The aim of this tutorial is to provide hands-on experience in open-domain text processing, covering the following topics: tokenisation, part-of-speech tagging, named entity recognition, parsing, semantic processing, and logical inference. The tutorial will comprise an overview of statistical modelling for natural language processing tasks and brief introductions to the above topics, followed by practical exercises. It will be centered around the state-of-the-art C&C tools and Boxer. No prerequisite knowledge is required; nevertheless basic knowledge of shell commands in linux/unix environments is a plus for getting the most out of this course.

Contents

1 Tokenisation 3

2 Part-of-speech tagging 5

3 Named entity recognition 8

4 Syntax 11

5 Semantics 13

6 Inference 15
Getting Started

The exercises of the tutorial require a running version of the C&C tools and Boxer. From the command line, run the following:

```
/home/yzhang/setup.sh
```

This will install everything you need in your home directory.

Note that all shell commands in this tutorial assume that your directory is the `candc` directory. Within this directory there should be the directories `candc/bin` containing all binaries, and a directory `candc/working/tutorial`, for storing temporary results.

The binaries have already been pre-built, so you’re ready to go.

Abbreviations

CCG  Combinatory Categorial Grammar
DRS  Discourse Representation Structure
DRT  Discourse Representation Theory
FOL  First-Order Logic
NER  Named entity recognition
POS  Part of speech
1 Tokenisation

Armchair material

Tokenisation is typically the first task in a pipeline of natural language processing tools. It usually involves two sub-tasks, which are often performed at the same time:

1. separating punctuation symbols from words;
2. detecting sentence boundaries.

In the output of this process white space is typically used as separation marks between tokens, and sentences are usually separated by new lines. (But there are other ways of doing this, e.g. using an XML markup language.)

At first it might sound like a tedious and rather trivial thing to do — how difficult can it be to identify punctuation symbols such as hyphens, commas, and full stops? The problem is that many punctuation symbols are ambiguous in their use. To give an example: a hyphen can be used in a football score, in a range of numbers, in a compound word, or to divide a word at the end of line.

For an accurate detection of sentence boundaries it is important to distinguish full stops in the use of abbreviations, and full stops when they are used to mark the end of a sentence (and an abbreviation could be the end of the sentence, in which case there is only one full stop!).

Hands-on stuff!

Exercise: 1.1 Tokenising

The following text is taken from Wikipedia (http://en.wikipedia.org/wiki/Kate_Bush).

Kate Bush (born 30 July 1958) is an English singer, songwriter, musician, and record producer. Her eclectic musical style and idiosyncratic lyrics have made her one of England’s most successful solo female performers of the past 30 years. Bush was signed by EMI at the age of 16 after being recommended by Pink Floyd’s David Gilmour. In 1978, aged 19, she topped the UK charts for four weeks with her debut song “Wuthering Heights”, becoming the first woman to have a UK number one with a self-written song.
Copy it in a file (called for instance katebush.txt) and store it in the working/tutorial directory. Make sure that all the text is on one line (e.g. paste the article into a file using emacs and manually remove the newlines).

Now tokenise it with the following command:

    bin/tokkie --input working/tutorial/katebush.txt

This will send the output of the tokeniser to the screen. Compare the tokenised text with the raw version.

**Exercise: 1.2 Quotes**
Try the following option of the tokeniser:

    bin/tokkie --input working/tutorial/katebush.txt --quotes delete

This removes all quotes from the original text (with an exception – what is it?).
Why would this be useful? Answer: quotes are hard to deal with (in parsing). Sometimes it is easier to ignore them!

**Exercise: 1.3 Output**
Use the --output option to redirect the output of the tokeniser to a file:

    bin/tokkie --input working/tutorial/katebush.txt --output working/tutorial/katebush.tok
2 Part-of-speech tagging

Armchair material

Words can be used in a variety of grammatical roles, for example nouns, adjectives, prepositions, verbs, and so on. These categories are the basic grammatical units of language and are called parts of speech. Part-of-speech tagging, or POS tagging, is the task of automatically labelling each token in the sentence with its part of speech. This is a crucial early step in understanding a sentence.

Now consider the following sentence:

The function of sleep, according to one school of thought, is to consolidate memory.

