FIT5217 week 4
CYK_Parser.ipynb

Objective of the tutorial.

- Learn how to **create parse tree of a sentence** using **CYK algorithm.**
  - parse tree
  - cyk algorithm
- what is **parse tree?** — A parse tree, also known as a **syntax tree**, is a **diagram** that represents the **syntactic structure** of a sentence **according to a given grammar.**
  - example: John hit the ball.
    - **what is the pos tagging for this sentence?**
    - John --> Noun
    - hit --> verb (an action)
    - the --> determiner
    - ball --> noun
    - result: [N, V, D, N]
    - the ball --> NP — NOUN PHRASE --> DETERMINER + NOUN
    - S --> root of the tree **(represents the whole sentence)** — **John hit the wall.**
    - S --> **breaks into two main parts:**
      - --> **a noun (N)** for the subject "John"
      - --> **a verb phrase (VP)** for the action and its object "hit the ball".
      - **VP** means verb+object **combination**, e.g., hit the ball.
      - --> here "hit the ball" is **VP**
      - **John (N) hit the Ball (VP)**
    - **VP** --> can be decomposed **into V and NP**
      - verb + noun phrase — hit the wall.
      - **NP (noun phrase)** is the combination of determinant and noun, e.g., the wall.
      - **hit (Verb) the Ball (Noun Phrase)**
    - **NP** --> can be decomposed into D and N
      - Determinant + Noun
      - **the (determiner) wall (Noun)**
    - **observation**: the last level of the tree bascially is "Noun Verb Determiner Noun"
  - Constituency-based parse tree
- what is **Context-Free Grammars(CFG)?**
  - S --> NP VP (A sentence can be broken down into a Noun Phrase followed by a Verb Phrase)
  - NP --> Det N (A Noun Phrase can be a Determiner followed by a Noun)
  - VP --> V NP (A Verb Phrase can be a Verb followed by a Noun Phrase)
  - ...and so on for determiners (Det), nouns (N), and verbs (V).
  - **Rules** about **how to decompose** a component **into smaller** components
- what is **CYK algorithm?**
  - **recap of lecture**
    - is an algorithm **used for parsing sentences** to **check** if they **belong to a given language** defined by a **Context-Free Grammar (CFG)**.
      - **so why we care about parse trees?**
        - it can used to **check grammartically error** — all the grammar rules --> and check one by one
        - if the sentence can be used to built parse trees --> **it means the sentence is valid**
    - 1. **Preparation:** Convert the CFG into Chomsky Normal Form (CNF). In CNF, each rule either produces two non-terminals or a single terminal.
      - Chomsky Normal Form (CNF) **is a way of simplifying and standardizing the rules** of a **Context-Free Grammar** (CFG)
      - **A grammar** is in **Chomsky Normal Form** if **all its production rules**
        - (the rules that define how to form sentences, e.g., NP --> Det N)
        - **follow one of these two patterns:**
          - non-terminal symbols — **Non-terminal Symbols**: These represent categories of phrases or types of words. They can be further broken down into other non-terminal or terminal symbols. They are like placeholders or slots for patterns in sentences. For example, "S" (sentence), "NP" (noun phrase), and "VP" (verb phrase) are non-terminal symbols.
            - S --> V NP
            - N --> John
          - terminal symbols
            - 1. **Binary rules:** A non-terminal symbol is replaced by exactly two non-terminal symbols. This takes the form `A → BC`, where `A`, `B`, and `C` are non-terminal symbols. It's like saying, "one component can be broken down into two specific parts."
              - S --> N VP
            - 2. **Terminal rules:** A non-terminal symbol is replaced by exactly one terminal symbol. In a sentence-tree diagram, This is written as `A → a`, where `A` is a non-terminal symbol, and `a` is a terminal symbol. It means "a component directly corresponds to a specific word or token."
              - N --> John
    - 2. **Initialization:** Create a table (a 2D array) where each cell will hold the possible non-terminals that can generate the substrings of the sentence. The table has as many rows as the number of words in the sentence.
