Computational First Position Repair Processing

Charles Threlkeld and Alexandru Ungureanu

May 11, 2018

1 Problem Statement

In natual dialog, people often correct themselves mid-statement. For example, if I say "I'm going to the library, no Halligan," a person will understand that I am going to Halligan. However, machines will often fail to process this utterance correctly. "I'm going to the library," is a full sentence, and the computer may recognize it as such, disregarding the repair. Alternatively, the machine may simply fail to parse the utterance and ask for a repetition. For a more exhaustive survey of the current research on linguistic repair, see Albert and de Ruiter[1]. Our algorithm aims to clarify the semantic meaning of substitution self-repair. It will dispose of any super-semantic meaning of the processed disfluencies.

2 Approach

Out approach uses the Indiana "Cooperative Remote Search Task" (CReST) Corpus[2] that was compiled from 2007 to 2010 by Kathleen Eberhard et al. at University of Notre Dame from 2007 to 2010. Eberhard et. al. collected and codified a pair of teammates performing a search task. The corpus was compiled explicitly as a data source for commonly problematic aspects of natural human dialogue in computational linguistics. The teammates are linked only by audio, and they were each privy to only certain relevant information for the task. This ensured that the scenario would elicit certain forms of problematic dialogue that are not necessarily present when participants have a larger shared knowledge base.

This corpus is broken down into several levels: utterances, token, parts of speech, and disfluencies (as well as a few others that we are ignoring for our purposes). the parts of speech and disfluencies are marked at a token level. To form a training and testing set, we processed the corpus into a data structure that combines these different tagging purposes into a single data structure so that we can tie together the utterance to its tokens, parts of speech, and

[1]Repair: the interface between interaction and cognition. Albert and de Ruiter 2018

[2]The Indiana "Cooperative Remote Search Task" (CReST) Corpus. Eberhard et al. 2010

disfluencies. The tokens, parts of speech, and disfluencies are lists of the same size and ordering, so that they can be easily associated computationally.

Once we were able to parse the data in this way, we isolated utterances that were tagged as disfluent and at least two tokens long. The assumptions here are: 1. Disfluent utterances signal that a repair is happening. If an utterance were fluent, then the speaker has already said what he intends, and may change the meaning later, but the utterance itself does not need repair. 2. If an utterance is only a single word, there is no semantics to repair. A simple "uh" utterance, for example, is disfluent, but cannot be repaired into a fully realized text. There is still information present, but it is meta-textual. For example, it could signal confusion or distraction, but tracking this sort of semantic information is beyond the scope of text repair.

The corpus yielded about one hundred repair candidate utterances. Reading through these, we narrowed this to 55 repairable utterances. Candidates were thrown out for several reasons. Some where textually fluent, but were marked with disfluencies by the corpus given the audio log. Some were more than one token, but only a single word (e.g. huh?). The remaining fifty-five should form a good, natural corpus with which to operate. Our sample size is a bit small, but should be large enough to be relevant on the scale normally used within computational linguistics. For compairson, van Deemter[3] outlines three experiments where there are 25, 14, and 34 subjects are used for the human control set. Our data set is of this same order, and so fits the standard set by other research.

In addition to our work preparing the repairable dialog, we have also built the scaffolding necessary to leverage our algorithm on novel data. The Sentencizer, Tokenizer, and POSTagger modules work together to create a data set that useable by our repair module. The disfluency data provided by the corpus was used to generate our training and test sets, but will not be necessary in order to leverage our repair mechanisms. Having disfluencies or repair tagged for a certain utterance is outside the scope of our work. If our algorithm is given text that does not need repair, it may become damaged as the features of the repair mechanism may flag false positives.

Of note in the POSTagger is that we use a different corpus to the CReST corpus. The reasons for this are twofold: 1. The data provided by the CReST corpus are limited to the words used in the specific task present in that experimentally generated data, and may not generalize easily to other domains. 2. The sheer number of unique tokens present in the CReST corpus is significantly smaller than that provided by our POS corpus. The wider data set allows for more robust, reliable tagging, which will be leveraged in the repair module.

