Extracting semantic relations using syntax: an R package for querying and reshaping dependency trees.

Kasper Welbers, Wouter van Atteveldt and Jan Kleinnijenhuis
VU University Amsterdam

Abstract

Most common methods for automatic text analysis in communication science ignore syntactic information, focusing on the occurrence and co-occurrence of individual words, and sometimes n-grams. This is remarkably effective for some purposes, but poses a limitation for fine-grained analyses into semantic relations such as *who says what to whom* and *according to what source*. One tested, effective method for moving beyond this bag-of-words assumption is to use a rule-based approach for extracting semantic information from syntactic dependency trees. Although this method can be used for a variety of purposes, its application is hindered by the lack of dedicated and accessible tools. In this paper we introduce the [package] R package, which is designed to make it easier to develop, test and apply rules for extracting useful semantic relations.

Introduction

Applications of automatic text analysis in social scientific research almost invariably rely on the *bag-of-words* assumption (Boumans & Trilling, 2016; Grimmer & Stewart, 2013; van Atteveldt & Peng, 2018). Texts are broken up into individual words, and hence represented only by the frequencies of words, regardless of the way in which these words are related to each other in syntax. This approach culls a substantial part of the information contained in a text, but has been proven to be very useful for a wide variety of text analysis tasks. From analyzing document or sentence level sentiment (Barberá, Boydstun, Linn, McMahon, & Nagler, 2016; Liu, 2012) and coding policy issues (Burscher, Vliegenthart, & De Vreese, 2015), to unsupervised techniques for automatically classifying texts into topics (Blei, Ng, & Jordan, 2003; Roberts et al., 2014) or positioning texts on an (ideological) dimension (Benoit & Laver, 2003; Slapin & Proksch, 2008), the mere frequencies of individual words seem to contain sufficient information.

However, for certain types of analysis the syntactic relations between words are invaluable. State-of-the-art approaches for sentiment analysis and opinion mining do not just look at word frequencies, but use methods that take the contexts of words into account (Cambria, Schuller, Xia, & Havasi, 2013; Nakov, Ritter, Rosenthal, Sebastiani, & Stoyanov, 2016). To extract sources and quotes from a text, the syntactic constructs that indicate which parts of a text are quotes and who the sources are needs to be used. More generally speaking, syntax can make the difference between knowing what entities and actions are mentioned in a text, and knowing which of these entities did what to whom and whether entities are talked about or quoted as sources.

Van Atteveldt, Sheafer, Shenhav, and Fogel-Dror (2017) presented a method for using syntax to extract sources, subjects and predicates from texts, and apply this for a fine-grained analysis of differences in citation and framing of news coverage about the Gaza war. This allows them, for example, to correctly distinguish who the attacker is in the sentences "Israel attacked Gaza" and "Israel was attacked by Hamas". The essence this method is to query and manipulate *dependency trees*, and to apply certain heuristics for dealing with syntactic complications such as *long-distance dependencies* and *argument drop* (see Bender, 2013). Although the method itself is not particularly complicated, and requires only a basic knowledge of dependency based grammar, the application is complicated by the tools that are required for working with dependency trees. In this paper we present an open-source R package that offers a dedicated and intuitive toolkit for using this method, and demonstrate its appliation by providing step-by-step instructions for extracting source-quote and subject-predicate clauses. In addition, a set of tools is provided for reshaping dependency trees, which makes it possible to simplify texts as a preprocessing step, to make it easier to extract semantic relations from complex sentences. Furthermore, we show that these tools for querying and reshaping dependency trees can more broadly be used for extracting useful semantic information.

Extracting semantic relations from texts

The ambition to automatically extract semantic relations from natural language is far from new. It lies at the center of Artificial Intelligence (AI) research aimed at building systems that can extract knowledge from text, and the longstanding goal to "build an autonomous agent that can read and understand text" (Kok & Domingos, 2008, 624). More recently, the relevance of this goal has been invigorated by the enormous growth of the internet, which now documents a vast ocean of knowledge and digital traces of human behaviour. The lion's share of this information is stored in natural language, so to use this information on a large scale, we need systems that can extract the facts and opinions that humans encode in the semantic relations between words.

Needless to say, making a computer understand human language is a highly ambitious challenge, that is far from close to completion. Luckily, the task becomes much easier if the goal is to extract more specific semantic relations, that are useful for specific problems. Current methods for automatic content analysis in communication research could already be greatly improved if we could just extract source-quote and subject-predicate clauses, to analyze who did what and according to whom. The main purpose of the [package] package is to provide a toolkit for extracting specific types of semantic relations from text, using a rule-based approach in which the researcher defines the patterns underlying these relations.

In this purpose [package] shares common ground with *semantic role labeling* (SRL) systems (Carreras & Màrquez, 2005; Palmer, Gildea, & Xue, 2010). SRL can be described as a field of techniques for extracting "'who did what to whom', 'when', and 'where' " (He, Lee, Lewis, & Zettlemoyer, 2017, 473) from a text. Here "who *did* what" often refers to any type of verb phrase, including actions such as "saying", "being" and "having". The outcome of an SML system is that these semantic roles (who, what, whom, etc.) are automatically extracted from a text. For example, in the sentence "Bob gave the dog a bone", we could say that "Bob" (who) "gave a bone" (what) to "the dog" (whom). Semantic roles can be general or specific. The role *who* can more specifically be labeled as a *source* if the sentence is a quote ("who said what") or the *holder* of an opinion in opinion mining ("who loves what").

