Text simplification through tree restructuring

Joscha Brügmann

2023-12-18

Summer Semester 2022

Advanced Natural Language Processing with Python

Dr. Kilian Evang, Tatiana Bladier

Abstract

This report discusses a programming project concerning text simplification, with focus on the syntactic aspects of what complicates human comprehension. The approach that has been chosen is to build a general tree restructuring solution for the use with syntactic structure trees. By formulating *restructure rules* that productively capture how to reduce the complexity of a structure tree, the program can simplify sentences and thus reduce the effort required for a human to parse them.

Contents

Introduction

There are at least two considerations when it comes to simplifying text; lexical complexity and syntactic complexity. A well structured text with simple grammatical phenomena may still be hard to understand if it uses very specific, complicated or simply infrequent words. The same goes the other way, a text containing simple words may employ complicated syntactic structures that end up hindering a readers understanding.

Evidence for the effects of modulating the complexity of syntactic structure has been shown in various studies. For example, center-embedded sentences compared to equivalent conjoined sentences elicit slower response times, reduced comprehension accuracy and higher levels of brain activity (Suh et al. 2007), multiple layers of self embedding decrease reader retention of sentences (Miller and Isard 1964), whether a sentence has canonical or non-canonical word order (SVO vs OVS) has an effect on response accuracy and pupil dilation in audio-visual picture-matching tasks (Wendt, Dau, and Hjortkjær 2016), object-relative sentences evoke heightened brain activation in comparison to subject-relative sentences (Constable et al. 2004).

This project aims to provide a means of simplifying the syntactic complexity of sentences by offering a general tree restructuring solution. Many grammatical theories employ trees as the main way of abstracting grammatical structure. By targeting trees in general, this project could be utilized for any of these grammatical theories, independent of the individual nuances. The implementation works through tree based pattern matching and transferring the matched nodes to a target tree structure. With this, one can delete parts of a tree, relabel nodes, add new nodes or completely reassemble a tree through user created rules. This relies on the input data already being trees, putting aside the issue of text-to-tree parsing.

Data and Resources

This is an enclosed tool that currently only relies on python version 3.10.0, its built in packages and nltk 3.7 (Bird, Klein, and Loper 2009). It does not rely on any specific data sets for its operation, instead it is intended to be used on sets of structure trees.

Project structure

The core of this project consists of '*TreeRestructureFunctions.py*', which contains the functions used for the restructuring implementation, and '*RestructureRule.py*', which provides the main programmatic interface in the shape of a class. '*restructure.py*' implements a console interface for accessing the main functions of the project while '*visualize.py*' is a small console utility to visualize json files containing the *trees* and *restructure-rules* used in this project. There is also a folder of test cases included, that were used in development to make sure some of the core components were working correctly. These can also be used, however, to become familiar with the format of data that is expected by the implementation.

Method

This section covers the workings of '*TreeRestructureFunctions.py*'. Explanations on the usage and function of the other modules can be found in the ReadMe file and the codes documentation. To get a general overview of the tree restructuring concept, please refer to the ReadMe file first.

After the input is handled and processed into the required format through '*restructure.py*' and '*RestructureRule.py*', '*TreeRestructureFunctions.py*' starts being used. First, the match/transfer rules that were defined are baked into a copy of the pattern tree. This means that for every node in the pattern tree, the nodes label is set to be a dictionary containing the specific settings for that node (2 b). The nodes original label is also included into its dictionary. This was done in order to simplify accessing the settings of nodes during recursive traversal. Next, all possible ways the pattern can match on the tree, with consideration to the matching settings of the nodes, are determined in an intermediate format (2 c).

The intermediate format uses three types of markers to track the positions in which the nodes of the pattern tree can be matched (1). A '*location marker*' holds two sets of coordinates. One for the location of a node within the pattern tree and one for the location of where that node is found in the tree that is being processed. A '*children marker*' is used when the associated node in the pattern tree is not a leaf. It holds both a location marker for the node and a list of all the markers for its children. With these two, an outline of how the pattern can be matched is built by correlating each node in the pattern tree with a node in the tree that is being processed. Whenever a node can be matched in multiple locations, an '*option marker*' is used. This marker opens up different strands of possible continuations for the match, which get resolved in the next processing step. There, the outline which the intermediate format provides gets expanded into full trees representing each possible match (2 d).