Here are the outputs of the same sentence labelled by two hypothetical taggers (the first a little better than the second):

The|DET function|NOUN of|PREP sleep|NOUN ,|PUN according|VERB to|PREP one|DET school|NOUN of|IN thought|NOUN ,|PUN is|VERB to|PREP consolidate|VERB memory|NOUN ,|PUN

The|DET function|VERB of|PREP sleep|VERB ,|PUN according|VERB to|PREP one|DET school|NOUN of|IN thought|VERB ,|PUN is|VERB to|PREP consolidate|VERB memory|NOUN ,|PUN

As you can see, there are some differences between the output of these taggers. Can you spot the errors in the second output? Differences in tagger output can be due to the following:

- ambiguities in natural language;
- choice of tagset.

Ambiguity Ambiguity is what makes POS tagging, and many NLP tasks, difficult. Many words have more than one possible syntactic category. Can you think of some?

The back door = adjective
On my back = noun
Win the voters back = adverb
Promised to back the bill = verb
The POS tagging problem is to determine the POS tag for a particular instance of a token. When we perform the task of POS tagging, we try to determine which of these syntactic categories (i.e., POS tags) is the most likely for a particular use of a token in a sentence.

**Tagset** Which and how many tags should we use? This is an issue researchers have been considering for a long time. Already in ancient Greek times, Aristotle distinguished between three categories: nouns, predicates, and conjunctions. Slightly later, a set of eight categories proposed by Dionysius Thrax, in the 2nd century BC, was one that was maintained (more or less unchanged) for a period of about two thousand years! And here is what Mark Twain says about the matter in his book “The Awful German Language”:

*There are ten parts of speech, and they are all troublesome.*

In the past decades, also due to the development of NLP techniques for more sophisticated processing, the tagsets used have been extended to comprise even up to 100 tags. Of course, tagsets are not always independent of the language, and often are also application dependent.

In this tutorial we use the Penn Treebank tagset (see Appendix), which comprises 36 different parts-of-speech plus punctuation. So, the sentence we have seen above would be assigned the following more specific tags.

```
The|DT function|NN of|IN sleep|NN ,| according|VBG to|TO one|CD school|NN of|IN thought|NN ,| is|VBZ to|TO consolidate|VB memory|NN .|
```

Well, how good are POS taggers? The performance of current state-of-the-art POS taggers shows an accuracy per token of around 97–98%. This is impressive! But what about accuracy per sentence? How many fully correctly tagged sentences do we get with a figure like this for token accuracy?

**Hands-on stuff!**

The POS-tagger is activated by the command `bin/pos`. Here is an example:

```
bin/pos --input working/tutorial/katebush.tok --model models/pos
```

As you can see from the result, the output consists of the tokens, a vertical bar, and the assigned part of speech. Use the –output option to redirect the pos-tagged sentences to a file:
Exercise: 2.1  Finding tagging mistakes
Try to find mistakes in the tagged text (there are perhaps two or three), and try to give an explanation of what caused the mistakes.

Exercise: 2.2  Part of speech tags
In the appendix of this document you will find a list of all tags used by the pos-tagger. Complete part of the table by filling in, say, 15 of the blanks by giving examples for each tag. (You might do this by taking some text, eg from the web, and finding examples of each tag being used.)

Exercise: 2.3  Tagging Homographs
Homographs are words that are spelled the same, but have different meanings (and usually different pronunciations). Let’s see what the POS-tagger makes of them! Try the following examples (thanks to A. Terry Bahill). Here you can go:

    bin/pos --model models/pos

and type the sentences on the command line, followed by enter; the pos tagger will tag the sentence and wait for the next input. Finish the input session by hitting Ctrl-c.

    A cat with nine lives lives around the corner.
    The dove dove into the deep grass.
    I did not object to that object.
    The landfill was so full it had to refuse refuse.
3 Named entity recognition

Armchair material

Named entities are phrases that contain names of any type which could be of interest for natural language processing tasks. A standard set of named entity types for NLP is: persons, organisations, locations, times and quantities. Other possibilities are genes, proteins, cell types, and any other sort of biological entity for those interested in processing biomedical research text. Or for someone building an application concerning published works, the named entities of interest might be films, books, paintings, and so on.

So, first one has to find out which sequences of words might denote entities of interest, and then assign a label to them. In other words, there are two steps to named entity recognition (NER):

1. detecting named entities;
2. classifying named entities.

However, these two steps are often performed in tandem and not separated as two distinct phases.