Present state-of-the-art SRL systems mostly rely on supervised machine-learning approaches. In particular, deep neural network models seem very promising, with several studies showing highly competitive resuls on the CoNLL-2005 (Carreras & Màrquez, 2005) and CoNLL-2012 (Pradhan et al., 2013) shared tasks for SRL[1]. The main advantage of using machine learning is that the model can learn to predict semantic roles based on complex combinations of input data, that would be difficult to program by hand. The deep neural networks that are currently on the rise have shown great boons for natural language processing tasks that go beyond the bag-of-words assumption (Goldberg, 2017), making notable progress in complex tasks such as sentiment classification (Nakov et al., 2016; Tang, Qin, & Liu, 2015). Recent studies even show good results in SRL models that do not include syntactic input (i.e. do not require grammar parsers), which suggests that deep learning models can independently learn the syntactic information that is required for extracting semantic roles (He et al., 2017; Zhou & Xu, 2015).

The main disadvantage of supervised machine learning methods for SRL is that they require a substantial amount of training data, that has to be meticulously created by hand. The training data that is currently available is mostly developed for English, and developed with the goals of AI and linguistic research in mind. In particular, one of the problems of most SRL systems is that they aim to identify all possible semantic roles in a text. This scope makes sense if the end goal is to create a model that can understand human language, but it sacrifices performance on the more specific types of semantic relations that would be of interest for communication research, because "given the large variety of possible actions and roles [...] systems based on these manually crafted resources almost always have data scarcity problems" (Van Atteveldt et al., 2017, 209). In this regard, the advantage of a rule-based approach is that rules can be developed for specific research interests in specific contexts, such as quotes by politicians, attacks by certain countries, or sentiment expressed about specific organizations. As a starting point, general rules can be developed that perform reasonably well for general types of semantic relations (e.g., "who did what", "who said what"), as demonstrated in the *Extracting quotes and clauses* section. This makes it possible to combine efforts of researchers in defining strong general rules, that can be tailored for specific use cases.

The rule-based and supervised machine learning approach can also be used in tandem. Current performance tests with the rule-based method, such as reported by Van Atteveldt et al. (2017), show that for many sentences the semantic roles can be correctly identified. For the sentences that go wrong, either due to patterns that are unaccounted for in the rules or mistakes in the dependency data, mistakes can manually be corrected to develop a training set. One of the future plans for the [package] package is to implement an annotation tool

[1] Shared tasks are a format for researchers to 'compete' on a task, by trying to achieve better results on the same dataset. The CoNLL-2005 and CoNLL-2012 datasets contain human developed semantic role annotations for training a model, and a standard procedure with test data to compare how well different types of models perform.

for manually checking and correcting the annotations, as a computer-assisted approach for creating the training data that is needed to train a good machine learning model. In theory, a deep recurrent neural network, such as the deep highway biLSTM model used by He et al. (2017), could be able to learn some of the complex patterns in language correct roles that the predefined rules could not capture, and be used as an additional or independent annotation tool depending on performance.

Finally, it should be noted that there are also unsupervised approaches for extracting semantic relations from texts. One of the main goals of unsupervised systems is to enable very large scale extraction without human input that is suitable for highly heterogeneous corpora. For instance, for Yates et al. (2007) and Kok and Domingos (2008) an important criterion for the methods they propose is that they should scale to the size of the Web (or at least, the Web as it was in 2008). The price for this corpus agnostic, unsupervised and huge-scale approach is that the type of relations that are extracted are very broad, and it is not possible to tailor the extraction for specific use cases or to optimize it for a more specific corpus. Since the corpora in communication research are often precise and narrow, and of a moderate size where scaling problems are not a major issue, these unsupervised approaches are of less interest.

The dependency tree

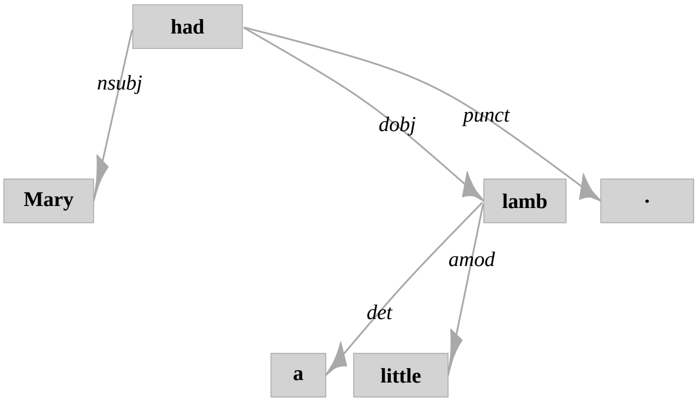
The method and tool discussed in this paper require that texts have first been preprocessed with a natural language processing (NLP) pipeline that includes a *dependency parser*. To apply the method, it is also necessary to have a basic understanding of what a *dependency tree* is. In this section we provide a brief introduction and show how to apply the spaCy pipeline (Honnibal & Johnson, 2015) to preprocess texts from within R.