The last step is the generation of the result trees for all matches by employing the transfer settings of the nodes. This means that for each of the trees that represent a found match, its nodes are transferred into a copy of the target tree structure according to the transfer settings of the node (2 e). Thus any sub-trees or parent-in-between trees are also transferred with the node if determined by the settings of that node. The result is a list of trees that includes all possible ways to match the pattern on a tree and transfer the possible matches to the target structure.

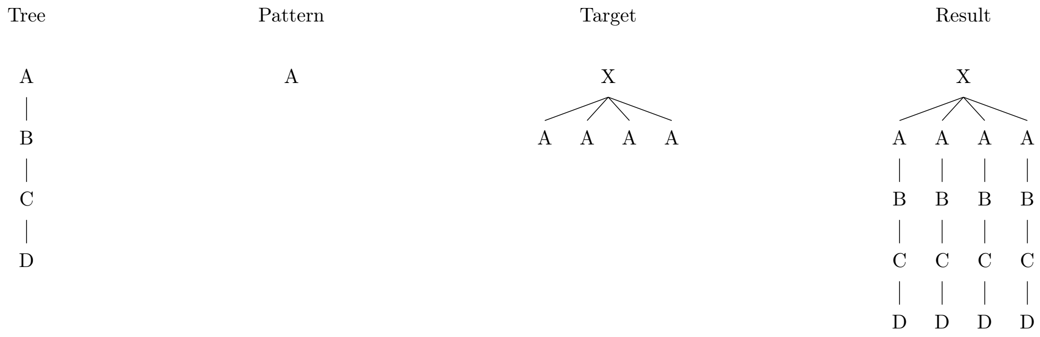
(1) Overview of markers in the intermediate format

Location marker: ('I am at', (*index in pattern*), (*index in tree*))

Children marker: ('I have children', (*location marker for node*), [*list of markers for children*])

Option marker: ('options', [*list of possible continuations*])

(2) (a) Overview for the '*duplicationTest*' files from the '*Testcases*' folder[1]



(b) Baked settings in pattern tree (results of '_ *bakeRules*')

```
({
'targetIndex':                    [[0], [1], [2], [3]],
'allowUnspecifiedChildren':       True,
'transferUnspecifiedChildren':    True,
'allowUnspecifiedParents':        True,
'transferUnspecifiedParents':     True,
'label':                          'A'
})
```

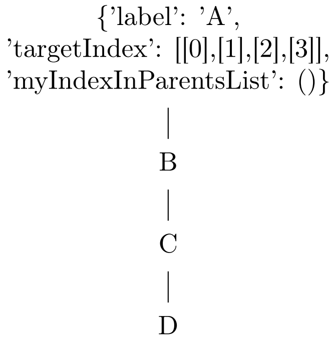
[1] Check the Appendix on page 9 for a few more complicated processing flow examples.

(c) Possible ways to match the pattern (results of '*howCanPatternApply*')

```
('I am at', (), ())
```

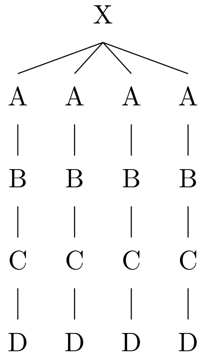
(d) Expanded into tree representation of match (results of '*patternApplications*')

```
({
'label':                  'A',
'targetIndex':            [[0], [1], [2], [3]],
'myIndexInParentsList':   ()}
        (B
                (C D)
        )
)
```


{'label': 'A',
'targetIndex': [[0],[1],[2],[3]],
'myIndexInParentsList': ()}
B
C
D

(e) Transferred into target tree structure (results of '*transfer*')

```
[(X
        (A (B (C D)))
        (A (B (C D)))
        (A (B (C D)))
        (A (B (C D)))
)]
```



Discussion

Taking the example of self embedding from Miller and Isard (1964) and building one rule to reduce and one rule to increase the levels of embedding, this implementation successfully generates the structure-trees expected for higher and lower levels of self embedding from any starting level. Since Miller and Isard (1964) showed that modulation of the level of self embedding has an effect on the retention rate of the sentences, this project can be used to modify the expected retention rate of these sentences through changing their level of self embedding. In other words, a rule that decreases the amount of self embedding has the effect of reducing the structural complexity of a sentence which correlates with a readers ability to retain the contents of the sentence. Such a rule thus achieves the goal of simplifying text for the purpose of human parsing.

