NaturalLanguageProcessing-Lecture11

Instructor (Christopher Manning):Hi, guys, and welcome back to 224n. Today what I'm gonna do is talk about how to do polynomial parsing of probabilistic context-free grammar. And this is probably as good a moment as any to warn people about Assignment 3. There's this slight "gotcha" in the third programming assignment where really we expect people to do two things. They're kind of driving the hearts of two different algorithms. So the first one is the max saint classifier for doing the name dense recognition, but we also then expect people to write the core of a probabilistic parsing algorithm. So don't get too cocky and overconfident and think if you got a max saint classifier done, you're almost finished because there's still more work you have to do. And so it's in this class that I'm that I'm gonna talk about the details of how to do that. Okay. I just mentioned that the last time there was this idea of Chomsky normal form grammars, which are a restricted form grammar, which admits a very simple of probable – of dynamic program parsing. And so I'll just illustrate this form quickly first. So the idea of Chomsky normal form grammar is you restrict your context-free grammar so you can only have rules of the form.

Non-terminal rewrites as to other non-terminals, all rules of the form. Non-terminal writes as a single terminal. So you can do this transformation for an arbitrary CFG with a couple of tiny edge cases, so there's an edge case, but if the original grammar admitted the empty string. You know, Chomsky normal form transformed one would lose the empty string from its language. But apart from a couple of teeny edge cases, you can do this transformation, and the way you do is there's two essential parts to it. One part is you want to find rules like this that have more than three kids and get rid of them. That's called binarization. One way of doing binarization is just to uniformly binarize all rules, even if at the cost of introducing extra nodes. And so for the assignment, you want to do this process of binarization and [inaudible] will talk more about binarization in the section on Friday. And also, that section on Friday is then the last section there is for this class. Okay. Then the other part of it is dealing with empties and unarys. So remember the two phenomena we have is if we start off with – this is a real pin treebank tree. This is the headline of an article, and the headline is just a tone. And that was parsed out by the pin people by saying, "Okay, this is a headline sentence, which has an empty subject. It's an imperative. And here's the verb and the verb phrase of a tone." So that's got both empties, and it's got unarys. Okay, so what we do in the preprocessing of the assignment is we just kind of strip out of trees all the empty material. So all of this stuff just disappears. Okay, there are better ways to treat empties, but we don't do that. And then you're left with this sort of long unary chain. And, I mean the way you can get rid of the Chomsky – get to the Chomsky normal form property from a unary chain is you just kind of delete stuff out of it because then you've go a terminal rewriting as a word. Okay, so let's suppose we've done that. So here is a nice tree from the pin treebank, which actually just happens to be in Chomsky normal form. So it has this property of a non-terminal, is either rewriting this to other non-terminals, or else it's rewriting as an individual word. Okay, and so corresponding to each of those rewrite rules; we're then going to get probabilities, which I talked about last time throughout PCFG. So the way we then do CKY parsing, which is Cocke-Kasami-Younger parsing. So this was an algorithm that dates from the '60s. So John Cocke was a famous IBM research scientist. He actually – I mean he's really more famous for other things. He's – I think he's most famous as the father of risk computing, but somehow he was always interested in natural language stuff as well. And so when early in the course we talked about the IBM statistical machine translation work, it was really John Cocke who, in terms of organizing that effort of getting IBM to fund and sponsor all the research on statistical machine translation. That really comes back to John Cocke. And so he came up with this polynomial time parsing algorithm about 1960. Now, he didn't actually write it down anywhere, so there's no paper published with John Cocke's name.

But it turns out that in 1960, computer science was a small enough field that you could get your name on an algorithm without actually writing it down. That doesn't work very well these days. I don't recommend that as a career strategy. And then there were two other people, Kasami and Younger, who both published independently papers on it about 1965. So the idea of Cocke-Kisami-Younger parsing is we have this data structure, which is a parse triangle. So really, it's a regular ray, but it looks pretty, although it's a lot more work in PowerPoint to turn it on its side like this because the property we're gonna have is each of these cells in these regular rays is gonna pick out a subsequence of this sentence. So these cells along here are the one-word subsequences. This is factory. This is payrolls. This is fell. And the cells along here are two-word subsequence. So this cell is factory payrolls. That cell is payrolls fell. And then this third row is the three word subsequences, so this cell here is payrolls fell in. And the four word ones, and then the top cell is for the entire sentence. And the nature of the CKY algorithm, if you start with the one-word cells, and you fill them in, and then you do the two word cells, the three word cells, the four word cells, and the five world cells. And so you work up in order of increasing span size. And using increasing span size as a method of exploring gives us straightforward, dynamic programming structure that's exploited by the algorithm. So that's what that all shows. So what we do at the beginning is we fill in words. So this is where we take advantage of having our grammar in Chomsky normal form. So that we had rules where non-terminal rewrote as a word, and those had a probability. And so we can write the non-terminals we can create and their probabilities into the cell. Okay, so then we want to move up to this second layer of cells. And in the second layer of cells, we want to work out every way we can make a non-terminal using the stuff beneath us. And so, there'll be, in principle, multiple ways to make something in the higher cell, and that will be in part because there are different things that we can use in these cells, and also – not really illustrated in this example – once you get higher up into the triangle, there'll be different ways in which you can use a binary rule, which you refer to as the split point, so that you could split the subsequence in different places to use a binary rule. Okay, so here, for this example, there are two ways in this simple little grammar I showed in which you can build a noun phrase, essentially using what's here. And it fits so you can write down, "Okay, I can build a noun phrase." There are these two ways of making it, and we can work out the probability of each one by multiplying what's here, multiplying what's there, and then multiplying by the probability of the rule in the grammar. And so that gives us a probability of building that piece of substructure. Okay, and then if our goal is to find the highest probability parse for a sentence, we consider all of the ways that we can build a noun phrase. And we find the one with the highest probability, and then we stick that one into the chart. Crucial, crucial property to remember – I mean in this example, the only thing that we're making is a noun phrase, but it's not in general the case that we find the one thing that we can build with the highest probability and stick it into the chart here.

