queue or stack for daughter is the easiest to understand.

• Graph-based algorithms have slightly higher complexity, are less prone to knock-on errors, don’t need special machinery to do long-distance dependencies
IV SEMANTICS & INFERENCE
What can you do with it?

Lots of work to find the words and their syntactic relations

Why?

Similarity-based tasks, pairwise tasks, deep reasoning
**** is a bat-and-ball game played between two teams of eleven players each on a **** field, at the centre of which is a rectangular 22-yard-long (20 metres) pitch with a target at each end called the wicket (a set of three wooden stumps upon which two bails sit).

Each phase of play is called an innings, during which one team bats, attempting to score as many runs as possible, whilst their opponents bowl and field, attempting to minimise the number of runs scored. When each innings ends, the teams usually swap roles for the next innings (i.e. the team that previously batted will bowl/field, and vice versa). The teams each bat for one or two innings, depending on the type of match. The winning team is the one that scores the most runs, including any extras gained (except when the result is not a win/loss result).

Before a match begins, the two team captains meet on the pitch for the toss (of a coin), with the winner deciding which team will bat first. Two players from the batting side, and all eleven players from the bowling/fielding side, then enter the field, and play proceeds by a member of the fielding team, known as the bowler, delivering (i.e., bowling) the ball from one end of the pitch towards the wicket at the other end, which is guarded by one of the batsmen, known as the striker. The striker’s role is to strike the ball well enough to score runs, if possible, while not being dismissed. The other batsman, known as the non-striker, waits at the opposite end of the pitch near the bowler.
***s are mammals of the order Chiroptera;[a] with their forelimbs adapted as wings, they are the only mammals naturally capable of true and sustained flight. ***s are more manoeuvrable than birds, flying with their very long spread-out digits covered with a thin membrane or patagium. The smallest ***, and arguably the smallest extant mammal, is Kitti’s hog-nosed ***, which is 2934 mm (1.141.34 in) in length, 15 cm (5.91 in) across the wings and 22.6 g (0.070.09 oz) in mass. The largest ***s are the flying foxes and the giant golden-crowned flying fox, Acerodon ju***us, which can weigh 1.6 kg (4 lb) and have a wingspan of 1.7 m (5 ft 7 in).

The second largest order of mammals, ***s comprise about 20% of all classified mammal species worldwide, with over 1,200 species. These were traditionally divided into two suborders: the largely fruit-eating mega***s, and the echolocating micro***s. But more recent evidence has supported dividing the order into Yinpterochiroptera and Yangochiroptera, with mega***s as members of the former along with several species of micro***s. Many ***s are insectivores, and most of the rest are frugivores (fruit-eaters). A few species feed on animals other than insects; for example, the vampire ***s feed on blood. Most ***s are nocturnal, and many roost in caves or other refuges; it is uncertain whether ***s have these behaviours to escape predators. ***s are present throughout the world, with the exception of extremely cold regions.
How did you work out what these pages were about?

Because the other words that appear on them are words that you would associate with cricket and bats.

- all
- an
- any
- as
- at
- attempting
- bails
- ball
- bat
- bats
- batsman
- batsmen
- batted
- batting
- before
- begins
- being
- between
- bowl
- bowler
- bowling
- by
- called
- captains
- centre
- coin
- deciding
- delivering
- depending
- dismissed
- during
- e
- each
- eleven
- end
- ends
- enough
- enter
- except
- extras
- field
- fielding
- first
- for
- from
- gained
- game
- guarded
- i
- if
- including
- innings
- is known
- long
- loss
- many
- match
- meet
- member
- metres
- minimise
- most
- near
- next
- non
- not
- number of
- on
- one
- opponents
- opposite
- or
- other
- phase
- pitch
- play
- played
- players
- possible
- previously
- proceeds
- rectangular
- result
- role
- roles
- runs
- s
- score
- scored
- scores
- set
- side
- sit
- strike
- striker
- stumps
- swap
- target
- team
- teams
- that the
- their
- then
- three
How did you work out what these pages were about?

Because the other words that appear on them are words that you would associate with cricket and bats.

a all an and any as at attempting bails ball bat bats batsman batsmen batted batting before begins being between bowl bowler bowling by called captains centre coin deciding delivering depending dismissed during e each eleven end ends enough enter except extras field fielding first for from gained game guarded i if including innings is known long loss many match meet member metres minimise most near next non not number of on one opponents opposite or other phase pitch play played players possible previously proceeds rectangular result role roles runs s score scored scores set side sit strike striker stumps swap target team teams that the their then three
Very common words don’t tell me much about a document: do something to compensate for the fact that some words occur in lots of places.

Divide by the likelihood that a random document will contain this word:

innings: 4.000, wicket: 2.000, fielding: 2.000, bowling: 2.000, batsman: 2.000, striker: 1.500, versa: 1.000, bowler: 1.000, batting: 1.000, swap: 0.500, rectangular: 0.500, non: 0.500, bat: 0.312, stump: 0.200, e: 0.182, eleven: 0.167, captain: 0.167, pitch: 0.154, bowl: 0.154, minimise: 0.143, deciding: 0.143, winning: 0.125, proceeds: 0.125, bail: 0.125, vice: 0.111, scores: 0.111, metre: 0.091, team: 0.086, toss: 0.067, wooden: 0.048, dismiss: 0.048, coin: 0.048, winner: 0.040, whilst: 0.040, ball: 0.039, yard: 0.037, score: 0.036, phase: 0.033, previously: 0.031, s: 0.029, field: 0.024, . . . , a: 0.001, on: 0.001, by: 0.001, their: 0.001, not: 0.001, for: 0.001, if: 0.000, all: 0.000, with: 0.000, an: 0.000, i: 0.000, or: 0.000, and: 0.000, that: 0.000, to: 0.000
Most people divide by the log of the likelihood that a random document will contain the target word – TF-IDF.

