Introduction to Natural Language Syntax and Parsing
Lecture 2: Introduction to Statistical Parsing

Stephen Clark

September 30, 2015

Automatic Parsing One way to characterise the natural language parsing problem is in terms of these three questions:

- Where does the grammar come from?
- What's the algorithm for generating possible parses?
- How do we decide between all the parses?

For the first question, there are three possibilities (and various combinations in between). One option is to have a linguist hand code the grammar, typically using a particular linguistic formalism such as HPSG. The resulting grammars are often referred to as *precision* grammars, since the information contained in them is detailed and precise, resulting in rich parse representations. The downside with this approach, common to all rule-based approaches to NLP, is that it is difficult to manually create grammars in this way which are robust and can apply to a wide variety of textual input.

The second approach, which is currently dominant in the literature, is to have a linguist annotate sentences with the desired parse output, and then learn the grammar automatically from this annotation. The set of annotated parses is often referred to as a *treebank*. This is the approach we'll be following in this course.

The third approach is to try and learn a grammar entirely from raw, or POS tagged, text, with some biases encoded in the model (either implicitly or explicitly) to help with the learning process. In some respects the third option is the most desirable, because it does not require the costly human annotation associated with treebank creation and it is (perhaps) the closest to how humans learn language. However, learning grammars from raw or POS tagged text has proven to be extremely difficult, and the performance of the resulting *unsupervised* parsers is way below that of their supervised counterparts which learn from manual annotation.

In terms of the parsing algorithm, there are a number of possibilities. For dependency parsing, which is the focus of these lectures, there are two dominant

approaches: graph based and transition based. The graph-based method uses a data structure called a chart to record all ways in which the words in the input sentence can be linked together. The transition-based method uses a queue and a stack to combine words, processing them from left to right by shifting words off the queue, and combining words on the stack.

Because of the massive ambiguity problem that was described in Lecture 1, we need a way to select one of the parses (or perhaps rank them and output a scored subset). For scoring we use a parsing *model*, trained on the available annotated data. Again there are many possibilities here — both probabilistic and non-probabilistic — but in these lectures we will focus on a simple (non-probabilistic) linear model, which is easy to train and yet surprisingly effective.

The Penn Treebank The Penn Treebank was created in the early 90s [2] and immediately sparked a parsing "competition" that is still continuing today. The part that has been used in this competition contains around 1M words of newswire text, manually annotated with phrase-structure trees. The annotation also contains *traces* and empty elements, marking aspects of predicate-argument structure which are not overtly realised in the surface sentence; however, the majority of work using the treebank has ignored this extra annotation. The treebank took around 3 years to create, using a handful of annotators, based on a detailed set of annotation guidelines. See my book chapter [1] for more commentary on the history of statistical parsing and the effect the treebank has had on NLP research (demonstrating the importance of resources for the advancement of the field more generally).

The treebank can also be used to generate a "dependency bank", consisting of pairs of sentences with dependency, rather than phrase-structure, trees, which can then be used to train a dependency parser. The dependencies are created using the notion of linguistic heads, which can be heuristically recovered from the phrase-structure annotation, and then used to generate head dependencies between words.

Problems with the PTB Parsing Task There is no doubting the influence that the PTB parsing task has had on NLP research, not just for parsing, but also for other tasks such as statistical machine translation where parsing models have been applied. However, some researchers and commentators became disillusioned with the central position that the PTB parsing task acquired in NLP research, arguing that the focus on English newswire text was detrimental to the field. There is also the problem that the same test set — roughly 2,400 sentences from Section 23 — has been used continually by, not only the same researchers, but the field as a whole. Hence there is the possibility that the field has been implicitly fitting models to the test set, even if not explicitly "cheating" by directly observing it during training and development.

In contrast, the dependency parsing community has developed a number of dependency banks, for many different languages, and also different domains within the same language. It is often suggested that manually annotating de-

pendency structure is easier than annotating phrase structure, and therefore dependency banks are easier to create (although I don't know of any scientific studies demonstrating this empirically). One of the reasons that Google has adopted dependency parsing as its main parsing paradigm is undoubtedly the availability of training data in many languages.

Dependency Parsing Head dependencies of the sort shown in the examples have proven useful for a variety of NLP tasks, such as Information Extraction, Question Answering and Machine Translation. The reason is that they provide an approximation to the underlying predicate-argument structure, expressing roughly who did what to whom. Recovery of the dependencies can also be performed accurately and efficiently (although the goal of 100% accuracy is still a long way off, especially with the more difficult dependencies arising from e.g. PP attachment and coordination).