Dependency-based syntactic parsing is a method for extracting the syntactic structure of a sentence, that outputs the sentence as a dependency graph (see e.g., Chen & Manning, 2014; Manning et al., 2014). A dependency graph can be defined as a set of labeled dependency relations between the words of a sentence (Kübler, McDonald, & Nivre, 2009, 12). It is a tree shaped graph in which each word can have one inward *relation* from a *parent* node, and multiple outward relations to *children*. A common (though not universal) property of a dependency graph is that a word can only have a single parent. One word is the *ROOT* of the tree, and thus does not have a parent. Furthermore, the dependency tree does not have cycles, meaning that a word is never directly or indirectly dependent on itself. For the method discussed in this paper these properties are assumed to be the case. For a strict definition of dependency graphs and a discussion of these properties, see chapter 2 of (Kübler et al., 2009).

An example of a dependency tree for the sentence "Mary had a little lamb." is shown in Figure 1. Here "Mary" has a *nsubj* relation with its parent "had", to denote that Mary is the *nominal subject* of the verb. The direct object (*dobj*) of the verb is "lamb". The node "little" is the adjectival modifier (*amod*) of "lamb", indicating that the lamb is little. The node "had" is the *ROOT*, and does not have a parent. Note that each node aside from the ROOT has exactly one inward arrow from a parent, which can be assumed to always be the case (the single-parent property).

The single-parent property conveniently allows the graph to be presented as a rectangular data frame in which each row is a unique token (also see the CoNLL-U format;

Figure 1. Example of a dependency graph.



Nivre et al., 2016). In Table 1 the output of the spaCy pipeline is shown as a data frame, which is also how it is produced in R using the spacyr package[2] (Benoit & Matsuo, 2017). The *parent* column contains the token_id of the parent token, and the *relation* column contains the type of relation of the token to its parent. In other words, the inward edge of the dependency graph is presented as a token attribute—even though it is formally an edge between tokens.

Thus, the single-parent property makes it possible to consider each unique token as a unique node in the dependency graph. From hereon, when we talk about nodes in the dependency graph, we will often refer to them simply as *tokens* to avoid confusion.

Table 1
Output of the spaCy parser for the sentence in Figure 1

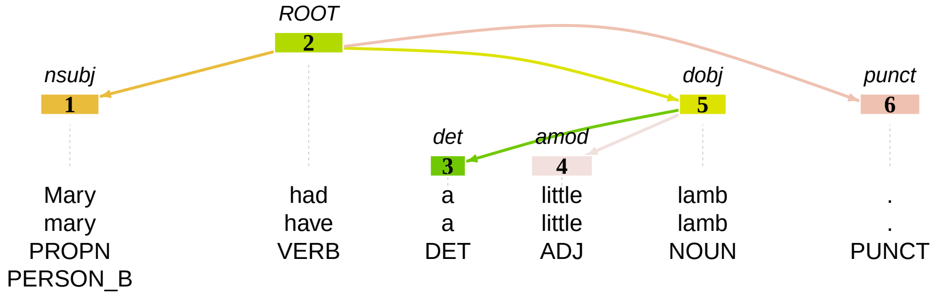
sentence	token_id	token	lemma	pos	parent	relation	entity
1	1	Mary	mary	PROPN	2	nsubj	PERSON_B
1	2	had	have	VERB		ROOT	
1	3	a	a	DET	5	det	
1	4	little	little	ADJ	5	amod	
1	5	lamb	lamb	NOUN	2	dobj	
1	6	.	.	PUNCT	2	punct	

To develop queries for the dependency tree, and to understand how the tree can be manipulated by using certain heuristics, it helps to visualize the tree in a way that conveys both the data frame structure and the graph. In the [package] package we implemented a function to produce the visualization shown in Figure 2. By default, the dependency graph is shown with the token_id as the node, and all other token attributes are included as labels at the bottom. In the current example, the labels at the bottom show the values of the token, lemma, pos (part-of-speeh tag) and entity columns.

Notably, the dependency relation is not shown on the edge between tokens, but in

[2] We changed some of the column names, which are not universal across parsers, to the names used in the [package] package. Specifically, we changed "sentence_id" to "sentence", "head_token_id" to "parent", and "dep_rel" to "relation". Furthermore, spaCy indicates a root with a parent relation to itself, but here we set the parent of the root to missing (NA in R).

Figure 2. The [package] visualization style for token data with dependency columns



italics above the token_id. This has two advantages for the current application. Firstly, it ensures that the relation is always clearly readable and easy to find. Secondly, it emphasizes that the relation between a token and its parent is in the data frame structured as an attribute of the token (i.e. in the *relation* column, on the same row as the token). When querying the token data, it helps to think of the dependency relation as a token attribute. Just keep in mind that in dependency theory the relation is more formally an edge attribute.

Syntactic parsing in R and preparing data for [package]

There are multiple NLP pipelines that can perform syntactic parsing. Among the most popular and best performing ones are the coreNLP (Manning et al., 2014) and spaCy (Honnibal & Johnson, 2015) pipelines. These pipelines primarily focus on English text, but also support some other languages, albeit with varying degrees of performance, and not always including a dependency parser. For some languages, there are also dedicated NLP pipelines that could provide better results. For Dutch, the Alpino parse (Van der Beek, Bouma, Malouf, & Van Noord, 2002) shows strong results (Van Atteveldt, Kleinnijenhuis, & Ruigrok, 2008), and for German there is the Parzu parser (Sennrich, Volk, & Schneider, 2013). However, these parsers can be more difficult to install, and do not have R bindings. To make it easier to use these parsers, and to use them directly from within R, the (masked for review) Python module can be use to set up a client-server based pipeline, that can be accessed in R with the (masked for review) package.