IDF was introduced, as 'term specificity', by Karen Spärck Jones in a 1972 paper. Although it has worked well as a heuristic, its theoretical foundations have been troublesome for at least three decades afterward, with many researchers trying to find information theoretic justifications for it. Wikipedia

This is typical of this kind of work. Does it work? Yes Does anyone know why? No
We should also normalise our vectors, because when we’re comparing them we don’t want to be fooled by the fact that one document is much longer than another (so all the scores are higher)

Divide each number in the original vector \( < v_1, v_2, \ldots, v_n > \) by \( \sum_{i=1}^{n} v_i \).

This is the conditional probability that you’ll see some word in the context defined by another: how likely is a sentence about cricket to contain the word ‘stumps’?

Divide \( \log(v_i) \) by \( \sum_{i=1}^{n} \log(v_i) \) (softmax)

This **isn’t** a probability, it’s just a number between 0 and 1. But it might behave nicely.

Divide each number in the TF-IDF vector \( < v_1, v_2, \ldots, v_n > \) by \( \sum_{i=1}^{n} v_i \).

This **isn’t** a probability, it’s just a number between 0 and 1. But it might behave nicely.
But the axes didn’t have to be called X and Y: 

\[
\begin{align*}
D1: & \{ \text{pet:}100, \text{ball:}6 \}, \\
D2: & \{ \text{pet:}12, \text{ball:}75 \}, \\
D3: & \{ \text{pet:}91, \text{ball:}18 \}, \\
D4: & \{ \text{pet:}24, \text{ball:}67 \}\end{align*}
\]
Three dimensional vectors (= lines): \{D1:\{X:100, Y:6, Z:0\},
I can’t even attempt to draw things in four dimensions, let alone higher than that.

But I can see that characterising something as a set of \( N \) features with numerical values is like making an \( N \)-dimensional vector.

And then I can imagine \( N \)-dimensional analogies to the distance between the ends of two vectors, or the angle between two vectors, as measures of similarity.
To see if two documents are similar, calculate the ‘**Euclidean distance**’ between their vectors or the ‘**cosine**’ of the angle between them.

If \(< x_1, x_2, \ldots, x_n >\) and \(< y_1, y_2, \ldots, y_n >\) are two \(n\)-dimensional vectors, then the Euclidean distance between them is

\[
\sqrt{(x_1 - y_1)^2 + \ldots + (x_n - y_n)^2}
\]

(ordinary analogy to distance in 2- or 3-D space)

If \(< x_1, x_2, \ldots, x_n >\) and \(< y_1, y_2, \ldots, y_n >\) are two \(n\)-dimensional vectors, then the cosine of the angle between them is

\[
\frac{\sum x_i \times y_i}{\sqrt{\sum x_i^2} \times \sqrt{\sum y_i^2}}
\]

(less obvious how this relates to cosine in 2-D space \((\approx \frac{x}{\sqrt{x^2 + y^2}})\): slightly fiddly to show the equivalence, doesn’t matter here)
If we take it that each word in the vector is a dimension, and that if a word is missing then the score along that dimension is 0, then we can use either of these to compare two vectors. $\cos(A, B) > \cos(A, B')$ if and only if $\text{dist}(A, B) < \text{dist}(A, B')$ (subject to a condition discussed below). Most people use $\cos$: not sure why.

But a vector is a rough approximation to ‘the meaning of the document’. So comparing vectors $\approx$ computing similarity between documents.
So: get four documents from searching for ‘cricket’, five from searching for ‘bacon’. Not the top five, because some of the cricket ones on the top page were just links to pages of statistics and scorecards, but no cheating.

<table>
<thead>
<tr>
<th></th>
<th>bacon-recip</th>
<th>Bacon.html</th>
<th>bacon_week</th>
<th>Cricket.Ru</th>
<th>Cricket.ht</th>
<th>easy-risot</th>
<th>Laws_of_cr</th>
<th>Pork, Bacon, Ha</th>
<th>Rules of Cric</th>
</tr>
</thead>
<tbody>
<tr>
<td>bacon-recip</td>
<td>0.00</td>
<td>0.90</td>
<td>0.96</td>
<td>0.97</td>
<td>0.95</td>
<td>0.86</td>
<td>0.98</td>
<td>0.87</td>
<td>0.92</td>
</tr>
<tr>
<td>Bacon</td>
<td>0.90</td>
<td>0.00</td>
<td>0.93</td>
<td>0.96</td>
<td>0.82</td>
<td>0.91</td>
<td>0.91</td>
<td>0.87</td>
<td>0.92</td>
</tr>
<tr>
<td>bacon_week</td>
<td>0.96</td>
<td>0.93</td>
<td>0.00</td>
<td>0.97</td>
<td>0.97</td>
<td>0.94</td>
<td>0.98</td>
<td>0.92</td>
<td>0.93</td>
</tr>
<tr>
<td>Cricket Rul-</td>
<td>0.97</td>
<td>0.96</td>
<td>0.97</td>
<td>0.00</td>
<td>0.83</td>
<td>0.96</td>
<td>0.82</td>
<td>0.98</td>
<td>0.93</td>
</tr>
<tr>
<td>Cricket</td>
<td>0.95</td>
<td>0.82</td>
<td>0.97</td>
<td>0.83</td>
<td>0.00</td>
<td>0.93</td>
<td>0.69</td>
<td>0.93</td>
<td>0.90</td>
</tr>
<tr>
<td>easy-risot</td>
<td>0.86</td>
<td>0.91</td>
<td>0.94</td>
<td>0.96</td>
<td>0.93</td>
<td>0.00</td>
<td>0.96</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>Laws_of_cr</td>
<td>0.98</td>
<td>0.91</td>
<td>0.98</td>
<td>0.82</td>
<td>0.69</td>
<td>0.96</td>
<td>0.00</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td>Pork, Bacon, Ha</td>
<td>0.87</td>
<td>0.87</td>
<td>0.92</td>
<td>0.98</td>
<td>0.93</td>
<td>0.90</td>
<td>0.96</td>
<td>0.00</td>
<td>0.90</td>
</tr>
<tr>
<td>Rules of Cric</td>
<td>0.98</td>
<td>0.98</td>
<td>0.99</td>
<td>0.96</td>
<td>0.93</td>
<td>0.93</td>
<td>0.91</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Differences aren’t enormous: zeroes down diagonal, everything else between 0.8 and 1.
Sort them: most similar to least similar. Although the differences are small, they’re pretty reliable. I think it would only get better if we had larger sets of documents