(3 c) repeats the tree diagrams from Miller and Isard (1964) for a sentence with four levels of self embedding and the corresponding tree without self embedding. (3 a) also shows the inbetween levels of self embedding for this sentence. The rules designed to increase and decrease the level of embedding in these sentences along with the trees can be found in the ’*Examples*’ folder of the project files. After navigating to the project directory, run e.g. ’`python restructure.py Examples/MillerIsard1964SelfEmbedding0.json Examples/MillerIsard1964EmbedRule.json --visualize`’ to create the tree with one level of embedding from the tree without any embedding and visualize the result. Repeatedly applying the ’*EmbedRule*’ on the results that get generated will further increase the level of embedding until you get to the maximum amount that the structure can sustain. At that point the rules pattern will no longer be applicable and your result will be empty. In the same manner, if you run ’`python restructure.py Examples/MillerIsard1964SelfEmbedding4.json Examples/MillerIsard1964UnembedRule.json --visualize`’, you will decrease the level of embedding of the tree with four embeddings by one. Repeating this with your result will end up producing the tree with zero embeddings, from where the ’*UnembedRule*’ is no longer applicable. If you just want to see all of the corresponding trees for (3 a), run (3 b).

One way of productively using this projects capabilities for text simplification would be to first, identify a number of structural phenomena that directly impact reader comprehension, then engineer generalized rules that can productively simplify those phenomena within the framework of a syntactic theory to finally employ those rules by running them on some set of tree-parsed sentences.

(3) (a) Examples for different levels of self-embedding (Miller and Isard 1964):

(0) She liked the man that visited the jeweler that made the ring that won the prize that was given at the fair;

(1) The man that she liked visited the jeweler that made the ring that won the prize that was given at the fair;

(2) The jeweler that the man that she liked visited made the ring that won the prize that was given at the fair;

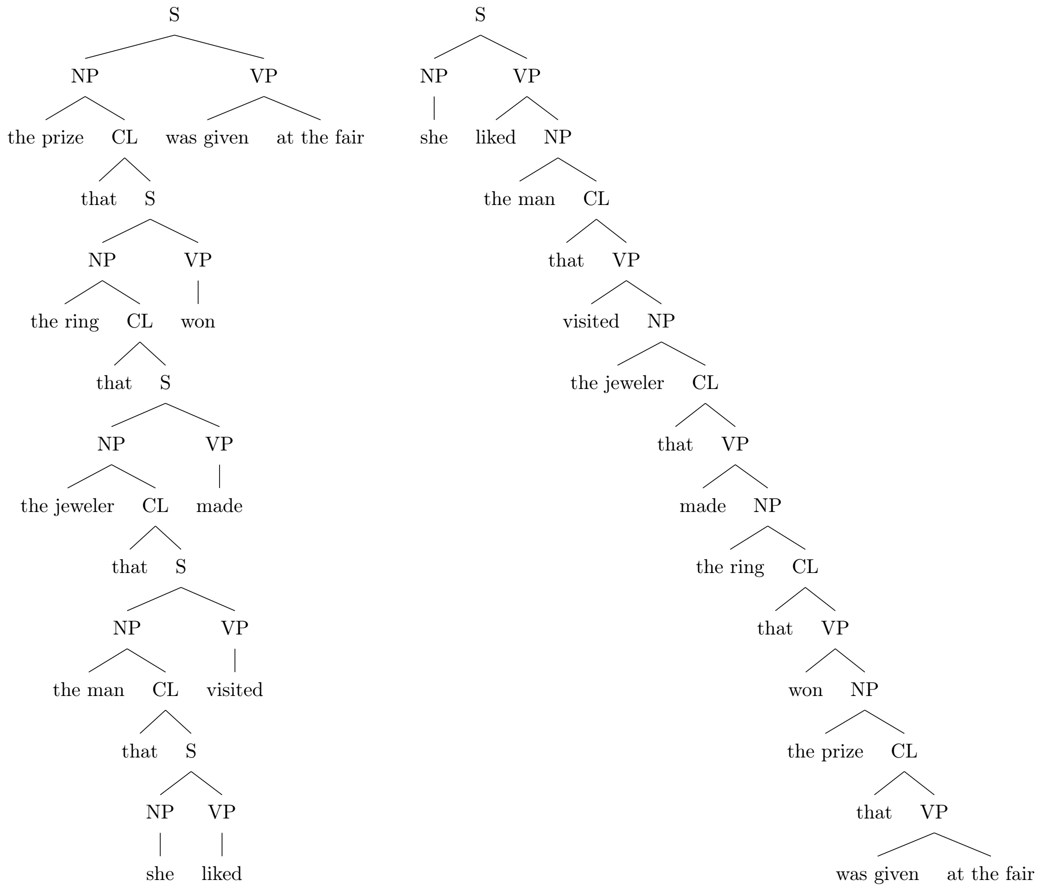
(3) The ring that the jeweler that the man that she liked visited made won the prize that was given at the fair;