Another possible reason for the success of dependency parsing is that the formalism is easy to understand, unlike, say, CCG, which we'll study later in the course. That's not to say that dependency grammar isn't a serious syntactic theory in linguistics — since it is — only that computer scientists with little linguistic training can easily understand it. Another possible advantage is that dependency parsers are almost entirely data driven, in the sense that the knowledge required to parse unseen sentences is acquired entirely from a treebank, with little or no manual intervention.

Dependency Trees Dependency graphs are graphs in a mathematical sense: sets of edges and nodes, where the nodes are the words in the sentence. A dummy word — $ in the example — is often placed at the beginning of the sentence to act as a dummy root of the graph. The graph is directed, since the notion of dependency incorporates the notion of head, with each edge pointing from head to dependent. A lot of the evaluations in dependency parsing use unlabelled graphs, but in practice it is likely to be useful to have grammatical labels on the edges, such as `subj` (subject), `mod` (modifier), and so on. Dependency graphs are typically restricted to be trees, so that each node has only one parent. Whether this is desirable from a linguistic viewpoint is debatable, since some constructions, such as control, result in some arguments having more than one parent. However, performing computations with trees is generally easier than with graphs.

Dependency Trees more Formally As well as the restriction to trees, dependency graphs are typically restricted in other ways, again to ease any computation (such as finding the highest scoring tree). Dependency trees are often defined to be connected (so it's possible to reach any node from any other node by moving along the edges, ignoring the direction of the edges); acyclic, so that it's not possible to return to any node by moving along the edges; single-head, so that each node only has one parent; and projective, so that none of the edges, when the graph is written in two dimensions, cross.

Crossing Dependencies The example shows a case in English with crossing dependencies, although cases such as these are relatively rare in English. In fact, the dependency banks for English typically consist entirely of projective dependencies. However, crossing dependencies are not rare in other languages, such as Czech, German and Dutch. The Wang and Zhang tutorial states that 23% of the sentences in the Prague Dependency Treebank of Czech contain at least one crossing dependency.

Graph-Based Models Graph-based models use scoring functions defined over the graphs (as opposed to shift-reduce models which score the parsing *actions*). Then a *decoding algorithm* is used to find the highest-scoring graph. Much of the research in dependency parsing is concerned with devising optimal, efficient decoding algorithms using dynamic programming over the graph (so that the decoder is guaranteed to return the highest-scoring graph). This is in contrast to shift-reduce approaches, which typically use heuristic search in combination with very rich feature sets.

Edge-Based Factorisation Model In order to define efficient dynamic programming decoders over the graph, it is crucial that the features used to define the model are kept sufficiently local to the edges. The extreme version of this idea, resulting in a *first order* model, is to define local scoring functions which only look at a single edge (but which are allowed to look at any part of the sentence). Then the score for the whole graph is the sum of the scores for each edge.

Edge-Based Linear Model There are various ways of defining the local scoring function, both probabilistic and non-probabilistic. Here we will use a simple, non-probabilistic form for the scoring function: a linear model. The features are typically indicator functions, taking the value zero or one, picking out particular aspects of the edge. Very rich models are required for good performance, resulting in millions of different features, each with a corresponding weight which needs to be estimated. However, recent work using neural networks shows how to obtain good performance without the need for so many features.

Example Features A later lecture will look at the features in more detail, but I'd like to provide some intuition now in order to describe the decoder in the next lecture (which uses the local scores to find the highest-scoring graph). The key idea is that the features in a first-order model cannot span more than one edge, but they are allowed to look anywhere in the sentence. Mathematically features are binary-valued functions, but a more intuitive way to think of a feature is that it captures a particular pattern in the graph. For example, saw_VBD_duck_NN captures the presence of a particular edge in the graph. VBD_PRP$_NN captures a particular sequence of POS tags between the words making up the edge. Note the extensive use of POS tags in the features, which is

designed to overcome the extreme sparsity that would result from only defining features in terms of words.

Readings for Today's Lecture There are various freely available tutorials on dependency parsing. The one that I have been stealing pictures from is the following:

- Recent Advances in Dependency Parsing, Qin Iris Wang and Yue Zhang. NAACL Tutorial, Los Angeles June 1, 2010.
  http://naaclhlt2010.isi.edu/tutorials/t7-slides.pdf

References

[1] Stephen Clark. Statistical parsing. In Clark, Fox, and Lappin, editors, *Handbook of Computational Linguistics and Natural Language Processing*, pages 333–363. Blackwell, 2010.

[2] Mitchell Marcus, Beatrice Santorini, and Mary Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.