Detecting Errors in Parsing of Speech Transcript primarily due to Active Linguistic Repair

Input

Transcript of *NPTEL* Lectures. Standard English Language tokenized text separated into sentences. The text files contain sentences separated by newlines. We will be using the following six text files for our analysis: Course 101101058, Lectures 14 to 19

Introduction

A repair is an alteration that is suggested or made by a speaker, the addressee, or audience in order to correct or clarify a previous conversational contribution.

It may occur at any of several points following the contribution in question, perhaps occurring in accordance with a conventional order of preference.

In our study, we will be studying self-initiated repair, in which the speaker of the utterance makes a repair without any prompting from another participant.

The problem of evaluating the correctness of parsing in an unsupervised manner is unsolved and kind of circular in nature. The only sound progress in this regard can be made by using a set of rules or principles which can effectively rule-out something from being the correct parse output of a particular sentence. In that regard, we manually verify a set of sentences from the following two text files which serves as our observation set: Course 101101058, Lectures 14 and 15.

We observe the patterns in the parsing of the above files to come up with a small set of intuitive rules, say $\mathscr{R}$ which can pinpoint a particular parse structure as grammatically incorrect, or show a possibility that some sort of linguistic repair has taken place within that very sentence.

We then evaluate how effective these $\mathscr{R}$ rules are in detecting phenomena in the other four text files serving as our verification set. This study will show if it is indeed possible to come up with a set of heuristics based on a small set of input sentences, which can be utilized in detecting errors & instances of repair in the parsed structure of sentences from a large array of transcripts coming from similar sources.

Parser

Spacy’s dependency parser.

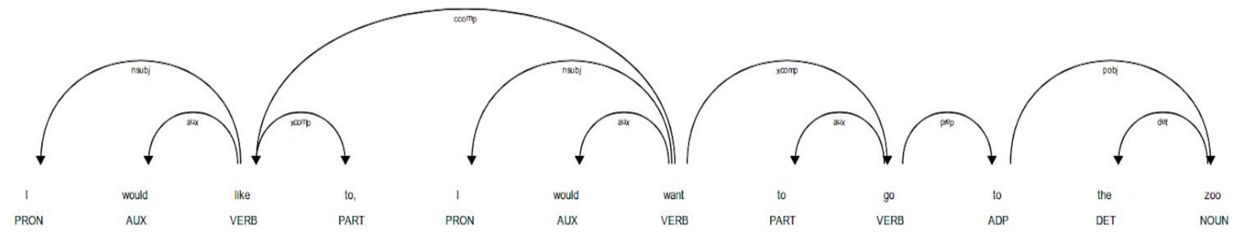
This is an industrial grade parser and gives much better results than Stanford CoreNLP. Although Spacy's parser draws inspiration from CoreNLP and uses a very similar set of rich linguistic features, it varies in its implementation. Also, spacy uses a trained supervised neural model to create modules which are both well tested and yield higher accuracies on available extrinsic and intrinsic tasks.

We see the spacy sometimes differs in the structure of a dependency parse, such as its way of handling conjunctions ‘cc’(s), however, their structure makes intuitive sense and is consistent across parsings of all sentences.

Violating Conditions

Let’s look at a toy example:
Sentence S_1 = "I would like to, I would want to go to the zoo."

We get the following parse output:

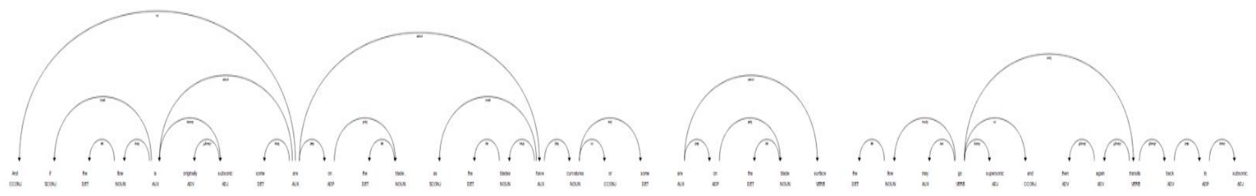


The parse structure shows an xcomp branch leading to a single ‘to’ which is a participle at the base level. This indicates that the phrase is incomplete. An xcomp branch must lead to a complete clause which is missing in this case. This error originates from the ungrammaticality of the sentence. In “I would like to”, the ‘to’ must lead to a complete clausal complement.

