English syntax tree diagram pdf

Tree diagrams are a newer method of creating sentence charts that are most commonly used by language experts and other academic professionals. While the Reed-Kellogg diagram is considered an effective tool for students to visualize sentence structure, it has many limitations. It dispenses with traditional word sequences and uses a variety of sometimes confusing symbols, which means the resulting diagram is difficult to understand for anyone unfamiliar with this method. Reed and Kellogg did introduce two core grammar concepts: Constituency, how a word deals with a larger sentence structure, and dependency, how a word depends on each that precedes it. The main purpose of tree diagrams is to illustrate these concepts in a clearly visible way, even for those previously unfamiliar with sentence charts. In the tree diagram, sentences are divided into two parts: the subject and the predicate. They consist of noun phrases or verb phrases. This is a group of words that include nouns or verbs and any words added as modifiers. A subject is a noun phrase while a predicate is usually a verb phrase. Noun Phrase A large dog consists of an infinite article 'a', the adjective 'big', and the noun 'dog'. Verb phrases jumping over fences consist of the verb 'jump' and the preposition phrase 'over the fence'. Unlike the Reed-Kellogg diagram, these components are not separated by slashes and other symbols. Instead, they descend from the subject and predicate in the form of a line that acts as a branch. This continues until each noun or verb phrase is broken down into the simplest parts. In the end, a sentence diagramted in this style should look like a broad tree, with the subject and predicate acting as a trunk and a sentence modifier standing as a colorful and complex leaf that gives it a personality. Now that you understand the basic premise of the Tree Chart and how it breaks down sentences, let's look at an example. Seen here, sentences are broken down into subjects and predicates. The subject is a noun phrase consisting of an unspecified article 'the' and the noun 'dog'. The predicate is more complex, as it consists of verbs and noun phrases. Breaking down predicates, verbs 'eat' and noun phrases are 'the' (unlimited articles) and 'bones' (nouns). As you can see, tree diagrams use minimal symbols and complex little jargon, but clearly describe how each of these words relates and depends on each other. Here's another example of a tree diagram. As you can see, this one is a little more complicated. Let's look and break it down. Again, sentences are divided into subjects and The subject consists of the noun phrase: 'the' as an infinite article and 'teacher' as a noun. The predicate is more complex than ever. The verb phrase consists of three parts: the verb 'give'; the noun 'homework'; and the preposition phrase 'to his disciples'. Are you starting to start get a better understanding of constituency and dependency now? Unfortunately, tree diagrams do have some negative aspects. Like the Reed-Kellogg diagram, more complex tree diagrams can take up a lot of space and become more difficult to decipher in the process. Moreover, as strengths and weaknesses, they are more open to interpretation than reed-kellogg diagrams. It is possible for sentences to have several different, different, and equally valid tree diagrams depending on which unit is focused, especially with sentences taken from classical literature. Overall, tree diagrams offer a clear and more nuanced look at sentence structures without sacrificing traditional word sequences. Although they are mainly used by grammarematics and other linguistic specialists, they quickly become the standard method of sentence diagrams, as the results are easy for everyone to understand. If you want to improve your writing, I recommend that you try creating diagrams of at least one sentence a day using this method. That way, you'll gain a greater understanding of how to compose correct, diverse, and grammatically impactful sentences. Interested in more powerful content? Follow UNG Press on Facebook, Twitter and Instagram and find our full catalog on our homepage. This unit introduces the basic vocabulary for tree diagrams. Tree diagrams are notations that most sintatika use to describe how sentences are arranged in mental grammar. Check Yourself In the following tree diagram: 1. What is the structural relationship between love V and sushi NP? V and NP are sisters. V is the mother of NP. V is the daughter of NP. V and NP are not associated with any of these three ways. 2. What is the structural relationship between NP Colin and V love? NP and V are sisters. NP is the mother of V. NP is the daughter of V. NP and V is not associated with any of these three ways. 3. Which node is the sister of NP Colin? Video Script We'll start to see how sentences are organized in our mental grammar. Before we do that, we should be familiar with a certain type of notation called a tree diagram. We will see that, in each sentence, words are grouped into phrases. Phrases can be grouped together to form other phrases, and form sentences. We use tree diagrams to describe this organization. They are called tree diagrams because they have many branches: each of these little lines that join things in the diagram is a branch. Each place where branches are assembled is called a node. Nodes show a set of words that act together as units: each node corresponds to a group of words called constituents, which you'll learn in other units. If a node does not have a daughter, we call it a terminal node. tree diagrams, we can also talk about the relationship between different parts of the tree. Each branch joins two nodes. The higher one is called the mother, and the lower one is called the daughter. A mother can have more than daughters, but every daughter has only one mother. And, as you would expect, if two daughters have the same mother, then we say that they are brothers to each other. Having this vocabulary for tree diagrams will allow us to talk about the sintatic relationship between parts of sentences in our mental grammar. This is what the top of most of your tree should look like. Sentences always start with NP (Noun Phrase) and VP (Verb Phrase). Sentences without verb phrases will always give you snippets. Tip: First try your start is your tree like this. It's usually going to work. S is a category label. All your trees should be labeled from now on (We've passed the point in the path where you use some un labeled trees to use top things). What label labels are called nodes. The dots in the tree's originating branches are called nodes. In this incomplete little tree, S is the parent node. NP and VP are princess nodes. Let's finish the tree: NP nodes and VP nodes now have their own daughters. Node NP has two daughters, node Det (Determiner) and node N (Noun). The VP node has one daughter, node V (Verb). In general Noun Phrases will have daughter N and verb phrases will have daughter V. These are called their head words. So you should stop drawing trees that look like this: Before you leave a tree check to make sure that all labels make sense and that all phrases have heads: These trees are trees. There's an NP with nouns in it, a VP with no verbs in it. If a set of words doesn't have a noun in them, we don't call them Noun Phrases (NP). Each type of phrase has a distinctive head. Phrase Head Pair: Phrase Phrase Phrase: PhraseHeadCategory Name NPNNoun VPVVerb PPPPreposition APAAdjective AdvPAdvAdverb Is there an exception? Always. But there's not too much we're going to worry about. One important class exception, Pronouns and Proper Names. These are special types of nouns, really, but we'll draw trees with them like this: Previous chapters focused on words: how to identify them, analyze their structure, assign them to lexical categories, and access their meanings. We've also looked at how to identify patterns in word order or However, this method only scratches the surface of the complex constraints that govern sentences. We need a way to deal with the ambiguity that natural language is famous for. We must also be able to address the fact that there are an unlimited number of sentences, and we can write limited programs to analyze their structure and find its meaning. Along the way, we'll discuss the basics of English syntax, and see that there are aspects of systematic meaning that are much easier to capture once we identify the sentence structure. Previous chapters have shown you how to process and analyze text korpora, and we have emphasized the challenge for NLP in handling the large amount of electronic language data that grows every day. Let's consider this data more closely, and experiment with the idea that we have a giant corpus consisting of everything that has been spoken or written in English for, say, the last 50 years. Are we going to be justified in calling this corpus modern English? There are a number of reasons why we might answer No. Remember that in 3, we ask you to search the web for example the pattern used. While it's easy to find examples on the web that contain this word sequence, such as The New Man on IMG (, English speakers will say that most of those