<table>
<thead>
<tr>
<th>bacon-recipes</th>
<th>bacon-reci 0.00</th>
<th>easy-risot 0.86</th>
<th>Pork, Baco 0.87</th>
<th>Bacon 0.90</th>
<th>Cricket 0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bacon</td>
<td>Bacon 0.00</td>
<td>Cricket 0.82</td>
<td>Pork, Baco 0.92</td>
<td>Bacon 0.93</td>
<td>Laws_of_cr 0.91</td>
</tr>
<tr>
<td>bacon_week</td>
<td>bacon_week 0.00</td>
<td>Pork, Baco 0.92</td>
<td>Laws_of_cr 0.82</td>
<td>Rules of_C 0.96</td>
<td>easy-risot 0.94</td>
</tr>
<tr>
<td>Cricket Rules</td>
<td>Cricket Ru 0.00</td>
<td>Laws_of_cr 0.69</td>
<td>Bacon 0.82</td>
<td>Bacon 0.91</td>
<td>Cricket Ru 0.96</td>
</tr>
<tr>
<td>Cricket</td>
<td>Cricket 0.00</td>
<td>bacon-reci 0.86</td>
<td>Pork, Baco 0.90</td>
<td>Bacon 0.91</td>
<td>Rules of_C 0.93</td>
</tr>
<tr>
<td>easy-risotto-wi</td>
<td>easy-risot 0.00</td>
<td>Cricket 0.69</td>
<td>Rules of_C 0.91</td>
<td>Bacon 0.91</td>
<td>Rules of_C 0.91</td>
</tr>
<tr>
<td>Laws_of_cricket</td>
<td>Laws_of_cr 0.00</td>
<td>bacon-reci 0.87</td>
<td>easy-risot 0.93</td>
<td>Rules of_C 0.91</td>
<td>bacon_week 0.92</td>
</tr>
<tr>
<td>Pork, Bacon, H</td>
<td>Pork, Baco 0.00</td>
<td>Laws_of_cr 0.91</td>
<td>easy-risot 0.90</td>
<td>Cricket Ru 0.96</td>
<td>bacon-reci 0.98</td>
</tr>
<tr>
<td>Rules of Cricke</td>
<td>Rules of C 0.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What can you do with it?
Documents are similar if they contain similar words.

Words are similar if they appear in similar documents?

But you probably need a lot of ‘documents’ to get any patterns here, because most words don’t appear in most documents.
Change the terminology slightly: words are similar if they appear in similar ‘contexts’, where a document is one kind of context.

What other kinds of context might we consider?

- next word, next 5 words, previous and following 2 words, . . .

- syntactically related word: if two verbs have similar objects maybe they’re similar words?
Next few bits are done with the BNC, because it’s big and reasonably accurately tagged, so I can roughly parse it: look for verbs followed by simple NPs, pick the verb and the head noun of the NP: goes wrong if I get the head of the NP wrong, and goes sort of wrong if the NP is actually the head of a subordinate clause: from ‘I know your friend likes me’ I’ll get ‘know-friend’.
Regex for finding verb-object pairs in the BNC:

**Verb-object:**

'justverbandobj': '(MV:verb) det? (adj|noun)* (OBJ:noun)'

No need to include options about names and pronouns for the object, because they’re not going to give us the kind of information we’re looking for anyway.

Runs about 70K words/sec, so can afford to do it for quite large amounts of data (BNC in 25’).
>>> VO, OV = getVerbObjPairs(BNC)
>>> VOmodel = MODEL(VO, df0=getdf(OV), df1=getdf(VO)).tfidf().normalised()
>>> VOModel.similarity("read", "write")
0.5290727208025825

>>> print ", ".join(VOModel.intersection("read", "write", N=20))
book: 0.067 (0.052), newspaper: 0.027 (0.000), paper: 0.020 (0.005), letter: 0.019 (0.039),
report: 0.019 (0.016), article: 0.018 (0.020), papers: 0.016 (0.002), story: 0.014 (0.009), bible: 0.012 (0.001), novel: 0.011 (0.012), magazine: 0.009 (0.000),
label: 0.009 (0.001), words: 0.009 (0.006), text: 0.008 (0.003), poem: 0.007 (0.018),
ote: 0.006 (0.009), passage: 0.006 (0.001), poetry: 0.006 (0.013), chapter: 0.006 (0.001), script: 0.005 (0.007)

>>> print ", ".join(VOModel.intersection("write", "read", N=20))
book: 0.052 (0.067), letter: 0.039 (0.019), letters: 0.026 (0.005), article: 0.020 (0.018),
song: 0.019 (0.000), poem: 0.018 (0.007), essay: 0.017 (0.001), report: 0.016 (0.019),
poetry: 0.013 (0.006), program: 0.012, novel: 0.012 (0.011), note: 0.009 (0.006),
ceque: 0.009, story: 0.009 (0.014), play: 0.008 (0.004), biography: 0.007 (0.001),
script: 0.007 (0.005), history: 0.006 (0.004), reply: 0.006 (0.001)
Not bad, nearly everything in either list is some kind of document, and there’s a fair bit of overlap: but note that I have no idea what the relationship between ‘read’ and ‘write’ is. Subsumption? Opposite?
>>> print ', '.join(VOModel.intersection("eat", "chew", N=20))
food: 0.028 (0.008), meat: 0.026 (0.010), meal: 0.019, breakfast: 0.018, disorder: 0.016, sandwich: 0.014, fish: 0.014, cake: 0.012, lunch: 0.012, chocolate: 0.011 (0.004), dinner: 0.010, bread: 0.009 (0.010), habit: 0.009, grass: 0.009, lot: 0.009 (0.003), supper: 0.009, fruit: 0.008, biscuit: 0.007, cheese: 0.007, apple: 0.007 (0.003)