(4) The prize that the ring that the jeweler that the man that she liked visited made won was given at the fair;

(b) Command to visualize all levels of self embedding:

```
python visualize.py Examples/MillerIsard1964SelfEmbedding0.
    json Examples/MillerIsard1964SelfEmbedding1.json Examples/
    MillerIsard1964SelfEmbedding2.json Examples/
    MillerIsard1964SelfEmbedding3.json Examples/
    MillerIsard1964SelfEmbedding4.json
```

(c) Structure tree with four levels of self embedding and the corresponding tree without self embedding, taken from Miller and Isard (1964) with a minor expansion:



Challenges and open issues

My initial approach to this project started by trying to make plain python regular expressions work with trees in bracket notation. I soon realized, however, that for any relevant node structures I wanted to search in the bracket notation, I needed recursive pattern matching. Since native python regular expressions don't support recursive patterns, I started building an independent solution that was specifically build for trees. In hindsight, the same could have most likely been accomplished using some pre-made implementation for recursive regular expressions. However, since I constantly felt like I was very close to making my independent solution work, I ended up getting absorbed in it instead of considering a third party package.

There are some limitations to the restructuring concept itself. For example, you do not have full control over where nodes that weren't specified in the pattern are transferred to in the target tree. I have thought of a separate conceptualization involving '*collection nodes*' which would consider all nodes part of themselves, up to the next pattern-specified node. This way one could specify transfer settings for nodes that aren't strictly part of the pattern. Another limitation is that the results of applying this program may not always lead to fully grammatical constructions since morphological requirements aren't taken into consideration. In those cases, further processing would be required to refine the results. It may be necessary to design rules with the expectation that the results of applying them will be incomplete or somewhat broken when it comes to grammatical agreement or references and to attempt fixing those mistakes with further processing. An example would be when a rule separates a sentence into multiple individual sentences as one might do for sentences that feature multiple propositions or embedding. The current implementation could be expanded to let you, depending on the environment of a constituent, pass additional information about grammatical properties that it should fulfill after transfer. This would allow feeding additional information into post processing. Providing additional context like this to the pattern could also help in making sure a rule is being applied under appropriate circumstances.

If I where to re-implement this project I would try to make the implementation more straight forward. For example, with multiple functions it would be worth considering to use iterative approaches instead of recursive ones and to avoid internally baking settings into trees. This could make the code more transparent to read and involve less copying. Getting rid of the intermediate format that is being generated by the '*howCanPatternApply*' function would also help with simplifying the implementation. I believe it could be skipped in favor of a more direct data flow. I opted to use this tactic during development because of trouble conceptualizing how to get to my intended results. Through an intermediate format I was able to get a step closer to the solution, which helped making clear how to move forward. However, finding a more direct implementation would be better. Another fairly big improvement in simplifying the implementation could be to not internally use and operate on the *nltk.Tree* tree implementation. I think it would have been better to create a small custom tree implementation or to simply use native python data types like standard lists to represent trees. This way a lot of the *nltk* specific implementation curiosities that needed to be worked around could have been avoided. Using *nltk.tree* for it's visualization options after independent internal processing would still be valid though. It would also be worth to reconsider the current json format of the rules and at least give the option to integrate the rules into the pattern tree in the file itself to mitigate user made indexing errors.