We put into the chart the highest probability way of making each category. So if there were other categories that we could make over this span, like we could make a sentence or we could make an adjective phrase. We'd also list here sentence and it's probability, adjective phrase and its probability because even though, looking at that cell, they might have lower probability than the probability of making a noun phrase, when we continue up the tree, depending on the grammar rules, it might turn out optimal to use those different categories. Okay. And so that's the way we made it. So I'm gonna go through that in a bit more detail. But rather than doing it for just the CKY over Chomsky normal form grammar, I want to introduce a version of extended CKY parsing, where you still have unarys in your grammar because it turn out that, although the simplest way to express the CKY algorithm is over this Chomsky normal form grammar, in practice, it's fairly easy to generalize it so it will work of grammars that have unarys in them. And you can even get it to work for grammars that have empties in them. But certainly for the assignment what you should be doing is writing a version of the CKY algorithm that will handle unarys. So the basic principle is you can just sort of add more stuff to the CKY algorithm, which makes it a bit uglier in a theoretical perspective, but is perfectly straightforward, to handle unarys and empties. But the thing that you really want to note is binarization or rules, that that's really the property that doing polynomial time parsability depends on. So every algorithm that does polynomial time context-free grammar parsing, it might be hidden in different places, but somewhere in the algorithm, it has to binarize rules because providing you binarize rules, you then have this property where you're combining two things together at a time. And you can do it in polynomial time, where if you have enery rules, you then have a problem where, "Okay, I can partition this thing into pieces and any place." And then you're still in the world where you have exponential algorithms. Okay. So here is the CKY algorithm in great detail. So the first bit of the algorithm initializes. So, you know, I empty. I have my arrays, and what I do is I have a sentence to parse, which is words. And I put into the diagonal cells at the bottom of my CKY chart all rules that are of the form A goes to a word. So I look up my lexicon and see what categories the words can have. And I stick that into the very bottom of the chart. Okay, so conceptually there are two phases. The first phase is doing the lexical look-up. And the second phase is filling in all the rest of the chart. And the second phase is on the second slide. If you had a Chomsky normal form grammar, you stop right there. But if you're going to allow unarys into your grammar, you have to add extra code in to handle unarys. And this is way having unarys is ugly because, as you'll see, on both this slide and the next slide that it turns out that actually more than half of the code is dealing with the generalization to handling unarys. So if we have unarys in the grammar, we would – might have a rule like – something like NP goes to N. And so we want to put those kind of rules into the same cell in the chart, because they still span the same number of words. And well, why it gets a bit more ugly on top of that, is that you then have a closure problem because well, we might also have another rule in the grammar that says S goes to NP as a unary rule. So if we then found that we want – that we could make an NP over some span. We then also have to find that we can make the S over the span. And in theory, at least, those unary chains can be quite long. And so the way this is handled here is by a simple loop where we keep on looping through a closure until nothing else is added to the chart. So I start off going through, and I say for every pair of non-terminals – if I've been able to find a way to make B, and as a rule, A goes to B in the grammar, and I work out the proper probability of that unary, and I stick it in the chart. And then I've added something new to the chart. And so long as one pass through, I've added something new to the chart, I'll keep on doing multiple passes through this until I've reached the – until something stops. Does this have to terminate or can it can go into a infinite loop? Yeah?

Student:[Inaudible]

Instructor (Christopher Manning):Right. Yeah. So exactly. This has to terminate and the reason it has to terminate is there's a finite number of non-terminals, and that if I'm going through another loop of the iteration, I have to have added on new non-terminal into that cell of the chart. And so that means I can only do this loop the number times that there are non-terminals, which is finite. Yeah, so – I mean doing it just like this is the way I recommend you doing it for the assignment. If you want to, you can do clever things where you precompute what kinds of unary chains are possible, and you can make things a little bit more efficient that way. But there's sort of no need to, unless you're really wanting maxed out efficiency, this is guaranteed to work. And in algorithmic terms, doing things this way doesn't change the complexity of the algorithm. It solves just fine. Okay, so then this is the second part of the algorithm, and again, everything done here is dealing with the unarys. And so it's sort of up here that is the main part.

Student: In terms of the nesting of the code, where does this – the first line reside [inaudible].

Instructor (Christopher Manning):I think it's right outside it. You know, it's inside the function, but otherwise it's not indented at all, right. Because all of the – the four loop, here, is doing the bottom row of the parse chart when I shot of diagonals. So you filled in the bottom row, and now, here, we're going to fill in all the higher rows. I guess that disadvantage of python-like thinking indentation works to show nesting. Okay, so what we do here is we have these four loops. So the outer one is what size subsequence we're working with. So we start off with two word subsequences, and we build up to the whole length of the string. Then we do for begin equals zero to number of words minus span. So now we're kind of gonna be walking along the cells that have the same span. And then I calculate the end point, which is just beginning plus span. And then what I'm gonna do is work out, "Okay, if I want to work out the stuff and say the word four to the word seven cell. Well, I'm assuming a binary rule. And so the ways I can make it is I can either make something with word four and words five to seven, words four to five, and words six to seven, or words four to six, and word seven. So that I have to consider the different ways in which I could make this subsequence out of two subsequences that together comprise that sub sequence. And so I iterate a third time over choosing a split point. And so that split point is corresponding to, if I wanted to do four versus five to seven, that five is then my split point in this algorithm. And then at this point, I then say, "Well, what rules do I have in my grammar? If I have a category B in the left, so underneath me, and the category. C in the right cell, underneath me. And there's a rule in the grammar, that's A goes to B, C. Well, then I'm gonna put that into this cell on the chart. Okay, and then I just keep on doing that. And then after that, for each cell, I again want to handle unarys. And the way I handle unarys is the is the same way that I did it before, that I'm doing this loop in each cell, where I'm considering if I can put in unarys and then go on once I can't put them in anymore. Okay, so here's a little picture, written very small, of this. So this is the same kind parse chart, apart from it's not turned on a diagonal this time. But so, here, this is the one-word cells. This is the two word cells; this is the three word cells heading up. Okay, so the first thing I do is I put in all the lexicon entries for the words and the sentence, being of different categories with some probability.

