Natural Language Systems: COMP34412

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

How does language work? 12
Architecture of a NL system 22
COMP24412 24
COMP34412 25
Course outline (draft) 26
What makes different sounds different? 34
Chords 36
Sounds & Fourier Transforms 44
Fourier analysis 45
Speech recognition 50
Spectrogram (Fourier analysis): AAA vs OOO 51
Formants 52
Spectrogram (Fourier analysis): CAT vs PAT 53
Spectrogram (quantised): CAT vs PAT 55
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

Speech synthesis

Pitch and duration

CHECKPOINT

The lexicon

Morphographemics, morphophonemics ('spelling rules')

Exercises for the reader

CHECKPOINT
Deterministic dependency parsing (Nivre et al. 2007; Nivre 2003)

Learning from a treebank

Headed phrase structure trees ≡ dependency trees

Learning a set of parsing rules

Long-distance dependency revisited

CHECKPOINT

Clustering

Detecting ambiguity

CHECKPOINT

Hand-coded lexical relations

Wordnet

Using Personalised Page Rank for WSD
‘Personalised page rank’
PPR with WordNet
Matrices & eigenvectors
CHECKPOINT
Textual entailment
Montague semantics
Bag of words
String-edit distance
Subsumption on parse trees
Domain rules
General principles
Where do you get rules from?
CHECKPOINT
Build a data structure that represents the meaning of the source?
Use it to generate the target
What else can we use transfer rules for? 565
The MT pyramid 573
Ambiguity in MT 579
Bilingual corpora, parallel corpora 584
Alignment 591
CHECKPOINT 611
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
How does language work?

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 ⊢ 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 data structures!
Architecture of a NL system

input utterance

SPEECH RECOGNISER

input text

PARSER

structural analysis

DECODER

background knowledge

canonical form

INFERENC ENGINE

anchored form

INFERENC ENGINE

updated minutes

GENERATOR

output text

SPEECH SYNTHESISER

output utterance

discourse model (minutes)

updated minutes
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)
How does language work?
input utterance

SPEECH RECOGNISER

input text

PARSER

dependency tree

Nothing!

dependency tree
discourse model (minutes)

background knowledge

anchored form

string edit distance, lexical relations, tree matching

entailment, equivalence

GENERATOR

output text

updated minutes

updated minutes

SPEECH SYNTHESISER

output utterance
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 vibrating reed</td>
<td>blow air over vibrating strings</td>
</tr>
<tr>
<td>column of air of variable length</td>
<td>column of air of 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.
What makes different sounds different?

‘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.

\[1\]

\(1\) (pictures produced using Praat: Google it, stick it on your machine)

Allan Ramsay, III Speech
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 ‘eeeee’
‘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
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])
```

![Graphs showing Fourier transforms of different 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.
Strong dark bands show you how the energy is distributed at different frequencies.
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?
But deciding the size of the chunks (size: how long is a chunk, what frequencies does it cover) is difficult

- If you pick the wrong boundaries, then things that are actually similar will look very different

- A small change in pitch is significant at low frequencies, but not at high ones. The difference between 50Hz and 60Hz is quite significant. The difference between 500Hz and 510Hz is inaudible.
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.
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

Representation as a vector of parameters

Using MEL frequency bins for quantisation
Most people use ‘Hidden Markov Models’ for the next bit.

Markov model: probabilistic model of what will happen next, given where you are now.
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)
• boil kettle (1.0)
• get milk (0.2)
• make tea (0.8)
• eat cake (1.0)
• make coffee (0.2)
• eat cake (1.0)

Figure 10: Things I might do in the kitchen
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 11, 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/?

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

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

$$p(S_j^i \mid O_i) \times P(S_{j-1}^i) \times P(S_{j-1}^i \rightarrow S_j^i)$$
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)

the (0.00) → dog (0.00)

a (0.00) \rightarrow the (0.00) \rightarrow a (0.00)

0.85 \rightarrow 0.27 \rightarrow 0.15

0.73 \rightarrow 0.21 \rightarrow 0.40

0.05 \rightarrow 0.66 \rightarrow 0.58

0.05 \rightarrow 0.05 \rightarrow 0.05

0.05 \rightarrow 0.05 \rightarrow 0.05

0.05 \rightarrow 0.05 \rightarrow 0.05

0.05 \rightarrow 0.05 \rightarrow 0.05

sleeps (0.00)

walks (0.00)

viterbi algorithm
Normalise after filling in this level

a (0.40) → cat (0.00)

the (0.60) → dog (0.00)

runs (0.00)
sleeps (0.00)
walks (0.00)
Transition from 'a' to 'cat': $p=0.27$ (0.40×0.85×0.79)

```
a (0.40) → cat (0.27) → runs (0.00)
the (0.60) → dog (0.00) → sleeps (0.00) → walks (0.00)
```
Transition from 'a' to 'dog': \( p = 0.01 \) (0.40 × 0.15 × 0.21)
Transition from 'the' to 'cat': $p = 0.13 \ (0.60 \times 0.27 \times 0.79)$
Transition from 'the' to 'dog': $p=0.09 \ (0.60 \times 0.73 \times 0.21)$
Normalise after filling in this level

a (0.40) → cat (0.74) → runs (0.00)

the (0.60) → dog (0.26) → sleeps (0.00) → walks (0.00)
Transition from 'cat' to 'runs': $p=0.02 \ (0.74 \times 0.05 \times 0.57)$
Transition from 'cat' to 'sleeps': $p=0.04$ (0.74×0.58×0.08)
Transition from 'cat' to 'walks': $p=0.09 \ (0.74 \times 0.37 \times 0.34)$

```
a (0.40) \rightarrow 0.85 \rightarrow \text{cat (0.74)} \rightarrow 0.05 \rightarrow \text{runs (0.02)}
the (0.60) \rightarrow 0.73 \rightarrow \text{dog (0.26)} \rightarrow 0.58 \rightarrow \text{sleeps (0.04)}
\rightarrow 0.15 \rightarrow \text{walks (0.09)}
```

Viterbi algorithm
Transition from 'dog' to 'runs': p=0.10 (0.26×0.66×0.57)
Transition from 'dog' to 'sleeps': $p=0.00$ (0.26×0.14×0.08)
Transition from 'dog' to 'walks': $p=0.02 \ (0.26 \times 0.20 \times 0.34)$
Normalise after filling in this level

a (0.40) → cat (0.74) → runs (0.43)
the (0.60) → dog (0.26) → sleeps (0.16) → walks (0.41)

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 $\approx$ short $\approx$ 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

--- Source: EXPT/train1.mfc ---

Sample Bytes: 52  Sample Kind: MFCC_K_0
Num Comps: 13  Sample Period: 10000.0 us
Num Samples: 125  File Format: HTK

--- Observation Structure ---

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

Samples: 0->1

...

Allan Ramsay, III Speech

What kinds of observations are we going to use?
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.
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 12: 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).

![Image of two graphs side by side.](image)

Figure 13: Martian men and women are quite distinct; Venussians 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 known 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.
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>

Allan Ramsay, III Speech -109- What kinds of observations are we going to use?
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/:
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/.
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 ‘\textit{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.

Allan Ramsay, III Speech -119- What kinds of observations are we going to use?
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

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

```
a = uh ;
the = th uh;
cat = c a t;
dog = d o g;
sleeps = s l i y p s;
runs = r u n s;
```
which in turn generates

Allan Ramsay, III Speech -125- What kinds of observations are we going to use?
(well, actually...)

Allan Ramsay, III Speech -126- What kinds of observations are we going to use?
That’s what the HTK does.

HTK was developed at Cambridge. They sold it to Microsoft, and for a few years you couldn’t get a copy. Then Microsoft made it publicly 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. 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 …)
Pitch and duration

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.

```python
>>> s = readsound("thecatsatonthemat.wav")
>>> setpitch(s, pitchchange=0.8).play()
>>> setpitch(s, pitchchange=1.2).play()
```
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.
```python
>>> a, b, c = stretch(setpitch(s, pitchchange=1.2), stretchby=1/1.2, c)
>>> c.play()
>>> pylab.plot(select(a, 5000, 5100).signal); pylab.plot(select(b, 50
```
To do it better, you need to work out the local frequency so that you can make the chunks exactly the right size. And then you should probably do something careful about how you merge them. For which you need to implement the PSOLA algorithm (Charpentier and Stella 1986).
• Formant vs concatenative vs diphone vs multiphone synthesis

  – Naturalness, comprehensibility

  – Need for recording large amounts of stuff?
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 ‘degirtt’ doesn’t look like English)

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

   b. He’s a complete unreconstructed Stalinist.

   c. * It’s just an unreconstruction.

(10) a. One of my best friends is watching old movies.

   b. One of my favourite activities is watching old movies.

(11) 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.
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’, ‘struct-ure-s’, .... 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. 14.

Figure 14: Letter trie
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
eexport PYTHONPATH=/opt/info/courses/COMP34411/PROGRAMS:$PYTHONPATH
in your .bash_profile file.

import lextrie

t = lextrie.TRIE()

t.addAll([('cat', 'noun'), ('cart', 'noun'), ...])
...

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

t.lookup('', 'cat', [], printing=True)
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’)
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];

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

- 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
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’?
Categorial descriptions

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

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

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

\[
\begin{array}{c}
\text{John} \\
\text{np}
\end{array} \quad 
\begin{array}{c}
\text{loves} \\
(s\backslash np)/np
\end{array} \quad 
\begin{array}{c}
\text{Mary} \\
\text{np}
\end{array}
\]

\[
\begin{array}{c}
\text{} \\
\text{} \\
\text{} \\
\text{} \\
\end{array}
\]

\[
\begin{array}{c}
\text{} \\
\text{s\backslash np} \\
\text{} \\
\text{s}
\end{array}
\]
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):

\[ \text{kick} = \text{N/NUM} \]
\[ \text{kick} = \text{V/TNS} \]

\[ \text{s} = \text{NUM} \]
\[ \text{s} = \text{TNS} \]

(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’)}
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

\[
\begin{align*}
X/Z & \Rightarrow X/Y \ Y/Z \\
X\backslash Z & \Rightarrow Y\backslash Z \ X\backslash Y \\
\end{align*}
\]

Raising

\[
\begin{align*}
Y/(Y/X) & \Rightarrow X \\
Y/(Y\backslash X) & \Rightarrow X \\
\end{align*}
\]

(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:

\[
\begin{array}{ccc}
\text{blanc} & \text{o} & \text{s} \\
\text{ADJ/GEN} & \text{GEN/NUM} & \text{NUM} \\
\hline
\text{ADJ/NUM} \\
\hline
\text{ADJ}
\end{array}
\]
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:

- ‘escribir’ 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')
Allan Ramsay, II Morphosyntax
Unseen: s, items found so far: []
Unseen: s, items found so far: [noun>agr]
Unseen: , items found so far: [noun]

**nouns:**

- **nouns:**
  - [noun] + [nouns] -> noun,
trie.lookup(t, 'carts')
Allan Ramsay, II Morphosyntax

-188-

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

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

Categorial descriptions
Unseen: sses, items found so far: []
Unseen: ses, items found so far: []
Allan Ramsay, II Morphosyntax

-201-

Categorial descriptions
Unseen: es, items found so far: [noun>agr, verb>tns]
Unseen: s, items found so far: [noun>agr, verb>tns]
47—t—48—i—49—o—50—n→[(noun>agr)<(verb>tns)]

[tens, agr]

Unseen: +s, items found so far: []
Spelling rule applied: [e] ==> [+]: [s, x0] → [s];: unseen was es and is now +s (seen ssik)
Unseen: , items found so far: [noun, verb]

noun>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: sing, 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: [verb>tns]
Unseen: , items found so far: [verb]

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 ‘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.

>>> 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 15: Probability of unknown words: 100M words
Figure 16: Probability of unknown words: 1M words
Figure 17: 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 18: 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}
Maximum-likelihood tagging

(the next few slides use the selection of the Penn Treebank that is available via the NLTK: 3.9K sentences, 90K words, about 23 words per sentence—much smaller than the BNC)

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 noun (‘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 backward and forward ‘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. People often use them in HMM-based algorithms (we will be looking at HMMs later when we look at speech processing). The one I’m going to use works as follows:
1. For each word, use a dictionary (with back-off to prefixes and suffixes) to calculate how likely it is to belong to each potential class (this would be used to find the ‘emission probabilities’ if we were using an HMM).

2. For each tag that has been assigned to the current word, look at how likely it is that this tag follows one of the tags assigned to the previous word: add these together (and likewise for following tags).

\[
P(tag(W_i) = T_j) = F(dict(W_i, t_j), \sum_k (P(tag(W_{i-1}) = T_k) \times P(T_k \rightarrow T_j)), \sum_k (dict(W_{i+1}, T_k) \times P(T_j \leftarrow T_k)))
\]
In other words, the likelihood that \( W_i \) has the tag \( T_j \) is some function \( F \) of the likelihood that is recorded for this pair in the dictionary, the sum of the forward transitions from the possible tags of the previous word, and the sum of the backward transitions from the next word. Note that we can use \( P(tag(W_{i-1}) = T_k) \) for the preceding word, but only \( dict(tag(W_{i+1}, T_k) \) for the following one.

What is \( F \)? Just pick something that works: I’m using \( F(d, f, b) = \sqrt{d} \times (f + b) \). Why? Because it works (actually \( d^{0.4} \times (f + 0.5 \times b) \) works very marginally better, but that’s too arbitrary even for me).
time flies like an arrow
time flies like an arrow

Forward transition probabilities

<table>
<thead>
<tr>
<th>A</th>
<th>N:0.80 V:0.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A:0.25 I:0.37 N:0.31 V:0.06</td>
</tr>
<tr>
<td>I</td>
<td>D:0.60 N:0.40</td>
</tr>
<tr>
<td>N</td>
<td>A:0.07 D:0.07 I:0.47 N:0.07 V:0.33</td>
</tr>
<tr>
<td>P</td>
<td>N:0.30 V:0.70</td>
</tr>
<tr>
<td>V</td>
<td>D:0.21 I:0.57 N:0.14 V:0.07</td>
</tr>
</tbody>
</table>
time flies like an arrow

Backward transition probabilities

<table>
<thead>
<tr>
<th></th>
<th>A:0.80</th>
<th>D:0.20</th>
<th>N:0.20</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>N:0.40</td>
<td>V:0.60</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>A:0.08</td>
<td>D:0.08</td>
<td>N:0.46</td>
</tr>
<tr>
<td>D</td>
<td>I:0.33</td>
<td>N:0.33</td>
<td>V:0.33</td>
</tr>
<tr>
<td>N</td>
<td>A:0.17</td>
<td>D:0.33</td>
<td>N:0.08</td>
</tr>
</tbody>
</table>
time flies like an arrow

Forward transition probabilities
A N:0.80 V:0.20
D A:0.25 I:0.37 N:0.31 V:0.06
I D:0.60 N:0.40
N A:0.07 D:0.07 I:0.47 N:0.07 V:0.33
P N:0.30 V:0.70
V D:0.21 I:0.57 N:0.14 V:0.07

Backward transition probabilities
A D:0.80 N:0.20
I N:0.40 V:0.60
V A:0.08 D:0.08 N:0.46 P:0.23 V:0.15
D I:0.33 N:0.33 V:0.33
N A:0.17 D:0.33 N:0.08 P:0.08 V:0.33

Allan Ramsay, II Morphosyntax

Maximum-likelihood tagging
time flies like an arrow

Forward transition probabilities
A: N:0.80 V:0.20
D: A:0.25 I:0.37 N:0.31 V:0.06
I: D:0.60 N:0.40
N: A:0.07 D:0.07 I:0.47 N:0.07 V:0.33
P: N:0.30 V:0.70
V: D:0.21 I:0.57 N:0.14 V:0.07

Backward transition probabilities
A: D:0.80 N:0.20
N: I:0.40 V:0.60
V: A:0.08 D:0.08 N:0.46 P:0.23 V:0.15
D: I:0.33 N:0.33 V:0.33
N: A:0.17 D:0.33 N:0.08 P:0.08 V:0.33

Allan Ramsay, II Morphosyntax

Maximum-likelihood tagging
time flies like an arrow

Forward transition probabilities

- A: N: 0.80 V: 0.20
- D: A: 0.25 I: 0.37 N: 0.31 V: 0.06
- I: D: 0.60 N: 0.40
- N: A: 0.07 D: 0.07 I: 0.47 N: 0.07 V: 0.33
- P: N: 0.30 V: 0.70
- V: D: 0.21 I: 0.57 N: 0.14 V: 0.07

Backward transition probabilities

- A: D: 0.80 N: 0.20
- I: N: 0.40 V: 0.60
- V: A: 0.08 D: 0.08 N: 0.46 P: 0.23 V: 0.15
- D: I: 0.33 N: 0.33 V: 0.33
- N: A: 0.17 D: 0.33 N: 0.08 P: 0.08 V: 0.33

Allan Ramsay, II Morphosyntax

Maximum-likelihood tagging
time: N:0.55, V:0.45
flies: N:0.49, V:0.51
like: I:0.56, V:0.44
an: D:1.00
arrow: N:0.50

Forward transition probabilities
A N:0.80 V:0.20
D A:0.25 I:0.37 N:0.31 V:0.06
I D:0.60 N:0.40
N A:0.07 D:0.07 I:0.47 N:0.07 V:0.33
P N:0.30 V:0.70
V D:0.21 I:0.57 N:0.14 V:0.07

Backward transition probabilities
A D:0.80 N:0.20
I N:0.40 V:0.60
V A:0.08 D:0.08 N:0.46 P:0.23 V:0.15
D I:0.33 N:0.33 V:0.33
N A:0.17 D:0.33 N:0.08 P:0.08 V:0.33
time flies like an arrow

Forward transition probabilities
A: N:0.80 V:0.20
D: A:0.25 I:0.37 N:0.31 V:0.06
I: D:0.60 N:0.40
N: A:0.07 D:0.07 I:0.47 N:0.07 V:0.33
P: N:0.30 V:0.70
V: D:0.21 I:0.57 N:0.14 V:0.07

Backward transition probabilities
A: D:0.80 N:0.20
I: N:0.40 V:0.60
V: A:0.08 D:0.08 N:0.46 P:0.23 V:0.15
D: I:0.33 N:0.33 V:0.33
N: A:0.17 D:0.33 N:0.08 P:0.08 V:0.33

Allan Ramsay, II Morphosyntax -262- Maximum-likelihood tagging
time 0.07 flies 0.47 like 0.60 an 0.31 arrow 0.47
N:0.55 N:0.49 I:0.56 D:1.00 N:0.60
V:0.45 V:0.51 V:0.44 V:0.40

Forward transition probabilities
A N:0.80 V:0.20
D A:0.25 I:0.37 N:0.31 V:0.06
I D:0.60 N:0.40
N A:0.07 D:0.07 I:0.47 N:0.07 V:0.33
P N:0.30 V:0.70
V D:0.21 I:0.57 N:0.14 V:0.07

Backward transition probabilities
A D:0.80 N:0.20
I N:0.40 V:0.60
V A:0.08 D:0.08 N:0.46 P:0.23 V:0.15
D I:0.33 N:0.33 V:0.33
N A:0.17 D:0.33 N:0.08 P:0.08 V:0.33

Allan Ramsay, II Morphosyntax
Maximum-likelihood tagging
%% mxl is the simulator used for these notes, not the real one