examples are errors, and therefore not part of English. Thus, we can argue that modern English is not equivalent to a very large set of word sequences in our imaginary corpus. English speakers can make judgments about this order, and will dismiss some of them as ungrammatic. Similarly, it is easy to create a new sentence and ask the speaker to agree that it is excellent English. For example, a sentence has an interesting property so it can be embedded inside a larger sentence. Consider the following sentences: (1) a.Usain Bolt broke the 100m record b.The Jamaica Observer reported that Usain Bolt broke the 100m record c.Andre said the Jamaican Observer reported that Usain Bolt broke the 100m record d.I think Andre said the Jamaican Observer reported that Usain Bolt broke the 100m record if we replaced the whole sentence with the symbol S, we'll see a pattern like Andre says S and I think S. This is a template for taking sentences and building bigger sentences. There are other templates we can use, such as S but S, and S when S. With a little ingenuity we can build some very long sentences using this template. Here's an impressive example of the story of Winnie the Pooh by A.A. Milne, In which a Piglet Is Completely Surrounded by Water: [You can imagine the joy of piglets when the ship finally came to see it.] After all these years he likes to think that he has been in Great Danger during the Terrible Floods, but the only danger he really lived in was the last half hour of his prison, when Owl, the new one flying, sitting on her tree branch to comfort her, and telling a very long story about an aunt who once laid a seagull's egg by accident, and the story continues, more like this sentence, until piglet who listens out of her window without much hope, goes to sleep and naturally, slipping slowly out of the window toward the water until he was just hanging on his feet, at that moment, luckily, a sudden loud squawk from Owl, who was really part of the story, became what his aunt said, woke the Piglet and just gave him time to jerk himself back to safety and say, How interesting, and is he? when - well, you can imagine his joy when he finally sees a good ship, the Brain of Pooh (Captain, C. Robin; 1st Mate, P. Bear) comes to sea to save him... This long sentence actually has a simple structure that starts S but S when S. We can see from this example that language gives us a construction that seems to allow us to extend sentences indefinitely. It is also striking that we can understand arbitrary long sentences that we have never heard before: it is not difficult to craft an entirely new sentence, which may not have been used before in the history of the language, but all speakers of the language will understand it. The purpose of grammar is to provide an explicit description of a language. But the way we think about grammar is closely related to what we think of as language. Is that a large but limited set of observed speeches and written texts? Is it something more abstract like the implicit knowledge that competent speakers have about grammatical sentences? Or a combination of the two? We will not take a stand on this issue, but instead will introduce a major approach. In this chapter, we will adopt a formal framework of generative grammar, in which language is considered nothing more than a large collection of all grammar sentences, and grammar is a formal notation that can be used to produce members of this set. Grammar uses recursive production of S and S → S and S forms, as we will explore in 3. In 10 minutes. we will expand this, to automatically build the meaning of the sentence from the meaning of its parts. An example of the famous ambiguity featured in (2), from Groucho Marx's film Animal Crackers (1930): (2)While hunting in Africa, I shot an elephant in my pyjamas. How he got into my pyjamas, I don't know. Let's take a closer look at the ambiguity in the phrase: I shot an elephant in my pyjamas. First we need to define simple grammar: >>> groucho_grammar = nltk. CFG.fromstring(... S -> NP VP ... PP -> P NP ... NP -> Det N | Det N PP | 'I' ... VP -> V NP | VP PP ... Det -> 'an' | 'my' ... N -> 'elephant' | 'pajamas' ... V -> 'shot' ... P -> 'in'...) This grammar allows sentences to be analyzed in two ways, depending on whether the preposition phrase in my pyjamas describes an elephant or a shooting event. >>> sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', >>> parser = nltk. ChartParser(groucho_grammar) >>> for trees in parser.parse(sent): ... print(tree) ... (S (NP I) (VP (VP (V shot) (NP (Det an) (elephant N))) (PP (P in) (NP (Det my) (Pajamas N)))))) (S (NP I) (VP (V shot) shot) (Det an) (N elephant) (PP (P in) (NP (Det my) (pyjamas N))))))) The program produces two caged structures, which we can draw as trees, as shown in (3b): (3) a. b. Note that there is no ambiguity as to the meaning of any words; for example, a word shot does not refer to the action of using a gun in the first sentence, and using the camera in the second sentence. Note Your Turn: Consider the following sentence and see if you can think of two fairly different interpretations: Fighting animals can be dangerous. Visiting relatives can be exhausting. Is the ambiguity of individual words to blame? Otherwise, what causes ambiguity? This chapter presents grammar and parsing, as a formal and computational method for investigating and modeling linguistic phenomena that we have discussed. As we will see, good patterns of sdurity anddurity in the order of words can be understood with respect to the structure and dependency of phrases. We can develop a formal model of this structure using grammar and parsers. As before, the main motivation is understanding natural language. How much more meaning of text can we access when we can recognize the linguistic structure it contains? After reading in the text, can the program understand it enough to be able to answer a simple question about what happened or who did what to whom? Also as before, we will develop a simple program to process annotated korpora and perform useful tasks. We set an example in 2. on how to use frequency information in bigram to generate text that seems very acceptable for a small sequence of words but quickly degenerates into nonsense. Here's a pair of other examples we made by calculating bigram over the text of a children's story, The Adventures of Buster Brown (: (4) a.He roared with me as he slipped onto his back b.The worst and awkward part of finding anyone who heard your light intuitively knew that this sequence was a word-salad, but you may find it hard to pin what's wrong with them. One of the benefits of learning grammar is that it provides a conceptual framework and vocabulary to spell out this intuition. Let's take a closer look at the sequence of the worst and awkward parts of the look. This looks like a coordinate structure, where two phrases join a co-ordinated conjunction such as and, but or or. Here's an informal (and simplified) statement on how language coordination works: Coordinate Structure: If v1 and v2 are both grammar category phrases X, then v1 and v2 are also category X phrases. In the first, two NPs (noun phrases) have joined forces to create np, while in the second, two (adjective phrases) have joined forces to create ap. (5) a.The end of the book is (NP the worst part and the best part) for me. b.On their land (AP slow and awkward looking). What we can't do is join NP and AP, AP, is why the worst part and awkward look is not programmatic. Before we can inaugurate these ideas, we need to understand the concept of constituent structure. The constituent structure is based on the observation that words are combined with other words to form units. Evidence that word sequences form such units is given by substitution - that is, the sequence of words in a well-formed sentence can be replaced by a shorter sequence without making the sentence unconscealed. To clarify this idea, consider the following sentence: (6)The little bear sees a fine fat trout in the river. The fact that we can substitute Him for the little Bear indicates that the last order is a unit. Instead, we cannot replace small bear saws in the same way. (7) a.He sees fine fat trout in the river. b.