Then I run the unary closure path, and I say, "Well look, if I make noun cats, I can also make mp goes to noun so I put that into the grammar. So then after that, I go through and do the two word sequences. And so once I put in the ways of making two words by combining things beneath me, I again do a unary closure on the sells in that diagonal. And then I'll go to the third diagonal, and etcetera. And eventually I fill in the chart and work it all up. Okay. So if you look at this part of the algorithm here, it's really manifest that what you have is a parsing algorithm that's cubic in the length of the sentence because you have these four loops right up here. So here's a four loop that goes through the length of the sentence. Here's where you begin, which is going through the length of the sentence. And here is where you're choosing a split point, that's also bound by the length of a sentence. So you have these three loops, which are up and bounded by the length of a sentence. And so you get an algorithm that's cubic in the length of the sentence. And the conceptual way to think about that is when you have one of these parse charts, the number of squares in that parse chart is N squared – I mean it's half N squared, but it's order N squared. And then for filling in each of those cells, you have to consider ways of combining things beneath you, so you have the different splits of things beneath you, and they're also order N. And so you end up as order N cubed. If you look in here, where I can see that every possible rule of the grammar, the way I've written it here, this algorithm is also G cubed, where G is the number of non-terminals in the grammar. So I'm just doing a loop through for every non-terminal, and so there are triples of these, and saying does this rule exist in the grammar? And naively that's the answer to this. If you start being a bit cleverer, you can reformulate how the bound on the grammar works so it's a little bit better than that, but this is kind of a worse case complexity never the less. Okay, yeah, but I mean I will comment on being a little bit cleverer. The way it's written here for going through the grammar isn't the way you want to write it if you want to have your parser run at a decent speed. The way you want to write that searching for rules part is that you've actually got data structures, and they're providing their start a code where you can ask questions about rules. And you can say, "Okay. In the left category I was able to build an NP and a PP; therefore, what rules can I make with that left corner?"

And get back a list of them. But that's the way you start to write an efficient parser as opposed to just naively doing this three nested, four loops through the non-terminal space, and then just asking, "Hmm, does that rule exist in the grammar or not?" Okay.

Okay. Okay. So we now have a parser, and hopefully at some point you will have a parser that works, that can do polynomial time parsing of PSFGs and return the best parse. So that's good. Let me just make one more remark about the status of unarys. A distinctive factor in this treebank parsing, if you just learn grammars of treebanks, is that unary rules cause trouble. And the reason why that is is that you can think of unary rules as just this sort of piece of magic that can transmute one category into another category. So here, this is a real sentence from the treebank, and this piece here is an empty node. So if you look at what else is there, you've got VP rewrites as S. S rewrites as VP. VP rewrites as VBZ. All of those are unary rules. And so, in particular, if you just look at this ones in terms of the phrasal categories, you literally have VP goes to S and S goes back to VP. So you've got this cycle where you can get from one to the other, and back again. I mean I'll note, incidentally, that if you look at – if you look at the – if you look carefully at this parse tree, it looks like the person who annotated just goofed up. This isn't really the right parse tree for this sentence, but it's the kind of complex construction that typically occurs in newswire, where you have these parentheticals put in. So it's, "Bob Ezdrexel, analyst Linda Dunn notes, its properties will be developed over 15 to 20 years." So that you have this parenthetical of somebody says somebody notes, which is conceptually taking the whole of this. The rest of the sentence is a compliment. There's actually kind of parenthetically put down inside the sentence. You've got lots of complex phenomena. But for the moment, just note these kind of unary rules. And so what you actually find if you just naively read off a treebank grammar from the pin treebank is that you find that those unary rules are such that most categories you can transmute into other categories. So if you work it out, this is the same span reachability of – in non-terminal categories in the pin treebank. And they don't collapse, but the thing to notice is that all of the main common categories, sentence, verb phrase, noun phrase, prepositional phrase, adjective phrase, adverb phrase, all of the main categories in the pin treebank you can change into one another by applying unary rules. And the ones that you can't are mainly kind of weird ones that don't occur much.

So there's a special category for lists, and that one you can't get anywhere from. And then there are special ones for "wh" phrases. And well, form those you can get into the strongly connected component, but you can't get out the other way. But basically, you can kind of just move things around from all the main categories. And you're defense against that happening a lot is that every time you apply unary rule, you take a probability hit because if the unary rule has a probability of 0.05, well then you've had to pay the cost of reducing the probability 20 times by applying unary rule. But it turns out that actually that cost isn't that high. I mean precisely because they are unary rules that you've only got sort of one kid beneath you, so there's sort of this bias that those rules tend to have higher probability because, all else being equal, there are just sort of less possibilities there than in binary rules, where you've got two kids beneath you. And so that tends to split up the probability mass a lot more. So a problem that you tend to get when you build treebank parsers if you're not careful is that you get kind of analysis being proposed with far too many unary rules happening, because they're effectively for the parser. They're the way – they're a way to fix up structure. So if you're staying down the bottom, thinking, "Hmm, this looks like a noun phrase, and the stuff up above it, it sort of says well, it be kind of convenient to have a verb phrase there according to my probabilities." The unary rules just say, "Easy, we can do that. We can turn a verb phrase into a noun phrase with the unary rule." And you'll only pay a relatively small hit in terms of probabilities. And so I'll come back to that later. Okay. So these kind of CKY parsers can be made pretty fast. So they're – there are other ways of doing polynomial time parsing, and I'll come back to some of them later.