Consider the following sentence from lecture 14 of NPTEL course number 101101058:

S_2 = “And if the flow is originally subsonic some are on the blade, as the blades have curvatures or some are on the blade surface the flow may go supersonic and then again transits back to subsonic.”

We get the following parse output:



Clearly, the above dependency tree is not even a connected graph. It is composed of three subtrees. This clearly is an error as a dependency structure is supposed to be a tree. The underlying intuition here is the fact that in order to qualify as a valid sentence, all its individual components must share some sort of connection. Again, the error arises purely due to ungrammaticality. Also, we can clearly see the repair taking place here. On realising that one portion wasn’t perfectly clear, the speaker says, “as the blades have curvatures or some are on the blade surface”, thereby repairing and repeating some portion of the previous utterance.

Analyzing a few sentences

To carry on the discussion further, kindly observe the following sampling of tokenized sentences from our observation set:

S_3 = “So all of these together has ensured that we have transonic compressors today , which are partly supersonic , partly subsonic , which means the compressors can go supersonic that means the flow can be in entirely supersonic and such compressors , supersonic compressors are indeed used in special circumstances specially in a rocket motors where you need very very compact compressors .”

S_4 = “In air craft engines , the designers decided to stick to transonic compressor other than going fully supersonic , which as I mentioned would have meant a lot of shocks , and lot of shock losses (vocalized-noise) and those lot of shock losses would have really brought down the efficiency of the compressor substantially which as you know show up in the form of fuel efficiency .”

S_5 = “This transition occurs because of the shocks, if it is the flow is originally supersonic through the shocks, normal shocks they would become subsonic.”

S_6 = “And if the flow is originally subsonic some are on the blade , as the blades have curvatures or some are on the blade surface the flow may go supersonic and then again transits back to subsonic .”

S_7 = “And at the root it is lowest hence and this is exactly what I was trying to mention a little earlier that the root this value of V 1 could jolivel be subsonic where as at the mean it could be near sonic and at the tip it could be go clearly supersonic .”

S_8 = “Very high compression ratio would normally require supersonic flow in both rotor and stator and this is what is often done in as I mentioned some of the rocket motor compressors on other hand in most of the air craft engines as of today only the rotor is supersonic where as the stators and by enlarge subsonic but , as we can see here it is entirely possible for stator to also go supersonic .”

S_9 = “Now each blade would have a shock standing in front of it and this shock you know is shown over here , so the flow as it enters the rotor in supersonic .”

S_{10} = “On the other hand , if we consider the fact that it is possible looking at the velocity diagram that we have done . It is entirely possible that the flow in the rotor is entirely subsonic , however the value of C 2 as they go into the stator indeed as gone supersonic .”

S_{11} = “This shocks as we know sharply diffuse the flow , it is a shock diffusion and hence the flow goes through partial supersonic diffusion and then rest of the diffusion is then done subsonically as we have understood before .”

S_{12} = “Now this is a problem , this is a serious problem because a very sharp leading edge is actually susceptible to faster and earlier stall when the flow is subsonic .”

S_{13} = “That means a sharp leading edge or a sharp leading body , a sharp nose body , does not know how to deal with subsonic flow , is very bad in subsonic flow , is very good in supersonic flow , is very bad in subsonic flow .”

S_{14} = “So early on in the compressor designers decided that they would not like to have any sharp edged , leading edged blade anywhere in the compressor design .”

S_{15} = “So it is necessary to ensure that the shocks are created or allowed to be created inside the blades in a controlled manner .”

S_{16} = “So when the flow is supersonic going over the blades the shocks that are created have certain amount of certain kind of shape and characteristics that are known and those things would create less of shock losses .”