```python
>>> x, y = mxl.test(text="the latest of the great loves of his life", N=9)
```
A lot depends on how F is defined. Should you give more weight to the dictionary score or to the context?

You could use an HMM to find the most likely route through the graph. Each word only has one tag, so look for the most likely overall sequence. Doesn’t use backward transition probabilities. Training can be slow.

The strategy here says ‘when considering the best tag for word $W_i$, use all your guesses about $W_{i-1}$ and $W_{i+1}$’. Works OK for me, with no need for a round of training.
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 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;

...

Retag the current word from T1 to T2 if its tag is T3 and the
tag of the previous two words is T4
#t4(T1, T2, T3, T4): T1 > T2 if tag[0]=T3 and tag[-1, -2]=T4;

...

Retag the current word from T1 to T2 if its tag is T3
#w0(T1, T2, T3): T1 > T2 if word[0]=T3;
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 \[ \text{tag}[0]=\text{UN} \];: gross score 371
Candidate rule #t3(UN, NN, UN, NN): UN > NN if \[ \text{tag}[0]=\text{UN}, \text{tag}[1, 2]=\text{NN} \];: gross score 205
Candidate rule #t4(UN, NN, UN, NN): UN > NN if \[ \text{tag}[0]=\text{UN}, \text{tag}[-1, -2]=\text{NN} \];: gross score 179
Candidate rule #t3(UN, NN, UN, VB): UN > NN if \[ \text{tag}[0]=\text{UN}, \text{tag}[1, 2]=\text{VB} \];: gross score 132
Candidate rule #t0(UN, CD, UN): UN > CD if \[ \text{tag}[0]=\text{UN} \];: gross score 123

...
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 next two words, 132 have a verb as one of the next two. These cases all 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 -271- 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 -583

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).
None of our taggers are perfect. No-one’s tagger is perfect. Knowing something about the mistakes it makes can help you compensate.

Confusion matrix: if the tagger says it’s an NN1, how likely is it that it’s actually a AJ0?
<table>
<thead>
<tr>
<th></th>
<th>NN</th>
<th>VB</th>
<th>IN</th>
<th>DT</th>
<th></th>
<th>JJ</th>
<th>SZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN</td>
<td>3028</td>
<td>129</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>155</td>
</tr>
<tr>
<td>VB</td>
<td>31</td>
<td>1117</td>
<td>1</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>7</td>
<td>1</td>
<td>973</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>DT</td>
<td>2</td>
<td></td>
<td></td>
<td>839</td>
<td></td>
<td></td>
<td>452</td>
</tr>
<tr>
<td></td>
<td>JJ</td>
<td>19</td>
<td>9</td>
<td>3</td>
<td></td>
<td></td>
<td>441</td>
</tr>
<tr>
<td></td>
<td>SZ</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>412</td>
</tr>
<tr>
<td></td>
<td>SA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CD</td>
<td>3</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>CC</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>TO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>RB</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PO</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 19: Confusion matrix
Suppose you had several taggers, each using slightly different information.

They will all make mistakes. They might all make systematic mistakes. They might all make different systematic mistakes.

Collect a confusion matrix for each, estimate their reliability on specific tags (e.g. the one above is 99.6% accurate when it says that something is a DT, but much less accurate when it says it’s a VB).

Take the result of the one that is most confident: given three Arabic taggers of which the best scores 0.965, we can get 0.995 by this approach Alabbas and Ramsay (2012).
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\(^2\). 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.

\(^2\)apart from ‘I’m bored of that’, which is ungrammatical
• Open and closed classes

• Backoff as a strategy

• Stemming vs morphology

• Tagging: corpus-based, affix-based, maximum-likelihood tagging, 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']]

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

>>> 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.

---

3Both interpretations do have to be in the dictionary: "loves" doesn’t occur even once as a noun in the UD treebank.
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 \(\approx\) 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 -290- Regular expressions
Dead easy to write. And regexes can be applied very efficiently. Can’t they?

```python
# First few are to match BNC tags
tag = ...  # First few are to match BNC tags
 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't 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 do n’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 do n'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))(((prep)(((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))(((prep)(((det0?(adj1 noun)*noun| name| pron))))*))*)))

Specification for 's' is just impossible. Hence undebuggable.
Not as lightning fast as we hoped. Problem is that they do have 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.
Cascaded regexes

Also known as supertagging Bangalore and Joshi (1999)

Find ‘islands’: if your tags are right, these can be found quite accurately.

You’ve now got a shorter string. But short strings are easier to parse. See if you can do anything with this, using other or the same regexes.

- Using different ones may be fast
- Using the same ones lets you describe recursive structures.

Have to get it right first time, because you’re rewriting the whole thing & you don’t have any chance of correcting any mistakes
ctags={'nn': choice(tag('NN.'),
    choice(tag('AJ.'), tag('noun'))+tag('nn'))},
'npo': tag('(AT|D.).')+tag('nn'),
'np': choice(tag('np0'), tag('np')+tag('pp')),
'pp': tag('PR.')+tag('np')
}
the fat old man

with a big nose

the fat old man

with a big nose

the fat old man

Allan Ramsay, II Morphosyntax

Cascaded regexes
with a big nose

the fat old man

Allan Ramsay, II Morphosyntax -303- Cascaded regexes
the fat old man

with

a

Cascaded regexes
The fat old man with a big nose.
Allan Ramsay, II Morphosyntax

Cascaded regexes
• 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! Cascading them lets you deal with recursive structures.

• 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 $\neq$ 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').

% % % % Found one: 23

\begin{center}
\begin{tikzpicture}
  \node (saw) {saw};
  \node (man) [below left of=saw] {man};
  \node (object) [below right of=saw] {I};
  \node (headlessEq) [below left of=man] {headlessEq};
  \node (headlessMod) [below right of=headlessEq] {headlessMod};
  \node (love) [below right of=headlessMod] {love,S};
  \node (who) [below left of=love] {who};
  \node (she) [below right of=love] {she};
  \node (the) [below right of=man] {the};
  \node (specifier) [below right of=the] {specifier};
  \node (agent) [below right of=she] {agent};
  \node (object) [below right of=she] {agent};
  \node (I) [below right of=object] {I};
  \draw (saw) -- (man);
  \draw (man) -- (headlessEq);
  \draw (headlessEq) -- (who);
  \draw (headlessEq) -- (she);
  \draw (man) -- (headlessMod);
  \draw (headlessMod) -- (love);
  \draw (man) -- (the);
  \draw (the) -- (specifier);
  \draw (specifier) -- (I);
\end{tikzpicture}
\end{center}

latexconll(23).
b. | ?- analyse(’I ate the peach which she said he wanted.’).
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. 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:

\{input: [DET, ADJ*, NN1, ...], stack: [NN1, ...]} \Rightarrow \text{shift}
\{input: [DET, ...], stack: [NN?, ...]} \Rightarrow \text{leftArc(det)}
...
\{input: [?, ...], stack: [?, ...]} \Rightarrow \text{leftArc(mod)}
\{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:} [\text{'old'}, \text{'man'}, \text{'ate'}, \text{'it'}], \text{stack:} [\text{'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
  VP
    NP
      NP
        PRP
        VBZ
        NP
          DT
          NN
          IN
          NP
            PRP
            NN
            NN
            NN
            NN
              our
              work
              force
              today