But in practice, there's sort of a bit of a trade off because you know, CKY parsing is good for systems hackers because it has this very simple data structure where you can just have these arrays and have type four loops and fill in these arrays. And so you can kind of make things very efficient and fast in a systemsy kind of a way, where most other forms of parsing, you have to kind of have cleverer data structures with pointers and all of that kind of stuff. So in our parser, the PCFG parser or the Stanford Parser that you can download, the PCFG parsing is just done with a highly optimized CKY algorithm. And so it can parse sentences of about 20 words in about half a second. So you can really, actually run it with treebank grammars pretty fast. And so that's the basis of in the assignment text that says you should try and get your PCFG parser to be able to parse 20 word sentences within five seconds because we thought you should at least be within an order of magnitude of how fast we could do it. But don't worry about that detail too much. It doesn't matter if it takes 10 or 20 seconds, providing it's just some kind of vaguely sensible number that can run. Okay, and so there are a couple of notes here on how getting things to run fairly quickly, which I will, I think, sort of skip. But this was the bit I was mentioning before. You want to make sure you have efficient grammar lexicon assessors. So you want to be able to have efficient data structures for the grammar, so you can say, "Okay. If the left child is this category, what are all of the rules that you have in the grammar that have this category as a left child?" So you can find quickly what to put higher up in the chart. Another thing that you want to do to make it efficient and fast is you want to make sure you check the zeros, all right, so if things have zero probabilities. So lots of things will be impossible over a span, and if they're impossible over a span, in probability terms they're zero. Or if your working in lump probabilities, which makes it a lot faster, then you can represent that as a negative infinity, long probability. And either way round, you want to test for things that are zeros because lots of things will be impossible. And the algorithm runs right if you just sort of cycle through and consider every way of building every thing, regardless of where the things under it were impossible. But you can get it to work a lot quicker if you kind of abort out of loops where you've noticed that one thing is impossible.

Okay, there are then sort of five slides that talk about a couple of other things, and one of the way to do things, and a few details on the complexity of the Ys. But I think I'll skip through those so I can make sure I can get to the last bit of what does relate to the stuff to do for the assignment today. Okay, but let me first just say a little bit about parser accuracy and how it's normally evaluated. So I mean clearly one way of evaluating parser accuracy is just to say, "Okay, I'm giving this sentence. Is the tree that it puts over that sentence correct, or is it wrong?" And you know, that's not necessarily such a bad way to do it. It's not necessarily bad for two reasons. Firstly, for the kind of parsers we build, that is the objective criteria, and you're trying to optimize. It's the parse correct rate is what your model is trying to optimize. Secondly, you know, for a lot of applications, if you mess up a parse in a major way that the – your treatment of the sentence might be useless. I mean it sort of depends on how you mess up. If you just sort of in some corner do some slight mess up, well then you're sort of okay. But a lot of other circumstances, if there's something that should be a verb hitting a verb phrase, and you're turning into a noun hitting a noun phrase, then probably you're just not going to be able to get out of the sentence the kind of stuff that you want to be able to use it usefully in application, even if there are considerable parts of the sentence that are parsed correctly. But at any rate, that's not normally what people do to evaluate parsers, and I mean think part of the reason for that is just the whole sentence correct rate runs somewhere around 25 to 30 percent, and people never like evaluation measures that they can't do well on. So the norm is to evaluate parsers based on getting parts of parsers correct. And in particular, for all the work that's been done in statistical parsing over the pin treebank over the last 15 years, the standard evaluation measure is done in terms of labeled precision, labeled recall of constituents. So what you do is you regard a parse as consisting of triples of a label, and a start and a finish point. So in the picture here, we have a constituency claim here that here's a verb phrase that extends from word position three to position nine. So that's a constituency claim. And there are other constituency claims, like here's a claim. There's a prepositional phrase from position six to nine.

So we have a set of constituency claims. And so we can just write down the set of all those constituency claims. And we can do that for both the parse tree returned and for the gold correct parse tree as done by the human annotators. And so what we can then do is calculate precision and recall, just like we did before, over these constituency claims. So that if the gold parse also has this prepositional phrase, that will count as one point of recall. And if it's in my set it will add to my precision, whereas if I said this whole VP is a constituent where the gold parse had the VP stopping here, then that will count as errors, and so I'll have a false positive claim of a category. Okay. And so then, just like before, the standard thing to do is then to give this F measure of precision and recall. The standard is to use F measure, but in this particular application, the F measure sort of essentially, it's not kind of interesting because, I mean, in practice everyone is putting a tree structure over sentences, so – which is sort of resembling parse tree. It's not the case that some people are returning a completely flat analysis of saying sentence goes to all the words, and therefore, doing terribly on recall, but they're precision is always 100 percent because the things that they do propose is always right. There's a sentence at the top. In practice for statistical parsers, their precision and recall is so near to balance that it's not really sensible to report two different numbers, and so you may as well just really be using the average. Okay, so that's the labeled precision, labeled recall F measure that's used for pin treebank parser accuracy. I mean it's not actually necessarily a very good measure. There are kind of – I won't go into a lot of detail now, but sort of there are various reasons to think it's not a very good measure. And most of those reasons point in the direction of using dependency-based evaluation. So in recent times, there's been a fair amount of advocacy of maybe we should be doing dependency-based evaluation of parsers. But never the less, on the ground, if you're wanting to write a parser and show that your parser's better than the other guys parser, it's still the case that this is just been the standard measure. The reason why it's sort of not clear it's such a good measure is – well, firstly here, these claims of this is a prepositional phrase, you know, it sort of seems like that's kind of far from what anyone is directly interested in for applications. Whereas the claim that this prepositional phrase modifies this noun, that's a dependency claim, that seems much closer to what people are interested in in applications.

But secondly, and perhaps more importantly, there's sort of a weird property of these labeled precision labeled recall measure, which is that errors compound, so that if you make a mistake in the attachment of something, you will then get this effect that that will also cause you to have errors up above that in the tree. And so that one mistake in attachment can cause multiple mistakes in label precision-labeled recall. Not withstanding that, it works okay as a measure. It's what people use. And it's what we use in the assignment. Okay, so the first obvious thing to do with a PCFG is to say, "Okay, well, what I'm going to do is I'm gonna take these treebank trees. I'll delete the empties from them." And for my parser I might binarize or whatever, but essentially I'm gonna take the trees that exist in the pin treebank, and just read off context-free grammar rules from the trees that are in the treebank. And I'm gonna count how often different rewrites occur, and that is going to give me a PCFG. And I'm gonna go and parse with it. And so if you just do that and nothing more, the question is, "Well, does that work excellently? Or does that work badly?" And the answer is that it sort of works, but some of the things that I'll want to do for the rest of this class, and also talk about in next class is some of the ways that it doesn't work so well, and some of the kind of work that people have done in statistical parsing to get something to work much better. Okay, so, well, what's on the good side? If you do that, if you build such a PCFG and go and parse with it, the good side is we've solved the problem of non – without doing anything more, we've solved the problem of non-robustness of parsing that if you just take this kind of treebank PCFG, and then try and go and parse sequences of words, it's actually provable. I guess it's a data distribution dependent fact, but it's actually provable that if you just read a PCFG off the pin treebank that that PCFG will parse every sequence of words over the vocabulary in the pin treebank. That there's effectively enough looseness in the grammar, enough ways of sort of middling around for those unary rules and things like that. That there is no sequence of words that's unparsable. So the language of the grammar is just vocabulary star. That there – so that means the grammar has no grammar-driven constraint whatsoever anymore, every sequence of words is parsable, it just might have a more or less good old weird structure. Okay, so that's good. We now have a robust parser.