>>> print ', '.join(VOModel.intersection("chew", "eat", N=20))
gum: 0.235 (0.000), cud: 0.071, fat: 0.049 (0.003), peyote: 0.047, betel: 0.047, tobacco: 0.041, mouthful: 0.029 (0.001), coca: 0.020, sugar-beet: 0.015, coriander: 0.015, end: 0.014, paan: 0.012 (0.001), baccy: 0.012, bone: 0.011, piece: 0.011 (0.004), bread: 0.010 (0.009), carpet: 0.010, things: 0.010 (0.002), meat: 0.010 (0.026), paper: 0.008 (0.001)

>>> print ', '.join(VOModel.intersection("devour", "eat", N=20))
still-living: 0.058, sea-flesh: 0.058, father(s): 0.058, timberwork: 0.037, roulade: 0.037, carcass: 0.024 (0.000), horoscope: 0.021, book: 0.019, spawn: 0.019, marshland: 0.019, hunk: 0.018 (0.001), novel: 0.017, blackbird: 0.017 (0.001), treatise: 0.016, polyp: 0.016, grub: 0.016 (0.001), sheen: 0.015, morsel: 0.014 (0.002), stag: 0.014
Overlap is weaker. More things that don’t sound edible for ‘devour’.

(50) The difference is that she never felt guilty about it, whereas I, who have devoured baby and child books since my first pregnancy test was positive, do. (see also ‘novels’)

(51) A monster eating a child does not just gnaw its head, but holds its wrists to stop it wriggling.

(52) Graffiti says: “We shall eat the Arabs out of tins.”
• ‘devour’ and ‘chew’ are subsets of ‘eat’ because they’re rarer than ‘eat’: (‘devour’: 1068, ‘chew’: 2286, ‘eat’ 30309)?

• ‘devour’ is more like ‘read’ than ‘eat’: the metaphorical interpretation is more common.

```python
>>> VOModel.similarity("devour", "eat")
0.116
>>> VOModel.similarity("devour", "eat")
0.046
```
>>> VOModel.subtract("read", "write")[:20]

newspaper: 0.026, paper: 0.015, book: 0.015, papers: 0.014, bible: 0.011,
magazine: 0.009, label: 0.008, passage: 0.005, story: 0.005, text: 0.005,
sign: 0.005, chapter: 0.004, gideon: 0.004, news: 0.004, instructions:
0.004, scripture: 0.004, lesson: 0.004, copy: 0.004, file: 0.003, page:
0.003

>>> VOModel.subtract("write", "read")[:20]

letters: 0.021, letter: 0.020, song: 0.019, essay: 0.016, program: 0.012,
poem: 0.010, cheque: 0.009, poetry: 0.008, biography: 0.006, reply:
0.005, dissertation: 0.005, screenplay: 0.004, fiction: 0.004, number:
0.004, play: 0.004, opera: 0.004, english: 0.004, software: 0.004, fore-
word: 0.004

People read long things and write short ones!
Kings and queens are like powerful/important people (queens are like kings, but kings aren’t like queens)
This is from the whole BNC. 100 million words. You need a \textbf{lot} of data. The BNC isn’t enough. Recent work (\texttt{word2vec}, \texttt{GLOVE}) is carried out using corpora of 6 billion - 40 billion words, i.e. 60-400 times as much as the experiments here.
Better models

What we’ve seen suggests that this is worth pursuing

It also suggests that there is a lot of work to do!

How should we collect the coocurrence data? What is a context?

• Syntactically related items
• Weighted? local? window
• Weighted? local? window of open class words
How should we convert a cooccurrence matrix into a set of vectors?

- Use TFIDF & simple normalisation? Use softmax instead of standard normalisation? Nice and quick, not bad at simple similarity

- Use TFIDF & simple normalisation, try to smooth out the noise by doing singular value decomposition (‘latent semantic analysis’, (Deerwester et al. 1990))?

- Do K-means clustering on some version of the cooccurrence matrix, using simple cos on the original as the distance measure

- Train a neural net with one hidden layer where inputs are coocurrence vectors and outputs individual words, use softmax on the hidden layer (Mikolov et al. 2013)

- Optimise some dead complicated function of the cooccurrence matrix (Chen et al. 2017)
• Vector space model of meaning: a ‘a word is known by the company it keeps’

• The role of contexts: document, window, syntactic relationship. But if you want syntactic relationships you have to be able to parse!

• TFIDF: common words don’t tell you much. Uncommon words that occur a lot in the documents you’re interested in tell you lots
- Vector space models tell you if items come from the same domain. They do **not** tell you **how** they are related. A document that said that evolution is nonsense would come out as being closely related to one that said it was true. Two words will score highly if they are opposites, or identical, or one subsumes another, or ...
Hand-coded lexical relations

That’s about as much as you can do about word meaning by just extracting information from corpora: co-occurrence patterns ≈ relatedness

But that’s not going to get you all that far: can’t even tell from it whether ‘I ate an apple’ ⊨ ‘I ate some fruit’, or whether it’s the other way round, or whether one of them contradicts the other, or . . .

So we’d like some finer-grained information: the only way I know of getting this into a computer is by hand-coding it (how do you get it into a person: by telling them).
Knowing that a man is a kind of a human, a human is a kind of mammal, a mammal is a kind of animal, an animal is a living thing, ... is fairly useful.

Useful repository of things like this: WordNet.

150K English words, all with up-down links. Well, actually words with sideways links to ‘synsets’ – labels for concepts that several words might point to.
One word can be linked to several synsets, one synset can have multiple daughters (note alternative synsets for ‘bifurcation’ that turn up under different heads)
How can I use this for calculating relations between words?

- Two kinds of relations: similarity and entailment. WordNet is going to be much better than any vector space model for entailment. What’s it like for similarity?

- WordNet relations are between synsets: do I have to do disambiguation before I can use it?
Could hardly be easier: that’s what WordNet hypernym relations are all about. All we need is transitivity of hypernyms.

‘man’: any living or extinct member of the family Hominidae characterized by superior intelligence, articulate speech, and erect carriage

‘primate’: any placental mammal of the order Primates; has good eyesight and flexible hands and feet

homo.n.02 → hominid.n.01 → primate.n.02
But that was a chain between synsets. I will always be interested in texts, which are made of words.