[3]Finetuning NLG through experiments with human subjects: the case of vague descriptions. Kees van Deemter. 2004

After scraping the CReST corpus, we hand-repaired each utterance, we hand-repaired the sentences to remove disfluencies. E.g. “just - just - can you - can you hold on a second, Brandy?” was repaired as “can you hold on a second, Brandy?”. You can see that the repaired version is simply a more fluent version of the semantics of the first. After performing these hand-repairs, we then randomized 70% (38 utterances) as our training set, and the remaining 17 as a test set.

We are splitting our training and testing sets so that the test set provides a distinct data set for assessing the effectiveness of our algorithm. Our features are meant to be linguistically transparent (see below), but we reserve the test set to ensure that we don’t overfit to the training set. The purpose of the training set, then, is to ensure that each repair feature that we add to our package is doing repair work. Similary to code tests, the Dice score (again, below) combined with the training set allow us to ensure that our features are both working in the programming sense, and working in the dialog repair sense.

With the data scraped, repaired, and randomized, we built the repair module. Our hypothesis is that we can generate algorithmically generated sentences that are semantically equivalent to the hand-repaired utterances. However, we would need to do further semantic processing in order to verify the semantic content. Instead, we use the syntax as a stand-in for semantics. There is a linguistic debate as to whether this is a reasonable substitution, but this is a first step in a a more full repair.

To that end, we tested our baseline Dice score (see below). For the baseline, we measured the Dice score of the unrepaired utterances to the hand-repaired utterances. This is a baseline because any feature that we add should cumulatively build on this baseline. For our training dataset, the Dice score was 0.77. We can then write features sequentially, ensuring that each monotonically adds to the Dice score. There is some hazard in this approach that we over-fit the data. However, even though our dataset is small, it is also varied. This was ensured both by the way the data was captured, as a feature of the CReST corpus and the randomization of the utterances for the training data.

With proper data and a testing method in place, we built the actual crux of the project: the repair mechanisms. First, in the repair module, we created a train method that would output the Dice score of repair done on the training set. Next, we built a series of features in the Feature module. Linguistically, disfluencies are not random. And so, each feature represents one specific type of disfluency that people exhibit. For example, one feature handles two-token utterances like “yeah okay”. In this case, the disfluency is a simple repetition of the same sentiment. The repair would entail dropping one token or the other. Semantically, the words should have the same content, and so as part of a semantic parser, we would have some leniency as to which token we choose for the utterance. In our feature, we chose the second token, with the thinking that if a second token wasn't a clarification of the first, then it would be unnecessary.

At the beginning of the project, the idea was have a process for choosing the most-likely sentence-repair chosen from a selection of possible sentences. The algorithm would have mapped very closely from the part-of-speech tagger from class. However, this approach would not work with linguistic repair features. First, we would need a much larger dataset. Generally, when doing machine learning of this type, a few thousand data points are necessary. Given the richness of natural language, this problem domain may need a few more orders of magnitude to train a meaningful algorithm.

Second, the features included in such an algorithm are unclear. Using our method, we can trace a direct line from linguistic theory to feature in our set. Word repetition (see above) is one standard disfluency marker with a clear repair. Extraneous utterances, such as "like" or "um" are also easily handled with our approach.

Third, our approach allows for several disfluencies to be disentangled and repaired. For example, if an utterance is "I want like some um chicken for dinner," mapping to "I want chicken for dinner" is straighforward using standard linguistic markers. In a traditionally trained algorithm, the "like" is ambiguous. Does the person want something similar to chicken or actual chicken? Similarly, the sentence "I want some dinner" is equally fluent as the candidate above, but loses the semantic content of the chicken. By adhering to more transparent linguistic formulae, we avoid throwing away semantically important information.

3 Further Work

As outlined above, our method of repair is linguistically transparent. Therefore, a good deal of work can be done by adding features to the Features module and to Features.featureList. These features represent the types of disfluencies that our algorithm currently handles, and so any additional features will be an asset to this library.

As an example of unimplemented features (due to time contraints), is one that leverages self-repair markers and our part-of-speech tagger in order to parse self-correction. E.g. "Pass me the blue book, I mean the red book" repairs to "Pass me the red book". This relies on identifying the self-correction marker "I mean", and recognizing by part-of-speech similarity that the object "red book" should replace the original "blue book" and the repair phrase can be dropped in the final semantics.