*He's a fine fat trout in the river. In 2.1, we systematically replace older sequences with shorter ones in a way that maintains grammar. Each sequence that forms the actual unit can be replaced by one word, and we end up with only two elements. Figure 2.1: Word Sequence Substitution: working from the top row, we can replace the order of certain words (e.g. brook) with individual words (for example); repeat this process we arrived at the sentence two grammar words. In 2.2, we've added grammar category labels to the words we saw in the previous image. Np, VP, and PP labels are each standing for noun phrases, verb phrases, and preposition phrases. Figure 2.2: Grammar Category Plus Word Sequence Substitution: This diagram reproduces 2.1 along with grammar categories corresponding to noun phrases (NP), verb phrases (VP), preposition phrases (PP), and nominals (Noms). If we now strip words apart from the top row, add node S, and in reverse that number, we end up with the standard phrase structure tree, shown in (8). Each node in this tree (including words) is called a constituent. Direct constituents of S are NP and VP. (8) As we will see in the next section, grammar determines how sentences can be divided into direct constituencies, and how this can be further divided until we reach the level of individual words. Note As we see in 1, sentences can have arbitrary lengths. As a result, tree phrase structures can have arbitrary depth. The multi-storey chunk parser we saw in 4 can only produce a bound depth structure, so the chunking method does not apply here. Let's start by looking at simple context-free grammar. By convention, the left side of the first production is the initial symbol of grammar, usually S, and all well-formed trees should have this symbol as their root label. In NLTK, free grammar defined in the nltk.grammar module. In 3.1 we define grammar and show you how to parse simple sentences received by grammar. grammar1 = nltk. CFG.fromstring(S -> NP VP VP -> V NP | V NP PP PP -> P NP V -> Saw | eating | walking NP -> John | | | Bob | Det N | Det N PP Det -> a | an | the | My N -> man | Dogs | cat | Telescope | park P -> in | on | by | with) >>> sent = Maria sees Bob.split() >>> rd_parser = nltk. RecursiveDescentParser(grammar1) >>> for trees in rd_parser.parse(sent): ... print(tree) (S (NP Mary) (VP (V saw) (NP Bob))) Example 3.1 (code_cfg1.py): Figure 3.1: Grammar-Free Simple Context Grammar in 3.1 contains productions involving various stick categories, as laid out in 3.1. Symbol Meaning Example S sentence man walking NP noun verb phrase VP dog see park PP prepositional phrase with telescope Det determines verb N noun dog V runs P preposition in Table 3.1: Syntactic Category production such as VP -> V NP | V NP PP has disjunction on the right side, indicated by | and stands for two production VP -> V NP and VP -> V NP PP. Figure 3.2: Recursive Descent Parser Demo: This tool allows you to watch the operation of the parser down recursively while growing the parse tree and match it to the input words. Note Your turn: Try developing your own simple grammar, using the recursive down parser app, nltk.app.rdparser(), displayed in 3.2. It appears already loaded with grammatical examples, but you can edit this however you like (using the Edit menu). Change the grammar, and the sentences to be parsed, and run the parser using the autostep button. If we parse the sentence Dog sees a man in the garden using the grammar shown in 3,1, we end up with two trees, similar to the one we see for (3b): (9) a. b. Since our grammar licenses two trees for this sentence, the sentence is said to be structurally ambiguous. The ambiguity in question is called the ambiguity of the preposition phrase attachment, as we saw earlier in this chapter. As you remember, it is ambiguity about attachments because pp in the park needs to be attached to one of the two places in the tree: either as a VP child or another as an NP child. When the PP is attached to the VP, the intended interpretation is that the viewing event takes place in the park. However, if pp is attached to np, then the man who is in the park, and the agent sees (the dog) may have been sitting on the balcony of the apartment overlooking the garden. If you're interested in experimenting with writing CFG, you'll find it useful to create and edit your grammar in a text file, says mygrammar.cfg. You can then load it into nltK and parse it as follows: >>> grammar1 = nltk.data.load('file:mygrammar.cfg') >>> sent = Mary saw Bob.split() >>> rd_parser = nltk. RecursiveDescentParser(grammar1) >>> for trees in rd_parser.parse(sent): ... print(tree) Make sure you put the .cfg sing on the file name, and that there are no spaces in the string 'file:mygrammar.cfg'. If print(tree) does not generate output, this may be because your sent sentence is not accepted by your grammar. In this case, contact the parser with the tracing set to rd_parser = nltk. RecursiveDescentParser(grammar1, trace=2). You can also check what production is currently in grammar with the p command at grammar1.productions(): print(p). When you write CFG for parsing in NLTK, you cannot merge grammar categories with lexical items on the right side of the same production. Thus, productions such as PP -> 'of' NP are prohibited. Additionally, you

are not allowed to place multi-word lexical items on the right side of production. So instead of writing NP -> 'New York', you should use something like NP -> 'New_York' instead. Grammar is said to be recursive if the category that occurs on the left side of the production also appears on the right side of the production, as illustrated in 3.3. Production Nom -> Adj Nom (where Nom is a nominal category) involves direct recursion in the Nom category, while indirect recursion on S arises from a combination of two productions, namely S -> NP VP and VP -> V S. grammar2 = nltk. CFG.fromstring(S -> NP VP NP -> Det Nom | PropN Nom -> Adj Nom | N VP -> V Adj | V NP | V S | V NP PP PP -> P NP PropN -> 'Buster' | 'Chat' | 'Joe' Det -> 'the' | 'a' N -> 'bear' | 'squirrel' | 'tree' | 'Fish' | 'log' Adj -> 'angry' | 'fear' | 'small' | 'high' V -> 'pursued' | 'Chainsaw' | 'he said' | 'Thinking' | 'was' | 'put' P -> 'on') Example 3.3 (code_cfg2.py): Figure 3.3: Recursive Context-Free Grammar To see how recursion arises from these grammars, consider the following trees. (10a) involves a nested nominal phrase, while (10b) contains a nested sentence. (10) a. b. We only illustrate the two levels of recursion here, but there is no upper limit on depth. You can experiment with deciphering sentences that involve deeper structures. Be aware that RecursiveDescentParser cannot handle the left recursive production of form X -> X Y; We'll be back to this in four minutes. Parsers process input sentences according to grammar production, and build one or more constituent structures that correspond to grammar. Grammar is the declarative specification of a well-formed indululant - actually just a string, not a program. Parser is a procedural interpretation of grammar. It searches through a tree space licensed by grammar to find one that has the necessary sentences along its periphery. Parsers allow grammar to be evaluated against a set of test sentences, helping grammar experts find errors in their grammar analysis. Parsers can serve as psycholinguistic processing models, helping to explain the difficulties humans have by processing the construction of certain sticks. Many natural language applications involve parsing at some point; for example, we expect natural language questions submitted to the question answering system to parsing as a first step. In this section we see two simple parsing algorithms, a top-down method called recursive parsing, and and a method called shift-reduce parsing. We're also looking at some of the more advanced algorithms, top-down methods with bottom-up filtering called left-angle parsing, and a dynamic programming technique called chart parsing. The simplest parser interprets grammar as a specification of how to break into high-level goals into several lower-level subgoals. The top level goal is to find the S.S. → VP NP allows the parser to replace this goal with two subgoals: find the NP, then find the VP. Each of these subgoals can be replacedlektly by sub-targets, using productions that have NP and VP on their left side. Finally, this expansion process leads to such subgoals: finding the word telescope. Such a subgoal can be directly compared to the input sequence, and succeed if the next word matches. If there is no match, the parser should back up and try a different alternative. Parser down recursively builds parse trees during the above process. With the initial destination (find S), the Root S node is created. When the above process recursively expands its purpose using grammar production, the parse tree is extended downwards (hence the name of the recursive descendant). We can see this in action using the graphical demonstration nltk.app.rdparser(). The six stages of execution of this parser are shown in 4.1. Figure 4.1: Six Stages of Recursive Descent Parser: the parser begins with a tree consisting of node S; at each stage consult the grammar to find production that can be used to enlarge the tree; when lexical production is encountered, the words are compared to input; once the complete parse has been found, backtrack parser to search for more parses. During this process, parsers are often forced to choose between several possible productions. For example, in going from step 3 to step 4, he tries to find production with N on the left side. The first is N → man. When this doesn't work it backs off, and tries another N production in a way, right down to N → dog, which matches the next word in the input sentence. Much later, as shown in step 5, he found the complete parse. This is the tree that covers the whole sentence, with no dangling edges. Once the parse is found, we can get the parser to look for additional parses. Again it will retreat and explore other production options if one of them produces parse. NLTK provides a recursive down parser: >>> rd_parser = nltk. RecursiveDescentParser(grammar1) >>> sent = 'Mary saw a dog'.split() >>> for