Do I have to solve the problem of lexical disambiguation before I can use WordNet?
• What information do you have when you are trying to do lexical disambiguation?
  Mainly the surrounding lexical context

• When would you want to determine lexical entailments?
  When you had two sentences that might be related. In any sensible context, they would be probably be about the same kinds of thing.
(53) a. I keep my boat tied up at the riverside
   b. I keep my boat tied up at the bank

(54) a. I keep my money in my money box
   b. I keep my money at the bank

In each case you’d want to say that the (a) example entailed the (b). Why? Because the (a) example provides the context for disambiguating the (b) example (Hobbs et al. 1993)
Lightning fast. And if it's not fast enough, you can precompile all possible links into a hash table (20 seconds) and then it’s **lightning fast** (hash-table lookup, 10M/sec)

Fairly high precision. Any errors that occur are because of the ‘**abductive**’ approach to disambiguation, but it’s actually a better strategy than anything else.

But WordNet itself is prone to human error, particularly missing links

```python
>>> abductiveSubsumption("man", "human")
[u'homo.n.02']
>>> abductiveSubsumption("woman", "human")
False
```
There are a number of simple path-based similarity measurements that you can do. Find the lowest common parent, find the lowest common parent and divide by distance to the top (WUP: Wu & Palmer, Fig 23), find the shortest path (PATH), ...

\[
\begin{align*}
WUP(f, g) &= \frac{\text{dist}(f, c) + \text{dist}(g, c)}{\text{dist}(c, a)} \\
WUP(b, c) &= \frac{\text{dist}(b, k) + \text{dist}(c, k)}{\text{dist}(k, a)} \\
WUP(x, z) &= \frac{\text{dist}(x, b) + \text{dist}(z, b)}{\text{dist}(b, a)}
\end{align*}
\]

Figure 23: Wu-Palmer distance
Again I would use the best similarity between any pair of synsets for two words as my measure of their similarity.
Allan Ramsay, IV SEMANTICS & INFERENCE

-477-  WordNet for similarity measurements

[ 'man', 'woman' ] 0.705882352941  WUP
[ 'man', 'woman' ] 0.333333333333  PATH
[ 'peach', 'pear' ] 0.923076923077  WUP
[ 'peach', 'pear' ] 0.333333333333  PATH
[ 'king', 'queen' ] 1.0  WUP
[ 'king', 'queen' ] 1.0  PATH
[ 'apple', 'fruit' ] 0.9  WUP
[ 'apple', 'fruit' ] 0.333333333333  PATH
[ 'novel', 'bible' ] 0.666666666667  WUP
[ 'novel', 'bible' ] 0.166666666667  PATH
These measures are based solely on the shape of the graph. You might be able to do better by using the ‘information content’ of each term.

How informative an event is depends on how often it occurs: if around 50% of people are male, and around 10% of people are bald, then telling you that someone is male is less informative than telling you they are bald. So the informativeness of an event $e$ is something like $1/\text{prob}(e)$ (standard definition is $-\text{prob}(e) \times \log(\text{prob}(e)))$.
Given a corpus, the informativeness of a word could be defined as the number of instances of that word or one of its subsets: ‘thing’ isn’t going to be very informative, because pretty well every noun in the corpus is going to be a subset of ‘thing’. ‘steed’ will be more informative than ‘horse’ because ‘steed’ is a subset of horse, so there will be more horses than steeds in the world described by the text.

So you could take the information content with respect to some corpus of the lowest common ancestor of two words as a measure of their similarity. And that might give you something which is informed by judgements about entailment (so it shouldn’t think that ‘love’ and ‘hate’ are very similar except that WordNet thinks they are!) but takes advantage of a corpus.

8except that WordNet thinks they are!
• The structure of WordNet: the key data structures (the ones that enter into ‘hypernym’/‘hyponym’ relations) are synsets, identified by unique integers.

• Using WordNet for determining entailment relations between words: abduction as a strategy for lexical disambiguation

• Using WordNet for determining similarity relations between words: Wu-Palmer as a metric
Textual entailment

Simple lexical relations are inadequate

What else might I try to do?

- What I care about is ‘entailment’/‘consequence’. If I say ‘I went mountain biking yesterday’ then you should be able to answer ‘Did I ride a bike yesterday?’.

- How can I determine consequence relations? By using logic
(55) Every man has a mother, John is a man ⊢ John has a mother

Translate from English into logic.

\[
\begin{align*}
\text{utt(claim,} & \quad \forall_A : \{\text{man}(A)\}, \\
& \quad \exists_B : \{\text{mother}(B)\}, \\
& \quad \exists_C, \\
& \quad \text{event}(C, \text{has}) \\
& \quad \& \ (\theta(C, \text{object}, B!3) \\
& \quad \& \ (\theta(C, \text{agent}, A!0) \\
& \quad \& \ \text{aspect(now, simple, C)))})])
\end{align*}
\]

\[
\text{utt(claim, man(ref(lambda(A, named(A, 'John'))))!0)}
\]
utt(query,
    exists(_A :: {mother(_A)}),
    exists(_B,
        (event(_B, have)
          & (theta(_B, object, _A!3)
          & (theta(_B, agent, ref(lambda(_C, named(_C, 'John')
            & aspect(now, simple, _B))))))))
claim(X): add X to your database

query(X): see whether your favourite theorem prover can derive X from what’s in the database. If not, see if it can **disprove** it.

- It’s the right thing to do (it’s the **only** thing to do)

- It’s too hard to do the conversion

- It’s too hard to do the theorem proving

- It’s too hard to collect the background knowledge
‘A text $T$ entails a hypothesis $H$’ ($T \vdash H$) if, typically, a human reading $T$ would infer that $H$ is most likely true (Dagan et al. 2005)

(Dagan’s use of ‘text’ and ‘hypothesis’ is confusing: I will talk of the background and the query)
$S$ entails $S'$ if every word in $S'$ is in $S$.

(56) I saw a man in the park $\vdash$ I saw a man

$S$ entails $S'$ if every word in $S'$ is subsumed by a word in $S$.

(57) I saw a man in the park $\vdash$ I saw a human
Slightly better: $S$ entails $S'$ if every word in $S'$ is subsumed by a word in $S$ where the subsuming terms respect the order of $S'$.