It has no bearing on our work force today.
```
What’s the most important thing in this tree?

It has no bearing on our work force today.
What’s the most important thing in this tree?
What’s the most important thing in the VP?
What's the most important thing in the VP?

S

NP-SBJ

PRP

It

VP

VBZ

has

NP

DT

no

NP

NN

bearing

IN

on

NP

PP-DIR

NP

NP-TMP

PRP

our

NN

work

NN

force

NN

today
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’.

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

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 work of our force today.
Learning a set of parsing rules

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. Examples below are done with ID3 Quinlan (1986)—easy to implement, easy to read the inferred rules, 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>shift</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.

\[(F_0, F_1, F_2, F_3, F_4 = Q_1, Q_2, S_1, S_2, \text{DIST})\]
What's the top item on the stack

Is it tagged as "TH" (i.e. the word "that")?

Yes: then what's the head of the queue

Is it tagged as "FW" (don't know what that is)?

Do a shift

Is it a verb?

What's the second item on the queue?

Is it a verb?

What's the second item on the stack?

Is it a pronoun?

Make "that" a dtr of the verb at the head of the Q

Do a shift

Is it a verb?

What's the second item on the stack?

Is it a pronoun?

Make "that" a dtr of the verb at the head of the Q

...
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)
(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)
A pint or a coffee & cake to anyone who can suggest a feature that makes things better. A meal for two if it gets me over 80% on the universal dependency treebank.
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’]]
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').

I
  ↕ saw
  ↘ the ← man
  ↘ who
  ↘ she ← love

saw

man
  ↘ object
  ↘ the specifier

headlessEq

headlessMod

love,S

*headlessEq

who
  ↘ object

she
  ↘ agent

I
  ↘ agent
b. | :- analyse('I ate the peach which she said he wanted.').
I eat the peach which she say he want.
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.
• Dependency grammar is good for
  – free word-order language
  – 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 operations. There are several solutions: looking deep into the queue or stack for daughter is the easiest to understand.

• N-fold cross validation
IV SEMANTICS & INFERENCE
**** is a bat-and-ball team sport. Many variations exist, with its most popular form played on an oval-shaped outdoor arena known as a field at the centre of which is a rectangular 22-yard (20.12 m) long pitch that is the focus of the game. A game (or match) is contested between two teams of eleven players each. One team bats, and will try to score as many runs as possible while the other team bowls and fields, trying to dismiss the batsmen and thus limit the runs scored by the batting team. A run is scored by the striking batsman hitting the ball with his bat, running to the opposite end of the pitch and touching the crease there without being dismissed. The teams switch between batting and fielding at the end of an innings.

There are also variations in the length of a game of ****. In professional **** this ranges from a limit of 20 overs per side (Twenty20) to a game played over 5 days (Test ****, which is the highest level of the game). Depending on the form of the match being played, there are different rules that govern how a game is won, lost, drawn or tied. The rules of two-innings games are known as the Laws of **** and maintained by the ICC and the Marylebone **** Club (MCC); additional Standard Playing Conditions for Test matches and One Day Internationals augment these laws. In one version of Indoor ****, matches include just 6 players per side and include two 12-over innings.

**** was first documented as being played in southern England in the 16th century. By the end of the 18th century, it had developed to the point where it had become the national sport of England. The expansion of the British Empire led to **** being played overseas and by the mid-19th century the first international matches were being held. Today, the game’s governing body, the International **** Council (ICC), has 104 member countries. With its greatest popularity in the Test playing countries, **** is the world’s second most popular sport after Association football.

Contents

- History
- Rules and Game-play
  - Objectives
  - Pitch, wickets and creases
  - Bat and ball
  - Umpires and scorers
  - Innings
From Wikipedia, the free encyclopedia
Jump to: navigation, search
This article is about the cured meat. For other uses, see **** (disambiguation).

Uncooked pork belly **** strips.

**** is a cured meat prepared from a pig. It is first cured using large quantities of salt, either in a brine or in a dry packing; the result is fresh **** (also green ****). Fresh **** may then be further dried for weeks or months (usually in cold air), boiled, or smoked. Fresh and dried **** must be cooked before eating. Boiled **** is ready to eat, as is some smoked ****, but either may be cooked further before eating.

**** is prepared from several different cuts of meat. It is usually made from side and back cuts of pork, except in the United States, where it is almost always prepared from pork belly (typically referred to as "streaky", "fatty", or "American style" outside of the US). The side cut has more meat and less fat than the belly. **** may be prepared from either of two distinct back cuts: fatback, which is almost pure fat, and pork loin, which is very lean. ****-cured pork loin is known as back ****.

**** may be eaten smoked, boiled, fried, baked, or grilled, or used as a minor ingredient to flavor dishes. **** is also used for barding and jarding roasts, especially game birds. The word is derived from the Old High German bazo, meaning "buttock", "ham" or "side of ****", and cognate with the Old French ****.[1]

In continental Europe, this part of the pig is usually not smoked like **** is in the United States; it is used primarily in cubes (lardons) as a cooking ingredient, valued both as a source of fat and for its flavor. In Italy, this is called pancetta and is usually cooked in small cubes or served uncooked and thinly sliced as part of an antipasto.

Meat from other animals, such as beef, lamb, chicken, goat, or turkey, may also be cut, cured, or otherwise prepared to resemble ****, and may even be referred to as ********[2] Such use is common in areas with significant Jewish and Muslim populations[1]. The USDA defines **** as "the cured belly of a swine carcass"; other cuts and characteristics must be separately qualified (e.g., smoked pork loin ***)}. For safety, **** must be treated for botulism, a parasite transfused which can be destroyed
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 bacon.

Well, not all the other words: [a, able, about, above, accepted, according, account, achieved, act, action, activity, actually, added, addition, affects, after, afterwards, again, against, agreed, aim, ais, all, allow, almanack, alone, already, also, although, always, america, amongst, amount, amp, an, and, another, any, anyway, appeal, appear, apply, april, are, area, arms, around, article, artillery, as, asking, associated, association, at, attempt, attribution, au, augment, august, australia, available, avoid, awarded, away, b, back, backyard, bad, bail, ball, based, basic, basket, bat, batsman, ...]
Collect some documents, e.g. by querying a search engine. I looked for documents about cricket and documents about bacon.

Remove the search terms: otherwise what we’re doing is circular—the documents are similar because they contained the words ‘cricket’ and ‘bacon’, so there’s no point in us trying to find out whether they are similar.
For each document, collect (and count) all the words it contains. These constitute a ‘vector space model’: [(a, 317), (abandoned, 1), (abide, 2), (able, 1), (about, 7), (above, 2), (absence, 2), (accepted, 2), (accessed, 1), (according, 3), (account 1), (accounting, 1), (accumulate, 2), (accuracy, 1), (accurate, 2), (achieved, 4), (acquire, 1), (across, 1), (act, 3), (action, 4), (actions, 1), (activity, 2), (acts, 1), (actually, 3), (adam, 1), (added, 1), (addition, 4), (additional, 6), (addressed, 1), (adjoined, 1), ...]
Four two dimensional vectors (lines): 

But the axes didn’t have to be called X and Y: 

\[
\begin{align*}
D_1: & \{\text{pet:100, ball:6}\}, \\
D_2: & \{\text{pet:12, ball:75}\}, \\
D_3: & \{\text{pet: 91, ball:18}\}, \\
D_4: & \{\text{pet:24, ball:67}\}
\end{align*}
\]
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

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

(obvious 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

$$\frac{\sum x_i \times y_i}{\sqrt{\sum x_i^2 \times \sum y_i^2}}$$

(less obvious how this relates to cosine in 2-D space ($= \frac{x}{\sqrt{x^2 + y^2}}$): 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.
If one document is much longer than another, it will have higher scores for each word. So the difference between them will be large.

Document 1: ‘Cricket is a game played by 11 fools, watched by 11000 others’.
   cricket:1, is:1, a:1, game:1, played:1, fools:1, watched:1, by:1, others:1

Document 2: ‘Cricket is a game played by 11 fools, watched by 11000 others. Cricket is a game played by 11 fools, watched by 11000 others’.
   cricket:2, is:2, a:2, game:2, played:2, fools:2, watched:2, by:2, others:2
Distance between Document 1 and Document 2 is

\[ \sqrt{1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1} = 3 \]

But surely these two documents are essentially identical, and should have a distance of 0?

Normalise it: divide each number in a vector by \( \sqrt{\sum_{i=1}^{n} s_i^2} \). Then if you have two vectors with nothing in common you’ll get a score of 1: if \( A = <a_1, a_2, a_3, 0, 0> \) and \( B = <0, 0, 0, b_1, b_2> \) then \( dict(A, B) \) will be

\[
\frac{\sqrt{(a_1 - 0)^2 + (a_2 - 0)^2 + (a_3 - 0)^2 + (0 - b_1)^2 + (0 - b_2)^2}}{\sqrt{a_1^2 + a_2^2 + a_3^2 + b_1^2 + b_2^2}} = 1
\]
‘a’, ‘the’ are the commonest words in any document. So the difference between how often these words occur in each of a pair of documents will dominate the scoring.

But they have almost nothing to do with whether the documents are about the same topics.
See how many documents contain the term. The more documents it occurs in, the less characteristic of each document it is. That’s the ‘document frequency’.

See how often it occurs in each individual document. The more often it occurs in an individual document, the more characteristic of that content of that document it is. That’s the ‘term frequency’.

Divide one by the other: that’s the term frequency $\times$ the inverse of the document frequency – the ‘TF-IDF’ score. Works quite well. If you’ve got huge numbers of documents, dividing by DF might be overkill, so people often take its $\log$ to smooth it out a bit. If you haven’t got very many documents, you might actually need to emphasise it, e.g. by dividing by some power.\footnote{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.\footnote{[3]}: Wikipedia}

Allan Ramsay, IV SEMANTICS & INference © 2019-2023
Quite a few words occur in only one document in a set. So their DF is low, and hence their TF-IDF for the document they appear in will be high.

But they cannot be used for relating one document to another. They can mask the underlying difference. So we may as well remove them.

Should probably stem them as well: if one document contains ‘cricketer’ and the other ‘cricketers’ we should take this as evidence that they are similar.
So: 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>
</tr>
</thead>
<tbody>
<tr>
<td>bacon-recip</td>
<td>0.00</td>
<td>0.90</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>Bacon</td>
<td>0.90</td>
<td>0.00</td>
<td>0.93</td>
<td>0.96</td>
</tr>
<tr>
<td>bacon_week</td>
<td>0.96</td>
<td>0.93</td>
<td>0.00</td>
<td>0.97</td>
</tr>
<tr>
<td>Cricket Rul</td>
<td>0.97</td>
<td>0.96</td>
<td>0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>Cricket</td>
<td>0.95</td>
<td>0.82</td>
<td>0.97</td>
<td>0.83</td>
</tr>
<tr>
<td>easy-risot</td>
<td>0.86</td>
<td>0.91</td>
<td>0.94</td>
<td>0.96</td>
</tr>
<tr>
<td>Laws_of_cr</td>
<td>0.98</td>
<td>0.91</td>
<td>0.98</td>
<td>0.82</td>
</tr>
<tr>
<td>Pork, Bacon, Ha</td>
<td>0.87</td>
<td>0.87</td>
<td>0.92</td>
<td>0.98</td>
</tr>
<tr>
<td>Rules of Cric</td>
<td>0.98</td>
<td>0.98</td>
<td>0.99</td>
<td>0.96</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>
</tr>
</thead>
<tbody>
<tr>
<td>Bacon</td>
<td>Bacon 0.00</td>
<td>Cricket 0.82</td>
<td>Pork, Baco 0.87</td>
</tr>
<tr>
<td>bacon_week</td>
<td>bacon_week 0.00</td>
<td>Pork, Baco 0.92</td>
<td>Bacon 0.93</td>
</tr>
<tr>
<td>Cricket Rules</td>
<td>Cricket Ru 0.00</td>
<td>Laws_of_cr 0.82</td>
<td>Cricket 0.83</td>
</tr>
<tr>
<td>Cricket</td>
<td>Cricket 0.00</td>
<td>Laws_of_cr 0.69</td>
<td>Bacon 0.82</td>
</tr>
<tr>
<td>easy-risotto-wi</td>
<td>easy-risot 0.00</td>
<td>bacon-reci 0.86</td>
<td>Pork, Baco 0.90</td>
</tr>
<tr>
<td>Laws_of_cricket</td>
<td>Laws_of_cr 0.00</td>
<td>Cricket 0.69</td>
<td>Cricket Ru 0.82</td>
</tr>
<tr>
<td>Pork, Bacon, Ha</td>
<td>Pork, Baco 0.00</td>
<td>Bacon 0.87</td>
<td>bacon-reci 0.87</td>
</tr>
<tr>
<td>Rules of Cricke</td>
<td>Rules of C 0.00</td>
<td>Laws_of_cr 0.91</td>
<td>easy-risot 0.93</td>
</tr>
</tbody>
</table>
Can we do any better?

*K-means clustering*: assume that there are two groups (hard to know how many groups to use: but here I know there are two, so I'll cheat)

Take the two least similar documents as ‘seeds’. If there are two groups, then surely these two don’t belong to the same group.

For each item, group it with the most similar seed.

For each group, make up a composite vector (by doing weighted averaging). People often call this the ‘centroid’ of the cluster.
What’s the distance for each document from the centroid for each cluster?

<table>
<thead>
<tr>
<th>Document</th>
<th>1st Distance</th>
<th>2nd Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>bacon-recipes</td>
<td>0.74</td>
<td>0.93</td>
</tr>
<tr>
<td>Bacon</td>
<td>0.26</td>
<td>0.81</td>
</tr>
<tr>
<td>bacon_week</td>
<td>0.83</td>
<td>0.96</td>
</tr>
<tr>
<td>Cricket Rules</td>
<td>0.94</td>
<td>0.73</td>
</tr>
<tr>
<td>Cricket</td>
<td>0.81</td>
<td>0.23</td>
</tr>
<tr>
<td>easy-risotto-wi</td>
<td>0.85</td>
<td>0.85</td>
</tr>
<tr>
<td>Laws_of_cricket</td>
<td>0.90</td>
<td>0.53</td>
</tr>
<tr>
<td>Pork, Bacon, Ha</td>
<td>0.70</td>
<td>0.91</td>
</tr>
<tr>
<td>Rules of Cricket</td>
<td>0.97</td>
<td>0.87</td>
</tr>
</tbody>
</table>
For most of them, the distance to the ‘right’ group is much less than it was to any one of the constituents.

- because the group actually contains the document?

- why does ‘easy risotto with bacon and peas’ not fit properly?
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, . . .

- 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 (OBJ:noun)'

adj is actually (AJ.)*(NN)*, i.e. zero or more adjectives and then 0 or more nouns. So could just be 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 500K words/sec, so can afford to do it for quite large amounts of data (BNC in 2’40’’).
>>> VO = sim.bnc.getDepRels(sim.bnc.BNC, "justverbandobj")
(or >>> VO = sim.load("VO.pck")

>>> sim.intersection("read", "write", VO, "OBJ") # Not doing TF-IDF weighting

read: book, 229 (265); newspaper, 80 (1); paper, 77 (21); letter, 52 (224); word, 47 (37); story, 41 (48); report, 38 (46); article, 37 (73); text, 27 (8); magazine, 22; label, 19 (1); passage, 18 (4); music, 18 (21); chapter, 18 (14); sign, 17; poem, 17 (61); novel, 17 (26); section, 16 (2); page, 16 (2); line, 16 (8);

write: book, 265 (229); letter, 224 (52); song, 76 (1); article, 73 (37); poem, 61 (17); story, 48 (41); program, 48; essay, 47 (2); report, 46 (38); word, 37 (47); poetry, 37 (9); note, 37 (6); piece, 29 (9); play, 28 (7); history, 27 (11); cheque, 27; novel, 26 (17); column, 24; account, 24 (9); things, 22 (15);
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?

This is from the whole BNC. 100 million words. You need a lot of data.
>>> intersection("eat", "devour", VO, "OBJ", 50)

eat:  food, 90 (1); meat, 65; meal, 52 (1); fish, 48 (1); breakfast, 33;
bread, 24; fruit, 22; lot, 18; cake, 16 (1); pie, 15; lunch, 15; grass, 14;
egg, 14; dinner, 12; day, 12; vegetable, 10; things, 10; thing, 10; diet, 10;
bite, 10; amount, 10; potato, 9; dog, 9; chocolate, 9; chicken, 9; supper,
8; sandwich, 8; chip, 8; cheese, 8; biscuit, 8; quantity, 7 (1); sweet, 6;
snack, 6; roll, 6; people, 6; load, 6; lamb, 6; bird, 6; berry, 6; apple, 6;
animal, 6; tea, 5; stuff, 5; steak, 5; spaghetti, 5; shit, 5; seed, 5; rest, 5;
product, 5; kind, 5;

devour:  world, 2; unfortunate, 1; treatise, 1; tissue, 1; timberwork, 1; tail,
1 (1); soul, 1; sort, 1 (2); sky, 1; quantity, 1 (7); principle, 1 (1); plant, 1
(4); part, 1 (1); nut, 1 (3); novel, 1; moth, 1; money, 1 (1); meal, 1 (52);
marshland, 1; man, 1 (2); line, 1; life, 1; land, 1; lady, 1; horoscope, 1;
forest, 1; food, 1 (90); fleet, 1; fish, 1 (48); father, 1; clump, 1; cake, 1
(16); book, 1; baby, 1 (1);
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.”

If we think it’s strong enough to believe in, reasonable hypothesis that ‘devour’ is a subset of ‘eat’ (because it’s less common, so is likely to be a special case?) (and because more of its arguments are shared?)
words = sim.getWORDS(["eat", "devour", "drink", "sip", "read", "write", "love", "hate", "kick", "hit"])
print words.selfcompare()  # this one does have TF-IDF weighting

<table>
<thead>
<tr>
<th></th>
<th>eat</th>
<th>devour</th>
<th>drink</th>
<th>sip</th>
<th>read</th>
<th>write</th>
<th>love</th>
<th>hate</th>
<th>kick</th>
<th>hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>eat</td>
<td>1.000</td>
<td>0.243</td>
<td>0.041</td>
<td>0.016</td>
<td>0.022</td>
<td>0.017</td>
<td>0.150</td>
<td>0.094</td>
<td>0.065</td>
<td>0.034</td>
</tr>
<tr>
<td>devour</td>
<td>0.243</td>
<td>1.000</td>
<td>0.013</td>
<td>0.004</td>
<td>0.318</td>
<td>0.238</td>
<td>0.217</td>
<td>0.083</td>
<td>0.019</td>
<td>0.037</td>
</tr>
<tr>
<td>drink</td>
<td>0.041</td>
<td>0.013</td>
<td>1.000</td>
<td>0.721</td>
<td>0.010</td>
<td>0.004</td>
<td>0.054</td>
<td>0.077</td>
<td>0.026</td>
<td>0.096</td>
</tr>
<tr>
<td>sip</td>
<td>0.016</td>
<td>0.004</td>
<td>0.721</td>
<td>1.000</td>
<td>0.005</td>
<td>0.000</td>
<td>0.041</td>
<td>0.035</td>
<td>0.009</td>
<td>0.031</td>
</tr>
<tr>
<td>read</td>
<td>0.022</td>
<td>0.318</td>
<td>0.010</td>
<td>0.005</td>
<td>1.000</td>
<td>0.752</td>
<td>0.155</td>
<td>0.085</td>
<td>0.013</td>
<td>0.041</td>
</tr>
<tr>
<td>write</td>
<td>0.017</td>
<td>0.238</td>
<td>0.004</td>
<td>0.000</td>
<td>0.752</td>
<td>1.000</td>
<td>0.199</td>
<td>0.065</td>
<td>0.008</td>
<td>0.029</td>
</tr>
<tr>
<td>love</td>
<td>0.150</td>
<td>0.217</td>
<td>0.054</td>
<td>0.041</td>
<td>0.155</td>
<td>0.199</td>
<td>1.000</td>
<td>0.608</td>
<td>0.090</td>
<td>0.141</td>
</tr>
<tr>
<td>hate</td>
<td>0.094</td>
<td>0.083</td>
<td>0.077</td>
<td>0.035</td>
<td>0.085</td>
<td>0.065</td>
<td>0.608</td>
<td>1.000</td>
<td>0.071</td>
<td>0.108</td>
</tr>
<tr>
<td>kick</td>
<td>0.065</td>
<td>0.019</td>
<td>0.026</td>
<td>0.009</td>
<td>0.013</td>
<td>0.008</td>
<td>0.090</td>
<td>0.071</td>
<td>1.000</td>
<td>0.519</td>
</tr>
<tr>
<td>hit</td>
<td>0.034</td>
<td>0.037</td>
<td>0.096</td>
<td>0.031</td>
<td>0.041</td>
<td>0.029</td>
<td>0.141</td>
<td>0.108</td>
<td>0.519</td>
<td>1.000</td>
</tr>
</tbody>
</table>
We can try to form clusters from them, using K-means clustering (as above).

To do K-means clustering, you have to do two things to start with:

- Decide on K

- Choose a set of initial seeds. The hope is that the choice of seeds won’t matter too much, because the algorithm can fix poor initial choices, but obviously you’d like to have a sensible starting point.
What happens with ["eat", "devour", "drink", "sip", "read", "write"] if we decide to start with two seeds?

Choose seeds by randomly selecting sets of N items and calculating the sum of their pairwise cosines (call this their ‘density’). Choose the one with the lowest density, because that means it’s the one with the most dissimilar members (if all N items were the same then this would be \( N(N+1)/2 \), if they had absolutely nothing in common it would be 0).
sim.findClusters(objects, verbs, potentialseeds=["eat", "devour", "drink", "sip", "<WORD drink>, <WORD sip>] 0.61
[<WORD drink>, <WORD read>] 0.01
[<WORD write>, <WORD sip>] 0.00
[<WORD devour>, <WORD sip>] 0.00
[<WORD eat>, <WORD write>] 0.02
[<WORD eat>, <WORD sip>] 0.02
[<WORD write>, <WORD read>] 0.72
[<WORD drink>, <WORD write>] 0.01
[<WORD sip>, <WORD read>] 0.00
[<WORD drink>, <WORD devour>] 0.02
[<WORD drink>, <WORD eat>] 0.06
[<WORD eat>, <WORD devour>] 0.25
[<WORD write>, <WORD devour>] 0.02
[<WORD eat>, <WORD read>] 0.02
[<WORD devour>, <WORD read>] 0.03
The sensible ones produce good quality clusters. The first two pick ‘sip’ as one of the seeds.