We also now have some sort of solution to this problem of grammar's causing you to give you way too many parsers for a sentence because using the CKY algorithm and the PCFG, we have a straightforward way to find the highest probability parse for a sentence. Through a building up of a CKY chart, we just kept the highest probability parse of a category over a span, and then when we get right to the top, we just look at all the cells, look at all of the constituents in the top cell, and say, "Which of those has the highest probability?" Return that one. And then that's the highest probability analysis of the whole sentence. I mean to put in a footnote there, if you actually do just what I said there, I mean that's sort of equivalent to having no start category in the grammar. So you're just saying, "What is the highest probability thing that I can make over the entire sentence? I'll return that." If it's a noun phrase, well, I just say the whole sentence is a noun phrase. And in practice, that's often the correct thing to do because of the fact that you find sentences that are only noun phrases in real text. Of course, if you wanted to insist that the thing at the top has to be a sentence, you can just look at the probability of the most likely sentence. Okay, so that's the good side of what we can do. There are then, also, some things that are not so good. So the first one is one that I'll go into in detail in just a moment is that PCFGs, as we have them now, have really, really strong independent assumptions. So that limits their utility as a parsing algorithm. Another of the supposedly good features of a PCFG is that this is a language model, and so you could naively think we're now in a perfect world because now we have something that's a language model that we could use for speech recognition and machine translations, spelling correction, all of these good things. But now, instead of it just being a Markoff model that looks at a sequence of words, now it's a grammar sensitive language model that knows about noun phrases and verb phrases and things like that. But if you actually try that out, your enthusiasm is very quickly knocked a very bad blow because it turns out that if you just use a simple PCFG as a language model in practice, it works a lot worse than using a trigram model. And the reason for that seems to be that PCFGs don't actually know much about words and how words go together precisely because of these, strong independence assumptions.

Now that we know about the label precision-labeled recall F1 measure, if you just read off a treebank grammar, the score you get is about 73 percent. A few fine points about how you deal with unknown words that is about that. And there are two ways you can react to that. One way of reacting to that is, "Hmm, that's not such a bad number." And in some sense it isn't such a bad number if you just have this treebank grammar. You know, it sort of parse, and most of the time, most constituents it places correctly over sentences, almost three-quarters of the time. But, you know, on the other hand, if a lot of sentences have about 20 non-terminals being postulated in them. I've – one fine point I should have mentioned is when I said you make a set of constituency claims, the convention is that this set of constituency claims ignores the claims about parts of speech. So the accuracy on parts of speech is reported separately as just an accuracy number. So you're not getting points in here for having said word 13 is a noun and word 14 is a verb. Those aren't counted as constituency claims. It's only the claims about phrase or categories above that. So if your typical sentence has about 20 phrasal category claims, and this is your accuracy, it means you're getting a bunch of things wrong. But you're getting about five things wrong in the average sentence, and that's quite a lot. Okay, and so – yeah, so one of the things that people first – so the first thing that people noted about this result and how they might improve things is that there seems to be a problem with PCFGs that they don't really know about information about words the way in n-gram model does. So if we have a bigram model, it has the probability of the next word being "Arabia" given that the previous word is "Saudi." That's kind of pretty powerful knowledge for a bigram language model that really helps predict words in sequence. If you think for a moment about a PCFG, it doesn't have any of that knowledge. So a basic PCFG, you'll have a rule for a noun phrase saying noun phrase goes to an NP. An NP, you can rewrite a noun phrase as a sequence of two proper nouns. But then you just have these independent lexicon rules saying a noun phrase can be Tokyo, a noun phrase can be Vancouver, a noun phrase can be Saudi. So a proper noun can be Saudi, a proper noun can be Arabia, a proper noun can be Angeles, and you know, you're just as likely to produce Saudi

Angeles, as you are to produce Los Arabia or Saudi Arabia. Right? The probability of those is just going to depend on the independent unigram probabilities of the words. And so that means you kind of don't actually get good language modeling performance.

Okay, and so the next direction in which work headed was to say, "Well, somehow, we need to – there are these profound facts about words and word distributions and somehow we need to be starting to put those into our PCFG models so that we'll be able to model things much better. Okay, and so the kind of idea you can learn about that from the syntax perspective as opposed to sort of a speech recognition perspective is actually for working out what phrasal rules are likely in different places, it just helps to know more about the words involved. And so one of the places that that's obvious that we talked about last time is attachment ambiguity. So we talked about examples like, "The astronomer saw the moon with a telescope." And the idea we had in mind was well, we'd be able to look at words and say, "see with telescope," you know, that sounds good. You know, that we'd use properties of the words to predict how to attach prepositional phrases. And that we saw data last time that if you use the words to predict prepositional phrase attachment, you can actually do quite a good job. You can accuracies in the high '80s. But again, if you think about it a little bit with PCFGs, the – a naïve PCFG doesn't actually have any ability to get that right because it's just gonna have rules like verb phrase goes to verb, noun phrase, prepositional phrase; noun phrase goes to noun phrase, prepositional phrase. Then it'll have rules like prepositional phrase goes to preposition, noun phrase. And then it'll have a rule like preposition goes to width. And so each of those rules is an independent probabilistic choice. And so it just doesn't have any ability to actually do sensible modeling of PP attachment decisions. And there are lots of other places where that turns up.