S_{17} = “So this is what the designer started doing when the transonic compressors first appeared , subsonic airfoils were no good , new airfoils had to be designed and hence and they have to be designed in a manner that the shocks that are created are under some kind of control when the compressor is operational .”

S_{17} = “However they were initially when used for transonic compressors the flow would go mildly supersonic on the blade surface and would somewhere over here the flow would go supersonic and it would create a small shock .”

Clearly, none of the above sentences are grammatically sound and pretty much all of them demonstrate active linguistic repair. However, their dependency parse is pretty much what we would expect for such a sentence, In that, the parser does a really good job. In most of these cases, the parse tree looks really convoluted but has no apparent errors. The ungrammaticality of the sentence itself is what leads to the weirdness of the parse structure. On parsing these sentences by hand, the result is pretty similar to what the machine produces. In case of differences, the machine seems to be doing a reasonable or better job. Hence, we cannot classify these “parse errors”.

How Repair Affects Results

Now, we’ll shift our attention to detecting instances of linguistic repair. Note that the detection of parse structure violation required some simple checks. But, here we will be proposing some algorithms to deal with self repair, which can be of the following three forms:

1. Repetitions: the speaker repeats some part of the utterance. For example, I * I like it.
2. Revisions (content replacement): the speaker modifies some part of the utterance. For example, We * I like it.
3. Restarts (also called false starts): a speaker abandons an utterance or constituent and then starts over. For example, It’s also * I like it.

As defined by the 2003 INTERSPEECH publication: Automatic Disfluency Identification in Conversational Speech Using Multiple Knowledge Sources.

The paper uses prosodic information as well to detect repair which we do not have. Instead, here we use the part-of-speech feature as used in the paper and we leverage our dependency parse tree to find possibilities that a repair of the above three forms may have taken place.

The key idea here is to build a standard graph out of the formatted dependency tree structure provided by a parser and to define some graph algorithms which can run on the created data structure.

Modules

We have the following modules to check for parse constraints violation based on the data-structure created above:

```
M1: Given a set of nodes S, Find the Root Node
Method: Requires a set of nodes, S as a parameter
iterate over S with iterator node: DO
    if parent of node is the node itself,
        then return node as the Root
```

```
M2: Given the root node of a tree, Find if it violates any ‘xcomp’ conditions
Method: Requires a ‘node’ as a parameter
let all_children ← set of all children of node
if all_children is an empty set and the dependency label of arc to node is ‘xcomp’,
    then return False,
        else return True

    else, let subtree_result ← True,
    iterate over all_children with iterator child: DO
        update subtree_result as (subtree_result and M2(child))

    return subtree_result
```

```
M3: Given the root node, find the size of the tree:
Method: Requires a ‘node’ as a parameter
```

```
let all_children ← set of all children of node
if all_children is an empty set,
    then we have encountered a leaf node and we return 1
    else,
        let subtree_size ← 1

        iterate over all_children with iterator child: DO
            update subtree_size as (subtree_size + M3(child))

        return subtree_size
```

```
M4: Given a set of nodes, check if the graph is connected or not
Method: Requires a set of nodes, S as a parameter
let root_token ← M1(S)
let tree_size ← M3(root_token)
if tree_size equals size of S,
    then return 1,
        else return 0
```

Foundations and Data Structures

Now, We will lay out the foundations for a set of algorithms for repair detection:

We create a global store ‘all_structs’ which will store structural information about subtrees rooted at every node in the dependency graph DS for the particular sentence in consideration.

Populating this store iteratively would have taken a complexity of O(N ^ 2), N being the number of nodes. Thus, we have to look for better alternatives. Dynamic programming on Trees comes in handy here.

```
dp [‘child’] = {'size': 1, 'hash': the word embedding of node text, 'node': the child node}
dp [‘node’] = {‘size’: 1 + sum of dp [‘i’] where ‘i’ is a child of node,
        ‘hash’: centroid of embedding of text in node with all hash [‘i’];
                                                   where ‘i’ is a child of node,
        ‘node’: the present node}
```

Using the above DP optimization, we can populate the store in O(N) instead.