So, how are we going to find these entities? One simple approach is just to have very long precompiled lists of names (usually called gazetteers), and match against the names in the lists. For example, we could compile lists of countries, cities, rivers, and so on, and the list might be quite exhaustive. However, it would be difficult to create an exhaustive list of companies, for example, since new companies are being created all the time.

But there’s another problem: ambiguity (yet again). Is “Washington” a person or a place? Does “Ericson” refer to a company or a person? Is “1984” a time expression or a measure phrase? Well, it depends on the context, of course. The NER module attempts to deal with this problem by tagging entities in context.

The NE tagger in our tutorial uses seven different classes, as shown in the table below.
<table>
<thead>
<tr>
<th>NE</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PER</td>
<td>person</td>
<td>Kate Bush</td>
</tr>
<tr>
<td>DAT</td>
<td>date</td>
<td>January 16</td>
</tr>
<tr>
<td>TIM</td>
<td>time</td>
<td>5 p.m.</td>
</tr>
<tr>
<td>LOC</td>
<td>location</td>
<td>England</td>
</tr>
<tr>
<td>MON</td>
<td>monetary expression</td>
<td>50 million dollar</td>
</tr>
<tr>
<td>ORG</td>
<td>organisation</td>
<td>Pink Floyd</td>
</tr>
<tr>
<td>PCT</td>
<td>percentage</td>
<td>10%</td>
</tr>
</tbody>
</table>
Hands-on stuff!

**Exercise: 3.1** Using the NE tagger
Try this to start the tagger:

```
bin/ner --input working/tutorial/katebush.pos --model models/ner
```

**Exercise: 3.2** Finding tagging mistakes
Try to find mistakes in the tagged text, and try to give an explanation of what caused the mistake.
4 Syntax

Armchair material

A parser assigns syntactic structure to a string, based on a grammar and lexicon. We follow Combinatory Categorial Grammar in defining the lexicon and grammar for English. CCG is a lexicalised theory of grammar: it has many lexical categories, but few grammar rules (combinatory rules as they are called in CCG).

We’ve just had a whole morning of the theory of CCG parsing, so let’s get straight on with the practical component.

Hands-on stuff!

To produce syntactic structure (in the form of CCG derivations) we use the C&C parser, a statistical parser for CCG, trained on a large database of CCG derivations (CCGbank). For convenience we will use a combined program – bin/candc – that also performs part-of-speech tagging and named entity recognition.

The parser expects one sentence per line, and the sentences need to be tokenised. Here is how you run the parser:

```
bin/candc --input <FILE> --models models/boxer --candc-printer boxer
```

What do these options mean? The `--models models/boxer` option selects a directory containing the statistical models needed by the various tools. The models are quite large, and so it takes a little while to load them into memory. The `--candc-printer boxer` option ensures that the output is displayed in the form of CCG derivations, which is what Boxer needs as input. It is possible to redirect the output to a file using the `--output <FILE>` option.

Exercise: 4.1 Parsing
Try the parser using the command above on some (tokenised) text. You can also try the parser on a raw text, and find out why tokenisation is important!

Exercise: 4.2 Inspecting the output

Create a file `working/tutorial/syn1.txt` with the following sentence:

```
Bill Gates stepped down as chief executive officer of Microsoft on January 12, 2000.
```
Tokenise it, then parse it. You’ll see that the output is in the form of a CCG derivation, which is the input Boxer needs, but it isn’t very readable. There is an alternative output format which represents the grammatical relations (GRs) between words:

```
bin/candc --input <FILE> --models models/boxer --candc-printer grs
```

As well as the relations between words, you’ll also see the POS tags, NE labels, and CCG lexical categories assigned to the words, shown on the final line of the output.

There are 3 types of GR in this example: `ncmod` (non-clausal modifier), `dobj` (direct object), and `ncsubj` (non-clausal subject). For example, *Gates* is the subject of *stepped* (i.e. Gates is doing the “stepping” (down)). The parser does okay on this sentence, although doesn’t quite get the analysis correct for *chief executive officer*. Can you spot the error?

**Exercise: 4.3 Parsing Homographs**
Try the parser on the sentences with homographs given in Exercise 2.3 on Page 7.
5  Semantics

In this part of the tutorial we will see how we can construct semantic representations on the basis of the syntactic derivations produced by the parser. The tool we’re going to use for this purpose is Boxer.