```python
>>> sim.findClusters(objects, verbs, potentialseeds=['eat', 'devour', 'drink', 'sip'],
                  potentialseeds=['eat', 'devour', 'drink', 'sip'])

Generation 0
<CLUSTER devour: 1, 1.00> <CLUSTER sip: 1, 1.00>
Generation 1
<CLUSTER devour:eat:read:write: 4, 0.18> <CLUSTER drink:sip: 2, 0.61>
```

In each case we get two clusters, one sensible one (with quite a high score) and one less sensible one (with a lower score).
Next one chose ‘drink’ and ‘write’. This led to a different (but still sensible) set of clusters, but it took an extra step.

```python
>>> sim.findClusters(objects, verbs, potentialseeds=['eat', 'devour'])
Generation 0
<CLUSTER drink: 1, 1.00>
<CLUSTER write: 1, 1.00>
Generation 1
<CLUSTER drink:eat:sip: 3, 0.23>
<CLUSTER **devour**:read:write: 3, 0.26>
Generation 2
<CLUSTER **devour**:drink:eat:sip: 4, 0.16>
<CLUSTER read:write: 2, 0.72>
```
K-means is actually stable enough that we can pick a very silly pair of seeds and it will do the right thing.

```python
sim.findClusters(objects, verbs, seeds=['drink', 'sip'], potentialseeds=['eat', 'devour'])
```

**Generation 0**
- `<CLUSTER drink: 1, 1.00>`
- `<CLUSTER sip: 1, 1.00>`

**Generation 1**
- `<CLUSTER devour: drink: eat: read: write: 5, 0.12>`
- `<CLUSTER sip: 1, 1.00>`

**Generation 2**
- `<CLUSTER devour: eat: read: write: 4, 0.18>`
- `<CLUSTER drink: sip: 2, 0.61>`

*drink* has actually hopped out of the cluster that it was originally a seed in. That won't always work, but it is reassuring: we don't have to get it absolutely right first time.
But there are, of course, three clusters here. By setting $K=2$, we've forced here to be two. What happens if we choose $K=3$?

Generation 0
<CLUSTER devour: 1, 1.00>
<CLUSTER sip: 1, 1.00>
<CLUSTER read: 1, 1.00>

Generation 1
<CLUSTER devour:eat: 2, 0.25>
<CLUSTER drink:sip: 2, 0.61>
<CLUSTER read:write: 2, 0.72>

Looks good. But how could the program know that this is better than what we got with $K=2$? We’re looking for some kind of compromise between the number of clusters and their density (we could get really high density by setting $K=6$ for this example). I don’t know what would be a good compromise.
Detecting ambiguity

Everything above treats words as unitary objects. How similar are the sentences ‘I bought some shares in Apple’ and ‘I bought some shares in apples’? How similar are ‘I keep my money tied up at the bank’ and ‘I keep my boat tied up at the bank’?

When we simply have a corpus, all we’ve got is the words. Is there any way of telling whether a word has more than one sense, simply using the techniques we’ve seen so far?
If a word has two meanings, then things it will relate to will be of different kinds.

‘I fired a gun’ and ‘I fired my secretary’ involve different senses of ‘fire’. How can I tell? Because guns and secretaries are different kinds of things.

‘I ran a marathon’ and ‘I ran an athletics club’ involve different senses of ‘run’. How can I tell? Because marathons and athletics clubs are different kinds of things.
(53) a. I fired a gun
   b. I fired a bullet
   c. I fired a weapon
   d. I fired a rifle
   e. I fired a pistol
   f. I fired a cannon
   g. I fired a torpedo
   h. I fired a rocket
   i. I fired a worker
   j. I fired a servant
   k. I fired a researcher
   l. I fired a faithful [...]
   m. I fired an employee
   n. I fired a crewman
   o. I fired a colleague
Looks good. Two reasonably dense clusters, so ‘fire’ probably has two interpretations.

```
sim.findClusters(verbs, objects, potentialseeds=['gun', 'bullet', 'weapon',
Generation 0
<CLUSTER employee: 1, 1.00>
<CLUSTER crewman: 1, 1.00>
Generation 1
<CLUSTER colleague:employee:researcher:servant:worker: 5, 0.38>
<CLUSTER bullet:cannon:crewman:gun:pistol:
    rifle:rocket:torpedo:weapon: 9, 0.32>
```
But I cheated. I took out loads of other words. The actual cooccurrence vector for ‘fire’ is

\[
[(\text{'shoot'}, 19), (\text{'gun'}, 11), (\text{'imagination'}, 7), (\text{'bullet'}, 5), (\text{'weapon'}, 4), (\text{'safety'}, 4), (\text{'shutter'}, 3), (\text{'rifle'}, 3), (\text{'pistol'}, 3), (\text{'cannon'}, 3), (\text{'blood'}, 3), (\text{'ball'}, 3), (\text{'turn'}, 2), (\text{'torpedo'}, 2), (\text{'stream'}, 2), (\text{'service'}, 2), (\text{'rocket'}, 2), (\text{'question'}, 2), (\text{'prevention'}, 2), (\text{'potential'}, 2), (\text{'pin'}, 2), (\text{'people'}, 2), (\text{'furnace'}, 2), (\text{'engine'}, 2), (\text{'bolt'}, 2), (\text{'blank'}, 2), (\text{'year'}, 1), (\text{'worker'}, 1), (\text{'work'}, 1), (\text{'train'}, 1), (\text{'synapsis'}, 1), (\text{'summer'}, 1), (\text{'success'}, 1), (\text{'stave'}, 1), (\text{'staple'}, 1), (\text{'son'}, 1), (\text{'shotgun'}, 1), (\text{'shell'}, 1), (\text{'servant'}, 1), (\text{'salamander'}, 1), (\text{'roof'}, 1), (\text{'risk'}, 1), (\text{'response'}, 1), (\text{'researcher'}, 1), (\text{'reply'}, 1), (\text{'replica'}, 1), (\text{'range'}, 1), (\text{'pupil'}, 1), (\text{'pulse'}, 1), (\text{'place'}, 1), (\text{'physios'}, 1), (\text{'penetrometers'}, 1), (\text{'oven'}, 1), (\text{'officer'}, 1), (\text{'night'}, 1), (\text{'newshot'}, 1), (\text{'neuron'}, 1)]
\]
Detecting ambiguity
• Unexpected but actually quite common things: ‘he fired hundreds of questions at me’, ‘he fired a barrage of questions’, ‘it fired my imagination’

• Things that look likely be mistaggings or misparses: ‘a fire officer’, ‘a fire safety inspection’, ‘fire place’

• And just weird things: ‘fire a hould’, ‘fire a pentrometer’. They tend to be singletons, but then so are some of the real ones (‘blunderbuss’, ‘colleague’)

But that’s what real data is like.
Run our algorithm on all the words associated with our target item, with increasingly large sets of seeds. Choose the one where the maximum average density.
• 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 . . .

• Clustering within the collection of vectors that make up a word may tell you whether it’s ambiguous.
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).
Words are mapped to ‘synsets’: collections of names for different senses (how many senses does a word have: does ‘bank’ have two senses (edge of body of water, financial institution)?

(54) a. I left my cheque book in the bank
    b. I keep my money in the bank
    c. The bank keep overcharging me
    d. I keep my boat tied up at the bank
    e. There are rats living in the bank of the river that runs through the park
How many senses of ‘bank’ does Wordnet provide? What do they each mean?