For the code in this paper we use the spaCy pipeline. Although spaCy runs in Python, the spacyr package in R (Benoit & Matsuo, 2017) provides bindings to easily parse texts from within R [3]. Once installed, the spacyr package can initialize spaCy and parse text with a few lines of code.

```
library(spacyr)
spacy_initialize()

tokens = spacy_parse('Mary had a little lamb.', dependency=T)
```

The *tokens* data.frame that is created here contains the tokens as presented in Table 1. To prepare the tokens for use in the [package] package, it first has to be converted to

[3] See https://cran.r-project.org/web/packages/spacyr/readme/README.html for instructions on how to install and use spacyr.

a *tokenIndex* class, which standardizes the data for the output of different parsers, and prepares the data for faster and more memory efficient processing. Specifically, it is a light wrapper around a *data.table* (Dowle & Srinivasan, 2017), which is an alternative to R's standard data.frame that enables fast binary search and can update the data by reference. This makes it possible to quickly navigate the dependency graph and perform queries. Once the *tokenIndex* is made, the tree can be visualized with the *plot_tree* function.

The following code creates the tokenIndex[4], and uses the dependency graph visualization feature, as shown in Figure 2.

```
library(MASKED)
tokens = as_tokenindex(tokens)
plot_tree(tokens)
```

In the current example only one sentence is parsed, but all functions in [package] are designed to be able to process multiple sentences and documents in the same data.frame. The plot_tree functions by default only plots the first sentence in the data, but other sentences can be specified in the function arguments.

Querying the token data and dependency tree

The rule-based method for extracting semantic relations from texts based on dependency-based syntax data can essentially be described as a special application of querying the token data and dependency tree. Given a dependency tree, and additional token attributes such as lemma and part-of-speech tags, queries can be developed to select specific sets of tokens that are syntactically related. In addition, certain heuristics can be applied to select larger parts of sentences based on assumptions about dependency trees. In this section we will first present how these queries and heuristics can be applied in [package], starting with a simple example.

Building a query with the *tquery* function

To explain the basics of creating queries in [package] we will build the following query:

- select all tokens where the pos tag is VERB, and that have a child with an *nsubj* relation
- give the verb token in this pattern the label "predicate"
- give the child token in this pattern the label "subject"

In [package] we can build this query (not yet execute it) with the *tquery* function (short for *tree query*). To select tokens based on certain attributes, we can pass named arguments, where the name of the argument refers to the column in the token data.frame, and the value is a character vector with lookup values. To develop a query to select all tokens where the pos tag is VERB, we use the following line of code.

[4] It is not strictly necessary to call *as_tokenindex* if a known format (such as the spacyr output) is used, since it is automatically called within [package] functions. In the remainder of this document we will therefore skip this step. Note, however, that it is more explicit and can in certain cases be much more efficient

```
tq <- tquery(pos = "VERB")
```

Note that this code only creates the query, here named *tq*. The query is not yet applied to the data. These steps are intentionally kept separate for the sake of clarity, because queries can become quite complex.

Now, to include in this query that the token should have a child with a *nsubj* relation, we can pass the *children* function as an argument to *tquery*. The *children* function can itself be seen as another *tquery*, and the syntax for selecting specific children tokens is identical. Thus, to include the condition that the verb should have a child with the *nsubj* relation, the query becomes:

```
tq <- tquery(pos = "VERB",
             children(relation = "nsubj"))
```

The current query only looks whether the pattern occurs. To use tokens that are selected, the user needs to *label* them. Labels can be assigned by using the *label* argument. In our example we wanted to label the verb token as “predicate”, and the child tokens as “subject”. The following code creates the full example query.

```
tq <- tquery(pos = "VERB", label = "predicate",
             children(relation = "nsubj", label = "subject"))
```

In the **Advanced queries** section below we will discuss more advanced features of the *tquery* function. In the next section we will first show how the basic query defined here can be applied.

Annotating the token data using a tquery

One of the most useful applications of a *tquery* is to use it to annotate the token data. By this we mean that the labels assigned to the tokens in the *tquery* are added to the token data.frame. This can be done with the *annotate* function, which takes as input the token data, the name for the new column, and one or multiple tqueries. In the following example we use the tquery defined above and use it to annotate the token data for the sentence “Mary had a little lamb.”, and call the new column “annotation”. By passing the query as a named argument, the name is included in the annotation id. Here we use the name “active”, because the query looks for the active subject.

```
tokens = annotate(tokens, "annotation", active = tq)
```

Table 2 shows the new token data with the additional annotation columns[5]. We see that three annotation columns have been added:

[5] Some columns have been omitted to fit the table on the page

Table 2
Token data.frame after applying a tquery with annotate heuristic

token_id	token	parent	relation	annotation	annotation_id	annotation_fill
1	Mary	2	nsubj	subject	active#text1.1.2	0
2	had		ROOT	predicate	active#text1.1.2	0
3	a	5	det	predicate	active#text1.1.2	2
4	little	5	amod	predicate	active#text1.1.2	2
5	lamb	2	dobj	predicate	active#text1.1.2	1
6	.	2	punct	predicate	active#text1.1.2	1

- **annotation** shows the label, as assigned in the *tquery*

- **annotation_id** contains the unique id of the query match. It is possible that a single sentence (and thus single dependency tree) contains multiple matches. The id enables us to see which set of labels belong together. In the current case, which "subject" is related to which "predicate". The id is a concatenation of the document id, sentence, and the first labeled token in the tquery, which makes them globally unique. If a query is given a name when applied (e.g., in the *annotate* function) this name is included in the id.

