COSC343: Artificial Intelligence

Lecture 23: Statistical Language Modelling

Alistair Knott

Dept. of Computer Science, University of Otago

In today's lecture

- Ambiguity in natural language
- Simple probability models for natural language

Techniques for resolving ambiguity

1. Use semantics.
   - Work out the meaning of each interpretation.
   - Which interpretation makes most sense?

2. Use statistics.
   - Which word sense is most likely, in the current context?
   - Which syntactic structure is most likely, in the current context?

Probabilistic models of natural language

To build a probabilistic model of a natural language L:

- We start by obtaining a corpus of text in language L.
- We may choose to enrich the corpus by annotating it, to identify important features explicitly.
  E.g. word senses, syntactic structures.
- We estimate probabilities of events in the corpus by counting how often they occur in the corpus.
- We use these probabilties to estimate probabilities of events in new (unseen) texts.

Example: Bayesian text classification

You saw this example in one of your tutes.

Training corpus:

- A set of emails, annotated (by a human) as either 'spam' or 'ham'.

Probabilities estimated from relative frequencies: for each word w_i in the corpus:

- $P(w_i|spam)$; $P(w_i|\neg spam)$

From these probabilities, using a naive Bayes model, we can take a document containing words $w_1 \ldots w_n$ and calculate

- $P(Spam|w_1 \wedge w_2 \wedge \ldots w_n)$

Dependencies between words

A naive Bayes model makes the assumption that the words in a document are (conditionally) independent of one another (given the document category).

- It's called a 'bag-of-words' model.

But if we want to build a model of sentence structure, rather than just classify documents, the whole point is that the words in a sentence are *not* fully independent of one another.

Recap: the two types of structure in human language

Recall from Lecture 20: knowing a natural language involves two things:

1. *General principles*, which are neatly encoded in a grammar:
   - Principles which group words into *general categories*.
   - Principles which define *hierarchical structure* in sentences.
2. A lot of *specific facts*:
   - Knowledge of individual word meanings
   - Knowledge of *idioms*: word combinations that occur particularly frequently.

Probabilistic models are good at capturing knowledge of idioms.

Idiomatic structure in language

Idioms are *commonly occurring* word sequences or patterns.

Collocations are combinations of words that are more likely than you would predict (e.g. from a grammar).

- To pull over, to tuck in, to pick up...
- How are you doing?
- Never mind...

Fixed expressions are groups of words that contribute a meaning *collectively*, rather than individually.

- To kick the bucket
- To take *X* to task...

Lots of expressions are in between:

- Don't worry about it

Probabilistic language models

A common way of using probability theory to model language is to adopt a Markovian model:

- The probability of a word W is dependent on the previous n words. (Also called an n-gram model.)

Note this is very different from a grammatical model! It ignores hierarchical structure in sentences altogether.

Building an n-gram language model

To build an n-gram model, we count frequencies of word sequences in a training corpus.

- If $n = 0$, we count the frequency of each individual word, and estimate the *prior* probability distribution for words.
- If $n = 1$, we count the frequency of each possible sequence of *two words*. Then if we're given a word, we can estimate the *conditional* probability distribution for the next word.

Some n-gram models

Here's a sequence generated by a model trained on unigrams in AIMA:

> *logical are as are confusion a may right tries agent goal the was diesel more object then information-gathering search is*

Here's one generated by a model trained on bigrams:

> *planning purely diagnostic expert systems are very similar computational approach would be represented compactly using tic-tac-toe a predicate*

Here's one generated by a model trained on trigrams:

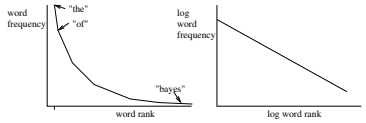
> *planning and scheduling are integrated the success of naive bayes model is just a possible prior source by that time*

Problems building a linear language model

Clearly, a higher n gives a better model.

But there's a sparse data problem: it's hard to get good estimates for sequences of length n when n gets large.

The problem is particularly severe because words are distributed according to Zipf's law:



Basically, there are lots of very rare words.
So to get good estimates of all word *sequences*, you need a huge corpus.

Uses of a linear language model

We can use a linear model for word sense disambiguation.

Many words are semantically ambiguous.

- E.g. *bank* can mean 'river edge' or 'financial institution'.)

To disambiguate:

- Identify the sense of each ambiguous word by hand in the training corpus.
- Then build an $n-$ary probability model.
- Now the *context* of an ambiguous word carries information about which sense was intended.
  - $P(\text{RIVERBANK}|swam, to, the)$ =0.000001
  - $P[\text{FINANCIALBANK}|swam, to, the)$ =0.000000000001

Uses of a linear language model

We can also use a linear language model in speech interpretation:

- Say the speaker's audio signal is consistent with several alternative word sequences:
  (e.g. *recognise speech*/*wreck a nice beach*)
- The probability model can be used to choose between these alternatives. (Which sequence is most probable?)

  $P(recognise, speech) = 0.0001$
  $P(wreck, a, nice, beach) = 0.00000001$

Uses of a linear language model

We can also use n-grams to improve a bag-of-words model of text classification.

- Instead of using single words in our Bayesian model of spam/ham, we could use *bigrams* in the training corpus. (Or n-grams.)

Uses of a linear language model

Linear language models can also be built with single letters.

We compute the probability of the next *letter*, given the previous n *letters*.