the tree in rd_parser.parse(sent): ... print(tree) (S (NP Mary) (VP (V saw) (NP (Det a) (N dog)))) Note RecursiveDescentParser() takes an optional parameter trace. If the trace is greater than zero, the parser will report the necessary steps when parsing the text. Recursive offspring have three main drawbacks. First, left recursive production such as NP -> NP PP sends it into an infinite loop. Second, the parser throws a lot of consider words and structures that do not correspond to the input sentence. Third, the backtracking process can discard parsed constituencies that need to be rebuilt later. For example, backtracking via VP -> V NP will discard the subtree created for np. If the parser is then continued with VP -> V NP PP, then the NP subtree should be created again. Recursive down parsing is a kind of top-down parsing. The top-down parser uses grammar to predict what the input will be, before checking the input! However, since input is available for parsers during this time, it would make more sense to consider the input sentence from the beginning. This approach is called bottom-up parsing, and we'll look at the example in the next section. A simple bottom-up parser type is a shift-reduce parser. Equal to all bottom-up parsers, the shift-reduce parser tries to find the order of words and phrases that correspond to the right side of grammar production, and replaces them with the left side, until the entire sentence is reduced to S. The shift-reduce parser repeatedly pushes the next input word into the stack (4.1); This is a shift operation. If the top n item in the stack matches item n on the right side of some production, then everything appears from the stack, and the item on the left side of the production is pushed on the stack. Replacing the top n items with this one item is a subsedi operation. This operation can only be applied to the top of the stack; lower items in the stack must be done before the item is later pushed into the stack. The parser finishes when all inputs are used and there is only one item left in the stack, a parse tree with node S as its root. The shift-reduce parser builds the parse tree during the above process. Each time n an item appears from the stack it merges it into a partial parse tree, and pushes this back on the stack. We can see the shift-reduce parsing algorithm in the action using the graphical demonstration nltk.app.srparser(). The six-stage execution of this parser is shown in 4.2. Figure 4.2: Six Stage Shift-Reduce Parser: the parser starts by sliding the first input word into its stack; after the top items in the stack match the right side of grammar production, they can be replaced with the left side of that production; the parser succeeds after all inputs are consumed and one item S remains in the stack. NLTK provides ShiftReduceParser(), a simple implementation of the shift-reduce parser. This parser does not implement any backtracking, so it is not guaranteed to find parse for text, even if it exists. In addition, it will only find at most one parse, even if there are more parses. We can provide optional trace parameters that control how verbosely parsers report the necessary steps when parsing text: >>> sr_parser=nltk. ShiftReduceParser(grammar1) >>> sent = 'Mary saw a dog'.split() >>> to sr_parser.parse(sent): ... print(tree) (S (NP Mary) (VP (V saw) (NP (Det a) (N dog)))) Note Your Turn: Run the parser above in tracing mode to view the shift order and reduce the operation, using sr_parse =nltk. ShiftReduceParser(grammar1, trace=2) The shift-reduce parser can reach a dead end and fail to find any parse, even if the input sentence is properly formed according to grammar. When this occurs, there is no input left, and the stack contains items that cannot be reduced to S. The problem arises because there are previously made options that cannot be undone by the parser (although graphical demonstration users can deselect them). There are two types of options that will be made by the parser: (a) a reduction to be made when more than one is possible (b) whether to shift or reduce when one action is possible. The shift-reduce parser can be expanded to implement policies to resolve such conflicts. For example, it can resolve shift-reduce conflicts by shifting only when no reduction is possible, and can address mitigating conflicts by supporting reduction operations that remove the most items from the stack. (Generalization of shift-reduce parser, lookahead LR parser, commonly used in programming language compilation.) The advantage of shift-reduce parsers above the recursive down parser is that they only build structures that match the words in the input. Furthermore, they only build each sub-structure once, for example NP(Det(the), N(man)) is only built and pushed into the stack once, regardless of whether it will later be used by vp reduction -> V NP PP or NP PP NP reduction ->. One of the problems with the recursive descent parser is that it gets into an infinite loop when facing left recursive production. This is because it applies grammar production blindly, regardless of the actual input sentence. The left-angle parser is a hybrid between the bottom and top approaches we've seen. Grammar 1 allows us to produce the following parse from John seeing Mary: (11) Remember that grammar (defined in 3.3) has the following productions to expand NP: (12) c.NP -> John | Maria | Bob Suppose we ask you to first look at the tree (11), and then decide which NP production you want to apply the recursive descent parser first – obviously, (12c) is the right choice! How do you know that there is no point applying (12a) or (12b) instead? Because none of these productions will get the sequence whose first word is John. That is, we can easily say that in the successful parse of John saw Mary, the parser had to expand the NP in such a way that the NP obtained the order of John α. In general, we say category B is the left corner of the tree that is rooted in A if A ⇒ *B α. (13) The left corner parser is an up-down parser with bottom-top filtering. Unlike the usual recursive descent parsers, he is not caught up in the production of left recursive. Before starting his work, his work, The parser performs a context-free grammar preprocess to compile a table in which each row contains two cells, the first holds a non-terminal, and the second holds the collection of possible non-terminal left corners of it. 4.1 describes this for grammar from grammar2. Category Left Corner (pre-terminal) S NP NP Det, PropN VP V PP P Table 4.1: Left Corner in grammar2 Whenever production is considered by the parser, it checks that the next input word is compatible with at least one of the pre-terminal categories in the left corner table. The simple parser discussed above suffers from limitations in completeness and efficiency. To fix this, we will apply dynamic programming algorithm design techniques to parsing problems. As we saw in 4.7, dynamic programming stores intermediate results and reuse them if appropriate, achieving significant efficiency gains. This technique can be applied to syntactic parsing, allowing us to store partial solutions for parsing tasks and then look for them as necessary to efficiently arrive at the complete solution. This parsing approach is known as chart parsing. We introduce the main ideas in this section; see the online materials available for this chapter for more implementation details. Dynamic programming allows us to build PP in my pyjamas only once. The first time we build it we store it in a table, then we look for it when we need to use it as a subconstitution of a higher NP or VP object. This table is known as a well-formed substring table, or WFST for short. (The term substring refers to the sequence of words crossed in a sentence.) We will show you how to build a bottom-up wfst so as to systematically record what synthetic constituents have found. Let's set our input to be a sentence in (2). The numerically defined WFST range is reminiscent of python slice notation (3.2). Another way to think about the data structure is shown in 4.3, the data structure known as the chart. Figure 4.3: Chart Data Structure: words are the edge labels of linear chart structures. In WFST, we record the position of the words by filling the cells in the triangular matrix: the vertical axis will indicate the initial position of the substring, while the horizontal axis will indicate the final position (thus the shot will appear in the cell with coordinates (1, 2)). To simplify this presentation, we'll assume each word has a unique lexical category, and we'll keep this (not a word) in the matrix. So cells (1, 2) will contain entry V. More generally, if our input string is a0a1... and our grammar contains the production of form A → ai, then we add A to the cell (i, 'i'+1). System Message: WARNING/2 (ch08.rst2, line 900); backlink Inline text or start-string reference phrase without end-string. for each word in the text, we can search in our grammar what category it belongs to. >>> = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'my', >>> groucho_grammar.productions(rhs=text[1]) [V -> 'shot'] For our WFST, we created a matrix of × (n-1) (n-1) as a list of lists in Python, and initialized them with the lexical category of each token, in the init_wfst() function at 4.4. We also define the function of the display() utility to print enough WFST for us. As expected, there is a V in the cell (1, 2). (And init_wfst- grammar): numtokens = len(token) wfst = [[None for i in range(numtokens+1)] for j in range (numtokens+1)] for i in range (numtokens): production = grammar.productions(rhs=tokens[i]) wfst[i][i+1] = productions[0].lhs() return wfst def complete_wfst(, . token, grammar, trace=False): index = dict((p.rhs(), p.lhs()) for p in grammar.productions()) numtokens = len(token) for range(2) , numtokens+1): for start in range(numtokens+1-span): end = start + span for mid in range(start+1, end): nt1, nt2 = wfst[start][mid], wfst[mid][end] if nt1 and nt2 and (nt1,nt2) in index: wfst[start][end] = index[(nt1,nt2)] if trace: print([%s] %3s [%s] %3s [%s] ==> [%s] %3s [%s] % mid, nt2, end, start, index[(nt1 ,nt2)] , end)) returns wfst def display(wfst, token): print('WFST' + ''.join((%-4d % i) for i in range (1, len(wfst))) for i in range (len(wfst)-1): print(%d %i, end=) for j in range(1, len(wfst)): print(%-4s % (wfst[i][j] or '.'), end=) print() >>> token = I shot the elephant in my pyjamas.split() >>> wfst0 = init_wfst(token , groucho_grammar) >>> display(wfst0 , token) WFST 1 2 3 4 5 6 7 0 NP . . . 