Armchair material

The semantic representations that we will produce are known as Discourse Representation Structures (DRSs), which are proposed in Discourse Representation Theory, a formal theory of natural language meaning dealing with a wide variety of linguistic phenomena.

The theory of DRT isn’t important for us today, but, simply put, a DRS consists of a pair of discourse referents (also known as the domain of the DRS) and a set of conditions. A discourse referent denotes an entity; a condition constrains the interpretation of this entity. DRSs are recursive structures — a DRS might contain other DRSs constructed with the logical operators negation, disjunction, or implication.

DRSs can be converted to formulas of first-order logic. In fact, for the purposes of this tutorial we’re just using DRT to demonstrate how a sentence of natural language can be converted into a first-order logical representation using a formal grammar.

Hands-on stuff!

**Exercise: 5.1 Boxing**

Create a file `working/tutorial/sem1.txt` with the following three sentences:

```
Bill saw a busy manager.
Bill saw every busy manager.
Bill saw no busy manager.
```

Use the tokeniser (Section 1) to tokenise these sentence in a file `working/tutorial/sem1.tok`. Then produce a CCG derivation with the parser (Section 4), using the boxer printer, and redirect the output in the file `working/tutorial/sem1.ccg`. Then run Boxer with the following command:

```
bin/boxer --input working/tutorial/sem1.ccg --box --resolve
```

This should produce DRSs both in Prolog format and pretty-printed boxes.
**Exercise: 5.2** DRS structure

Compare the DRS structures of the three DRSs. The + indicates a conjunction of DRSs, the \( \Rightarrow \) an implication, and \( \neg \) a negation. Try to paraphrase the content of the DRSs in English.

**Exercise: 5.3** Output formats

Run the same command as before but now add the option `--format no` and then again with `--format xml`.

**Exercise: 5.4** Boxing

Create a file `working/tutorial/sem2.txt` with the following three lines:

```
<META>text1
Mr. Jones bought an expensive car.
He is a busy manager.
```

Tokenise this file, then parse it and store the output in a file `working/tutorial/sem2.ccg`. Now run Boxer on this file, once without the `--resolve` option, and once with, then compare the results:

```
bin/boxer --input working/tutorial/sem2.ccg --format no --box
bin/boxer --input working/tutorial/sem2.ccg --format no --box --resolve
```

The `<META>` tag causes the sentences following it to be analysed in one DRS, rather than in a separate DRS for each sentence. You can have more than one `<META>` tag in an input file.
6 Inference

Here we will see how we can use techniques from automated deduction to draw logical inferences from texts. We will do this with the help of Nutcracker, a system for recognising textual entailment.

Armchair material

The two tools we are going to use are a theorem prover and a model builder for first-order logic (FOL). We will translate the DRSs generated by Boxer to FOL and then give it to the theorem prover and model builder. As the name suggests, a theorem prover attempts to find out whether a given input formula is a theorem. In other words, it checks whether the input is a validity, that is, true in all possible models. On the other hand, a model builder attempts to construct a model for a given input.

How can we make use of these tools? Imagine a text is inconsistent, such as the following sentence:

Mr Jones is a woman.

Say we produced a semantic representation for this sentence, and generated background knowledge that Mr entails a male human entity, and woman a female human entity, and that male and female are disjoint properties. We all put this in one big formula – let’s call it $\phi$. Now when given $\phi$ to a model builder, the model builder will fail to find a model, because there isn’t one satisfying the input ($\phi$ doesn’t have a model, or $\phi$ is not satisfiable). It might be that in such cases the model builder is going on forever (or for a very long time) trying to find a model.

Therefore we also use a theorem prover. But we don’t give it $\phi$. We give it the negation, $\neg \phi$. Why? Well think about it: if $\phi$ isn’t true in any model, then the negation of $\phi$ must be true in all models. And that’s precisely what theorem provers are good at finding out.

In sum: the theorem prover and model builder are complementary tools. We always use them together. When we give $\phi$ to the model builder, we give $\neg \phi$ to the theorem prover. If the theorem prover finds a result, we know that the input text is inconsistent, and if the model builder finds a result, we know that the text is consistent. So basically, we’re now able to perform consistency checking for texts.
Hands-on stuff!

**Exercise: 6.1** Consistency checking
Make a directory `working/tutorial/rte`. Create two text files in this directory: a file named `t`, and a file named `h` (t for “text” and h for “hypothesis”). Make the contents of the first file the sentence “Bill Gates bought a car.”, and the contents of the second “Bill Gates bought no car.”. Then run Nutcracker:

```
bin/nc --dir working/tutorial/rte
```

Does the text entail the hypothesis? What does Nutcracker conclude?