```python
>>> wordnet.synsets("bank")
[Synset('bank.n.01'), Synset('depository_financial_institution.n.01'),
 Synset('bank.n.03'), Synset('bank.n.04'), Synset('bank.n.05'),
 Synset('bank.n.06'), Synset('bank.n.07'), Synset('savings_bank.n.02'),
 Synset('bank.n.09'), Synset('bank.n.10'), Synset('bank.v.01'),
 Synset('bank.v.02'), Synset('bank.v.03'), Synset('bank.v.04'),
 Synset('bank.v.05'), Synset('deposit.v.02'), Synset('bank.v.07'),
 Synset('trust.v.01')]
```

Which is pretty useless. **Word meanings only do anything by entering into relations with other words.**
Synsets are related to each other by superset and opposite relations. So the only way I can make any sense of the set of synset identifiers is by looking at the other items they are linked to.
for x in wordnet.synsets("bank"): print x.hypernyms()

[Synset('slope.n.01')]
[Synset('financial_institution.n.01')]
[Synset('ridge.n.01')]
[Synset('array.n.01')]
[Synset('reserve.n.02')]
[Synset('funds.n.01')]
[Synset('slope.n.01')]
[Synset('container.n.01')]
[Synset('depository.n.01')]
[Synset('flight_maneuver.n.01')]
[Synset('tip.v.01')]
[Synset('enclose.v.03')]
[Synset('transact.v.01')]
[Synset('act.v.04')]
[Synset('work.v.02')]
[Synset('give.v.03')]
[Synset('cover.v.01')]
[Synset('believe.v.01')]
Are they the ones you expected? Will any of them make sense for ‘I left my cheque book in the bank’? What’s the relation between ‘financial organisation’, ‘repository’ and monetary_resource? And most of them aren’t actually words anyway.

They’ve been written by people: so they contain mistakes and inconsistencies.
(the sets of opposites are even less reliable:

artifact natural_object
artefact natural_object
overachievement underachievement
appearance disappearance
appearance disappearing
debarkation embarkation
...
failing passing
failing pass
failing qualifying
call_option put_option
...
boy female_child
boy girl
boy little_girl
boy daughter

What does opposite mean? It can't be just non-intersection, but I really can't see what else all these have in common)
What should you be able to do?

(55) John and Mary got divorced ⊢ John and May used to be married but are no longer so.
Best I can get out of Wordnet:

?- getSensesAndHypernyms(divorce).
divorce-100916338: [separation-100916124]

?- getSensesAndHypernyms(marriage).
mariage-100788835: [ritual-100781685, rite-100781685]
mariage-106668788: [family-106657904, family_unit-106657904]
mariage-111778901: [marital_status-111778725]
mariage-112181303: [union-112178618, unification-112178618]

?- opposites(union).
union disunion
union separation

(56) John and Mary got divorced ⊨ John and May are not married.
What do I get by comparing object lists?

```python
>>> x = sim.sortTable(w['marri']['headNoun']) [('13-year-old', 1), ('adcock', 1), ('albertina', 1), ('aldi', 1), ('ali', 1), ('allan', 1), ('allenbi', 1), ('ann', 2), ('anytim', 1), ('art', 2), ('ashley-coop', 1), ('bailey', 1), ('baronet', 1), ('baron', 1), ('bellow', 1), ('blanch', 1), ..., ('cellist', 1), ('cousin', 9), (...('daughter', 11), ..., ('humbug', 1), ('idiot', 1), ..., ('widow', 2), ('wife', 2), ('wive', 1), ...] (187 cases)
```

```python
>>> x = sim.sortTable(w['divorc']['headNoun']) [('arm', 1), ('battl', 1), ..., ('daughter', 1), ..., ('husband', 2), ..., ('wife', 3)] (20 cases)
```
(I think a fair number of these are cases like ‘His married daughter comes to visit him on Sundays’; and you don’t get as many cases of ‘wife’ or ‘husband’ as you might expect because that’s so bound up with the meaning of the verbs)
It’s not enough: but you shouldn’t expect to get more out of sets of subset-superset and opposite links, or sets of cooccurrence patterns. Meanings are more complicated than that.
Using Personalised Page Rank for WSD

The words in the surrounding context provide the information that is used for disambiguation (‘word sense disambiguation’, ‘WSD’)

But how do we know which words are linked to the one whose meaning we are trying to determine? Because they occur in the same contexts

How do we know which words are linked to the target sense? Because they’ve been disambiguated for us in WordNet
What we want to know is: how likely is it that we want the second synset for bank given that it occurred in the context ‘I keep my boat tied up at the bank?’
‘Page rank’: the ‘random surfer model’. If a page has \( N \) links, then a visitor who makes a random choice about where to go next has a \( \frac{1}{N} \) chance of ending up on any one of these.

Markov model:

\[
\text{prob}(S^{t+1}) = \sum\text{prob}(S^t_i) \times \text{prob}(S_i \rightarrow S)
\]
p1 = simpleppr.initprobs()
simpleppr.run(p1, n=10, template=False)
Page rank: network simulator
Page rank: network simulator

A (0.00) B (0.33) C (0.33) D (0.33)

Allan Ramsay, IV SEMANTICS & INFERENCE -424- Using Personalised Page Rank for WSD
Page rank: network simulator

A (0.33) B (0.17)

C (0.33) D (0.17)
Page rank: network simulator

Allan Ramsay, IV SEMANTICS & INFERENCE  -426- Using Personalised Page Rank for WSD
Page rank: network simulator

A (0.28) → B (0.22) with weight 0.33
C (0.28) → D (0.22) with weight 0.50
A (0.28) → C (0.28) with weight 0.50
B (0.22) → A (0.28) with weight 0.50
B (0.22) → C (0.28) with weight 0.50
Page rank: network simulator

A (0.25) — 0.33 — B (0.20)
C (0.31) — 0.50 — D (0.23)

Using Personalised Page Rank for WSD
After a while this will converge.

A 1.00, C 0.00, B 0.00, D 0.00,
A 0.00, C 0.33, B 0.33, D 0.33,
A 0.33, C 0.33, B 0.17, D 0.17,
A 0.25, C 0.28, B 0.19, D 0.28,
A 0.28, C 0.28, B 0.22, D 0.22,
A 0.25, C 0.31, B 0.20, D 0.23,
A 0.27, C 0.29, B 0.20, D 0.24,
A 0.26, C 0.29, B 0.21, D 0.24,
A 0.26, C 0.30, B 0.21, D 0.23,
If we make links bidirectional, it can take much longer to converge.
Page rank: network simulator

A (0.00) → B (0.00) → C (0.00) → D (0.00)

A (0.00) ← B (0.00) ← C (0.00) ← D (0.00)
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.1813</td>
<td>0.0708</td>
<td>0.1771</td>
<td>0.3583</td>
</tr>
<tr>
<td>2</td>
<td>0.2273</td>
<td>0.0514</td>
<td>0.2036</td>
<td>0.2769</td>
</tr>
<tr>
<td>3</td>
<td>0.1927</td>
<td>0.0644</td>
<td>0.1821</td>
<td>0.3125</td>
</tr>
<tr>
<td>4</td>
<td>0.2078</td>
<td>0.0546</td>
<td>0.1874</td>
<td>0.2844</td>
</tr>
<tr>
<td>5</td>
<td>0.1958</td>
<td>0.0589</td>
<td>0.1797</td>
<td>0.2932</td>
</tr>
<tr>
<td>6</td>
<td>0.1996</td>
<td>0.0555</td>
<td>0.1801</td>
<td>0.2833</td>
</tr>
<tr>
<td>7</td>
<td>0.1954</td>
<td>0.0566</td>
<td>0.1769</td>
<td>0.2846</td>
</tr>
<tr>
<td>8</td>
<td>0.1960</td>
<td>0.0554</td>
<td>0.1763</td>
<td>0.2808</td>
</tr>
<tr>
<td>9</td>
<td>0.1943</td>
<td>0.0555</td>
<td>0.1748</td>
<td>0.2804</td>
</tr>
<tr>
<td>10</td>
<td>0.1942</td>
<td>0.0551</td>
<td>0.1742</td>
<td>0.2787</td>
</tr>
<tr>
<td>11</td>
<td>0.1934</td>
<td>0.0550</td>
<td>0.1735</td>
<td>0.2781</td>
</tr>
<tr>
<td>12</td>
<td>0.1932</td>
<td>0.0548</td>
<td>0.1730</td>
<td>0.2772</td>
</tr>
<tr>
<td>13</td>
<td>0.1928</td>
<td>0.0547</td>
<td>0.1726</td>
<td>0.2768</td>
</tr>
<tr>
<td>14</td>
<td>0.1926</td>
<td>0.0546</td>
<td>0.1723</td>
<td>0.2763</td>
</tr>
<tr>
<td>15</td>
<td>0.1924</td>
<td>0.0546</td>
<td>0.1720</td>
<td>0.2760</td>
</tr>
<tr>
<td>16</td>
<td>0.1923</td>
<td>0.0545</td>
<td>0.1718</td>
<td>0.2757</td>
</tr>
<tr>
<td>17</td>
<td>0.1922</td>
<td>0.0545</td>
<td>0.1717</td>
<td>0.2755</td>
</tr>
<tr>
<td>18</td>
<td>0.1921</td>
<td>0.0545</td>
<td>0.1716</td>
<td>0.2754</td>
</tr>
<tr>
<td>19</td>
<td>0.1920</td>
<td>0.0544</td>
<td>0.1715</td>
<td>0.2753</td>
</tr>
<tr>
<td>20</td>
<td>0.1920</td>
<td>0.0544</td>
<td>0.1714</td>
<td>0.2752</td>
</tr>
</tbody>
</table>

Allan Ramsay, IV SEMANTICS & INFRINGEMENT -432- Using Personalised Page Rank for WSD
Under certain conditions it’s **bound** to converge.

- No ‘sinks’ (no pages with no way out: a page which contains nothing but broken links is a sink)

- No inescapable loops: if my page points only to yours and yours points only to mine then you can’t get out of them.

- But you can’t rely on these conditions holding
  
  — Failure of either of these will lead to ‘**probability mass**’ being lost, because it will be injected into such nodes but it won’t go anywhere.

  — In the end everything will become zero.
Page rank: network simulator

\[ A \ (1.00) \rightarrow \ C \ (0.00) \rightarrow \ D \ (0.00) \rightarrow \ B \ (0.00) \rightarrow \ A \ (1.00) \]

- A → C: 0.33
- C → D: 0.50
- D → B: 0.50
- B → A: 0.33
- C → D: 0.50
- D → C: 0.50
- C → B: 0.33
- B → C: 0.33

Using Personalised Page Rank for WSD
After a while this will converge.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.33</td>
<td>0.33</td>
<td>0.33</td>
</tr>
<tr>
<td>3</td>
<td>0.33</td>
<td>0.00</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>4</td>
<td>0.08</td>
<td>0.11</td>
<td>0.19</td>
<td>0.11</td>
</tr>
<tr>
<td>5</td>
<td>0.11</td>
<td>0.03</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>6</td>
<td>0.06</td>
<td>0.04</td>
<td>0.08</td>
<td>0.05</td>
</tr>
<tr>
<td>7</td>
<td>0.04</td>
<td>0.02</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>8</td>
<td>0.03</td>
<td>0.01</td>
<td>0.03</td>
<td>0.02</td>
</tr>
<tr>
<td>9</td>
<td>0.02</td>
<td>0.01</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>10</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>11</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>12</td>
<td>0.01</td>
<td>0.00</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>13</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>16</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Allan Ramsay, IV SEMANTICS & INFERENCE -435- Using Personalised Page Rank for WSD
So we feed some probability mass in at every stage: that’s like our surfer randomly jumping somewhere completely random at every stage.

The ‘damping’ parameter says how likely he is to jump to somewhere linked to this page; otherwise there’s a \( \frac{1-damping}{|nodes|} \) chance of going to any node.