Summary and conclusion

Overall, I believe that the present implementation may offer a promising approach towards one solution for restructuring and simplifying syntactic trees. More effort needs to be put into identifying generalized and productive rules and the specific implementation would benefit from refactoring and consideration to efficiency. The core concept could also be re-conceptualized a bit to offer more control over node matching and transferring.

References

Bird, Steven, Ewan Klein, and Edward Loper (2009). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit.* O'Reilly Media, Inc. URL: https://www.nltk.org.

Constable, R. Todd et al. (2004). "Sentence complexity and input modality effects in sentence comprehension: an fMRI study". In: *NeuroImage* 22.1, pp. 11–21. DOI: 10.1016/j.neuroimage.2004.01.001.

Miller, George A. and Stephen Isard (1964). "Free recall of self-embedded english sentences". In: *Information and Control* 7.3, pp. 292–303. DOI: 10.1016/S0019-9958(64)90310-9.

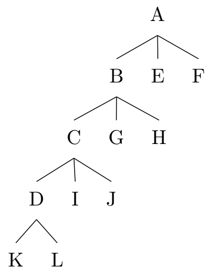
Suh, Soyoung et al. (2007). "Effects of syntactic complexity in L1 and L2; An fMRI study of Korean-English bilinguals". In: *Brain Research* 1136, pp. 178–189. DOI: 10.1016/j.brainres.2006.12.043.

Wendt, Dorothea, Torsten Dau, and Jens Hjortkjær (2016). "Impact of background noise and sentence complexity on processing demands during sentence comprehension". In: *Frontiers in psychology* 7, p. 345. DOI: 10.3389/fpsyg.2016.00345.

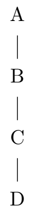
Appendix

Processing steps for the '*levelling*' files from the '*Testcases*' folder:

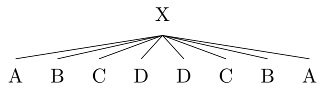
Tree



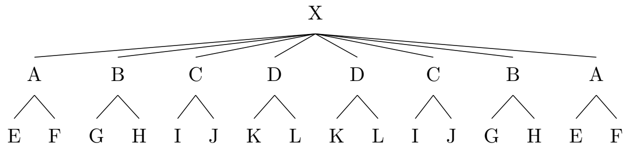
Pattern



Target



Result



1. Baked settings in pattern tree (results of '_*bakeRules*')

```
({'targetIndex':                  [[0], [7]],
'allowUnspecifiedChildren':       True,
'transferUnspecifiedChildren':    True,
'allowUnspecifiedParents':        True,
'transferUnspecifiedParents':     True,
'label':                          'A'}
        ({'targetIndex':                  [[1], [6]],
        'allowUnspecifiedChildren':       True,
        'transferUnspecifiedChildren':    True,
        'allowUnspecifiedParents':        True,
        'transferUnspecifiedParents':     True,
        'label':                          'B'}
                ({'targetIndex':                  [[2], [5]],
                'allowUnspecifiedChildren':       True,
                'transferUnspecifiedChildren':    True,
                'allowUnspecifiedParents':        True,
                'transferUnspecifiedParents':     True,
                'label':                          'C'}
                        {'targetIndex':                   [[3], [4]],
                        'allowUnspecifiedChildren':       True,
                        'transferUnspecifiedChildren':    True,
                        'allowUnspecifiedParents':        True,
                        'transferUnspecifiedParents':     True,
                        'label':                          'D'})))
```

2. Possible ways to match the pattern (results of '*howCanPatternApply*')

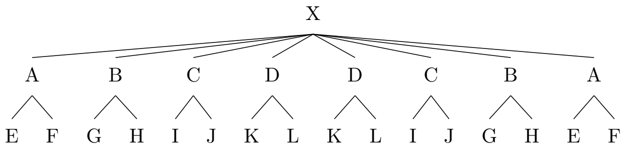
```
('I have children', ('I am at', (), ()),
          [('I have children', ('I am at', (0,), (0,)),
                    [('I have children', ('I am at', (0, 0), (0, 0)),
                              [('I am at', (0, 0, 0), (0, 0, 0))]
                    )]
          )]
)
```