And so it happens in coordination. So another prominent kind of ambiguity in English is the scope of coordination. So here we have, "The dog's in the house and the cats." So this could be a narrow-scope coordination where you have the noun phrase "the house and the cats," or it can be wide-scope coordination where you have "the dog's in the house and the cats," which is obviously the likely one here. And why is it likely? Well, again it seems to have to do with words, right? Cats and dogs is a good pair. That makes sense to coordinate. So you want to be building the noun phrase "the dog's in the house." Whereas house and cats just isn't a likely thing to coordinate, and you're not likely to have dogs that are in cats, that's a bit bizarre. So the dyslexical facts that's determined what's good. This third example is working out the structure of verb phrases, which I show on the next slide, that when you're building phrases, what kind of verb phrases is likely to – are likely depends a lot on what the head verb is. So if you have a head verb like "put," "put" really wants and almost always has both an object noun phrase and the prepositional phrase after it. So "Sue put the book on the table" is good, where "Sue put the book" basically doesn't work. "Sue put on the table" basically doesn't work. Whereas lots of other verbs are transitive and will only take an object, so "Sue likes the book" is normal. "Sue likes on the table," that's weird. So we need to know about the verb to know how to predict things. Okay, and so that led to a lot of work on doing lexicalized parsing. And I'm gonna come back to that and talk about it more next time. But for the rest of this time, I'm gonna talk about some work that I did with a former student, Dan Kline, on how much you can improve PSFG parsing by doing unlexicalized parsing. And so the hypothesis of this work – I mean this was, in some sense, a piece of work. There was a piece of data analysis work that, really, everyone else in the field had been sort of very obsessed with this idea of using lexicalization to help parsing. And it's not that it's a bad idea. It's a good idea because things like PP attachment and conjunction scope, you want to know about the properties of words. It's every bit a good idea, but my intuition was – well, actually there's a lot of other information that's structural information that you can use to improve the quality of parsers. And nobody was thinking about how you could exploit better structural information in your grammar to come up with better parsers. And so this was attempting to show that there was a lot of structural information of that sort.

And you could do enormously better than a naïve PCFG by having a better encoding of the structural information in a parse. And so I'll talk about that for the rest of the time today. And then, really, for – what you were meant to do for the assignment is first of all, just get a PCFG parser working, and then do a baby bit of this kind of work or making use of ways to structurally improve your grammar. Okay, so I've already said kind of bad independence assumptions three times, but here's a kind of a nice way, I think, of thinking about the independence assumptions of a PCFG. So when you have, you know, you're tree in a PCFG and you can focus in on any category of that PCFG, and if you do, you have the stuff that's above and around it, which is the blue stuff here. And you have the stuff that's under the NP, the red stuff. And effectively, in a PCFG, any node in the PCFG is a bottleneck for information flow. So the probability of anything that's built down here depends only on knowing that this is a down phrase. It's completely independent of any of the words and structure that's outside of that. Similarly, the probability of all the stuff outside of this depends only on knowing that this span is being parsed as a noun phrase. It knows nothing about the internal structure of this noun phrase. Now, the problem is, then, just simply saying that this is a noun phrase gives you incredibly little information that we know that there are all sorts of dependencies of information in a sentence that connect across this boundary. And you know I was illustrating some of them on the previous slide, motivating, lexicalization, so things like PP attachment, coordination, scope, that you wanted to know about stuff that was inside the noun phrase, or stuff that was inside a prepositional phrase to work out how likely the entire sentence was. And yet it's not represented in this category. Okay, well, if we want to fix that problem, how can we fix that problem? And the obvious way of fixing that problem, which is used both for lexicalized grammars and the stuff that I'm going to show right now is, well, what if we just put more information here. So suppose that we said not just that this was a noun phrase, but we said it was a noun phrase with a prepositional phrase modifier? Or we said this noun phrase was a pronoun? If we just sort of added extra information here, well, then we'd get more information flow between the red part and the blue part. And so then the question is how much information do we need? Like, how much information is it vital to know about this red part to be able to inform the structure you choose up above in the blue part?

You know, one answer is to say; you need to know absolutely everything. There are no independence assumptions. And, I mean in some sense, that's probably right, but you might wonder; is there key information about this that we can tell to the blue part, which would be sufficient for it to be able to choose the right parse in a lot of circumstances? And the idea that motivates working out whether there's information apart from lexical information that would be useful to know is, if you sort of think of linguists putting structures over sentences, that by and large, the kind of structure that you put over sentence isn't actually dependent on knowing the particular words, that there are kind – Linguists normally think just in terms of categories of things you can do with an adjective and things that you can do with an adverb, and how you can combine them together. But details of which particular adjective or verb it is will influence which is the most likely parse in terms of attachment ambiguities. But it shouldn't really influence much, just what sort of structural patterns are good structural patterns of the language, whereas the perception was that the treebank PCFGs didn't even correctly capture, a lot of times, the structural patterns of the language. And I'll show a couple of examples of that in a moment. So here's a couple of more examples of the independence assumptions of a PCFG being too strong. All right, so that if you look at the category "noun phrase," and then consider noun phrases in different places, it turns out that how noun phrases expand really varies, depending on where they appear. But that's not what we were predicting here, where we're saying the possibilities of expanding a noun phrase are exactly the same no matter where that noun phrase occurs in the sentence. And a simple example of that is if you look at noun phrases that are noun phrase subjects, a lot of noun phrase subjects are pronouns. So, in the Wall Street Journal, one-fifth of noun phrases subjects are pronouns, he, she, it, I, you. And then that 10 percent are determiner noun, nine percent have verb prepositional phrase post modifier. However, if you look at noun phrases that – object noun phrases inside a verb phrase, the rewrite probabilities are just stunningly different. So only about four percent of them are pronouns. And almost a quarter of them have a prepositional phrase modifier. And so, well, if you're got more linguistic background, you can say, "Oh, yeah. That makes a lot of sense."

