Python Grammar Checking Program for English using Constituency Parse Tree

Yuzhi Tang, Hongshou Ge, Zheng Luan

Tuesday, April 16, 2021

Problem Description and Research Question

The scope of our project is in the field of Natural Language Processing (NLP). NLP is a prominent subfield of Artificial Intelligence, and it is mainly concerned with the interactions between computers and human language. Nowadays, NLP allows computers to recognize text and speech, and it can interpret them to measure sentiment and determine which parts are important.

Our group finds that a basic strategy to represent human language is to use syntax trees. We are inspired by the use of abstract syntax trees to check syntax correctness for Python programs, and we believe the same idea can be applied to check grammar correctness for natural languages. In particular, **the aim of our project is to create a Python program that could check the grammar correctness of an English sentence, and return feedback on the grammatical errors that may be present in the sentence**. Due to the many grammatical rules for the English language and the limited time for the project, only several grammar rules will be implemented as part of the grammar-checking program; however, one should note that such a program would be highly scalable.

Our approach to design the program is to first parse the English sentence into a constituency parse tree by using the Berkeley Neural Parser library (benepar). The root of a constituency parse tree represents the sentence (S). Internal values of a constituency parse tree represent non-terminal categories of the English grammar, for example, noun phrases (NP), verb phrases (VP), etc. The leaves of a constituency parse tree represent terminal categories of the English grammar (e.g. nouns (N), verbs (V), determiners (D)), which are the actual words in the sentence. Figure 1 is an example of a constituency parse tree.

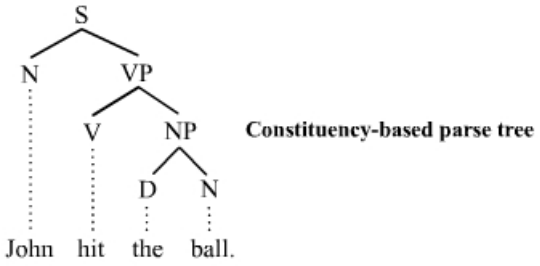

Figure 1: an example of constituency parse tree (File:Parse tree 1.jpg, 2011)

Next, we will represent the parse tree generated by the benepar library using our own constituency parse tree class (`GrammarCheckingTree`). This process is done using a translation function that we write (see computational plan for details). We want to translate the parse tree because this will give us greater freedom for manipulating the tree itself, which is needed for grammar checking. Grammar-checking code is implemented as methods in `GrammarCheckingTree`. This choice is partly inspired by the `evaluate` method of the abstract syntax tree classes for a Python program (e.g. the `Num` class, the `BinOp` class, etc).

Finally, grammar checking on a sentence is accomplished by calling the grammar-checking method on the root value of the `GrammarCheckingTree` object that represents the sentence (see computational plan for details). We will test the capability of the grammar checking program by providing it with numerous sentences and assess its

performance on the sentences.

We hope by making the grammar checking program, we could help people with their English writings. For example, it could serve as an educational tool for English Language Learners, who can use it to check the grammar correctness of their writings. Furthermore, in the Discussion section, we will be elaborating on the potentials for further development and limitations of creating a grammar-checking program using a constituency parse tree data structure to represent the sentence.

Computational Overview

GrammarCheckingTree class

The `GrammarCheckingTree` class is stored in `grammar_checking_tree.py`. The class inherits from the `GrammarTree` class, which is a standard recursive tree data structure that represents a constituent parse tree of an English sentence. The `GrammarCheckingTree` class has additional grammar checking methods. The `GrammarCheckingTree` class has two attributes: `root`, which is a dictionary with two keys `"label"` and `"text"` that store strings, and `subtrees`, which is a list of `GrammarCheckingTree` objects. The value of `"label"` is the constituent tag [1] of the represented constituent parse tree. If a `GrammarCheckingTree` object represents a word, the word itself is stored in `root["text"]`; otherwise, `root["text"]` is just an empty string. Figure 2 shows the detailed structure of a `GrammarCheckingTree` object that represents the sentence: "John hit the ball.".