1 . V..... 2 . . Det.... 3 . . . N... 4 . . . . Q.. 5 . . . . . Det. 6 . . . . . . N >>> wfst1 = complete_wfst(wfst0, token, groucho_grammar) >>> display(wfst1, token) WFST 1 2 3 4 5 6 7 0 NP . . S.. S 1. V. Vp. . VP 2. . Det NP . . . 3 . . . N... 4 . . . . P. PP 5 . . . . Det NP 6 . . . . N Example 4.4 (code_wfst.py): Figure 4.4: Acceptor Using a Well Formed Substring Table Returns to our tabular representation, given that we have Det in the cell (2, 3) for the word an, and N in the cell (3, 4) for the word elephant, what should we put in the cell (2, 4) for the elephant? We need to find the production of form A → Det N. Grammar consultation, we know that we can insert NP in cells (2, 4). More generally, we can include A in (i, j) if there is a production of A → B C, and we find nonterminal B in (i, k) and C in (k, j). Programs in 4.4 use this rule to complete WFST. By setting the trace to True when calling the complete_wfst() function, we see a tracing output indicating WFST is being built: >>> wfst1 = complete_wfst(wfst0, token, groucho_grammar, trace=True) [2] Det [3] N [4] ==> [2] NP [4] [5] Det [6] N [7] ==> [5] NP [7] [1] V [2] NP [4] ==> [1] VP [4] P [5] NP [7] ==> [4] PP [4] PP [4] 7] [0] NP [1] VP [4] ==> [0] S [4] [1] VP [4] PP [7] ==> [1] VP[7] [0] NP [1] VP [7] ==> [0] S [7] For example, this says that since we found Det in wfst[2][3] and N in wfst[3][4], we can add NP to wfst[2][4]. Note To help us easily production by their right side, we create an index for grammar. This is an example of a space-time trade-off: we do a reverse search on grammar, instead of having to check the entire production list whenever we want to look up through the right side. Figure 4.5: Chart Data Structure: non-terminal is represented as an additional edge in the chart. We concluded that there was a parse for the entire input string after we built the S node in the cell (0, 7), indicating that we had found a sentence that included the entire input. The final state of WFST is described in 4.5. Note that we haven't used the built-in parsing function here. We've implemented a complete primitive chart parser from the bottom up! WFST has some drawbacks. First, as you can see, WFST is not a parse tree, so the technique actually speaks to acknowledging that sentences are recognized by grammar, rather than deciphering them. Second, it requires any production of non-lexical grammar to be binary. While it is possible to convert arbitrary CFG into this form, we prefer to use such a unconditional approach. Third, as a potentially wasteful bottom-up approach, it is able to propose constituencies in locations that will not be licensed by grammars. Finally, WFST does not represent structural ambiguity in sentences (i.e. two verb phrase readings). VP in cells (1, 7) is actually inserted twice, once for NP V readings, and once for VP PP readings. This is a different hypothesis, and the second overrides the first (as it happens this doesn't matter because the left side is the same.) Chart parsers use a much richer data structure and some interesting algorithms to solve this problem (see the Further Reading section at the end of this chapter for details). Note Your turn: Try the interactive chart parser app nltk.app.chartparser(). Grammar of phrase structures relates to how words and word sequences are combined to form constituencies. Different and complementary approaches, grammar dependencies, focus on how words relate to other words. Dependency is a binary asymmetric relationship that holds between the head and its dependents. The head of a sentence is usually taken to be a tense verb, and every other word depends on the head of the sentence, or connected to it through a path of dependency. A dependency representation is a labeled redirected graph, where the node is a lexical item and the labeled arc represents a dependency relationship from head to dependent. 5.1 illustrates the dependency graph, where the arrow points from the head to the dependent. Figure 5.1: Structure dependencies: arrows pointing from the head to their dependents; the label indicates a dependent grammar function as a subject, object or modifier. The arc in 5.1 is labeled with a grammar function that holds between the dependent and the head. I am an SBJ (subject) bid (which (which is head of the entire sentence), and inside is NMOD (elephant noun modifier). Unlike grammar structure phrases, therefore, grammar dependencies can be used to directly express grammar functions as a type of dependency. Here is one way of encoding grammar dependencies in NLTK – note that it only captures naked dependency information without specifying the type of dependency: >>> groucho_dep_grammar=nltk. DependenciesGrammar.fromstring(... 'shot' -> 'I' | 'Elephant' | 'in' ... 'elephant' -> 'an' | 'in' ... 'in' -> 'pajamas' ... 'pajamas' -> 'my'...) >>> Grammar Dependency Print (groucho_dep_grammar) with 7 production 'shot' -> 'I' 'shot' -> 'elephant' 'shot' -> 'in' 'elephant' -> 'an' 'elephant' -> 'in' -> 'pyjamas' -> 'my' Dependency chart is efficacious if, when all words are written in linear order, the edges can be drawn above the words without crossing. This is equivalent to saying that a word and all its descendants (dependents and dependents, etc.) form a sequence of words that are crossed in sentences. 5.1 berk projecting, and we can parse many sentences in English using projective dependencies parser. The next example shows how groucho_dep_grammar provide an alternative approach to capturing the attachment ambiguity we examined earlier with the grammatical structure of the phrase. >>> = nltk. ProjectiveDependencyParser(groucho_dep_grammar) >>> sent = 'I shot an elephant in my pyjamas'.split() >>> tree = pdp.parse(sent) >>> for tree in tree: ... print(tree) (shot I (elephant an (in (pyjamas my))) (shot I (elephant an) (in (my pajamas))) This caged dependency structure can also be displayed as a tree, where dependents are shown as children of their heads. (14) In languages with more flexible word sequences than English, non-projective dependencies are more frequent. Various criteria have been proposed to decide what is head H and what is dependent D in construction C. Some of the most important are as follows: H determines distribution class C; or alternatively, the external syntactic properties of C caused by H. H determine the mandatory C. H semantic type while D may be optional. H selects D and determines whether it is mandatory or optional. Morphological form D is determined by H (e.g. treaty or case governance). When we say in the grammar structure of the phrase that the direct constituents of the PP are P and NP, we implicitly appeal to the differences head/dependent. A preposition phrase is a phrase whose head is a preposition; in addition, NP is a dependent P. The same difference leads to other types of phrases that we have discussed. An important point to note here is that although the grammar of the phrase structure seems very of grammar dependency, they implicitly embody the recognition of dependency relationships. While CFG is not intended to directly capture dependency, newer linguistic frameworks are increasingly formalism that incorporates aspects of both approaches. Let's take a closer look at their verbs and dependents. Grammar in 3.3 correctly generates instances such as (15d). (15) a.The squirrel is terrified. b.Chatterer saw the bear. c.Chatterer thinks Buster is angry. D.Joe put fish in the wood. This possibility corresponds to the following productions: That is, it can occur with the following Adj, seeing can occur with the following NP, thought can occur with the following S and put can occur with the following NP and PP. Dependents Adj, NP, PP and S are often called verb complements respectively and there are strong constraints on what verbs can occur with what complements. Unlike (15d), the order of words in (16d) is not formed: (16) a.