- **annotation_fill** shows whether a token was directly selected by the query (fill = 0) or whether the token is a child of one of the selected tokens (fill > 0), in which case the fill indicates how far down the tree the child was found.

The fill column requires elaboration. The default behavior of a *tquery* is that if a token is selected and given a label, all of its children that are not yet labeled are automatically also selected and given the same label. In the example, this means that all the children of "had" are labeled as "predicate", except for "Mary" which is already labeled as "subject". This can most clearly be seen in Figure 2. Note that for "lamb" the *annotation_fill* value is 1, because it is a direct child of "had", whereas "little" has the value 2, because it is two levels below "had".

We call this the *fill* heuristic, because it enables us to easily fill a branch of a dependency tree by querying the parent tokens. It is based on the idea that in a dependency tree the children of a token contain information about itself (i.e. the children depend on the parent). Thus, if tokens in a dependency tree are selected with a query, the fill heuristics ensures that all information about these tokens is included. We call this a *heuristic*, because it should only be seen as a practically useful trick based on a general assumption about how the semantic relations between words are structured in a dependency tree. As we will show below, there are cases where using the default *fill* settings does not make sense, and we offer additional tools for dealing with these cases. Nevertheless, heuristics such as this make it much easier to extract useful semantic relations. Furthermore, since the accuracy of dependency parsers is not perfect, and since many texts do not always adhere to common or even correct syntax, heuristics can sometimes better deal with messy data.

Developing advanced queries

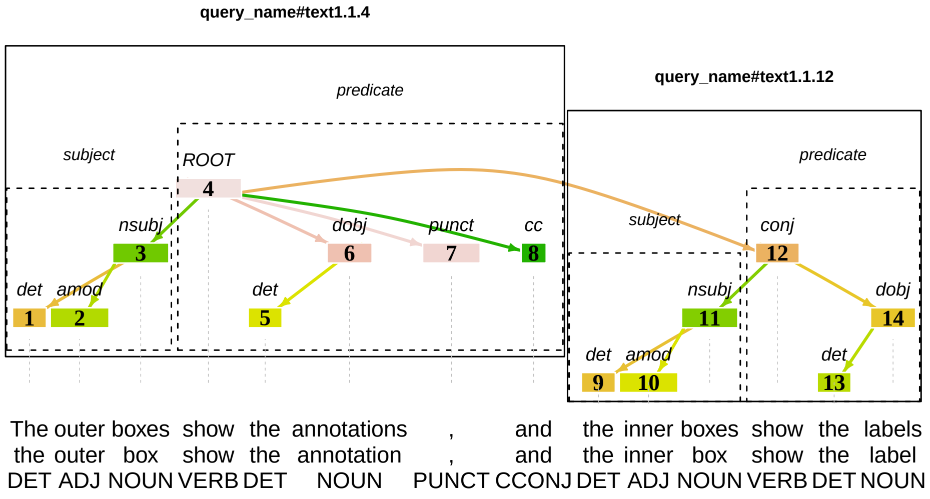
The basic query created above showcases the general logic of the *tquery* format and behaviour. Here we discuss several more advanced features for creating better queries.

Testing queries

To develop and test queries, the *plot_tree* function can display the annotations over the dependency tree. The following code shows a workflow based on magrittr piping[6], that annotates a token and plots the result. This makes it easier to test a query for different sentences and update the query. The output is shown in Figure 3.

```
tokens = spacy_parse('The outer boxes show the annotations
                      and the inner boxes show the labels', dependency=T)
tq <- tquery(pos = "VERB", label = "predicate",
             children(relation = "nsubj", label = "subject"))

tokens %>%
  annotate('annotation', query_name = tq) %>%
  plot_tree(annotation = 'annotation')
```

Figure 3. Using the *plot_tree* function to display annotations over the dependency tree.


query_name#text1.1.4
query_name#text1.1.12
predicate
subject
ROOT
4
nsubj
dobj
punct
cc
3
6
7
8
det
amod
det
1
2
5
predicate
subject
conj
12
nsubj
dobj
11
14
det
amod
det
9
10
13
The outer boxes show the annotations , and the inner boxes show the labels
the outer box show the annotation , and the inner box show the label
DET ADJ NOUN VERB DET NOUN PUNCT CCONJ DET ADJ NOUN VERB DET NOUN

Querying deeper patterns

Above we showed how the *children* function is used in a *tquery* to select specific children of a token. More generally, a *tquery* can be used to select any pattern in a dependency tree. Multiple *children* functions can be nested side by side, and *children* can also be nested

[6] The *magrittr* package is a widely used package for piping in R, that uses the *x %>% f()* syntax to use *x* as the first argument of the function *f*. For more detailed instructions, please consult the *matrittr* package documentation (Bache & Wickham, 2014)

in *children* to look deeper down the tree. To look upward in the tree, the *parents* function can be used in the same, recursive way.