She should think about that in terms of information structure of discourse, that the typical pattern is you have an established topic, who will be the subject of the sentence. And since that topic has been referred to previously, it's quite likely that the topic can be referred to as a pronoun, whereas conversely, what's appearing as the object of the sentence is much more likely to be new information. And therefore, if you're introducing new information, you're most likely to use a full noun phrase. And indeed, you're very likely to add on to that noun phrase something like a prepositional phrase or a relative clause, which is giving the hearer more information about that noun phrase because it's the first time it's used in the discourse. And so, you know, this all makes sense in terms of higher levels of linguistic theory, but it points out that there's kind of a lot of information that's just structural information about how noun phrases rewrite that is completely being failed – completely failing to be captured by a basic PCFG. Okay, and then lexicalization, just to mention again, quickly, once more, you can also think of it as what you want to do is encode more information into non-terminal nodes. So here we have "The lawyer question the witness." For basic PCFG, there's no – there's just no information flow between "question" and "witness" because each of these nodes in between is an independence claim. So that's bad news because what we really want to be doing is saying is, "'Question witness,' is that a plausible verb-object combination?" And similarly, is "lawyer question;" is that a plausible subject-verb combination? So what you do in lexicalized grammars, just briefly, is that you define for each rule a head, which is in the fairly obvious way. So that a noun is the head of a noun phrase, a verb is the head of a verb phrase, and the verb phrase is the head of a sentence. And then what you do is you percolate up the headwords, up these head chains. So "witness" goes up to the noun phrase. "Question" goes up from the verb, to the verb phrase, to the sentence. And lawyer goes up to this noun phrase. And if you percolate headwords up in that fashion, you then have rewrite rules that aren't only about categories, but are also about words. So here we have S "questioned" rewrites as noun phrase "lawyer," verb phrase "questioned." And this seems bad news from the implementational perspective because the number of non-terminals in the grammar would now be humongous. It's the previous – it's the size of the previous set of non-terminals times the number of words in the grammar.

And so you have to do clever stuff for lexicalized grammar to make it tractable. And so we don't do that for the assignment. But in terms of information – look, now we're saying is it likely to have a sentence headed by questioned, have a noun phrase subject headed by lawyer, that we precisely are getting this information about words that we'd like to have back in our model and of the kind that we had in bigram and trigram models, only now it's also sensitive to syntactic structure. Okay, so I've tried to motivate these independence assumptions being bad. You know, one question is well, they're bad, but all probabilistic models make independence assumptions. Do these probabilistic models actually do harm? Do these independence assumptions do harm? Can you see places where they cause you to do the wrong thing? And the answer is, well, yeah. Actually, it's easy to find places, if you just run a naïve PCFG, where it does the wrong thing because of the independence assumptions. Here's one example, which is a slightly technical one. But in the pin treebank, basic noun phrases are given a flat analysis. So it's just something like "big board composite trading" should be all flat. But if you just make a simple PCFG and you try and parse "big board composite trading," what you'll actually get back is "big board" as being grouped into a noun phrase, that's then being put into this bigger noun phrase like this. Well, how is that possible, if this is the structure that's always used in the pin treebank? The reason it was possible is that rules like NP goes to NP, adjective noun do occur in the pin treebank, but for only one construction. They're used for possessive constructions. So if it was – if the construction was something like "Chicago's composite trading," this would be the right tree structure on top. But the category noun phrase doesn't record whether there's a possessive noun phrase down here or not, and so that information is lost. And so the wrong structure gets used in these kinds of examples. Okay. So the strategy here is to say, well, what about if we just break up the symbols, so we sub-categorize the symbols into finer symbols? So we can record information this useful. So what if, when it is a possessive phrase, like "Fidelity's new ad," instead of calling this category noun phrase, we call it noun phrase possessive. So we've split noun phrase into two categories. Then we've weakened the independence assumption between down here and out here, and therefore, you'll only be able to use this parse structure for possessive constructions.

And so you won't be able to make the mistake on the previous slide. This is useful. Another thing that's useful – in fact, if you do one thing, the simplest thing to do that is really useful is this process being proposed earlier by Mark Johnson, which is parent annotation. So parent annotation means that you split up your categories by just annotating onto each category what its parent category is. So this is now saying, "I'm an adverb phrase, which appears under a verb phrase." And this one's saying, "I'm a verb phrase that's appearing under a sentence." So if you're looking at the whole tree, that looks very redundant because well, look, adverb. Here's the verb. Why do you have to write it down here? But if you think in terms of the probabilistic independence assumptions, it means that when you're expanding this adverb phrase, that you know it's under a verb phrase. And it turns out that's an enormous amount of additional information because first, something like an adverb phrase, if it appears under a noun phrase, it normally has failure restricted forms, like it's often very, or extremely, or something like that, whereas if an adverb phrase appears under a verb phrase, there are all sorts of different things it's likely to be, like sometimes, occasionally, etcetera. And this also would work for the case I looked at before of pronouns being common in subject position. If we do this, we can model that because we can say an NP under S is very likely to be a pronoun, where we can say an NP that inside a verb phrase, an object NP, is much more likely to rewrite as something that has a prepositional phrase inside it. Okay, so basically, that's the entire strategy. We take these categories and we split them into subcategories that preserve more information. And I'm just gonna write them as strings like this, NP under an NP, that's possessive. So that they are just literally, you know, non-terminal names that are kind of big and sort of semantically interpretable in our heads. If you want to you can also think of them as feature-based categories. And if you've seen feature-based categories in something like Ling120, if you're an [inaudible] assist student that that feature-based grammar has essentially come out of the same direction of thinking about these complex categories as a bunch of features. Okay, so for pin treebank parsing, which is what this work has done, you know. There's this sort of – I've talked about the pin treebank and there's effectively this sort of stand that's set up where you train on part of the treebank and test on another part. I've talked about evaluation accuracy. And the one other figure I'll use is what is the size of the grammar, which is the number of non-terminals in the grammar? And some of which of the basic non-terminals, and others of which come from the binarization of the grammar. Okay, and so what these experiments essentially do is start with the basic PCFG and then gradually kind of split the categories to try and get a more accurate PCFG. And I'll skip lexicalization here. Okay, so the first thing that you can do – and you might want to try for the assignment, and this is again something that will be talked about more in the section – is this process called here horizontal markupization.