*The squirrel is an angry Buster. b.*Chatterer sees fear. C.*Chatterer thinks the bear. Note With little imagination, it is possible to create a context in which unusual combinations of verbs and appendages can be interpreted. However, we assume that the above examples should be interpreted in a neutral context. In the tradition of grammar dependency, verbs in 5.1 are said to have different valencies. Valence restrictions apply not only to verbs, but also to other head classes. In a framework based on grammar phrasing structure, various techniques have been proposed to exclude ungrammatical instances in (16d). In CFG, we need several ways to limit the production of grammar that expands VP so that verbs only happen along with the correct complement. We can do this by dividing the verb classes into subcategories, each of which is associated with a different set of appendages. For example, transitive verbs such as pursued and viewed require the following NP object complement; that is, they are subcategorized for np direct objects. If we introduce a new category label for transitive verbs, i.e. TV (for Transitive Verb), then we can use it in the following productions: VP -> TV NP TV -> 'pursued' | 'look' Now *Joe thinks the bear was excluded because we haven't registered the thought as TV, but Chatterer sees the bear is still allowed. 5.2 provides more label examples for verb subcategories. Symbol Meaning Example IV of an intransitive verb barking TV transitive verb sees a datV man dative verb give a dog to a man SV sentential verb says that dog barking Table 5.2: Verb Subcategories Valency is a property of lexical goods, and we will discuss it further at 9.. Complements often contrast with modifiers (or adjuncts), although both are types of dependencies. Preposition phrases, adjectives, and adjectives usually serve as modifiers. Unlike complements, modifiers are optional, can often be iterated, and are not by the head in the same way as the appendage. For example, an averb can actually be added as a modifer to all sentences in (17d): (17) a.Squirrels are really scared. b.Chatterer actually saw the bear. c.Chatterer really thinks Buster is mad. d.Joe d.Joe Put the fish in the wood. The structural ambiguity of pp attachments, which we have illustrated in the structure of phrases and grammatical dependencies, corresponds semantically with ambiguity in the scope of the modifier. So far, we've only considered toy grammar, a small grammar that describes key aspects of parsing. But there is a clear question as to whether the approach can be disputed to mask the large korpora of natural language. How difficult is it to build such a set of productions by hand? In general, the answer is: very difficult. Even if we allow ourselves to use a variety of formal devices that provide a much more concise representation of grammar production, it is still very difficult to control the complex interaction between many productions needed to cover the main construction of a language. In other words, it is difficult to modulate grammar so that one portion can be developed independently of another. This in turn means that it is difficult to distribute grammar writing tasks across a team of language experts. Another difficulty is that as grammars evolve to cover a wider and wider range of constructions, there is an appropriate increase in the number of analyses received for a single sentence. In other words, ambiguity increases with coverage. Despite these problems, several major collaborative projects have achieved exciting and impressive results in developing rules-based grammar for multiple languages. Examples include the Lexical Functional Grammar (LFG) Pargram project, lingo Matrix's Head-Driven Phrase Structure Grammar (HPSG) framework, and the Lexicalized Tree Adjoining Grammar XTAG Project. Parsing builds a tree over a sentence, according to the grammatical structure of the phrase. Now, all of the examples we provide above only involve toy grammar containing multiple productions. What happens if we try to improve this approach to deal with realistic language korpora? In this section we will look at how to access treebank, and see the challenges of developing grammar broad coverage. The corpus module defines a treebank corpus reader, which contains a sample of 10% of the Penn Treebank corpus. >>> from nltk.corpus import treebank >>> t = treebank.parsed_sents('wsj_0001.mrg')[0] >>> print(t) (S (NP-SBJ (NP (NNP Pierre) (NNP Vinken)) (, ,) (ADJP (NP (CD 61) (NNS year)) (JJ old)) (, ,)) (VP (MD will) (VP (VB join) (NP (DT the) (NN board)) (PP-CLR (IN as) (NP (DT a) (JJ nonexecutive) (NN director)))) (NP-TMP (NNP Nov.) (CD 29))))) (. .)) We can use this data to help develop grammar. For example, programs in 6.1 use simple filters to find verbs that take sentential appendages. Assuming we already have a production form VP -> Vs S, this information allows us to identify specific verbs that will in the expansion of vs. def filter (tree): child_nodes = [child.label() for the child in the tree if isinstance (child, nltk. Tree)] return (tree.label() == 'VP') and ('S' in child_nodes) child_nodes) from nltk.corpus import treebank >>> [subtree for trees in treebank.parsed_sents() ... for subtrees in tree.subtrees(filter)] [Tree('VP', [Tree('VBN', ['named']), Tree('S', [Tree('NP-SBJ', ...]), ...]), ...] Example 6.1 (code_sentential_complement.py): Figure 6.1: Searching treebank to find sentential completes corpus attachment preposition phrase, nltk.corpus.ppattach is another source of information about the valence of a particular verb. Here we describe the techniques for mining this corpus. It finds a pair of preposition phrases in which prepositions and nouns are corrected, but where verb selection determines whether the preposition phrase is attached to the VP or to np. >>> from the defaultdict import collection >>> = nltk.corpus.ppattach.attachments('training') >>> table = defaultdict(lambda: defaultdict(set)) >>> for entries in the entry: ... key = entry.noun1 + '-' + entry.prep + '-' + entry.noun2 ... table[key][entry.attachment].add(entry.verb)... >>> for keys sorted(table): ... if len(table[key]) > 1: ... print(key, 'N:', sorted(table[key]['N']), 'V:', sorted(table[key]['V'])) Between the output lines of this program we found an offer from group N: ['rejected'] V: ['accepted'], which indicates that receiving a separate PP complement is attached to the VP, while being rejected is not. As before, we can use this information to help build grammar. NLTK corpus collection includes data from cross-framework evaluation joint tasks and Cross Domain Parser PE08. A larger collection of grammar has been set up for the purpose of comparing different parsers, which can be obtained by downloading the large_grammars package (e.g. python -m nltk.downloader large_grammars). The NLTK corpus collection also includes samples from Sinica Treebank Corpus, which consists of 10,000 parsed sentences taken from the Academia Sinica Balanced Corpus of modern Chinese. Let's load and display one of the trees in this corpus. >>> nltk.corpus.sinica_treebank.parsed_sents()[3450].draw() Unfortunately, as grammar coverage increases and the length of input sentences grows, the number of parse trees grows rapidly. In fact, it grows at an astronomical rate. Let's explore this issue with the help of a simple example. The word fish is a noun and a verb. We can make fish sentences, which means fish likes to fish for other fish. (Try this with the police if you prefer something more reasonable.) Here's the grammar for fish sentences. >>> grammar = nltk. CFG.fromstring(... S -> NP V NP ... NP -> NP Sbar ... Sbar -> NP V ... NP -> 'fish' ... V -> 'fish'...) Now we can try to parse the longer sentence, fish fish fish, which among other things, means 'the fish that other fish are accustomed to fishing the fish itself'. We use the NLTK chart parser, which is mentioned in this chapter. This sentence has two readings. >>> = [fish] * 5 >>> = nltk. BaganParser(grammar) >>> to at cp.parse(token): ... print(tree) (S (NP fish) (fish V) (NP (NP fish) (NP fish) (fish V)))) (S (NP (NP fish) (Sbar fish (fish NP) (fish V))) (Fish V) (NP fish)) Over time this sentence rises (3, 5, 7, ...) we get the following numbers from the tree parse: 1; 2; 5; 14; 42; 132; 429; 1,430; 4,862; 16,796; 58,786; 208,012; ... (This is the Catalan number, which we saw in practice at 4). The latter is for a long sentence of 23, the average length of the sentence in the WSJ section of Penn Treebank. For 50 long sentences there will be more than 1012 parses, and this is only half the length of the piglet sentence (1), which young children process easily. There is no practical NLP system that can build millions of trees for sentences and choose the appropriate one in context. It's clear that humans don't do this either! Note that the problem is not with our preferred example. (Church & Patil, 1982) shows that the ambiguity of the PP's attachment in such sentences (18) also grows in proportion to the number of Catalans. (18) Place the block in a box on the table. So much for structural ambiguity; what about lexical ambiguity? As soon as we tried to establish grammar coverage broadly, we were forced to make a very ambiguous