For syntactical clarity, it is recommended to start each nested query on a new line. In certain script editors, including in RStudio, this will also automatically jump in deeper levels further to the right. This allows deep patterns in the dependency tree to be specified in a way that is relatively easy to interpret, as seen in the following example.

```
family <- tquery(label = "node",
                 parents(label = "parent",
                         parents(label = "grandparent")),
                 children(label = "noun_child", pos = "NOUN"),
                 children(label = "remaining_child"))
```

Note the behaviour of the two children. The “noun_child” has the condition that the part-of-speech tag should be “NOUN”, whereas the “remaining_child” does not have a condition. The same token cannot (and should not) be assigned twice in the same pattern. In this case, “remaining_child” will thus only match tokens for which the part-of-speech tag is not “NOUN”.

Tailoring the fill heuristic

By default, the fill heuristic includes all direct and indirect children of a selected and labeled token that have not yet been labeled. Alternatively, the fill heuristic can be tailored to only include children that meet certain criteria. This can be done by nesting the *fill* argument in a *tquery*. The following example only labels fill children that are proper nouns.

```
tquery(relation = "nsubj", label = "subject",
       fill(pos = "propn"))
```

The *fill* function is in fact a special type of the *children* function, in which by default the *depth* argument, that controls how far down the tree the query should look, is set to *Infinite*, and the *req* argument is set to FALSE to indicate that the fill tokens are optional (i.e. not required).

An additional argument in the *fill* function is *connected*, which determines what should be done if certain of the children tokens do not meet the specified condition. If *connected* is TRUE, fill will stop traveling down the tree the moment it reaches a child that does not meet the condition, thus ignoring the remainder of the branch, even if in this branch there are children that do meet the condition. If *connected* is FALSE, all children that meet the condition will be labeled. This can for instance be used in combination with the NOT operator to cut of relative clauses.

```
tquery(relation = "nsubj", label = "subject",
       fill(NOT(relation = "relcl"), connected = TRUE))
```

In some cases it can be useful to turn of the fill heuristic entirely. This can be done in two ways. Firstly, within a query the fill heuristic for a specific token can be turned off by setting the fill argument to FALSE. Secondly, functions that apply a query, such as *annotate* also have a fill argument, that can be set to FALSE to ignore all the fill heuristics within a query.

Chaining queries

Queries can be chained together, so that in cases where a token would match more than one query, only the first match is used. For example, in the following codeblock we define two queries, for *passive* and *active* subject-predicate relations. In these subject-predicate relations, we always consider the actor that does something as the subject (who, in “who does what to whom”).

```
passive = tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("agent"), label = "subject"))

active =  tquery(pos = "VERB*", label = "predicate",
                 children(relation = c("nsubj", "nsubjpass"), label = "subject"))
```

The following example sentence contains both an active and passive subject-predicate relation. The tokens and dependency tree is shown in Figure 4

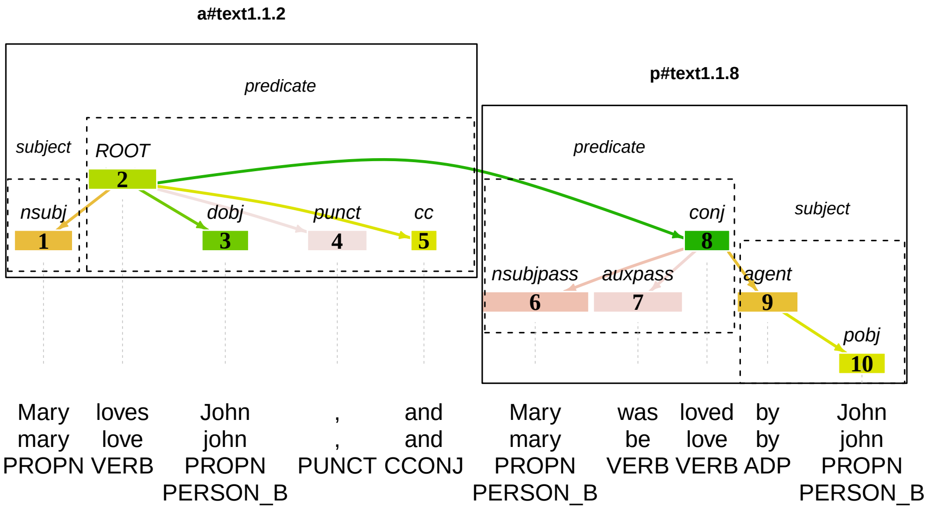
```
tokens = spacy_parse('Mary loves John, and Mary was loved by John', dependency=T)
plot_tree(tokens)
```

In the first part, “Mary loves John”, Mary is the subject that loves John. In the second part, “Mary was loved by John”, Mary is the passive subject that is loved, and not the one that *does* the loving. Thus, for the subject-predicate relation we want to label John as the subject that loves Mary. However, the *active* query would match both parts, and incorrectly label Mary as the subject in both sentences[7]. This is prevented by chaining the queries together, with the more specific *passive* query applied first. To chain queries, we simply pass multiple *tqueries* to the *annotate* function, with the order in the list determining the order in the chain. It is recommended to use named arguments, in which case the name will be used in the annotation id. This is convenient for keeping track of which query was used for which annotation. Instead of using named arguments, a named list can also be provided.

```
tokens %>%
    annotate("subject", pas=passive, act=active) %>%
    plot_tree(annotation = "subject")
```

[7] In the current case the *active* query did not have to allow the "nsubjpass" relation for the "subject" child. However, as discussed in the *Reshaping the dependency tree* section, there can be cases where this is necessary due to heuristics for dealing with argument drop.