Another example that would require more extention, would be to include stemming in our package. Without a stemmer, correction of tense or plurality does not take place. So “Hand me the book - books” goes unrepaired, as “book” and “books” are not recognized as the same token.

Our algorithm could also be recursive, such that repair candidates themselves are then repaired until the repair and repair-candidate are equivalent. This would allow us to cover cases where there are multiple disfluencies in a single utterance, which is another benefit of using the transparently linguistically-derived features.

4 Evaluation

When judging referring expressions in computational linguistics, one common method is the Dice metric defined as follows:

$$s(A,B) = \frac{2 * \|A \cap B\|}{\|A\| + \|B\|}$$

That is, for two expressions, we measure their similarity by counting the tokens they share and dividing by the total number of tokens. A perfect score of 1 would mean that the intersecting tokens of the two sets are equal in size to the addition of the two sets (i.e. they are the same set). There are some concerns with the Dice metric[4] For example, a single missing token is penalized more heavily in the linear judgement than is adding an extra token. Secondly, the Dice metric is symmetric, so $s(A,B) = s(B,A)$, which has similar problems. Further, the semantic content of the repaired sentence (produced freely by a human) could rephrase in such a way that the tokens have almost no overlap, but the semantics are extremely similar.
However, given these caveats, Dice is still a reliable method for a first pass on the effectiveness of our algorithm. It has the advantages of being easily implemented and objective. Were we to judge on semantics, we would need an entire NLU pipeline to format both utterances into (e.g.) a logical format, and then a method to compare the similarity of the logics. Or, we could explore the similarities subjectively, but any subjective judgement is going to be obscured by the biases of any judge. (E.g. the creators have incentive to judge leniently, but classmates may have incentive to judge harshly.)

We began with a baseline Dice score of 0.77 on our training set. After implementing our features, we achieved a Dice score of 0.82 with our training set. In our test set, our baseline Dice score was 0.85, and the score when using our repair feature set was 0.88.

[4]Beyond DICE: measuring the quality of a referring expression. Kees van Deemter. 2010.

5 Materials

5.1 Data List

We used a fifty-five utterances of dialog recognized as exhibiting disfluency. The larger our corpus, the more general we can make our algorithm, but since this type of repair happens almost exclusively in spoken dialog, and not prepared speech or written text, we were limited to a spoken corpus of natural dialog. If we were able to find a larger dataset, we could test new linguistic features of repair.

We have found the Cooperative Remove Search Task (CReST) corpus that was put together by Kathleen Eberhard et al. in 2010. It codes for disfluencies and should serve our needs. It has already been tagged for part-of-speech, so we can use it for both training and testing of our algorithm. Though the current implementation is deterministic and the scope does not include identification of repair, if more work can be done, then it may be possible to do a more mechanized train and test. Details can be found in Eberhard 2010[5].

5.2 Code Libraries

We anticipate to use breeze linear algebra and optimization libraries and possibly some other statistics scala libraries.

5.3 Additional Material

We have located several papers by psycholinguists studying the mechanisms of dialogue repair as well as the current state of robotic implementation of these mechanisms.

6 Results

At this point, we designed and identified the different components of our project: the Sentenceizer (splits an input file into the sentences to be repaired), the Tokenizer (splits a sentence into tokens), the POSTagger (tags each of the tokens of a sentence with a part of speech found most appropriate by our POS tagging algorithm), and the Repair driver (based on the POS taggings of the tokens and the flagged repair phrases, repairs the phrase and outputs the resulting sentence).

In addition to setting up the scala project and the source file for each component, we implemented a naive training for the POS Tagging component of our project and created a naive implementation of the POSTagger class, which will take a list of tokens, and, based on the training phase learning, assign the ”appropriate” part of speech tag to each token. Also, we wrote a

[5]Eberhard et al. The Indiana “Cooperative Remote Search Task” (CReST) Corpus 20017

simplifier that maps Penn tags to the more general tags Adjective, Adverb, Conjunction, Determiner, Noun, Number, Preposition, Verb, Beginning. Currently, the tagger works at about 95% accuracy according to the testing data. It currently operates on the corpus from assignment 6, so results may be somewhat smaller when operating on the distinct CReST corpus.