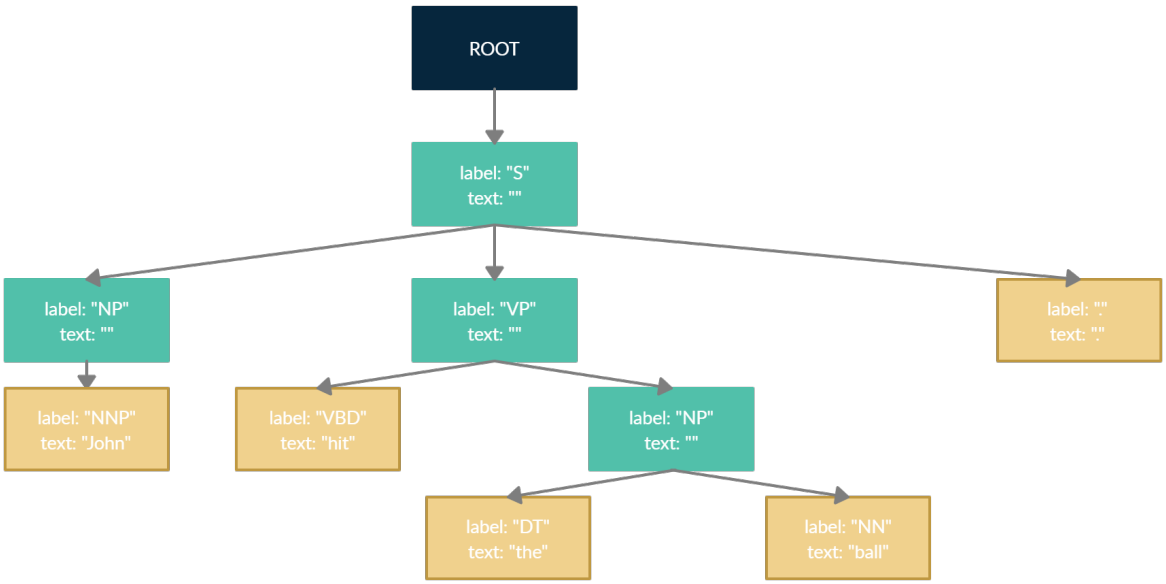

Figure 2: Structure of a `GrammarCheckingTree` object (Tree diagram generated using creately.com)

Creating GrammarCheckingTree objects

The `translate` function in `translator.py` returns a list of `GrammarCheckingTree` objects, each representing a sentence based on an input English paragraph. First, the `translator.py` module downloads the necessary constituency parse model. Then in `translate`, using the `nlp` function, the benepar library creates a list of constituent parse trees based on the input paragraph. Then, `_create_grammar_tree` creates and returns the corresponding `GrammarCheckingTree` object for each of the constituent parse trees.

[1] For a full list of constituent tags, see http://www.surdeanu.info/mihai/teaching/ista555-fall13/readings/PennTreebankConstituents.html.

Grammar-checking methods

The grammar checking codes are implemented as methods of the `GrammarCheckingTree` class. The followings are the grammar rules that are currently implemented in the program (please reference the code in `grammar_checking_tree.py`).

1. Check whether the given sentence is a complete sentence (`check_complete_sentence`).

   For all the complete sentences, they must have at least 1 noun and 1 verb in them. In this method, the logic is as follows:

   $$\exists \text{ Noun phrase in the sentence } \wedge \exists \text{ Verb phrase in the sentence } \implies \text{ The sentence is complete}$$

2. Check whether the adjectives in the sentence are at the right positions (`check_adjective`).

   If the given sentence is not an interrogative sentence, the position of an adjective has two cases:

   (1). The adjective is before a noun (e.g. He is a cool boy)

   (2). The adjective is after a linking-verb (e.g. he is cool)

   This is a recursive method and contains following steps:

   1. check whether there is any adj or adjp in the given GrammarCheckingTree. If true, keep going. Else, the result is no adj in the given sentence or use adj incorrectly.