Another, important optimization to make here is to make the parent pointers in the tree explicit such that we may traverse upwards. This neat trick reduces the number of steps of finding the height of a node from (2 * N - 1) to H, since we do not need to search for the node anymore when its height is inquired for. The parent pointers simply take us up to the root.

Now, once we have stored the necessary node variables in a struct, checking if one struct was actually a descendant of the other can be done in O (max (height (node_1), height (node_2))). Say we want to check if 2 stored structs: struct_A and struct_B are bound by an ancestor-descendant relation or not, we assign struct_B to be the struct having the higher node and initialize an iterator variable, say token with 'node' parameter in struct_A. Then:

```
while parent of token is not the same the node: DO
    if parent of token is the same as the 'node' parameter of struct_B,
        then return True
otherwise, return False
```

Feature Engineering

Now, we need some features for pairs of nodes which captures the spatial proximity between them. 2 such intuitive features come to mind:

1. The length of the shortest path between the 2 nodes
2. The difference in their index of appearance in the depth-first search traversal

Feature 2 can be easily computed using our store which maintains all nodes in order of their depth-first traversal.

Getting Feature 1 in linear time is the tricky issue here. We use another popular trick to achieve this. Note that:

path_length (A, B) = path_length (A, root) + path_length (B, root)
- 2 * path_length (root, LCA(A, B)) — (**I**)

- where LCA(A, B) is the least common ancestor of A and B

Hence, the problem reduces to finding the LCA of nodes in O (height).

For this, we create 2 paths: Path_A storing nodes from A to the root and Path_B storing nodes from B to the root. Now, we reverse each path and iterate over them from the root. The moment we hit an uncommon node, we return the previous node as the LCA. Or if we have completely traversed one out of the two paths, we return the last node of the said path as the LCA. Clearly, this takes 3 * height_of_tree operations and is clearly O (height_of_tree).

Then, using (**I**), we obtain the value for feature 1.

Similarity Checker

Our similarity metric checks two parameters:

1. The cosine similarity
2. The L2 norm of the difference between the embeddings

Repetitions using Parse structure

Now, we can define our main repetition checker which operates on the global store of all structs. We iterate over pairs of structs and ignore the following cases:

a. The two structs have nodes which appeared which were less than ‘nhops’ hops away in the DFS traversal of the tree
b. The shortest path between the two nodes is of length less than ‘lpath’
c. The two nodes are bound by an ancestor-descendant relationship

Now, if the cosine similarity between the two nodes is less than threshold ‘tcos’, and the L2 norm of difference between embeddings is less than threshold ‘tnorm’, we classify the sentence as one having a high possibility of repair, based on its parse structure.

Handling Overlaps in Smaller Windows

We also handle repetitions in smaller windows of size say, seven. In every window of seven words, if we find two lexical tokens having a cosine similarity greater than 0.95, we classify that as a possible lexical repetition in a small window. Note that, stop words and function words are not considered for repair identification. Thus, using these two metrics we have effectively captured repairs of all the three forms defined above.

What about word overlaps?

The interesting thing here is that doing something like a simple bigram overlap within a sentence does not capture repair effectively. We encountered some bigram overlaps in sentences having repair in our observation set. However, in our verification set of 865, sentences, the number of sentences having bigram overlap in maximum windows of size 31 was only two. This experiment proves that simply looking at word overlap will not provide effective repair information over distances.

Hyperparameter Tuning

We proposed the above scheme of classification as it makes intuitive sense and has a small set of tunable parameters. Clearly, since we do not have a wide range of gold standard labelled data, there has to be a human in the loop. In that, we manually set parameter values to check what works best for a small set of sentences from our observation set bootstrapped with a few toy examples.