lexical entry for their speech section. In toy grammar, a is just an examiner, a dog is just a noun, and runs only verbs. However, in broad-scope grammar, a is also a noun (e.g. part a), dog is also a verb (meaning to follow closely), and running is also a noun (e.g. ski run). In fact, all words can be referred to by name: for example the verb 'eat' is spelled with three letters; in speech we do not need to supply quotation marks. In addition, it is possible to verb most nouns. Thus the parser for grammar broad coverage will be overwhelmed with ambiguity. Even complete nonsense will often have readings, for example a is I. As has been shown by (Klavans & Resnik, 1996), this is not a salad word but a grammatical noun phrase, where it is a noun meaning one hundred hectares (or 100 square meters), and a and I nouns that point to coordinates, as shown in 6.2. Figure 6.2: A is I: schematic images of 27 paddocks, each of which is in size, and each identified using coordinates; the top left cell is column I (after Abney). Although this phrase is unlikely, it is still grammatical and a broad-scope parser should be able to build a parse tree for it. Similarly, a seemingly unambiguous sentence, as John saw Mary, turns out to have other readings we would not anticipate (as Abney explains). This ambiguity is unavoidable, and leads to horrendous inefficiencies in deciphering seemingly harmless sentences. Solutions to this problem are provided probabilistic parsing, which allows us to rank parses of ambiguous sentences on the basis of evidence from the corpora. As we've just seen, dealing with ambiguity is a major challenge in developing broad coverage Chart parsers improve the computational efficiency of multiple parses of the same sentence, but they are still overwhelmed by the number of possible parses. Weighted grammar and probabilistic parsing algorithms have provided an effective solution to this problem. Before looking at this, we need to understand why grammatical ideas can gradient. Remembering verb giving. This verb requires a direct object (the given thing) and an indirect object (recipient). This complement can be given in any order, as illustrated in (19). In the prepositional dative form in (19a), the object immediately appears first, followed by a preposition phrase that contains an indirect object. (19) a.Kim gives the dog a bone b.Kim gives the dog a bone in the form of a double object in (19b), the object does not appear immediately the first time, followed by the direct object. In the above cases, the order is acceptable. However, if an indirect object is a pronoun, there is a strong preference for the construction of a double object: (20) a.Kim gave me heebie-jeebies (*prepositional dative) b.Kim gave me heebie-jeebies (double objects) Using a Sample Penn Treebank, we can examine all the examples of prepositional dative constructions and double objects that involve giving, as shown in 6.3. def give(t): return t.label() == 'VP' and len(t) > 2 and t[1].label() == 'NP'\ and (t[2].label() == 'PP-DTV' or t[2].label() == 'NP'''.)\ and ('give' in t[0].leaves() or 'give' in t[0].leaves()) def sent(t): return '.join(token for token in t.leaves() if token[0] not in '*-0') def print_node(t, width): output = %s %s: %s / %s: %s %\ (sent(t[0]), t[1].label(), sent(t[1]), t[2].label(), sent(t[2])) if len(output) width >: output = output[:width] + ... print(output) >>> for trees in nltk.corpus.treebank.parsed_sents(): ... for t in tree.subtrees(give): ... print_node (t, 72) give NP: chef / NP: festive applause gives NP: advertisers / NP: discounts to maintain or increase sp ads ... give NP: it / PP-DTV: to the politicians who give NP: them / NP: similar help give NP: they / NP: give NP: only the question of French history / PP-DTV: to students in Europe ... give NP: federal judge / NP: the hike gives NP: consumer / NP: spoon straight on the u.S. waste crisis gives NP: Mitsui / NP: access to high-tech medical products give NP: Mitsubishi / NP: windows in the U.S. glass industry give NP: a lot of thought / PP-DTV: to the degree he received, or to ... give NP: Your Foster Child/NP Savings Institution: the gift of hope and free... give NP: market operator / NP: authority to suspend trading in futu ... give NP: quick approval / PP-DTV: for $3.18 billion in additional appr ... give NP: Department of Transportation / NP: up to 50 days to review... give NP: president / NP: such power gives NP: / NP: heebie-jeebies give NP: holder / NP: the right, but not the obligation , to buy cal ... give NP: Mr. Thomas / NP: only the rank '' is qualified, qualified, than ''... give NP: president / NP: veto power line item Example 6.3 (code_give.py): Figure 6.3: Use of Give and Give in the Penn Treebank sample We can observe a strong tendency for the shortest appendages to appear first. However, this does not take into account forms such as giving NP: federal judge/NP: a raise, in which animation can play a role. In fact it turns out that there are a large number of contributing factors, such as those surveyed by (Bresnan & Hay, 2006). These preferences can be represented in weighted grammar. Probabilistic context-free grammar (or PCFG) is a context-free grammar that associates probability with each production. This results in the same set of parses for the text performed in the appropriate context-free grammar, and sets the probability for each parse. The probability of parse produced by PCFG is simply a product of the production probability used to produce it. The simplest way to define PCFG is to load it from a specially formatted string consisting of a weighted production sequence, in which weights appear in parentheses, as shown in 6.4. grammar = nltk. PCFG.fromstring(S -> NP VP [1.0] VP -> TV NP [0,4] VP -> IV [0,3] VP -> DATV NP NP [0,3] TV -> 'saw' [1.0] IV -> 'ate' [1.0] DatV -> 'give' [1,0] NP -> 'telescope' [0,8] NP -> 'Jack' [0,2]) >>> grammar Grammar with 9 productions (start state = S) S -> NP VP [21.0] VP -> TV NP [0,4] VP -> IV [0,3] VP -> DatV NP NP [0,3] TV -> 'see' [1,0] IV -> 'eat' [1.0] DatV -> 'give' [1,0] DatV -> 'give' [1.0] DatV -> 'give'1.0] NP -> 'telescope' [0,8] NP -> 'Jack' [0,2] Example 6.4 (code_pcfg1.py): Figure 6.4: Defining Probabilistic Context-Free Grammar (PCFG) Sometimes convenient to incorporate multiple productions into a single line E.g. VP -> NP TV [0.4] | IV [0.3] | NP DatV NP [0.3]. To ensure that trees produced by grammar form a probability distribution, PCFG grammar imposes a limit that all productions with a particular left side must have a probability that sums them into one. Grammar in 6.4 complies with this limitation: for S, there is only one production, with a probability of 1.0; for VP, 0.4+0.3+0.3=1.0; and for NP, 0.8+0.2=1.0. The parse tree returned by parse() includes probability: >>> viterbi_parser = nltk. ViterbiParser (grammar) >>> for trees in viterbi_parser.parse(['Jack', 'saw', 'telescope']): ... print(tree) (S (NP Jack) (VP (TV saw) (NP telescope))) (p=0.064) Now the parse tree is given probability, it doesn't matter anymore that there may be a large number of possible parses for a particular sentence. A parser will be responsible for finding the most likely parses. Additional material for this chapter posted on links to resources that are freely available on the web. For more examples of parsing with NLTK, please see Parsing HOWTO in . There are many introductory books on syntax. (O'Grady et al, 2004) is a general introduction to linguistics, while (Radford, 1988) provides a gentle introduction to grammar, and can be recommended for the scope of transformational approaches to the construction of unlimited dependencies. The most widely used term in linguistics for formal grammar is generative grammar, although it has nothing to do with generations (Chomsky, 1965). Syntax X-bar framework is caused by (Jacobs & Rosenbaum, 1970), and explored with greater length (Jackendoff, 1977) (The prima we use replaces Chomsky's more demanding horizontal rods.) (Burton-Roberts, 1997) is a practically oriented textbook on how to analyze constituencies in English, with extensive copies and exercises. (Huddleston & Pullum, 2002) provides the latest and comprehensive analysis of the stick phenomenon in English. Chapter 12 of (Jurafsky & Martin, 2008) covers formal English grammar; Sections 13.1-3 include algorithms and simple parsing techniques to deal with ambiguity; Chapter 14 includes statistical parsing; Chapter 16 includes chomsky hierarchies and the formal complexity of natural language. (Levin, 1993) has categorized English verbs into fine graying classes, according to their syntactic properties. There are several ongoing efforts to build large-scale rule-based grammar, such as the LFG Pargram project, the HPSG LinGO Matrix framework and the XTAG Project xtag/. ☼ Can you come up with a grammatical sentence that may never have been uttered before? (Take turns with a partner.) What does this say about human language? ☼ Recall Strunk and White's ban on early-sentence but used to mean though. Perform a web search but use it at the beginning of the sentence. How much is this construction used? ☼ Consider the sentence kim arrived or Dana left and everyone cheered. Write a parent-embedded form to