   2. check the type of the given tree

   (1) The base case: if the type is ADJ, means we recursive to the leaf. We will check the accumulative list `result_so_far, which is List[bool]`. If all(result_so_far) and its length is greater than 0, the result is no error. else, result is 'hard to determinate: it may lack linking-verb or use adj incorrectly'.

   (2) If it is a question sentence, we will set a variable `whether_question = True` and check whether adj is after noun (eg. Is he cool?).

   (3) If it is a ADVP, return 'adj can not be a adverb'. If the type is FRAG, return 'you may lacks some linking verb or noun around adj'.

   (4) If the type is NP (noun phrase), check whether the adj is before the noun.

   if question sentence $\wedge$ (adj before noun) $\implies$ 'it is a question sentence and difficult to determinate'

   elif not question sentence $\wedge$ (adj before noun) $\implies$ `result_so_far`.append(True).

   else, noun before adj $\wedge$ question sentence $\implies$ 'This is a question sentence and may no mistake'.

   noun before adj $\wedge$ not(question sentence) $\implies$ 'There may no linking verb before adj'.

   If no error return right now, recursive the subtree of this GrammarCheckingTree. If no error is found, return 'can not easily judge: no error so far'

   (5) If the type is VP (verb phrase),

   the VP starts with a adj $\implies$ 'adj is in wrong position' (VP should start with a verb).

   linking-verb(be-verb) + adj $\implies$ `result_so_far`.append(True).

   If no error return right now or not these cases above, recursive the subtree of this GrammarCheckingTree. If no error is found, result is no error so far.

   (6) If the type is not one of the type listed above, recursive the subtree. If no error is found, result is no error so far.

3. Check whether the verb-ing is at right position (`check_verb`)

   If the given sentence is not a question sentence, the case of verb-ing is be-verb/like + verb-ing.

   This is a recursive method and contains following steps:

   1. check whether there is any verb-ing in the given GrammarCheckingTree. If true, keep going. Else, result is no verb_ing in given sentence or use verb_ing incorrectly.

   2. check the type of the given tree

   (1) The base case: if the type is VBG, means we recursive to the leaf. We will check the accumulative list `result_so_far, which is List[bool]`. If all(result_so_far) and its length is greater than 0, the result is no error. else, result is 'hard to determinate: it may lack linking-verb or use adj incorrectly'.

(2) If the type is SBAR (Clause introduced by a (possibly empty) subordinating conjunction), recursive the subtree of this GrammarCheckingTree.

(3) If the type is VP or S, it starts with be-verb/like + verbing $\implies$ `result_so_far`.append(True).

not starts with be-verb/like $\implies$ recursive its subtree.

(4) If the type is not one of the type listed above, recursive the subtree. If no error is detected, result is no error is found.

4. Check parallelism `check_parallelism`

   Check whether both sides of a conjunction have the same grammatical structure.

   First, check whether the given `GrammarCheckingTree` object contains a conjunction. If it contains, check whether the subtrees surrounding the conjunction are the same type of subtrees.

   Different types of subtrees $\implies$ "hard to determinate: the left side of the conjunction is not parallel to the right side.'

5. The following methods have very similar inspirations: `plural_noun_singular_verb`, `singular_noun_plural_verb`, `check_noun_to_verb`, `existence_of_subject`. The method `check_end_punctuation` uses a different strategy, but is nevertheless not a complex method.

   The first four methods all focus on trees with label 'S'. Each tree with label 'S' can be regarded as a sentence (may be a complete sentence or an incomplete sentence such as attributive clause). For each 'S' tree, I call recursive helper methods (often `contain_type`) to detect them. If any constituent of a full sentence is wrong, then the sentence as a whole will be wrong. Thus, I use this strategy to check the grammar correctness for simple, complex, and compound sentences.

   Among the above five methods, `check_noun_to_verb` is slightly different. Actually, it is a combination of `plural_noun_singular_verb` and `singular_noun_plural_verb`. This means it does not do totally new operations. It just calls the two methods as helper methods.