      - **terminal symbols** on the top
      - it is a triangular matrix, and it is empty at first
      - we will then use this set of rules to fill the triangular matrix
    - 3. **Filling the Table:** Starting from the bottom, fill the table by checking if the CFG rules can produce the substrings represented by each table cell. This involves checking:
      - Single words against rules producing terminals.
      - Combinations of two cells for rules producing two non-terminals.
      - **S → NP VP**
      - **S → X1 VP**
      - **X1 → Aux NP**
      - **S → book | include | prefer**
      - **S → Verb NP**
      - **S → VP PP**
      - **NP → I | he | she | me**
      - **NP → Houston | NWA**
      - **Nominal → book | flight | meal | money**
      - **Nominal → Nominal Noun**
      - **Nominal → Nominal PP**
      - **VP → book | include | prefer**
      - **VP → Verb NP**
      - **VP → VP PP**
      - **PP → Prep NP**
      - **Det → the | a | that | this**
      - **Noun → book | flight | meal | money**
      - **Verb → book | include | prefer**
      - **Pronoun → I | he | she | me**
      - **Proper-Noun → Houston | NWA**
      - **Aux → does**
      - **Prep → from | to | on | near | through**
      - **NP → Det Nominal**
      - **VP → Verb NP**
      - **S → Verb NP**
      - **VP → VP PP**
      - **S → VP PP**
    - 4. **Result:** After filling the table, if the cell representing the entire sentence (the top right cell) includes the start symbol of the CFG, the sentence belongs to the language defined by the CFG.
      - extract parse trees from results
        - finding parsing tree **starting from top right corner**
        - extract parse tree based on rules

Implementation

- **1. initialise Grammar object**
  - 1. initialise Grammar object
  - 2. parsing a sentence to find parse tables
  - 3. get parse trees from parse table
  - load the data
    - what the txt file looks like
  - initialization of a grammar object
    - **parameters of the object:**
      - grammar_rules --> storing all the rules
      - parse_table
      - length
    - **process:**
      - 1. **load grammar rules from a file**
      - 2. **print out the loaded rules**
  - 1.load grammar rules from file
    - a dictionary, each key's corresponding value is a list
    - **all grammar rules**
      - first 5 binary rules --> leading two 2 non-terminal symbols
      - last 6 terminal rules --> leading to one terminal symbol
    - self.grammar_rules
  - 2. print out the loaded rules
  - print_rules() --> print out the rules in grammar_rules
    - printed results
- **2. parsing a sentence**
  - **self.productions are empty lists if set as None**
    - initialization
  - production_rules class
    - production basically is the grammar rule, e.g., S --> NP VP
    - result in this case is S
    - left child of the rule — p1 is NP
    - right child of the rule — p2 is VP
    - initialization
    - "__gen_type__ ", "__gen_lhs__ ", "__gen_rhs__ " These are **dunder(double underscore) methods**, meaning they are **reserved by Python** and have a specific purpose. **__gen_type__** returns the result of the rule, **__gen_lhs__** returns the left-side (LHS) and **__gen_rhs__** returns the right-side (RHS).
    - S --> V NP — BINARY RULE
    - N --> JOHN — TERMINAL RULE
    - CNF
  - **in a cell,** it holds a **productions list** storing production rules
  - **self.parse_table**
    - initialise a parse table, a triangular matrix, where **each cell will hold possible productions** that can generate the substrings of the sentence.
  - 1. set up 4 properties:
    - - number of trees
    - - tokens
    - - length
    - - parse table (initialisation)
  - production rules — cells objects — grammar object --> parse table
  - what the self.tokens look like
  - **apply_rules()**
    - NP --> astronomers
    - NP --> saw
    - V --> saw
    - check the grammar rules related to the input
  - 2. set up the parse_table
    - --> LOOPING THROUGH WORD ONE BY ONE IN A SENTENCE
  - parse function --> take in a sentence and
    - **1. set up four properties:**
      - - number of trees
      - - tokens
      - - length
      - - parse table
    - **2. filling in the parse table**
    - **3. check if the number of trees in the parse table**
      - --> if 0, the sentence violated the grammar rules
      - --> else, the sentence is valid
    - - if the word not existed in the grammar_rules --> raise error
    - - for all corresponding found rules
    - **w** is the production rule being found, e.g., 'NP --> Saw', 'V --> Saw'
    - **r** is the rules list
    - **x** is the index for a word, e.g., 0, 1, 2...