(58) I saw a man in the park ⊢ I saw a human
(59) John saw a man in the park ̸⊢ A man saw John
String-edit distance

Subtler version of the last one. What would it take to turn $S$ into $S'$ by adding, deleting & exchanging words?

This is a standard task: it underlies (with some changes) spelling correction algorithms, DNA sequencing, the scoring algorithm for the HTK, ... 

Illustrate it as spelling correction.
The standard algorithm for doing this is called ‘**dynamic time warping**’ (at least it is in speech contexts: may have other names for other applications).

1. Make a grid, where the lengths of each segment are marked off along the two axes.

2. Work your way through the grid: look at the places you can get to from where you are now.

   - If you’ve never been there before, then getting there from here is the cheapest way you know of for getting there
   - If you’ve been there before, but you can get there more cheaply from here, then record that getting there from here is the best route (and record the cost)

Guaranteed that for each point in the grid you know the cheapest way of getting there. If you keep a backpointer, you can trace your way back from the bottom-right to the top-left.
Time complexity is $N \times M$: obvious enough — you have to inspect every node! Nodes a long way off the diagonal are unlikely to lie on the best path (and if they do then the best path probably isn’t very good), so you could restrict yourself to nodes where $\text{abs}(i - j)$ is less than some threshold, in which case you only have to look at $\max(M, N) \times K$ of them.
>>> a = dtw.array('whichs', 'witches')
>>> a.findPath()
>>> a.getAlignment()

(>>> a.findPath(latex=True, out="temp.tex"))
<table>
<thead>
<tr>
<th></th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>w</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

String-edit distance
<table>
<thead>
<tr>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>X</td>
<td>w</td>
<td>h</td>
<td>i</td>
<td>c</td>
<td>h</td>
<td>s</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>X</td>
<td>w</td>
<td>h</td>
<td>i</td>
<td>c</td>
<td>h</td>
<td>s</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

String-edit distance
<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

String-edit distance
String-edit distance
<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>i</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Allan Ramsay, IV SEMANTICS & INFRINGEMENT
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>t</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Allan Ramsay, IV SEMANTICS & INFEERENCE

String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE  -507-  String-edit distance
String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE -509- String-edit distance
<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>c</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Allan Ramsay, IV SEMANTICS & INference
String-edit distance
Allan Ramsay, IV SEMANTICS & INference

String-edit distance
Allan Ramsay, IV SEMANTICS & INference

String-edit distance
String-edit distance

Allan Ramsay, IV SEMANTICS & INFERENCE


<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Column</td>
<td>X</td>
<td>w</td>
<td>h</td>
<td>i</td>
<td>c</td>
<td>h</td>
<td>s</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>h</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Allan Ramsay, IV SEMANTICS & INference

String-edit distance
String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE -520- String-edit distance
String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance

```
X  w  h  i  c  h  s
X 0  2  4  6  8  10  12
w 2  0  2  4  6  8  10
i 4  2  3  2  4  6  8
h 10 8  6  -1 -1 -1 -1
```

```
e  -1  -1  -1  -1  -1  -1
s  -1  -1  -1  -1  -1  -1
```
<table>
<thead>
<tr>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>h</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>w</td>
<td>h</td>
<td>i</td>
<td>c</td>
<td>h</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>h</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>-1</td>
</tr>
<tr>
<td>e</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>s</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
String-edit distance
Allan Ramsay, IV SEMANTICS & INference -526- String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
String-edit distance
String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
X  w  h  i  c  h  s
X  0  2  4  6  8  10  12
w  2  0  2  4  6  8  10
i  4  2  3  2  4  6  8
h  6  4  5  4  5  7  9
c  8  6  7  6  4  6  8
h  10  8  6  8  6  4  6
e  12  10  8  9  8  6  7
s  -1  -1  -1  -1  -1  -1  -1
Allan Ramsay, IV SEMANTICS & INFERENCE

String-edit distance
<table>
<thead>
<tr>
<th>X</th>
<th>w</th>
<th>h</th>
<th>i</th>
<th>c</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>i</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>h</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>s</td>
<td>14</td>
<td>12</td>
<td>13</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
String-edit distance
String-edit distance
String-edit distance
In general an exchange is preferable to an insert followed by a deletion, so \( \text{cost}(XCH) \) should be less than \( \text{cost}(INS) + \text{cost}(DEL) \) (I used 3 for \( \text{cost}(XCH) \) and 2 for \( \text{cost}(INS) \) and \( \text{cost}(DEL) \) above).

But you can use cost functions that depend on what you are swapping: for spelling correction, exchange is cheap for adjacent keys on the keyboard, . . .

Useful extension: swap two items. Nice for spelling correction – make swap cheap for keys that you would type with different hands.
For textual entailment, exchange is free if the source item is a WordNet subset of the target item, deletion is free for modifiers, ... 

EXCH = 0 if the WordNet entailment relation from earlier on holds between W1 and W2, 3 otherwise

DELETE = 0 for adjectives (preps?) (use the tagger we trained earlier)

>>> teAlignment("I saw a fat old man", "I saw a person", tagger) 0
Missing links (‘I watched a woman’ ⊨ ‘I watched a person’, ‘I watched a woman’ ⊭ ‘I watched a human’)

Stupid links: ‘I watched a peach’ ⊨ ‘I watched a person’

Morphology: ‘he sees a man’ ⊨ ‘he watches a man’, ‘he saw a man’ ⊭ ‘he watched a man’
Or try to learn costs: cost is $I_{pos}$ for an insert, $D_{pos}$ for a delete, $(W_{sim} \times \text{sim} + W_{opp} \times \text{opp} + \ldots)$ where $I_{pos}$ is the cost of inserting an item with a given POS tag, $D_{pos}$ is the cost of deleting an item with a given POS tag, $\text{sim}$ is your favourite similarity measure and $W_{sim}$ is how much you think similarity matters, $\text{opp}$ is 0 or 1 depending on whether the words are WordNet opposites and $W_{opp}$ is how much you think that oppositeness matters, . . .