Figure 4. A dependency tree with an active and passive subject-predicate relation



The output is shown in Figure 4. Note that chaining only applies to direct matches, before using the fill heuristic. In the output the fill heuristic would also have included the second part of the sentence as part of the predicate for the first query, because token 12 is a child of token 4. But if another query matches the fill children, the fill will stop where the query begins. To summarize, direct matches always have precedence over fill matches, and for direct matches precedence is determined by the position in the chain.

Alternatively, we can use magritrr piping to chain queries together. As long as the same column name is used to store the annotations, the earlier annotations will be kept (unless the overwrite argument is used to reset the annotation column).

```
tokens = tokens %>%
  annotate("subject", pas=passive) %>%
  annotate("subject", act=active)
```

We could also have solved this specific issue by making the active query more specific. In general, there will be multiple ways to deal with problems with different pros and cons, and the [package] aims to facilitate multiple solutions. The advantage of chaining is that it offers a convenient approach for combining many queries without having to think about accidentally having them overlap. The positions of queries in the chain can be used to systematically improve precision and recall. Very specific queries tend to have high precision but low recall: the subjects that are found are likely to be correct, but there will be many false negatives (i.e. subjects in the data that are not found). Conversely, very broad queries tend to have high recall but low precision: many of the subjects will be found, but there will be many false positives (i.e. matches that are not real subjects). By putting specific queries at the start of the chain, the specific cases can be dealt with to improve precision. Broader queries further down the chain can then focus on the remaining cases to improve the recall without accidentally capturing the cases that are already labeled.

Using flags and wildcards in lookup values

The *tquery* function, and the *children* and *parents* functions, can select nodes by passing vectors of lookup values. As shown above, this is done by passing named arguments, where the name refers to a column in the token data, and the value is a vector of lookup values. By default, these values need to be an exact, case sensitive match. Alternatively, certain wildcards can be used, and flags can be used to make the search case insensitive, use regular expressions.

Two special characters in lookup values are treated as wildcards. An asterisk (*) wildcard can be used as a placeholder for any number of undefined characters, and a question mark (?) can be used as a placeholder for a single undefined character. For example, some parsers use more detailed part-of-speech tags, such as the Penn Treebank (Santorini, 1990) that denotes different verb forms and tenses (VB, VBD, VBG, VBN, VBP, VBZ). To select all verbs, the lookup value VB* can be used. The following queries are thus identical for tokens where the Penn Treebank part-of-speech tags are used.

```
tquery(pos = "VB*", label = "verb")
tquery(pos = c("VB", "VBD", "VBG", "VBN", "VBP", "VBZ"), label = "verb")
```

Flags can be added with special suffixes to the argument name. The suffix is a double underscore, followed by a character that selects the flag. There are currently three types of flags:

- **I** sets **case insensitive** search
- **R** sets **regular expression** search, using the implementation in base R (see ?regex).
- **F** sets **fixed** search, meaning exact matches with wildcards disabled.

```
tquery(token__R = "[0-9]")    ## Use regex to select tokens that contain a number
```

Note that using wildcards and the I and R flags makes the lookup operation slower. The columns that are used in the query are indexed the first time they are used to enable fast binary search. If a wildcard, regex or case insensitive search is used, the lookup terms are first matched to the vocubulary to find the exact terms used in the column, which are then used for the binary search.

Boolean operators: AND, OR and NOT

By default, if multiple lookup columns are used, the lookup values need to match for all of the used columns. In other words, multiple lookup columns are by default connected with an AND operator. For example, consider the following *tquery*:

```
tquery(pos = "VERB", lemma = "say")  ## pos = "VERB" AND lemma = "say"
```

This query looks for tokens where the part-of-speech tag is "VERB", *AND* the lemma is "say". Alternatively, it is possible to use the OR and NOT operators. In [package] the boolean operators *AND*, *OR* and *NOT* are functions that can be nested in the *tquery* function[8]. The lookup terms can then be passed to these function to use that boolean operator. The NOT operator can in particular be useful. The following query uses the *NOT* function to find all tokens where the lemma is NOT "say".

```
tquery(NOT(pos = "say"))  ## NOT pos = "say"
```

This can be combined with other operators. The following query looks for tokens where the part-of-speech tag is "VERB", AND the lemma is NOT "say". This is for instance useful if we have a list of verbs that indicate a speech act (e.g., "say", "state", "argue"), and the goal is to separately look for speech acts and other types of verbs (as used below in the extracting clauses section).

```
tquery(pos = "VERB", NOT(lemma = "say"))  ## pos = "VERB" AND NOT lemma = "say"
```

Excluding certain children or parents

In some cases, it is necessary to state in a query that a node cannot have certain parent or children. For this, the special *not_parents* and *not_children* functions can be used. These have the same lookup style as the *parents* and *children* functions, that can be used to exclude specific types of parents or children. If no lookup values are used, the token should not have parents or children at all.

In certain parsers (e.g., Alpino for Dutch) the dependency relation of a subject does not indicate whether the subject is active or passive[9]. For a query that looks for active subjects, we can then state that the verb cannot have a child that indicates that the verb is passive. In the following example we look for subjects of which the parent VERB does not have a child that is a passive auxiliary[10] ("auxpass").

```
tquery(relation = "nsubj", label = "direct_subject",
       parents(pos = "VERB",
               not_children(relation = "auxpass")))
```

[8] One of the future development goals is to implement a more conventional syntax for using boolean operators in *tquery*.