3. Expanded into tree representation of match (results of '*patternApplications*')

```
({'label':                'A',
'targetIndex':            [[0], [7]],
'myIndexInParentsList':   ()}
          ({'label':                'B',
          'targetIndex':            [[1], [6]],
          'myIndexInParentsList':   (0,)}
                    ({'label':                'C',
                    'targetIndex':            [[2], [5]],
                    'myIndexInParentsList':   (0,)}
                              ({'label':                'D',
                              'targetIndex':            [[3], [4]],
                              'myIndexInParentsList':   (0,)}
                              K
                              L)
                    I
                    J)
          G
          H)
E
F)
```

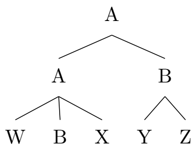
4. Transferred into target tree structure (results of '*transfer*')

```
[(X
(A E F) (B G H) (C I J) (D K L)
(D K L) (C I J) (B G H) (A E F))]
```

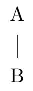


Processing steps for the '*permutationTestSimple*' files from the '*Testcases*' folder:

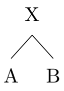
Tree



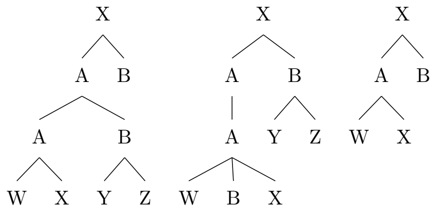
Pattern



Target



Results



1. Baked settings in pattern tree (results of '*_bakeRules*')

```
({'targetIndex':                   [[0]],
'allowUnspecifiedChildren':      True,
'transferUnspecifiedChildren':   True,
'allowUnspecifiedParents':       True,
'transferUnspecifiedParents':    True,
'label':                         'A'}
        {'targetIndex':                   [[1]],
        'allowUnspecifiedChildren':      True,
        'transferUnspecifiedChildren':   True,
        'allowUnspecifiedParents':       True,
        'transferUnspecifiedParents':    True,
        'label':                         'B'})
```

2. Possible ways to match the pattern (results of '*howCanPatternApply*')

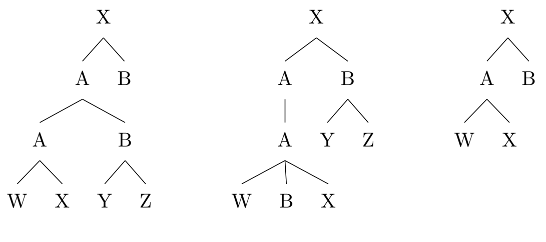
```
('options',
  [
    ('options',
        [
                ('I have children', ('I am at', (), ()),
                        [('I am at', (0,), (0, 1))]
                ),
                ('I have children', ('I am at', (), ()),
                        [('I am at', (0,), (1,))]
                )
        ]
    ),
    ('I have children', ('I am at', (), (0,)),
        [('I am at', (0,), (0, 1))]
    )])
```

3. Expanded into tree representations of match (results of '*patternApplications*')

```
[
  ({'label': 'A', 'targetIndex': [[0]], 'myIndexInParentsList': ()}
    (A
        W
         ({'label': 'B', 'targetIndex': [[1]], 'myIndexInParentsList': (0, 1)})
        X
    )
         (B Y Z)
    ),
  ({'label': 'A', 'targetIndex': [[0]], 'myIndexInParentsList': ()}
    (A W B X)
    ({'label': 'B', 'targetIndex': [[1]], 'myIndexInParentsList': (1,)}
        Y
        Z)
    ),
  ({'label': 'A', 'targetIndex': [[0]], 'myIndexInParentsList': (0,)}
   W
    ({'label': 'B', 'targetIndex': [[1]], 'myIndexInParentsList': (1,)})
    X
  )
]
```

4. Transferred into target tree structure (results of '*transfer*')

```
[
          (X
                    (A
                              (A W X)
                              (B  Y  Z)
                    )
                    (B)
          ),
          (X
                    (A
                              (A W B  X)
                    )
                    (B  Y  Z)
          ),
          (X
                    (A W X)
                    (B)
          )
]
```


X
A
B
A
B
W
X
Y
Z
X
A
B
A
Y
Z
W
B
X
X
A
B
W
X