```python
simpleppr = ppr.ppr(links='A->B,A->C,A->D,B->C,B->D,D->A,D->C')
p1 = simpleppr.initprobs()
simpleppr.run(p1, preferred=['A', 'B', 'C', 'D'], damping=0.85, n=20)
```

Allan Ramsay, IV SEMANTICS & INFERENCE -436- Using Personalised Page Rank for WSD
Page rank: network simulator

A (0.00) B (0.00) C (0.00) D (0.00)

0.33 0.33 0.33 0.50

0.04 0.04 0.04 0.04

Allan Ramsay, IV SEMANTICS & INFERENCE -437- Using Personalised Page Rank for WSD
Page rank: network simulator

A (0.14) B (0.11)
C (0.32) D (0.21)

A → B: 0.33
B → C: 0.04
C → D: 0.50
D → A: 0.04

Allan Ramsay, IV SEMANTICS & INFERENCE -438- Using Personalised Page Rank for WSD
Page rank: network simulator

A (0.13) B (0.08) C (0.22) D (0.12)

0.33 0.33 0.33 0.50

A (0.13) B (0.08)

0.33 0.33 0.04 0.04 0.04 0.04

C (0.22) D (0.12)

0.33 0.50 0.50 0.50 0.04 0.04

Allan Ramsay, IV SEMANTICS & INFERENCE -439- Using Personalised Page Rank for WSD
Page rank: network simulator

A (0.09) B (0.07) C (0.16) D (0.11)

A → B: 0.33
A → C: 0.33
A → D: 0.50
B → A: 0.33
B → C: 0.04
B → D: 0.50
C → A: 0.50
C → B: 0.04
C → D: 0.50
D → A: 0.04
D → B: 0.04
D → C: 0.04

Allan Ramsay, IV SEMANTICS & INference -440- Using Personalised Page Rank for WSD
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1437</td>
<td>0.1083</td>
<td>0.3208</td>
<td>0.2146</td>
</tr>
<tr>
<td>0.1287</td>
<td>0.0782</td>
<td>0.2155</td>
<td>0.1243</td>
</tr>
<tr>
<td>0.0903</td>
<td>0.0740</td>
<td>0.1600</td>
<td>0.1072</td>
</tr>
<tr>
<td>0.0831</td>
<td>0.0631</td>
<td>0.1401</td>
<td>0.0945</td>
</tr>
<tr>
<td>0.0777</td>
<td>0.0610</td>
<td>0.1280</td>
<td>0.0878</td>
</tr>
<tr>
<td>0.0748</td>
<td>0.0595</td>
<td>0.1228</td>
<td>0.0854</td>
</tr>
<tr>
<td>0.0738</td>
<td>0.0587</td>
<td>0.1203</td>
<td>0.0840</td>
</tr>
<tr>
<td>0.0732</td>
<td>0.0584</td>
<td>0.1191</td>
<td>0.0834</td>
</tr>
<tr>
<td>0.0729</td>
<td>0.0582</td>
<td>0.1185</td>
<td>0.0831</td>
</tr>
<tr>
<td>0.0728</td>
<td>0.0582</td>
<td>0.1182</td>
<td>0.0829</td>
</tr>
<tr>
<td>0.0727</td>
<td>0.0581</td>
<td>0.1181</td>
<td>0.0828</td>
</tr>
</tbody>
</table>

...
Doing this gets you the ‘most important’ nodes. It’s how Google started Page et al. (1999) (don’t know how similar it is to what they do now).

But I’m most likely to be interested in things that my friends are doing and reading about and are interested in.

Then their home pages should have some extra influence. I’ll reserve a bit of the probability mass, and at each cycle I’ll divide that (evenly) among my friends’ pages. Then their pages and ones that they are interested in will become more important to me.
Allow a subset of nodes to have more of the injected mass (in the simulation below I’m letting the preferred nodes have all the mass, but when I start doing it with WordNet I give extra mass, but not all of it to the preferred nodes).

```
simpleppr = ppr.ppr(links='A->B,A->C,A->D,B->C,B->D,D->A,D->C')
p1 = simpleppr.initprobs()
simpleppr.run(p1, preferred=['A','D'], damping=0.85, n=20)
```

A 0.1813, C 0.2833, B 0.0708, D 0.2521, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1821, C 0.1886, B 0.0514, D 0.1565, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1415, C 0.1399, B 0.0516, D 0.1484, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1381, C 0.1251, B 0.0401, D 0.1370, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1332, C 0.1144, B 0.0391, D 0.1312, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1307, C 0.1101, B 0.0377, D 0.1294, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1300, C 0.1081, B 0.0370, D 0.1281, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1294, C 0.1070, B 0.0368, D 0.1276, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1292, C 0.1065, B 0.0367, D 0.1273, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1291, C 0.1063, B 0.0366, D 0.1272, A 0.1290, C 0.1061, B 0.0366, D 0.1271,
A 0.1291, C 0.1062, B 0.0366, D 0.1271,
Links are between synsets, i.e. between word senses.

- subset/superset links

- between a synset and the synsets of words that appear in its gloss.
From a word we can find its synsets (so from a sentence we can find the synsets of all the words)

"bank" = [Synset('bank.n.01'), Synset('depository_financial_institution'),
          Synset('bank.n.03'), Synset('bank.n.04'),
          Synset('bank.n.05'), Synset('bank.n.06'), Synset('bank.n.07'),
          Synset('savings_bank.n.02'), Synset('bank.n.09'), Synset('bank.v.01'),
          Synset('bank.v.02'), Synset('bank.v.03'), Synset('bank.v.04'),
          Synset('bank.v.05'), Synset('deposit.v.01'),
          Synset('bank.v.07'), Synset('trust.v.01')]
From a synset we can find up/down links:

```python
>>> bank = wordnet.synsets("bank")
>>> bank[0].hypernyms()
[Synset('slope.n.01')]
>>> bank[0].hyponyms()
[Synset('riverbank.n.01'), Synset('waterside.n.01')]
```

So you could make a network out of these: ‘bank.n.01’–’slope.n.1’, ‘bank.n.01’–’riverbank.n.01’, ‘bank.n.01’–’waterside.n.01’, ...
From a synset we can find a definition (‘gloss’):

```python
>>> bank[0].definition()
u’sloping land (especially the slope beside a body of water)’
```

And then from each open-class word in the definition we can get a set of synsets:

```python
>>> wordnet.synsets('water')
[Synset('water.n.01'), Synset('body_of_water.n.01'), Synset('water.n.
```

And then we get ’bank.n.01’–’water.n.01’, ....
Run page rank on this, you’ll get the most prominent synsets for a word.

Run **personalised** page rank, where the personalised links are the synsets of the words in the target sentence, you’ll get the most prominent synsets given the context. Glosses involve word, but links are between synsets. We have to give extra probability mass to all the synsets for all the words in the context, even though many of these are not relevant.

This is the best current knowledge-based WSD algorithm Agirre et al. (2014).
Matrices & eigenvectors

Some numbers: 29054 nodes, 2319281 links. Which sounds like a lot.

You can represent a network as an array.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>c</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td>d</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

Figure 21: Matrix representation of a network

This says $a \to b, b \to c, b \to d, c \to a, d \to a, d \to b, d \to c$. 
This lets you apply various bits of maths that apply to matrices, e.g. the stable set of probabilities that emerges after we’ve run it for a few steps is an ‘eigenvector’ of the probability matrix.

More importantly for us, it lets us use algorithms that can multiply ‘sparse’ matrices efficiently. If we actually drew our matrix out as in Fig. 21, it would be 29054*29054 = 844134916 cells, of which 844134916-2319281 = 841815635 would be empty. Using the Python sparse matrix representation lets us do a complete update in 0.04 seconds.
Standard version gives you the nodes that are most important given the context. That’s a mixture of most frequent (most frequent in glosses, which \( \approx \) most frequent in the general language) and relevant to other items in the context.

Might be good to choose the ones whose importance is boosted most by the context.

Some combination of the two is probably optimal. I haven’t run a million experiments to see what combination to use.
Good ones

(57) bank–the pilot pulled on the joystick
(58) bank–my money is safe there
(59) bank–I rowed my boat there
(60) secretary–the boss fired her
(61) secretary–antique furniture
Lots of bad ones. Some are due to the fallibility of WordNet. Why does it get the wrong interpretation of ‘fire’ in

```python
>>> sppr.chooseInterpretation("the boss fired his secretary", "the boss fired his secretary")
```

Because the right interpretation isn’t linked as a synset of ‘fire’!
• disambiguates all the words in the target material in one pass.

• doesn’t require someone to sit down and disambiguate a large amount of material (is the riverbank sense of ‘bank’ really the most common? How many words would you have to look at to be sure of that?) (but it did require someone to write the glosses).

• Isn’t as good as just picking the most frequent word sense! (Supervised) data-driven beats rule-based, but takes far more effort (that’s not what people usually say but it’s true).
• The structure of WordNet: the key data structures (the ones that enter into ‘hypernym’/‘hyponym’ relations) are synsets, identified by unique integers.

• Personalised page rank:
  – main algorithm, need for boost to cope with leaks, adding preferences
  – application to WSD via wordnet: using glosses to build the network, personalisation is through words but links are between synsets, boost rather than absolute value
  – use of matrix and sparse matrix representations of graphs
Because WordNet entries are hand-coded, they are fallible. But what you can do without hand-coded knowledge is very limited (what you know about English is a mixture of observation (corpus-based) and what you were told (hand-coded)).
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
Every man has a mother, John is a man ⊢ John has a mother

Translate from English into logic.

\[
\text{utt(claim,}
\begin{align*}
&\text{forall(_A :: \{\text{man(_A)}\},} \\
&\qquad \text{exists(_B :: \{\text{mother(_B)}\},} \\
&\qquad \qquad \text{exists(_C,} \\
&\qquad \qquad \quad \text{event(_C, has)} \\
&\qquad \qquad \quad \& (\text{theta(_C, object, _B!3)} \\
&\qquad \qquad \quad \& (\text{theta(_C, agent, _A!0)} \\
&\qquad \qquad \quad \& \text{aspect(now, simple, _C)))))))))
\end{align*}
\]

\[
\text{utt(claim, man(ref(lambda(_A, named(_A, 'John')))!0))}
\]
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 $S$.

(63) I saw a man in the park ⊢ I saw a man

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

(64) I saw a man in the park ⊢ 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 $S'$.

(65) I saw a man in the park ⊬ I saw a human

(66) 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: we’re going to see it a several more times, it underlies (with some changes) spelling correction algorithms, DNA sequencing, . . .

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>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>-1</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>
<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>
<tr>
<td></td>
<td>w</td>
<td>h</td>
<td>i</td>
<td>c</td>
<td>h</td>
<td>s</td>
<td></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>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td></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></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></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></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></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></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></td>
</tr>
<tr>
<td></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>-1</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>
<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>
<tr>
<td></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>-1</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>
<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>
<tr>
<td></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>-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
String-edit distance
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>2</td>
<td>4</td>
<td>6</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>7</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
Allan Ramsay, IV SEMANTICS & INference

String-edit distance
String-edit distance
Allan Ramsay, IV SE\textsc{EMANTICS} & IN\textsc{ERENCE} -483- String-edit distance

\begin{verbatim}
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  7  9  11
t  6  7  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}
c  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}
h  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}
e  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}
s  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}  \text{-1}
\end{verbatim}
String-edit distance
String-edit distance
String-edit distance
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>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>-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
Allan Ramsay, IV SEMANTICS & INference -493- 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>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>8</td>
<td>10</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

Allan Ramsay, IV SEMANTICS & INFERENCE -496-
Allan Ramsay, IV SEMANTICS & INference

String-edit distance

Which s

0 2 4 6 8 10 12

w 2 0 2 4 6 8 10 10

i 4 2 3 2 4 6 8 8

t 6 4 5 4 5 7 9 9

c 8 6 7 6 4 8 10 10

h 10 8 6 -1 -1 -1 -1 -1

e -1 -1 -1 -1 -1 -1 -1 -1

s -1 -1 -1 -1 -1 -1 -1 -1
Allan Ramsay, IV SEMANTICS & INference -499- 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>6</td>
<td>8</td>
</tr>
<tr>
<td>t</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>h</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>9</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

Allan Ramsay, IV SEMANTICS & INFERENCE
String-edit distance

Allan Ramsay, IV SEMANTICS & INFERENCE

-505-
String-edit distance

Allan Ramsay, IV SEMANTICS & INference

-507-
String-edit distance
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, . . .

For textual entailment, exchange is free if the source item is a WordNet subset of the target item, deletion is free for modifiers, . . .
(you can precompute all possible supersets of every word with a superset in WordNet reasonably quickly (a bit under 3 seconds).
And then you can more or less instantly check whether one word is a subset of another:

```python
>>> 'human' in taggedhyps['n']['man']
True

>>> taggedhyps['n']['man']
{'artefact': 4, 'help': 2, 'helper': 2, 'being': 5, 'entity': 7, 'homo': 1, 'hominid': 2, 'supporter': 2, 'unit': 7, 'underling': 1, 'instrumentation': 3, 'g':
'brute': 8, 'equipment': 2, 'animal': 8, 'manservant': 1, 'cause': 5, 'creature'::
'abstraction': 2, 'somebody': 4, 'eutherian': 4, 'assistant': 2, 'object': 8, 'wo

>>> taggedhyps['n']['woman']
{'cleaner': 1, 'workingman': 3, 'adult': 1, 'people': 2, 'being': 3, 'entity': 5,'group': 3, 'stratum': 1, 'employee': 4, 'cause': 3, 'abstraction': 4, 'somebody':
'laborer': 2, 'jack': 2, 'class': 1, 'laborer': 2, 'whole': 5, 'workman': 3, 'so
'mortal': 2, 'grownup': 1, 'organism': 3, 'grouping': 3})
EXCH = 0 if the two words have the same part of speech and the first is subset of the second (some interpretation of the first is a subset of some interpretation of the second: abductive disambiguation (Hobbs et al. 1993))

DELETE = 0: would it be better to set it to 0 for modifiers (adjectives, preps)?