Then you’ve got a set of weights that you can vary to emphasise the cost of each operation: so you can tune these using your favourite optimisation algorithm (genetic algorithms, particle swarm optimisation, artificial bee colony, . . .)

Can be extended to apply to trees (Alabbas and Ramsay 2013). The algorithm gets difficult to follow (Bille 2005), and the complexity goes through the roof, but trees are a better representation of the information carried by the text.
Subsumption on parse trees

Define inference operations directly on parse trees (probably easiest with dependency trees).

Split it into domain rules and inference principles.
(60) a. John and Mary got divorced ⊨ John and Mary used to be married.
   b. John and Mary used to be married ⊨ John and Mary are not married.
got:VB

John:NN divorced:NN

and:CC Mary:NN

⇒

used:VB

John:NN to:TO

and:CC Mary:NN be:VB

married:NN

(note that the dependency relations are about right even though the tags for ‘divorced’ and ‘married’ are wrong)

(and to some extent it doesn’t matter anyway: if it gets ‘John and Mary used to be married wrong’ it will probably get ‘Peter used to drink too much’ wrong in the same way)
Better to use variables: add $X, Y, \ldots$ to English.

(61) a. $X$ and $Y$ used to be married $\vdash X$ and $Y$ are not married
b. $X$ and $Y$ used to be in love $\vdash X$ and $Y$ are not in love
c. John used to live in Dublin ⊨ John does not live in Dublin
used:VB
X:NN
to:TO
Y:NN

⇒

does:VB
X:NN
not:RB
Y:NN
Given \( \{ P \Rightarrow Q, P \} \) you can derive \( Q' \) where \( Q' \subseteq Q \).

\[
P \Rightarrow Q \\
\begin{array}{c}
t_1 \\
\ldots \\
t_i \\
\ldots \\
t_n \\
\Rightarrow \\
\begin{array}{c}
t_1 \\
\ldots \\
t_n \\
\end{array}
\end{array}
\]

\[
P'
\]

\[
Q'
\]

where \( t > t' \) and \( t_k \vdash t'_k \).
To do this properly is a bit intricate. I want to find out whether there is *some* way of matching two trees, where I’m prepared to ignore bits of the second one.

The problem is that early choices about what to skip or not skip may have ramifications later in the process.
A good way to deal with this is by using ‘continuation programming’.

- Every function has an extra argument, its ‘continuation’. This is the thing you would like to do next if the function does what it should.

- If a function does what it should, then you call its continuation: it will only return if it hasn’t managed to do its job (because if it did do its job, then it would have called the continuation)

- In the end, the thing that was set as the original continuation will get called. This will need to circumvent the normal return procedure, by throwing an exception
This is an efficient way of handling search that requires back-tracking, because you can use the normal call-and-return stack to keep the choice points on; and the implementers of your favourite programming language will have done a good job of implementing the call-and-return stack efficiently.

The key is that a function only returns if it has **failed** to do its job. If it does what it’s supposed to, it will call the continuation.

(constructing a continuation takes just under $10^{-7}$ seconds, so the cost of constructing the chain of continuations is minimal)
Simple symmetric matching with variables (unification):

```python
def match(x, y, contn):
    if x and y are the same then do contn
    if x or y is an unbound variable, bind it to the other and do contn
    unbind it (remember: return means failure)
    if x and y are lists
        contn = "do match(tl(x), tl(y), contn) when I tell you to"
        match(hd(x), hd(y), contn)
```

Not every language makes contn = "do match(tl(x), tl(y), contn) when I tell you to" easy.
Simple symmetric matching with variables (unification):

```python
def match(x, y, contn):
    if x == y: contn()
    if x or y is an unbound variable: bind(x, y); contn()
    unbind(x, y)
    if x and y are lists
        contn = lambda: match(x[1:], y[1:], contn)
        match(x[0], y[0], contn)
```

Not every language makes `contn = "do match(tl(x), tl(y), contn) when I tell you to"` easy. Python does with λ-functions.
Asymmetric matching with a subsumption lattice

def match(x, y, hypernyms, contn):
    if x == y or y in hypernyms[x]: contn()
    if x or y is an unbound variable: bind(x, y); contn()
    unbind(x, y)
    if x and y are lists
        contn = lambda: match(x[1:], y[1:], hypernyms, contn)
        match(x[0], y[0], contn)

X and Y are the same doesn’t have to mean X==Y. We can use our subsumption lattice (but it’s now asymmetric)
def match(x, y, hypernyms, contn):
    if x == y or y in hypernyms[x]: contn()
    if x or y is an unbound variable: bind(x, y); contn()
    unbind(x, y)
    if x and y are lists
        newcontn = lambda: match(x[1:], y[1:], hypernyms, contn)
        match(x[0], y[0], hypernyms, newcontn)
    This step skipped over elements of Y rather
    than X in the version I gave out earlier (and it didn’t
    actually decrement Y, so the loop wouldn’t have terminated)
    if len(x) > len(y):
        match(x[1:], y, hypernyms, contn)
Complexity was linear in size of the terms being matched. Now exponential in the difference in size between t1 and t2. Small innocent-looking changes can have radical effects on complexity.
Then for the inference engine we do

```python
def prove(goal, rules, hypernyms, contn):
    find a useful rule
    prove the subgoals and then do the contn

def proveall(goals, rules, hypernyms, contn):
    if goals == []:
        do the continuation
    else:
        prove the first:
        contn is to prove the rest and then do the original contn
```
Then for the inference engine we do\(^9\)

```python
def prove(goal, rules, hyponyms, contn):
    f = functor[goal]
    for concept in [f]+hyponyms[f]:
        for rule in rules[concept]:
            match(rule.hd, x,
            lambda: proveall(rule.subgoals, rules, hyponyms, contn))

def proveall(goals, rules, hyponyms, contn):
    if goals == []:
        contn()
    else:
        prove(goals[0], rules, lambda: proveall(goals[1:], rules, hyponyms, contn))
```

\(^9\)hyponyms is the inverse of hypernyms
This is essentially a fairly efficient implementation of Prolog in Python: but because I’m defining the matching algorithm for myself, I can make it do different things.