   `check_end_punctuation` is comparatively the simpler of the methods. Because the library we used offered us a chance to solely focus on the last element in `self.subtrees`. Therefore, the strategy for this method is different though also simpler.

Reporting results

We will report how well our grammar-checking program performs for English sentences. Numerous English sentences of different sentence structures, some with grammatical errors of the types implemented in the program and some without grammatical errors, will be selected from credible sources or manually generated as inputs to the program. We will then observe how well our grammar-checking program can correctly recognize the grammatical errors in each of the sentences. We will report how well our program performs on the chosen sentences (in a mostly qualitative manner), and based on the result, we will also report on the potentials for further development and limitations of creating a grammar-checking program using a constituency parse tree data structure.

Changes made from project proposal

We changed the data structure of how we represent the constituent parse tree generated by the benepar library. In the proposal, we intended to use multiple classes to represent the constituent parse tree based on different sentence levels (i.e. the clause level, the phrase level, and the word level). However, we later found that for our purpose of making a grammar-checking program, the structural information stored by the three classes can be condensed into a single class (i.e. `GrammarCheckingTree` class) without loosing much information. Furthermore, one of our group members had to leave due to personal/family reasons, and as such we had to reduce the number of grammar rules that are to be implemented in the program due to limited man-power. Nevertheless, owing to the data structure employed in the program, the program itself is highly scalable in the sense that any new grammar-checking methods can be implemented, in principle, with little to no need to modify existing code.

Instruction for Running Program

1. Install all Python libraries listed under the requirements.txt file. The libraries can be installed using Pycharm directly.

2. To see the result of the grammar-checking program's performance on the pre-selected sentences, run the module `main.py`. The pytest result is printed onto the console.

3. To check the grammar correctness for new sentences, run the module `demo.py` and use the function `demo_check_grammar` on the new sentences.

Discussion of Results

Our project has achieved the goal of checking whether a given sentence has grammar error in some aspects, including check whether the sentence has noun (subject) and verb (predicate), the use of adj, and the use of verb in 'ing' form.

However, due to the limitation of the library we used (and the difficulty of constituent parsing on English sentences), we can not point out the error of a sentence for some cases. For example, suppose we use check_adj to check the sentence 'The man who cool'(Obviously it is a wrong sentence because it lacks a linking-verb), the outcome is out of anticipation: it returns 'no adj inside or use adj wrongly' (this should be returned only when there is no adj in the given sentence), though it should have returned: 'adj in wrong position, maybe lack linking-verb' for this case. This problem is caused by the fact that the library does not recognize the 'cool' in this sentence as an adj. Instead, according to the graph of the tree, the type of 'cool' is VBP, which means non-3rd person singular verb. As a result, our method can not find any adj in the given sentence 'The man who cool'. Therefore the method returns 'no adj inside or use adj wrongly'. Similarly, for some errors in complex sentences that contain clauses, we can only return 'it is hard to determinate.' due to the irregular library output.

There is one limitation for `check_end_punctuation`, that is it cannot analyse the end punctuation that are not commonly used. For example, although it is valid for an English sentence to end with quotation mark or ellipsis (at least by some formatting standards), here we ignore those cases to decrease method complexity. Nevertheless, we are able to analyze the end punctuation that are commonly used for sentences: '.', '!' and '?'.

In terms of `existence_of_subject`, one of the limitations comes from the library. partly due to that the library is designed for mostly translating correct English sentences. Sometimes it will translate an incorrect input sentence (clearly lacking a noun phrase) in such a way as to treat non noun phrases as noun phrases to make the sentence seem 'less incorrect' (this is due to the subtleties of the inner workings of the benepar library translation algorithm, which we will not go into here). As a result, the accuracy of the method `existence_of_subject` is relatively low.