>>> 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 \( I_{\text{pos}} \) for an insert, \( D_{\text{pos}} \) for a delete, \((W_{\text{sim}} \times \text{sim} + W_{\text{opp}} \times \text{opp} + \ldots)\) where \( I_{\text{pos}} \) is the cost of inserting an item with a given POS tag, \( D_{\text{pos}} \) is the cost of deleting an item with a given POS tag, \( \text{sim} \) is your favourite similarity measure and \( W_{\text{sim}} \) is how much you think similarity matters, \( \text{opp} \) is 0 or 1 depending on whether the words are WordNet opposites and \( W_{\text{opp}} \) is how much you think that oppositeness matters, \ldots

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, \ldots)

Can be extended to apply to trees (Alabas 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
Define inference operations directly on parse trees (probably easiest with dependency trees).

Split it into domain rules and inference principles.
(67) a. John and Mary got divorced $\vdash$ John and Mary used to be married.

b. John and Mary used to be married $\vdash$ John and Mary are not married.
got:VB  ⇒  used:VB

John:NN  divorced:NN  John:NN  to:TO

and:CC  Mary:NN  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)
John and Mary used to be married.

⇒

John and Mary are not married.

Domain rules

Allan Ramsay, IV SEMANTICS & INference
Better to use variables: add \( X, Y, \ldots \) to English.

(68) 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 \[\rightarrow\] does:VB
\[
\text{X:NN} \quad \text{to:TO} \quad \text{X:NN} \\
\quad \text{Y:NN} \quad \not:RB \quad \text{Y:NN}
\]
Given \( \{P \Rightarrow Q, P\} \) you can derive \( Q' \) where \( Q' \subseteq Q \).

\[
P \Rightarrow Q
\]

\[
P'
\]

\[
Q'
\]

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 backtracking, 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):

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):

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 $\lambda$-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)
Approximate matching with variables and a subsumption lattice

```python
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], hypernyms, contn)
        while len(x) <= len(y):
            match(x, tl(y), hypernyms, contn)
```

Complexity was linear in size of the terms being matched. Now exponential in the difference in size between t1 and t2.
Then for the inference engine we do

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

```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))
```

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

5hyponyms is the inverse of hypernyms

Allan Ramsay, IV SEMANTICS & INFERENCE -539- General principles
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))
...
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: ['+', '-'], \ldots\}

\begin{verbatim}
doubted:VB
   I:PR liked:VB
      that:TH she:PR me:PR
\end{verbatim}

\{'doubt',VB,+,
   [{'I',PR,+,[]},{'like',NN,-,
      [{'that',TH,-,[]},{'she',PR,-,[]},{'me',PR,-,[]}]}]}
t1 = COMP34411.te.makeTree(COMP34411.fp.parse("I doubted that she loved ")

\[
t2 = \text{COMP34411.te.makeTree(COMP34411.fp.parse("I doubted that she liked ")

\[
\text{COMP34411.te.subtree(t1, t2)}
\]

\[
\text{COMP34411.te.subtree(t2, t1)}
\]

Allan Ramsay, IV SEMANTICS & INference -545- General principles
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’ ⊢ ‘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 (pseudocode), not detail (code))
‘How can you get your computer to translate documents that are written in some foreign language?’

A lot of my examples will come from Arabic: a few of you know Arabic, but they won’t have any advantage over the others. They may even have a disadvantage, because they won’t have to think about the examples as hard.

Arabic has one very weird property: you leave out half the vowels when you write it, so it’s just like writing txt msgs used to be. I will generally leave out the vowels in example sentences, and I’ll often put them in in things like parse trees.
What’s the goal of translation?

To construct text or speech in the target language that expresses the same message as the text or speech in the source.

How do we do that?

Build a data structure that represents the meaning of the source, use it to generate some text in the target language.
Build a data structure that represents the meaning of the source?

We've seen what data structures that represent the meaning in logic look like:

```prolog
?- analyse('the man wrote a book.').

utt(claim,
   exists(_A :: {book(_A)},
      exists(_B :: {past(now, _B)},
         exists(_C,
            (event(_C, 'wr/i/o/te')
             & (theta(_C, object, _A!3)
             & (theta(_C, agent, ref(lambda(_D, man(_D)))!0)
             & aspect(_B, simplePast, _C))))))

****/
```

It’s complex: it needs to be complex to preserve all the information that was in the text)
Use it to generate the target

Convert the meaning representation into the target language:

\[
| \texttt{?- analyse(}'ktb \textit{Alrjl ktb.}', \texttt{arabic}). \\
\texttt{utt(claim,} \\
\quad \texttt{exists(}_A:: \{\texttt{past(now, }_A)\},} \\
\quad \texttt{exists(}_B,} \\
\quad \quad \texttt{(exists(}_C:: \{\texttt{'}k?t?b'(}_C)\},} \\
\quad \quad \quad \texttt{(event(}_B, \texttt{'}k?t?b')} \\
\quad \quad \quad \quad \texttt{& (theta(}_B, \texttt{agent, ref(l lambda(}_D, \texttt{'}r?j?l'(}_D))))} \\
\quad \quad \quad \quad \texttt{& theta(}_B, \texttt{object, }_C!0))))} \\
\quad \quad \quad \quad \texttt{& aspect(}_A, \texttt{simple, }_B)))}}} \\
\texttt{\texttt{))}}
\]

Generate a string of words from this.
How would I convert from an English meaning rep to an Arabic one?

man ==> r?j?l
book ==> k?t?b
write ==> k?t?b
...

(I’m going to have to do something like that, whatever I do)
How will I generate an Arabic sentence from this meaning rep?

VERY HARD: it’s very difficult to map the various bits of the meaning representation to words that might have contributed those bits. You can work out that ‘r?j?l’(_D) comes from ‘Alrjl’, but how do you allocate aspect(_A, simple, _B) to the form of the verb.

I don’t know of anyone who can do this.
Is there any other kind of datastructure that represents the meaning?

The parse tree! That’s what we did when we were doing inference earlier on.
The fact that the subject of the English sentence is ‘the man’ corresponds fairly directly to the fact that the subject of the target Arabic sentence is ‘Alrjl’; even better, the fact that English verb is simple past tense corresponds fairly directly to the fact that the Arabic verb is in the perfect tense (we’ll come back to this: English has more tenses than Arabic, so that’s likely to cause us a problem)
[.,
  (arg(claim)
  - [wrote,
      (arg(object) - [book, (identity - [a])]),
      (arg(agent) - [man, (identity - [the])])])]

==> 

[.,
  (arg(claim)
  - [kataba,
      (arg(agent) - [Alrajulou]),
      (arg(object) - [kutubF])])]

Allan Ramsay, V MACHINE TRANSLATION -558- Use it to generate the target
Easy enough to do the substitutions:

[.,
 (arg(claim)
  - [wrote,
    (arg(object) - [book, (identity - [a])]),
    (arg(agent) - [man, (identity - [the])])])]

==>
[.,
 (arg(claim)
  - [kataba,
    (arg(object) - [kutbF, (identity - [a])]),
    (arg(agent) - [rajul, (identity - [the])])])]

Allan Ramsay, V MACHINE TRANSLATION -559- Use it to generate the target
But there were some bits that didn’t get translated. Arabic has no indefinite article (‘a’), and the definite article is written attached to the noun:

‘Alrajul’ rather than ‘Al rajul’

And the arguments that I got by analysing the Arabic are in a different order from how they appear in the English’:

‘yktb Alrjl ktb’ rather than ‘Alrjl yktb ktb’
So I have to start messing around with my trees. I could cope with the definite article by treating it as separate word, and attaching it to the adjacent item in the written form (in the same way that when I use the reduced form of ‘not’ in English I attach it to the previous word — ‘I don’t think I understand this’). That’s just about the writing system—not very exciting (like the fact that different languages use different character sets, or that languages that use the Arabic character set are written from right→left).

But I have to delete the indefinite one. That’s more problematic.
The meaning is carried by the words and the mode of combination: which includes syntactic features. So if I include these in my trees, maybe I can do something:

wrote

book ([-def])  man ([+def])

a  the

kataba

Alrajul ([+def])  kutubF ([-def])
I need a rule like

\[ X ([\text{-def}]) \quad X' ([\text{-def}]) \]

\[ \quad \Rightarrow \quad \]

\[ a \]

And then I apply this rule recursively to my English tree.
This rule ‘transfers’ information about semantically significant aspects of the English tree to the Arabic one.
What else can we use transfer rules for?

Other places where items get deleted/inserted.

| ?- analyse('I am writing a book.').

am [+aux]

writeing

book([-def]) I [+def, +pron]

a

==>

| ?- analyse('nktb ktb.', arabic).

nakotubu

0 [+def] kutubF [-def]

The auxiliary (‘am’ has disappeared); and so has the pronoun ‘I’.
Delete the auxiliary from the English

\[ X \ [+\text{aux}] \]
\[
\begin{array}{c}
\text{-------} \\
Y
\end{array}
\]
\[
\begin{array}{c}
\text{---} \\
Y' \\
\text{-------}
\end{array}
\]

Delete the pronoun from the English

\[ X \]
\[
\begin{array}{c}
\text{------------------------} \\
Y \ [+\text{pron}, +\text{nom}]
\end{array}
\]
\[
\begin{array}{c}
\text{------------} \\
Y' \\
\text{---}
\end{array}
\]
These look like they will do what’s needed; and they also look like they should be reversible (i.e. that after I’ve applied them I will be able to generate the surface form from tree, not that the same rules will work in either direction: which auxiliary should I insert when going from Arabic to English, which pronoun ...?).
But you need to be careful about the morphology:

?- analyse('He is writing a book.').
-------------------------------

is [+aux, 3rdsing]
-------------------------------

writeing
-------------------------------

book([-def]) he ([+def, +pron])
--------
a

=>

?- analyse('yktb ktb.', arabic).
-------------------------------

takotubu[3rdsing]
-------------------------------

0([-def]) kutubF([-def])

Allan Ramsay, V MACHINE TRANSLATION
One language may make distinctions that the other doesn’t:

- English has ‘they’, Arabic has ‘yktbAn’ (‘they both, masculine’), ‘tktbAn’ ‘they both, feminine’, ‘yktbwn’ (‘they all, masculine’) and ‘yktbn’ (‘they all, feminine’). So you can’t tell how to translate ‘they’ without knowing a lot about the context of the utterance, because you have to know who ‘they’ are.

- Arabic has present (‘Oktb’), English has simple present (‘I write’) and present progressive (‘I am writing’) (lots of languages don’t have this distinction—French, Dutch, German). So you can’t tell how to translate ‘Oktb’ without knowing a lot about the context of the utterance.
Hard to write a program that can cope with this (also hard for language learners: if your native language doesn’t make a distinction between $X$ and $X'$, it’s hard for you to learn how to make that distinction in a new language. Non-English speakers find it very hard to learn when to use the ‘bare plural’ – ‘I like Italian peaches’ rather than ‘I like the Italian peaches’/I like some Italian peaches/I like all Italian peaches).
‘Lexical gaps’, ‘support verbs’: source language may have a single word where target language requires some longer expression.

(69) a. I am looking for a unicorn.
    b. Je suis à la recherche d’une licorne

(try ‘They dined off quince’, ‘we lunched at the Ritz’ in standard translation engines). Particularly common where one language has a proper verb for something and the other has a ‘support/light verb’—‘do a sum’ vs ‘prove a theorem’, ‘have tea’ vs ‘dine’, …
Idioms:

die [agr=AGR, vform=VFORM]                      =>                      etre [agr=AGR, vform=VFORM]
-----------------------------------------------           -----------------------------------------------
             X                        X’              mort [agr=AGR]

kick [agr=AGR, vform=VFORM]                      =>                      etre [agr=AGR, vform=VFORM]
-----------------------------------------------           -----------------------------------------------
        X                  bucket                X’              mort [agr=AGR]
         -------          the
My transfer rules can operate at different ‘levels of abstraction’.

Word for word pairs

I ==> je  he ==> il
I ==> ich  he ==> er
I ==> ik  he ==> hij
je ==> I  il ==> he
ich ==> I  er ==> he
ik ==> I  hij ==> he
je ==> ich  il ==> er
I ==> ∅ (Italian, Spanish)  he ==> ∅ (Italian, Spanish)
je ==> ∅ (Italian, Spanish)  il ==> ∅ (Italian, Spanish)
...
Pronouns become pronouns:

\[\text{[form}=X, \text{pro, case}=C, \text{person}=P, \text{number}=N, \text{gender}=G\]\n
\[==>[\text{form}=X', \text{pro, case}=C, \text{person}=P, \text{number}=N, \text{gender}=G]\]

'he' = \[\text{[form}="he", \text{pro, case}=\text{subj, person}=3\text{rd, number}=\text{singular, gender}=\text{masc}\]\n
'il' = \[\text{[form}="il", \text{pro, case}=\text{subj, person}=3\text{rd, number}=\text{singular, gender}=\text{masc}\]\n
'er' = \[\text{[form}="er", \text{pro, case}=\text{subj, person}=3\text{rd, number}=\text{singular, gender}=\text{masc}\]\n
One rule for all pronoun cases: I can use it for translating from English to French, or from German to Dutch, or, ...
High level rules cover more cases, and are more portable between languages. The rule above will let me translate from English into French, German, Dutch, . . . , so long as I know what the 1st person singular subject-case pronoun (or the third person masculine singular subject-case pronoun, or . . . ) is in the target language. Which isn’t much better than translating from English into French by knowing that ‘I’ is ‘je’.
But it also tells me how to translate ‘il’ into Dutch: I don’t need English → French and English → German and English → Dutch and French → English and French → German and French → Dutch and German → English and German → French and German → Dutch dictionaries: I just need English → I and French → I and German → I and Dutch → I, where I is some ‘interlingua’.
So if we could do it that way, we’d have $N \rightarrow L$ interlingua dictionaries (and perhaps $N$ interlingua to L dictionaries) for $N$ languages, rather than $N \times N - 1$ dictionaries.