And so this has to do with how you binarize the rules, so that if you start up with flat categories, say like, you find in the pin treebank noun phrase goes to proper noun, proper noun, proper noun, and you can even find noun phrase goes to proper noun, proper noun, proper noun, proper noun, proper noun, these big flat rules. And so if you binarize these rules to make them binary, the kind of obvious way to do it is to do exact binarization so you remember your left context. So here, you're saying this is the ways to continue expanding a noun phrase that began with two proper nouns. And so you've got a distribution over that. And the context of where you are up to is being represented with these dotted rules. So this is saying, "I'm expanding a noun phrase in which you've already seen two proper nouns, and then what else can come next?" And the things that could come next might be another proper noun, or maybe there'll be a common noun because then you could have something like "United States election." Okay, and so if you do this kind of binarization, you get an exact binarization, which is equivalent to your original grammar. But – and one idea is to say, "Well maybe I should generalize a little by not remembering everything that I've seen to the left, but only a little bit of the stuff that I've seen to the left." And so what you do is mark-upize the grammar in exactly the same way as we Markoff models for language models. And so you end up merging these states, and saying, "Okay, I'm expanding a noun phrase. The last thing was a noun phrase, and I'm forgetting the stuff that comes before that." So markupization gives you a way of generalizing your grammar. And so, if you've mark-upized your grammar, you shrink the size of the grammar in terms of the number of non-terminals in it a lot. And it turns out the performance actually improves, effectively because your grammar generalizes better. So the category's here, is infinite means no markupization. Two means second auto markupization. One means first auto markupization. Zeroth means zero auto markupization, which would here mean your just saying, "I'm in a noun phrase. I've foregone everything that I've seen before. Tell me the distribution over noun phrases." And the 2V in here is kind of doing something in between, which was actually optimal of using either first or second auto markupization, depending on how much data you'd seen for a category. And so if you do that, you can improve your PCFG parsing accuracy by about a percent. I'd already mentioned the idea of parent annotation. You can think of parent annotation as – there's access of parent annotation. There's also like a markupization process, where it's kind of the opposite.

In this case, if you just have a basic PCFG, that's like having a first order vertical markupization because you're expanding a node based on knowing only what the parent category is. And that suggests, like, well maybe we could say, "I want to know the parent category and the grandparent category. That's a second order vertical markupization. Or what if I wanted to know the great-grandparent category as well. And then I can go to a third order vertical markupization. And so as you do that, the size of your grammar increases because you've got categories like this, VP under S, rather than just VP. But it turns out that using parent information just enormously improves the quality of a PCFG, so that the single thing that you can do that causes the biggest accuracy increase in the PCFG that anyone has ever found is just to annotate all categories with their parents. So if you do nothing else, you get about four and a half percentage increase in your PCFG accuracy. You can, by putting in grandparents, even get a bit more than that. Okay, and so that kind of gives you a space in which you can do both. And as you kind of go along these axis, the number of categories in your grammar gets bigger and bigger, and your accuracy is kind of max somewhere in the middle. And so, for a particular grammar that we used as a basis for further exploration, we used this variable first, second order markupization, both horizontally and vertically. And so that was then giving 77.80th one. So we're up by about 4.8 percent from doing a basic treebank grammar. So then heading beyond that, the idea was to completely, by hand, introduce other states splits into the grammar that will improve the quality of the grammar. And so the mechanism for doing that was just to look at what the grammar did wrong, diagnose why it did it wrong, and put in states splits that recorded extra information that was necessary. And I won't go through all of this in detail, but the kind of the nature of it is, as I mentioned to you, unary rules often go out of control in PCFGs. So "Revenue was 449 million, including net interest," it's built a structure with this unary rule, but it's kind of not right. It's not the right structure. And one way to fix that is to split states in the grammar to mark whether they're the parent of a unary rule or not. And if you put in that split, the PCFG will then chose the right structure for this sentence, and fix that error. And that gives you about half a percent of accuracy.

Another place where you kind of get bad presentation in pin treebank PCFGs is the parts of speech tags are actually quite coarse and bad in some respects. One of the places where they're worst is in the things that are labeled "in" for preposition. It turns out in the pin treebank all kinds of things labeled "in" for preposition, including things such as subordinating conjunctions in traditional grammar, like "if advertising works." So here, for this sentence, the analysis that the grammar – the parser gave was "if" is a preposition, and then "advertising works" is just being parsed as a noun phrase, which is just wrong. And work should be being parsed as a verb going into a verb phrase. And that analysis should be plausible because "if" is a subordinating conjunction that should take a clausal compliment. So we can fix that. We can split the preposition tag into sub-cases that represent the different sub-classes of preposition, and if we do that, we then get a fixed-up analysis for this sentence. And that splitting of the preposition, just by itself, is actually very useful. You can get a couple of percent in accuracy by doing just that. And so essentially, we kept on doing that same trick as much as we could and kept on doing it. And, you know, essentially what stops you doing just this trick at some point is that your grammar starts to become too sparse because there's nothing subtle and clever being done here. There's no smoothing of the grammar of any sort. We're just keeping one splitting things. And so, as you split, split more, your grammar starts to become too sparse. And you start to have holes and coverage. But nevertheless, you can split it a fair way, and make the accuracy go up quite a lot. And so you end up with a grammar that kind of has categories like this. This is a VP under a sentence, which is headed by a finite verb, and the VP has a verb inside it. So I've got these kind of split categories that record quite a bit of information. And the result that we got out of that was that we could build an unlexicalized PCFG that got through 86 and a bit F measure. So that's a ton higher than where we started off from, which was only 73. And it was actually quite competitive with some lexicalized parsers. So the interesting thing was that just showed that there was enormous amount of structural information that you don't capture using a naïve PCFG over pin treebank, which can enormously improve parsing, even before you get to the kinds of places like PP attachment, where you actually need to know like word properties to make parsing better. Okay, that's it for today. And then next time I'll start into lexicalized parsing and stuff like that.

[End of Audio]

Duration: 77 minutes