**Exercise: 6.2** Background knowledge
Nutcracker calculates background knowledge using the WordNet lexical database. It uses this background knowledge while it draws inferences. Let’s look at some examples:

```
t:John has a dog.
h:John has an animal.

t:John has an animal.
h:John has a dog.

t:John likes no animal.
h:John likes a dog.
```

What does Nutcracker predict for these three textual entailment pairs? Does t entail h in each case? Find out what the background knowledge ontology looks like by viewing the contents of the file `mwn.pl` in the example directory.

**Exercise: 6.3** Looking under the hood
After using Nutcracker, have a look at the temporary files generated for drawing inferences. They are stored in the directory `working/tmp`.
## Appendix: Part of Speech (POS) tagset

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description (Penn Treebank tagset)</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Coordinating conjunction</td>
<td></td>
</tr>
<tr>
<td>CD</td>
<td>Cardinal number</td>
<td></td>
</tr>
<tr>
<td>DT</td>
<td>Determiner</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>Existential “there”</td>
<td></td>
</tr>
<tr>
<td>FW</td>
<td>Foreign word</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>Preposition or subordinating conjunction</td>
<td></td>
</tr>
<tr>
<td>JJ</td>
<td>Adjective</td>
<td></td>
</tr>
<tr>
<td>JJR</td>
<td>Adjective, comparative</td>
<td></td>
</tr>
<tr>
<td>JJS</td>
<td>Adjective, superlative</td>
<td></td>
</tr>
<tr>
<td>LS</td>
<td>List item marker</td>
<td></td>
</tr>
<tr>
<td>MD</td>
<td>Modal</td>
<td></td>
</tr>
<tr>
<td>NN</td>
<td>Noun, singular or mass</td>
<td></td>
</tr>
<tr>
<td>NNS</td>
<td>Noun, plural</td>
<td></td>
</tr>
<tr>
<td>NNP</td>
<td>Proper noun, singular</td>
<td></td>
</tr>
<tr>
<td>NNPS</td>
<td>Proper noun, plural</td>
<td></td>
</tr>
<tr>
<td>PDT</td>
<td>Predeterminer</td>
<td></td>
</tr>
<tr>
<td>POS</td>
<td>Possessive ending</td>
<td></td>
</tr>
<tr>
<td>PRP</td>
<td>Personal pronoun</td>
<td></td>
</tr>
<tr>
<td>PRP$</td>
<td>Possessive pronoun</td>
<td></td>
</tr>
<tr>
<td>RB</td>
<td>Adverb</td>
<td></td>
</tr>
<tr>
<td>RBR</td>
<td>Adverb, comparative</td>
<td></td>
</tr>
<tr>
<td>RBS</td>
<td>Adverb, superlative</td>
<td></td>
</tr>
<tr>
<td>RP</td>
<td>Particle</td>
<td></td>
</tr>
<tr>
<td>SYM</td>
<td>Symbol</td>
<td></td>
</tr>
<tr>
<td>TO</td>
<td>“to”</td>
<td></td>
</tr>
<tr>
<td>UH</td>
<td>Interjection</td>
<td></td>
</tr>
<tr>
<td>VB</td>
<td>Verb, base form</td>
<td></td>
</tr>
<tr>
<td>VBD</td>
<td>Verb, past tense</td>
<td></td>
</tr>
<tr>
<td>VBG</td>
<td>Verb, gerund or present participle</td>
<td></td>
</tr>
<tr>
<td>VBN</td>
<td>Verb, past participle</td>
<td></td>
</tr>
<tr>
<td>VBP</td>
<td>Verb, non-3rd person singular present</td>
<td></td>
</tr>
<tr>
<td>VBZ</td>
<td>Verb, 3rd person singular present</td>
<td></td>
</tr>
<tr>
<td>WDT</td>
<td>Wh-determiner</td>
<td></td>
</tr>
<tr>
<td>WP</td>
<td>Wh-pronoun</td>
<td></td>
</tr>
<tr>
<td>WP$</td>
<td>Possessive wh-pronoun</td>
<td></td>
</tr>
<tr>
<td>WRB</td>
<td>Wh-adverb</td>
<td></td>
</tr>
</tbody>
</table>