The ‘machine translation pyramid’

What can we use as an interlingua? That’s the key.
(these rules are very like the rules we were using for textual entailment: a good translation should be ‘truth preserving’. The differences are

• in textual entailment you don’t have to generate the surface form of the derived sentence
• in machine translation you don’t have to do very long chains of inference

You can see tasks such as text simplification as being translations from one variant of a language to another with a more restricted vocabulary or simpler grammatical rules)
Remember that we have three kinds of ambiguity: structural, lexical and scope.

If we get them wrong in the analysis of the source sentence, we’re likely to have a problem with the target.
'Structural ambiguity' often turns out to be unproblematic:

(70)  
  a. I saw the man in the park with a telescope  
  b. J’ai vue l’homme dans le parc avec un telescope  
  c. Ik zag de man in het park met een telescoop.  

  a. I saw the man with a big nose  
  b. J’ai vue l’homme avec un grand nez  
  c. Ik zag de man met een grote neus

(71)  
  a. strawberry jam jar, glass jam jar  
  b. pot de confiture de fraises, pot de confiture en verre  
  c. aardbeienjam jar, glazen jampot: the English must have been disambiguated to get these two.

8I used Google translate for lots of the next few slides
Much the same is true for ‘scope ambiguities’. If two languages have similar sets of quantifiers, and similar ways of expressing ideas, then they are likely to display the same scope ambiguities:

(69) a. I am looking for a unicorn.
    b. Je suis à la recherche d’une licorne
But ‘lexical ambiguity’ doesn’t tend to transfer:

(72) a. I keep my money tied up at the bank.
   b. Je garde mon argent liée à la banque.

(73) a. I keep my money tied up at the bank.
   b. Je garde mon bateau amarré à la rive.
Where do we get rules and bilingual dictionaries from?

Can we extract them from corpora?

Bilingual corpora, parallel corpora, translation corpora
Is there anything you can do if you just have two corpora, in different languages and with no reason to believe that they are connected?

The following table is from two disjoint chunks of the BNC (2.7M sentences each, around 30 words per sentence).
Allan Ramsay, V MACHINE TRANSLATION -585- Bilingual corpora, parallel corpora
The BNC is supposed to be ‘balanced’ text, drawn from all sorts of sources. It seems plausible that if the distribution of the top words is like this here, it will be pretty like this across any very large corpus.

It **might** turn out that the distribution in a similar balanced corpus for another language will be similar. In which case you could use it to make guesses, at least about very common words.

But French people might be less interested than we are in when things happen, or in public affairs.
It would probably work better if we could split the corpus into chunks that were about the same kinds of thing.

How could I split two corpora in different languages into chunks that were about the same kinds of thing if I didn't know how to translate between them?

Use proper names: the French for Barack Obama is Barack Obama. Find the commonest name in the English corpus, extract all documents in either that contain it. They all have something in common. Then do the same again with the remainder, . . .
The more closely matched the two corpora are, the easier it gets.

(74) a. I hold the post of Professor of Formal Linguistics. What this means is that my research is directed at obtaining precise formal descriptions of the way that natural language works. People have studied language for thousands of years.

b. Je detiens le poste de professeur de linguistique formelle. Ce qui veut dire que ma recherche vise a obtenir des descriptions formelles précises de la façon dont fonctionne le langage naturel. Les gens ont étudié la langue pendant des milliers d’années.
If you had nice sentence-by-sentence literal translations of this kind, you could easily collect potential translation pairs.

- Match the i’th word in the English to the i’th word in the French. But English might be more/less wordy than French.

- Match the $i_{E}$’th word in the English to the $i_{F}$’th word in the French

- Match the $i_{E}$’th word in the English to the $i_{F}$’th word in the French if they are roughly the same length

- Match the $i_{E}$’th word in the English to the $i_{F}$’th word in the French if they are roughly the same length or share consonants

...
That will get you lots of good suggestions: works best if

• for each source language sentence there is one target language sentence

• the two languages use about the same number of words to express an idea

• they have much the same word order

• they have a common ancestry
Translations are supposed to preserve the meaning of the originals.

And aspects of the meaning of the original are carried by the way it is broken up into paragraphs and sentences.

So you can hope that the number and order of the sentences in the two texts will be similar. **You can’t assume they will be the same.**
Suppose my two texts were as follows:

e1 e2 e3. e4 e5. e6 e7 e8. e9 e10 e11 and e12 e13 e14 and e15.
f1 f2 f3 and f4 f5 and f6 and f7 f8. f9 f10 f11. f12 f13 f14. f15.

You’d want to say that \{e1 e2 e3. e4 e5. e6 e7 e8.\} and
\{f1 f2 f3 and f4 f5 and f6 and f7 f8.\} should be aligned, and
likewise \{e9 e10 e11 and e12 e13 e14 and e15\} and \{f9 f10 f11. f12 f13 f14. f15.\} – the groups of sentences I’ve linked contain similar numbers of words, so are likely to express similar ideas.
Align them with our ‘**dynamic time warping**’ algorithm.

Recall that you can use cost functions that depend on what you are swapping: try $\frac{\text{len}(w)}{\text{len}(w')}$, where $w$ is the longer of the two words, so that it costs you less to swap two words of similar lengths. Try that on my passage about professors of linguistics.
import nltk
import dtw

english = nltk.tokenize.wordpunct_tokenize("I hold the post of Professor of Formal Linguistics. What this means is that my research is directed at obtaining precise formal descriptions of the way that natural language works. People have studied language for thousands of years. ")
french = nltk.tokenize.wordpunct_tokenize("Je detiens le poste de veut dire que ma recherche vise a obtenir des descriptions formelles precises de la facon dont fonctionne le langage naturel. Les gens ont etudie la langue pendant des milliers d’annees. ")

a = dtw.array(english, french, EXCHANGE=dtw.matchStrings)
alignment = a.showAlignment()
print dtw.csvalign(alignment)
I hold the post of Professor of Formal Linguistics. What this means is that my research is directed at obtaining precise descriptions of the natural language, which works the way it does. People have studied language for thousands of years, and have obtained descriptions that function in a precise way. Je detiens le poste de professeur de linguistique formelle. Ce que ma recherche vise est obtenir des descriptions formelles précises de la façon dont le langage naturel fonctionne. Les gens ont étudié la langue pendant des milliers d’années.
Not too bad: but I’m missing some obvious things, where the English and French words are very similar: ‘obtaining’ ≈ ‘obtenir’, ‘studied’ ≈ ‘etudie’

I’d like to pay attention to whether words look like each other when I’m doing my alignment.

How can I measure whether two words look like each other? By aligning them!

(pay extra attention to punctuation, since it’s most likely to correspond to punctuation in the other language)
I hold the post of Professor of Formal Linguistics. What this means is that my research is directed at obtaining precise descriptions of the way that natural language works. People have studied the language for thousands of years. All people have studied the language for thousands of years because they wanted to understand how it works. The way it works is complex and involves many different elements. However, by using formal linguistic analysis, we can create a more precise understanding of how natural language functions.
• Distinctions that are present in one language and not the other: ‘the’ → ‘le’ and ‘la’

• Words that are missing in one but present in the other: English bare NPs (‘precise formal descriptions’, ‘natural language’, ‘language’) are realised as definite NPs in French (‘des descriptions formelles précises’, ‘le langage naturel’, ‘la langue’)

• Lexical ambiguity: ‘natural language’=‘le langage naturel’, ‘language’=‘la langue’

• Local changes in word order: ‘natural language’=‘le langage naturel’, ‘Professor of Formal Linguistics’=‘professeur de linguistique formelle’ (very like spelling error transpositions: fix the DTW algorithm to allow swaps as well as insertions, deletions, exchanges)
Assuming that I’ve aligned my sentences reasonably, I can start looking for pairs.

Working assumption: if $W^L_1$ in $S^L_1$ and $W^L_2$ in $S^L_2$ have been matched during the alignment process then $W^L_2$ might be a translation of $W^L_1$.

Count all such pairs: most likely translation is commonest item.
Some experiments with the only translation corpus I’ve got, which is the Qur’an: good testbed, because

- alignment is difficult
  - we’ve got verse-by-verse alignment, but some verses are very long
  - hard to do word-by-word alignment within verses because Arabic words can be compounds, and because Arabic word order is fairly different from English

- lack of short vowels means that written Arabic is HUGELY ambiguous
• writing system and morphology mean that it can be very difficult to see that two forms correspond to the same lexeme: ['bAlktAb', 'bktAb', 'bktAbkm', 'bktAby', 'ktAbA', 'ktAbh', 'ktAbhA', 'ktAbhm', 'ktAbk', 'ktAbnA', 'ktAbyh', 'ktb', 'AlktAbA', 'AlktAbh', 'ALktAbhA', 'AlktAbhm', 'AlktAbk', 'ALktAbnA', 'ALktAbyh', 'Alktb', 'lktAb', 'lktb', 'wAlktAb', 'wbAlktAb', 'wktAb', 'wktbh'] are all forms of ‘book’.
Just do it:

Figure 22: Base proposer, accuracy against number of words matched
What did it actually do?

'the'->'Allh' ('lh') (score=5491.00): True
'and'->'Allh' ('lh') (score=5426.00): False
'allah'->'Allh' ('lh') (score=4016.00): True
'of'->'Allh' ('lh') (score=2888.00): False
'in'->'Allh' ('lh') (score=2182.00): False
'to'->'Allh' ('lh') (score=2180.00): False
'they'->'Allh' ('lh') (score=1473.00): False

Everything gets translated as Allah!

(scoring algorithm is wrong, because it's accepted Allh as a variant of Al)
A translation ought to occur roughly the same number of times as the word it is translating. So multiply count by ratio of $W^S$ and $W^T$. A good translation will have a ratio near to 1. Helps a bit: at least we get a stronger link between allah and Allh.

'allah'->'Allh' ('lh') (score=2636.76): True
'the'->'Allh' ('lh') (score=2541.82): True
'and'->'Allh' ('lh') (score=2484.70): False
'of'->'Allh' ('lh') (score=1525.24): False
'to'->'Allh' ('lh') (score=1112.81): False
'in'->'Allh' ('lh') (score=1092.99): False
'they'->'Allh' ('lh') (score=718.27): False
'that'->'Allh' ('lh') (score=682.20): False

But if Allh is the translation of allah, surely it isn't the translation of the as well.
Block \(W^T\) as a translation of \(W^S\) if you’ve already used it as the translation of \(W^{S'}\):

Figure 23: Basic algorithm, don’t use the same word more than once: accuracy against frequency
Better for longer: but why the dip?
I’m still being fooled by very common words that have no explicit counterpart in the Arabic.
Don’t accept a suggestion if the ratio between the frequency of the two words is greater than $t$ for some $t$: try $t = 2$.

Figure 24: As above, but with filter (not sure what’s happened to the X-axis)
So: we can make sensible guesses about translation equivalents with very little information.

What other improvements can we make?

- Only match things if they have the same part-of-speech tags
  - The fact that my taggers won’t be 100% accurate doesn’t matter very much: I’m unlikely to accidentally assign two potential equivalents the same tags.
  - But you can’t rely on translations to preserve POS tags: translation into Spanish of ‘I ran to the station’
• Transfer rules: format, levels of detail, algorithm for application, limitations, MT pyramid

• Translation equivalence $\approx$ logical equivalence: transfer rules $\approx$ inference rules in textual entailment

• Ambiguity: kinds of ambiguity, significance for MT

• Bilingual dictionaries: $1 \rightarrow N$, $N \rightarrow 1$ mappings, multiword expressions, extraction from corpora (alignment, dynamic time warping)
THE END!
Index

back off, 228
bigram, 252
BNC (British National Corpus), 236
categorial descriptions, 168
centroid, 378
closed, 227
consequence, 457
contexts, 382
continuation programming, 532
cosine, 369
damping, 436
deltas, 56
density, 391

Dependency grammar, 310
Derivational morphology:, 160
diphones, 134
document frequency, 374
dynamic time warping, 465, 593
eigenvector, 450
emission probabilities, 71, 253
entailment, 457
Euclidean distance, 369
F-measure, 246
fast Fourier transform, 47
FFT, 47
flat start, 107
Formant synthesisers, 129
formants, 52
Gaussian mixture models, 102
gloss, 447
gross score, 270
head percolation table, 337
Hidden Markov Models, 64
hyponym, 455
hyponym, 455
Inflectional morphology:, 159
interlingua, 576
islands, 300
K-means clustering, 378
leftArc:, 317
lexical ambiguity, 582
Lexical gaps, 571
liquids, 41
machine translation pyramid, 577
morphemes, 145
movement, 299
N-fold cross validation, 345
Nasals, 38
net score, 271
non-deterministic, 308
normal distributions, 100
obstruents, 40
Open, 227
oracle, 318
out-of-place items, 288
Page rank, 421
parsing accuracy, 344
part of speech tag, 233
Penn treebank, 236  
Personalised page rank, 442  
phonology, 152  
Porter stemmer, 232  
Precision, 246  
probability mass, 433  
random surfer model, 421  
realisation rules, 152  
Recall, 246  
reestimate, 119  
rightArc:, 317  
saturated, 168  
scope ambiguities, 581  
seeds, 378  
shift:, 317  
sibilants, 41  
spelling rules, 151  
stem, 232  
Structural ambiguity, 580  
support verbs, 571  
synsets, 408  
term frequency, 374  
TF-IDF, 374  
transfers, 564  
transition probabilities, 71  
transitions, 250  
treebank, 236  
trills, 41  
truth preserving, 578  
two-level morphology, 157  
Universal dependencies, 236  
unsaturated, 168
Unsaturated lexical items, 225
vector space model, 364
Vowels, 34
word sense disambiguation, 419
WSD, 419
References


URL http://dx.doi.org/10.1007/978-3-642-33409-2_12


URL http://dl.acm.org/citation.cfm?id=973306.973310


Evans, R., Gazdar, G., 1989. Inference in DATR. In: Proceedings, Fourth Meeting of the European Chapter of the Association for Computational
URL http://dx.doi.org/10.1162/neco.2006.18.7.1527