• Store rules so that any rule whose functor is a subset of the functor of the goal will be found: to prove that ‘X went to Y’, see if you have a rule which has ‘X walked to Y’ as its head.

• Do partial asymmetric matching using the hypernym table
from te import *

>>> untaggedhyps = getTaggedHyps(useTags=False)
>>> t1 = text2term("I saw a woman")
>>> t2 = text2term("I saw a person")
>>> t3 = text2term("I saw a woman in the park")
>>> tryit(lambda: ptp.match(t1, t2, hypernyms=untaggedhyps))
...

Allan Ramsay, IV SEMANTICS & INFERENCE
-565- General principles
This will get things like ‘Joe watched the new Harry Potter film on Friday’ ⊢ ‘Joe saw the new Harry Potter film’ and ‘I believe she loves me’ ⊢ ‘I believe she likes me’ right.
But it will get ‘Joe didn’t watch the new Harry Potter film on Friday’ ⊨ ‘Joe didn’t see the new Harry Potter film’ and ‘I doubt that she loves me’ ⊨ ‘I doubt that she likes me’ wrong.

We need to mix classical and textual entailment.
Label subtrees as being positive or negative: use a table of polarity switching words: {doubt:[1, -1], ...}

polarityTable = {"doubt": [1, -1],
                "not": [-1],
                ...}

def fixPolarity(tree0, polarity=1):
    mark the hd with 'polarity'
    try:
        plist = polarityTable[tree0[0]]
        for each dtr di do fixPolarity using polarity*plist[i]
    except:
        do fixPolarity to each dtr using polarity
    return tree1+dtrs
def match(x, y, hypernyms, polarity, contn):
    if polarity == 1:
        if x == y or y in hypernyms[x]: contn()
        if x or y is an unbound variable: bind(x, y); contn()
        unbind(x, y)
    if x and y are lists
        contn = lambda: match(x[1:], y[1:], hypernyms, contn)
        match(x[0], y[0], hypernyms, contn)
        while len(x) <= len(y):
            match(x, tl(y), hypernyms, contn)
    else:
        match(y, x, hypernyms, -polarity, contn)
(62) I doubt she likes him ⊨ I doubt she loves him

(63) I do not doubt that she loves him ⊨ I do not doubt that she likes him
Where do you get rules from?

Write them. Yuk.

Collect them from a corpus. If S1 and S2 mean the same then you could make a rule out of them. But how would you get sentences that meant the same from a corpus, and how would you know they meant the same.

News websites. On any given day, most news websites will carry some articles about the same stories. And if two articles are about the same story then they may contain sentences that mean the same.
Use our similarity measures to collect articles about the same story. Fairly low threshold, using the fact that proper names have low IDF scores.

Use our similarity measures to collect sentences (phrases?) that mean the same thing. High threshold.
Tempting

- you don’t have to write conversion rules for building semantic representations

- you’ll be able to do something, and probably something sensible, even if you haven’t managed to produce a proper parse tree

- Coarse strategies (bag-of-words, ordered bag-of-words, string edit distance) will have high recall but poor precision.

- Dynamic time warping algorithm!
You need to get rules from somewhere. Don’t want to write them, but collecting them from corpora isn’t straightforward. News article strategy will get you fairly technical ones, but it won’t get you things like ‘$X$ is bad for $Y$’ $\vdash$ ‘$Y$ should not do $X$’. I know of no resource that contains this stuff!

You may find yourself doing so many tree conversions that you might just as well have gone for standard logical form.

Managing backtracking by continuations (principles (pseudo-code), not detail (code))
THE END!
Index

abductive, 474
auto-correlation, 173–175
back off, 265
bigram, 289
BNC (British National Corpus), 273
categorial descriptions, 205
closed, 264
consequence, 481
contexts, 450
continuation programming, 555
cosine, 445
deltas, 55

Dependency grammar, 338
Derivational morphology:, 197
Dijkstra’s algorithm, 147
diphones, 143
dynamic time warping, 489
Edmond’s algorithm, 389
emission probabilities, 80
entailment, 481
Euclidean distance, 445
F-measure, 283
fast Fourier transform, 46
FFT, 46
flat start, 116
Formant synthesisers, 138
formants, 51
frames, 60
Gaussian mixture models, 111
gross score, 308
head percolation table, 365
Hidden Markov Models, 73
hyponym, 480
hypernym, 480
Inflectional morphology:, 196
information content, 478
latent semantic analysis, 462
leftArc:, 345
liquids, 40
MALT, 345
Mel frequency cepstral coefficients, 69
morphemes, 183
movement, 334
N-fold cross validation, 373
Nasals, 37
net score, 309
non-deterministic, 336
normal distributions, 109
obstruents, 39
Open, 264
oracle, 346
out-of-place items, 323
parsing accuracy, 372
part of speech tag, 270
Penn treebank, 273
phonology, 189
Porter stemmer, 269

Allan Ramsay, IV SEMANTICS & INFEERENCE -578- Where do you get rules from?
Where do you get rules from?
References


URL citeseer.ist.psu.edu/evans89inference.html 213

Hinton, G. E., Osindero, S., Teh, Y.-W., Jul. 2006. A fast learning algo-
URL http://dx.doi.org/10.1162/neco.2006.18.7.1527 136


Jurafsky, D., Martin, J., 2000. Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition. Prentice Hall, London. 194


dependency parsing using spanning tree algorithms. In: In Proceedings of
Human Language Technology Conference and Conference on Empirical
Methods in Natural Language Processing. pp. 523–530. 388

Mikolov, T., Chen, K.,Carrado, G., Dean, J.,2013. Efficient Estimation

Proceedings of the 8th International Workshop on Parsing Technologies
(IWPT. Citeseer, pp. 149–160. 4, 336

Nivre, J.,2006. Inductive dependency parsing. Vol. 34 of Text, Speech and
Language Technology. Springer. 345

4 (3), 138–152. 345

Nivre, J., Hall, J.,Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov,
S., Marsi, E.,2007. MaltParser: A language-independent system for data-
driven dependency parsing. Natural Language Engineering 13 (02), 95–
135. 4, 336

Allan Ramsay, IV SEMANTICS & INFERENCE

Where do you get rules from?