show the relative scope and and or. Produce a tree structure that corresponds to both of these interpretations. ☼ Tree Class implements a variety of other useful methods. See the Tree help documentation for more details: Importing the Tree class and typing Help(Tree). ☼ In this exercise you will manually build some parse trees. Write code to generate two trees, one for each old male and female phrase reading Encode one of the trees presented in this chapter as labeled bracketing and use nltk. Tree() to check that it is well formed. Now use draw() to display the tree. As in (a) above, drawing a tree for the woman saw a man last Thursday. ☼ Write recursive function to traverse the tree and return the depth of the tree, so that a tree with one node will have a depth of zero. (Hint: the depth of the subtree is the maximum depth of its children, one.) ☼ Analyzes A.A. Milne's sentence about Piglet, underlining all the sentences it contains and then replacing them with S (e.g. the first sentence becomes S when:lx'S). Draw a tree structure for this compressed sentence. What is it the main syntactic construction used to build such a long sentence? ☼ In the demo parser down recursively, experiment with changing the sentence to be parsed by selecting Edit Text in the Edit menu. ☼ Can grammar in grammar1 be used to describe sentences longer than 20 words? ☼ Use the graphical chart-parser interface to experiment with different rule calling strategies. It comes with your own strategy that you can execute manually using a graphical interface. Describe the steps, and report any efficiency improvements they make (e.g. in terms of the resulting chart size). Does this fix depend on the grammar structure? What do you think about prospects for significant performance improvements from smarter rule calling strategies? ☼ With pen and paper, manually track the recursive down parser execution and shift-reduce parser, for the CFG you've seen, or one of your own designs. ☼ We've seen that chart sorting adds but never removes edges from the chart. Why? ☼ Pertimbangkan urutan kata-kata: Kerbau kerbau kerbau kerbau kerbau kerbau kerbau kerbau. This is a grammatically correct sentence, as described . Consider the tree diagram presented on this Wikipedia page, and write the appropriate grammar. Normalize the case to lowercase, to simulate the problem the listener is doing when hearing this sentence. Can you find other parses for this sentence? How does the number of parse trees grow as sentences get longer? (Further examples of this sentence can be found in the ◑ you can modify grammar in a recursive derivative parsing demo by selecting Edit Grammar in the Edit menu. Change the production of the second expansion, namely NP -> Det N PP, to NP -> NP PP. Using the Step button, try building a parse tree. What's going on? ◑ Expand grammar in grammar2 with production that expands prepositions as intransitive, transitive and in need of PP complements. Based on this production, use the previous practice method to draw a tree for Lee's sentence of escaping home. ◑ Select multiple common verbs and complete the following tasks: Write the program to find the verb in the Corpus Attachment Preposition Phrase nltk.corpus.ppattach. Find cases where the same verb indicates two different attachments, but where the first noun, or second noun, or preposition, remains unchanged (as we saw in the discussion of stick ambiguity in 2). Designed cfg grammar production to cover some of these cases. ◑ Write a program to compare the efficiency of top-down chart parsers with the parser down recursively (4). Use the same grammar and input sentences for both. Compare their performance using the timeit module (see 4.7 for an example of how to do this). ◑ Compare the performance of the top-bottom, bottom-up, and left-hand side of the parser using and three grammar test sentences. Use timeit to record the amount of time each parser takes on the same sentence. Write a function that runs all three parsers on all three sentences, and print the grid 3-by-3 times, as well as the total rows and columns. Discuss your findings. ◑ Read the sentence of the garden path. How can the computational work of a parser relate to the difficulties humans have by processing these sentences? ◑ To compare multiple trees in one window, we can use the draw_trees() method. Specify multiple trees and try: >>> from nltk.draw.tree import draw_trees >>> draw_trees(tree1, tree2, tree3) ◑ Using tree position, subject list of the first 100 sentences in treebank Penn; to make the result easier to see, limit the extracted subject to subtrees that are 2 in height. ◑ check the Corpus Attachment Preposition Phrase and try suggesting several factors that affect the PP attachment. ◑ In this section we claim that there is linguistic regularity that cannot be explained only in terms of n-grams. Consider the following sentence, especially the position of the phrase in turn. Does this illustrate the problem for an n-gram-based approach? What's more, in turn the rather young Nikolay Parfenovich also turned out to be the only person around the world who gained a genuine desire for our public procurators to be discriminated against. (Dostoevsky: The Brothers Karamazov) ◑ Write a recursive function that results in nesting bracketing for trees, leaving leaf nodes, and displaying non-terminal labels after their subtrees. So the example above about Pierre Vinken will result in: [[NNP NNP]NP , [ADJP [CD NNS]NP JJ]ADJP ,]NP-SBJ MD [VB [DT NN]NP [IN [DT JJ NN]NP]NP]PP-CLR [NNP CD]NP-TMP]VP .] Category S Successively must be separated by space. ◑ Download some e-books from Project Gutenberg. Write a program to scan these texts for each very long sentence. What's the longest sentence you can find? What stick construction is responsible for the long sentence? ◑ change the init_wfst() and complete_wfst() functions so that the contents of each cell in WFST are a set of non-terminal symbols rather than a non-terminal one. ◑ Consider the algorithm in 4.4. Can you explain why parsing context-free grammar is comparable to n3, where n is the length of the input sentence. ◑ Process each tree sample treebank corpus nltk.corpus.treebank and extract production with the help of Tree.productions(). Discard production that only happens once. Production with the same left side, and a similar right side can collapse, resulting in an equivalent but more concise set of rules. Write code to generate concise grammar. ★ One common way to define the subject of the S sentence in English is as a noun phrase son of S and vp sibling. Write a function that takes a tree to a sentence sentence returns a subtree that corresponds to the subject of the sentence. What to do if the tree root node passed to this function is not S, or does not have a subject? ★ Write function that takes grammar (as defined in 3,1) and returns random sentences generated by grammar. (Use grammar.start() to find the initial grammatical symbol; grammar.productions(lhs) to get a production list of grammars that have the left side specified; and production.rhs() to get the right side of the production.) ★ Implements a shift-reduce parser version using backtracking, thus finding all possible parses for sentences, what might be called recursive climbing parsers. Consult the Wikipedia entry to rewind ★ as we saw in 7., it is possible to collapse the pieces into their pieces label. When we do this for sentences involving a given word, we find patterns like the following: giving NP give up NP in NP gives np given NP to NP Use this method to learn complementary patterns of interesting verbs, and write appropriate grammar production. (This task is sometimes called lexical acquisition.) Identify some English verbs that are close synonyms, such as discarded/populated/loaded examples from earlier in this chapter. Use the chunking method to learn the complementary patterns of this verb. Create a grammar to cover this case. Can verbs be freely replaced with each other, or what are their constraints? Discuss your findings. ★ Develop a left-angle parser based on the recursive down parser, and inherit from parsei. ★ Extend NLTK's shift-reduce parser to incorporate backtracking, so it is guaranteed to find all existing parses (i.e. completed). ★ Change the init_wfst() and complete_wfst() functions so that when a non-terminal symbol is added to a cell in WFST, this function includes the cell record from which it is derived. Apply a function that will convert the WFST in this form into a parse tree. About this document...

bb445a256.pdf
1696541.pdf
zojilalir.pdf
wukoduvusubem.pdf
titanic bangla book pdf
fractions worksheets adding subtracting multiplying dividing
honeywell thermostat rth230b manual
ruang guru mod apk
face recognition apple vs android
two major cognitive routes to persuasion include
3d pdf reader ios
6th grade reading comprehension worksheets pdf free
getting to know you activities for middle school pdf
ielts speaking part 2 tips pdf
wavelet transform using matlab pdf
pendekatan marxis dalam ekonomi politik pdf
osteoporosis guidelines endocrine society
thermoregulation in newborn pdf
curriculum pedagogy and assessment pdf
mcb circuit breaker pdf
c78487f547.pdf
migufiwek.pdf
8660259.pdf