    - **t** is the word, e.g., saw
    - for each production rule
      - --> call the add_production() to add rules, e.g.,
      - NP --> Saw
      - --> None
    - **add_production(self, result, p1,p2)**
    - add_production() is **a function in the Cell Class** --> adding production_rule
    - **self.productions** --> is a list
    - a **production rule** basically is made up of
    - **result, p1 and p2.**
    - **j =1**
    - **i = 0**
    - Book the flight through Houston
  - 3. get parse trees from parse table
    - self.length --> length of the sentence
    - get production rules
    - get_rules() --> getting production lines
    - 1. checks if **there is a CFG rule that can combine a rule from t1 with a rule from t2.**
      - e.g., t1 may be [NP, VP]
      - t2 may be [VP]
      - based on S --> NP, VP
      - we can find the derivation
    - 2. If such a rule (r) is found, it **represents a valid derivation** for the substring and is **added to the parse table.**
    - "Applied Rule: **S[2,1] --> NP[1,1] VP[1,2]**" means that the **rule S --> NP VP** was applied to **create a Sentence (S) in the parse table**
      - --> at **row 2, column 1**,
      - --> by **combining a Noun Phrase (NP) at row 1, column 1,** and
      - --> **a Verb Phrase (VP) at row 1, column 2.**
    - Applied Rule: S[1,1] --> NP[1,1] VP[1,2]
    - Applied Rule: NP[1,1] --> Det[1,1] N[1,1]
    - Applied Rule: VP[1,2] --> V[1,1] NP[1,2]
    - Applied Rule: S[2,1] --> NP[1,1] VP[2,2]
    - Applied Rule: VP[1,2] --> VP[1,2] PP[2,2]
    - Applied Rule: S[1,1] --> NP[1,1] VP[1,2]
    - Each line indicates a successful application of a CFG rule when building the table
  - 4. parse tree function --> take in a sentence
    - Part 1
    - Part 2
    - Part 3
    - The sentence IS accepted in the language
    - The sentence IS NOT accepted in the language
    - Output exceeds the size limit. Open the full output data in a text editor
    - Grammar file reading successfully. Rules readed:
    - S --> NP VP
    - S --> X1 VP
    - X1 --> Aux NP
    - VP --> book
    - VP --> Verb NP
    - S --> book
    - NP --> Det N
    - NP --> Houston
    - Noun --> flight
    - Noun --> meal
    - Noun --> money
    - Nominal --> Nominal Noun
    - Nominal --> Nominal PP
    - VP --> book
    - VP --> Verb NP
    - VP --> VP PP
    - PP --> Prep NP
    - Det --> the
    - Det --> a
    - Det --> that
    - Det --> this
    - Noun --> book
    - Verb --> book
    - Verb --> include
    - Verb --> prefer
    - Pronoun --> I
    - Pronoun --> he
    - Pronoun --> she
    - Pronoun --> me
    - Proper-Noun --> Houston
    - Proper-Noun --> NWA
    - Aux --> does
    - Prep --> from
    - Prep --> to
    - Prep --> on
    - Prep --> near
    - Prep --> through
    - Applied Rule: S[1,1] --> NP[1,1] VP[1,2]
    - Applied Rule: NP[1,1] --> Det[1,1] N[1,1]
    - Applied Rule: VP[1,2] --> V[1,1] NP[1,2]
    - Applied Rule: S[2,1] --> NP[1,1] VP[2,2]
    - Applied Rule: VP[1,2] --> VP[1,2] PP[2,2]
    - Applied Rule: S[1,1] --> NP[1,1] VP[1,2]
    - Applied Rule: S[3,1] --> NP[3,1] VP[3,2]
    - The sentence IS accepted in the language
    - The sentence IS NOT accepted in the language
    - ('NP', 'V') ('Det') ('N') ('N') ('Det') ('N')
    - ---- ---- ---- ---- ---- ---- ----
    - ---- ---- ---- ---- ---- ---- ----
    - ---- ---- ---- ---- ---- ---- ----