- E.g. $P(p|h, e, l) = ?$
- $P(l|h, e, l) = ?$

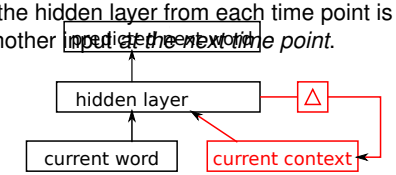
This is how 'predictive text' algorithms are learned.

Learning linear language models with neural networks

There's a very neat variation on a backprop neural network that can learn a kind of probabilistic language model.

In a regular feedforward network, there's an input layer, a hidden layer and an output layer. In a simple recurrent network (invented by Jeff Elman, 1990):

- The network takes each word in the training corpus one by one, and is trained to predict the *next word*. (Using regular backprop.)
- To help it, the hidden layer from each time point is copied, and used as another input at the next time point.



A simple training grammar

Elman trained his network on sentences produced by a simple grammar: e.g. *woman break glass*, *monster eat man*.

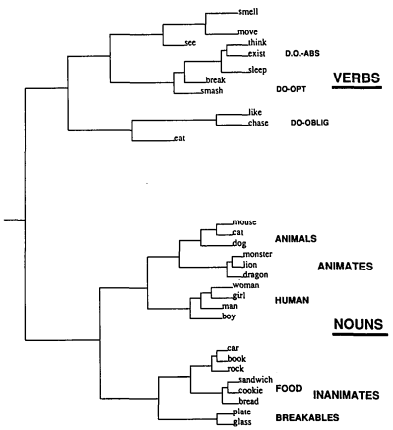
WORD 1	WORD 2	WORD 3
NOUN-HUM	VERB-EAT	NOUN-FOOD
NOUN-HUM	VERB-PERCEPT	NOUN-INANIM
NOUN-HUM	VERB-DESTROY	NOUN-FRAG
NOUN-HUM	VERB-INTRAN	
NOUN-HUM	VERB-TRAN	NOUN-HUM
NOUN-HUM	VERB-AGPAT	NOUN-INANIM
NOUN-HUM	VERB-AGPAT	
NOUN-ANIM	VERB-EAT	NOUN-FOOD
NOUN-ANIM	VERB-TRAN	NOUN-ANIM
NOUN-ANIM	VERB-AGPAT	NOUN-INANIM
NOUN-ANIM	VERB-AGPAT	
NOUN-INANIM	VERB-AGPAT	
NOUN-AGRESS	VERB-DESTROY	NOUN-FRAG
NOUN-AGRESS	VERB-EAT	NOUN-HUM
NOUN-AGRESS	VERB-EAT	NOUN-ANIM
NOUN-AGRESS	VERB-EAT	NOUN-FOOD

NOUN-HUM	man, woman
NOUN-ANIM	cat, mouse
NOUN-INANIM	book, rock
NOUN-AGRESS	dragon, monster
NOUN-FRAG	glass, plate
NOUN-FOOD	cookie, break
VERB-INTRAN	think, sleep
VERB-TRAN	see, chase
VERB-AGPAT	move, break
VERB-PERCEPT	smell, see
VERB-DESTROY	break, smash
VERB-EAT	eat

Analysis of the trained network

After training, if we build a dendrogram of the patterns of activity produced in the hidden layer by each word in the training lexicon, we see that words cluster into syntactic categories!

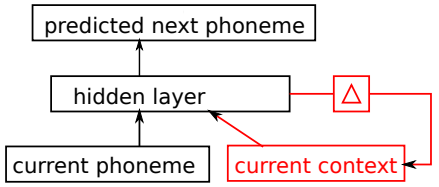
The network doesn't have to be 'taught' what syntactic categories are. They emerge very clearly from the data.



An Elman network for next letter/phoneme prediction

An Elman network can also be trained using letters (or phonemes) as input.

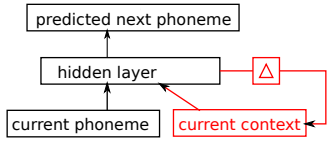
- In this setup, it learns to predict the next phoneme based on the previous *n* phonemes.



- E.g. *M,a,n,y,y,e,a,r,s,a,g,o,a,b,o,y,a,n,d,g,i,r,l,l,i,v,e,d,b,y,t,h,e,s,e,a*

An Elman network for next letter/phoneme prediction

An interesting thing to do is look at the trained network's *confidence* in its predictions about the next letter.



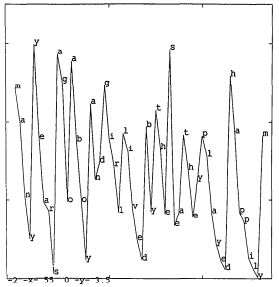
- We can do this by treating the predicted next phoneme layer as a probability distribution (by scaling activations to sum to 1).
- We can then compute the information gained by learning the actual next letter.

Recall Lecture 6: $I(\langle P_1, \ldots, P_n \rangle) = \sum_{i=1}^{n} -P_i \log_2 P_i$

High confidence = low information.

Learning what words are

If you plot the network's confidence for each prediction, you see that the places where it's *least confident* about the next letter correspond to *word boundaries*.



An Elman network can learn what *words* are! No-one has to teach it!

The limits of a linear language model

We've seen that in order to describe the range of sentences in a human language, we need to build a grammar, which assigns each sentence a hierarchical structure.

- On this model, sentences are not simple sequences—they're *trees*.
- Our model of semantic interpretation uses these trees to build sentence meanings.

What we need to do is to build a probability model which works with a grammar.

- I'll talk about that in the next lecture.

Summary and reading

Reading for this lecture: AIMA Section 22.1
Reading for next lecture: AIMA Section 23.2.1.