Let me talk about `plural_noun_singular_verb`, `singular_noun_plural_verb` and `check_noun_to_verb` as a whole. The reason why I separate them is to make it easy to call any of them individually. It is very common to see English learner match nouns and verbs mistakenly (violating subject-verb agreement). To make it obvious and explicit, I separate them and print slightly different feedback for each of the grammar errors. The reason why I combined the two methods in `check_noun_to_verb` is because that they are very much related to each other and therefore should be categorized as the same sort. Both of their structures are very similar and their design inspiration is the same. If a sentence only contains intransitive verbs, then the above three methods are likely to work well even when the sentence is a compound or complex sentence. When a sentence contains transitive verbs or passive voice, the three methods can not detect grammar errors effectively, because they are likely to treat the object as the subject. We tried to fix this issue, but all the attempts do not yield satisfactory results. Thus we have to admit that this method is not effective for sentences containing transitive verbs or the passive voice.

For future studies, we recommend finding a constituent parsing model that can accurately recognize the part of speech of words, even if the input sentence is not grammatically correct (to solve the issue we faced in this project, where the 'cool' in 'The man who cool' is classified as a 3rd person singular verb, rather than a adjective). For this we recommend looking into constituent parsing models that are statistical based (instead of rule based). Using techniques such as supervised learning for training such models, they may perform better than rule based models when it comes to translating grammatically incorrect sentences.

Conclusion

In this project, we built a grammar checking program. We used benepar library for translating an English sentence into a `GrammarCheckingTree` class (i.e. a recursive tree structure). Then, grammar checking and returning feedback is done by calling methods (recursive and non-recursive) on the `GrammarCheckingTree` object representing the sentence. We found that a major limitation we faced when writing and testing the grammar checking methods is that the constituent parse tree (generated by benepar) does not give a meaningful object for analysis at times when it comes to grammatically incorrect input sentences. Although this sheds doubt as to whether using a constituent parse tree structure to analyze the grammar correctness of a sentence is effective, we recommend future studies to employ statistical based constituent parsing models as a means for possible improvement.

References

1. Commons.wikimedia.org. 2011. File:Parse tree 1.jpg. [online] Available at: https://commons.wikimedia.org/wiki/File: \\Parse_tree_1.jpg.

2. Berkeley Neural Parser. spaCy. (2021). Retrieved 15 March 2021, from https://spacy.io/universe/project/self-attentive-parser/.

3. Grammar: Sentence structure and types of sentences. (n.d.). Retrieved March 15, 2021, from https://academicguides.waldenu.edu/writingcenter/grammar/sentencestructure.

4. Natural language processing. (2021, March 09). Retrieved March 15, 2021, from https://en.wikipedia.org/wiki/Natural_language_processing.

5. Parse tree. (2021, March 06). Retrieved March 15, 2021, from https://en.wikipedia.org/wiki/Parse_tree.

6. Sharma, A. (2021). Part-of-Speech(POS) Tag — Dependency Parsing — Constituency Parsing. Analytics Vidhya. Retrieved 15 March 2021, from https://www.analyticsvidhya.com/blog/2020/07/part-of-speechpos-tagging-dependency-parsing-and-constituency-parsing-in-nlp/

7. Penn Treebank Constituent Tags. Surdeanu.info. (2021). Retrieved 15 March 2021, from http://www.surdeanu.info/mihai/teaching/ista555-fall13/readings/PennTreebankConstituents.html.

8. Liu, D., & Badr, M. (2021). 14.1 Introduction to Abstract Syntax Trees. Teach.cs.toronto.edu. Retrieved 15 March 2021, from https://www.teach.cs.toronto.edu/ csc110y/fall/notes/14-abstract-syntax-trees/01-abstract-syntax-trees.html.

9. Grammar Rules. (2021). [Ebook]. Retrieved 16 April 2021, from https://www.languagecouncils.sg/goodenglish/-/media/sgem/document/additional-sgem-resources/pdf/grammar-rules-_-speak-good-english-movement.pdf?la=en.

10. benepar. PyPI. (2021). Retrieved 17 April 2021, from https://pypi.org/project/benepar/#usage.