Also, note that none of the above algorithms come as part of any package or library. Therefore, we had to implement all of them from scratch. We used the following hyperparameter values:

nhops = 6
lpath = 8
tcos = 0.975
tnorm = 1.15

Results

The following is purely based on our verification set:

- Percentage of Sentences violating parsing constraints: 4.86%
- Percentage of wrongly parsed Sentences showing Repetitions based on Parse: 71.43%
- Percentage of wrongly parsed Sentences showing Repetitions based on Window: 42.86%

	Values
Sentence Count	865
Not a Tree	40

'xcomp' Condition Violated	3
Possible Repetitions using Window	253
Possible Repetitions using Parse	627
Bigrams repeated	2
Parsing Constraint violated	42
Violation + Repetition with Window	18
Violation + Repetition with Parse	30

For the Qualitative Evaluation, kindly refer to the given Jupyter-Notebook. As an illustration, consider the following sentence which has been classified as one with violation + repetition with parse:

“Now , we know , that the degree of reaction should never be 0 anywhere on the blade or definitely , never be less than 0 anywhere on the rotor blade .”

utterance_1 = “should never be 0 anywhere on the blade”
utterance_2 = “never be less than 0 anywhere on the rotor blade .”

Clearly, utterance_2 repairs utterance_1, by providing a clarification. It is therefore a correction or rather a modification. Also, note that this would not get captured purely looking at lexical overlap. This is getting captured purely due to our proposed graph algorithm.

Hence, we conclude that our proposed algorithm does work out in reality. To capture how effective our algorithm is, we need some sort of a supervised dataset for a proper evaluation.

Related Work

Here are some papers who have worked on some form of parse-based approach to repair. Other than that, most of the work uses prosodic cues to detect repair in speech. Prosodic disjuncture is detected by a decision tree-based ensemble classifier that uses acoustic cues to identify where normal prosody seems to be interrupted.

Very little work has been done by taking dependency parse-based approach to linguistic repair. One notable work is that done by *Honnibal and Johnson (2014)* in a paper name *Joint Incremental Disfluency Detection and Dependency Parsing*. Their incremental dependency parsing model jointly performs disfluency detection. The model handles speech repairs using a novel non-monotonic transition system and includes several novel classes of features.

They use the arc-eager transition system (Nivre,2003, 2008), which consists of four parsing actions: **S**hift, **L**eft-Arc, **R**ight-Arc and Re**d**uce. The Shift action moves the first item of the buffer onto the stack. The Right-Arc does the same, but also adds an arc, so that the top two items on the stack are connected. The Reduce move and the Left-Arc both pop the stack, but the Left-Arc first adds an arc from the first word of the buffer to the word on top of the stack. Constraints on the Reduce and Left-Arc moves ensure that every word is assigned exactly one head in the final configuration.

Rasooli and Tetreault (2013) in their paper *Joint Parsing and Disfluency Detection in Linear Time* describe a joint model of dependency parsing and disfluency detection. They introduce a second classification step, where they first decide whether to apply a disfluency transition, or a regular parsing move. Disfluency transitions operate either over a sequence of words before the start of the buffer, or a sequence of words from the start of the buffer forward.

The algorithm we proposed has its foundations in how tree-LSTMs work. The interpretability in tree-LSTMs is a big hurdle. Our approach tries to combine the ideas of storing some sort of structured value in the tree nodes and thereby combining them hierarchically. We use the dependency tree structure as our underlying tree and go on populating node values in a bottom-up manner.

References:

1. https://glossary.sil.org/term/repair
2. Joint Parsing and Disfluency Detection in Linear Time: https://www.aclweb.org/anthology/D13-1013.pdf
3. Joint Incremental Disfluency Detection and Dependency Parsing: https://www.aclweb.org/anthology/Q14-1011.pdf
4. Automatic Disfluency Identification in Conversational Speech Using MultipleKnowledge Sources: http://www.cs.columbia.edu/~julia/papers/liu03.pdf
5. Improved Semantic Representations FromTree-Structured Long Short-Term Memory Networks: https://arxiv.org/pdf/1503.00075.pdf
6. https://universaldependencies.org/en/dep/xcomp.html
7. https://spacy.io/