[9] As discussed below, this can also result from reshaping the parse tree with argument drop heuristics

[10] These are words such as "has", that can be the child of a verb such as "been" to create the passive "has been"

Using optional tokens

The default behaviour of *tquery* is that all parts of the query have to be found. This can be cumbersome if there are tokens that can be considered optional. For example, in a set of queries for finding clauses, we might want to label a direct object (*dobj*) if a direct object exists, but also select the pattern if there is no direct object. This would require every query to be duplicated, with the more specific queries (that include the direct object) positioned earlier in the chain. A more elegant and efficient solution is then to make the direct object token optional, by setting the *req* (required) argument to FALSE.

```
tquery(pos = "VERB", label = "predicate",
       children(relation = "nsubj", label = "subject"),
       children(relation = "dobj", label = "object", req = FALSE)
```

Extracting quotes and clauses

Using the *tquery* and *annotate* functions, we can replicate the queries for extracting quotes and clauses presented by Van Atteveldt et al. (2017). For this example we developed the queries for the spaCy pipeline output, whereas Van Atteveldt et al. used CoreNLP (Manning et al., 2014). This requires some minor alterations, and we have not extensively tested whether the validity of these results is the same. The purpose of this example is only to demonstrate the recommended workflow.

Extracting quotes

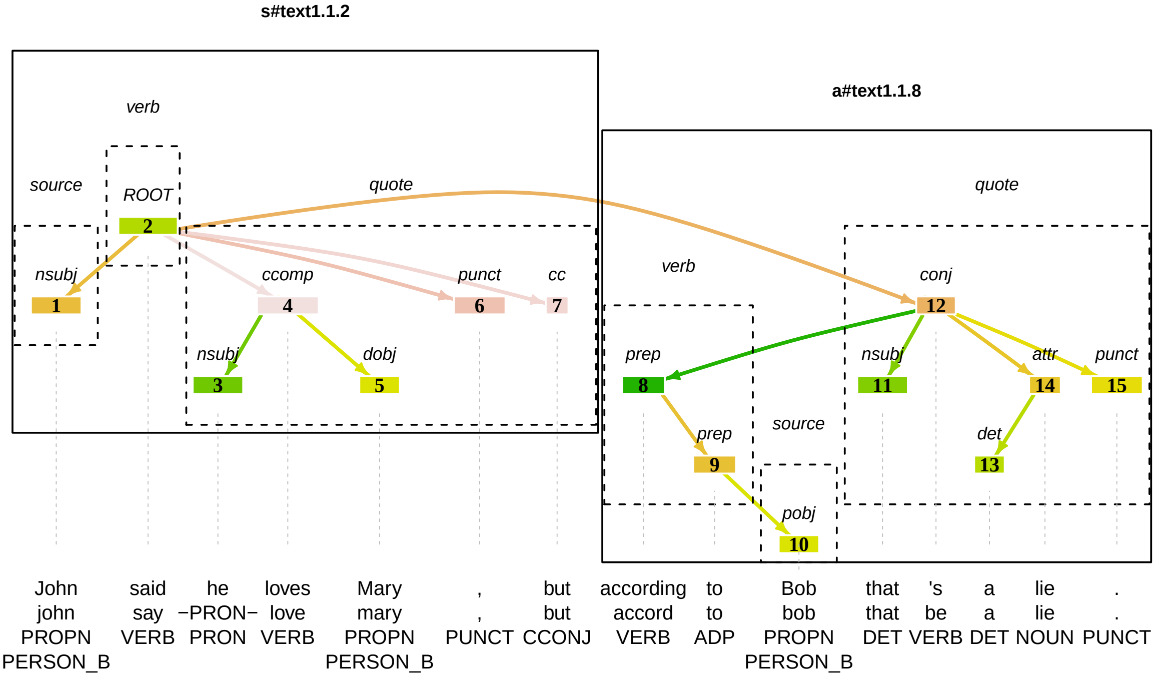
Van Atteveldt et al. (2017, pp. 210-211) describe two simple patterns for extracting quotes. The first pattern looks at the use of explicit speech verbs (e.g., say, state, argue) and the subject or passive agent of this verb. There is no field in the token data that indicates whether a verb is a speech verb, so instead we use a character vector of speech verbs in lemma form in the query (the specific list of speech verbs used by Van Atteveldt et al. is provided in their online appendix). This can also be used to specify what type of speech verbs are used for a particular study, or to make separate queries for different types of speech verbs. The second pattern deals with the common “according to” case. In CoreNLP, this is indicated with a special type of relation (*nmod:according_to*), but this is not the case in spaCy, so instead we look for the lemma “accord” with the child “to”, and from here specify the rest of the pattern. In the following code, we first make a vector of speech verbs, and then create the two queries.

```
say_verbs = c("say", "tell", "show", "admit", "exclaim")    ## abbreviated

speechverb = tquery(lemma = say_verbs, label = "verb",
                    children(relation = c("nsubj", "agent"), label = "source"),
                    children(label = "quote"))
according = tquery(lemma = "accord", label = "verb",
                   children(relation = "prep",
                            children(relation = "pobj", label = "source")),
                   parents(label = "quote"))
```

In the following code we apply the queries. An important detail is that we should chain these queries together, with the *according* query going first (as explained in the *Chaining queries* subsection above). The reason is that the *speechverb* query now very greedily labels the "quote" tokens. To demonstrate this, we show the results for a sentence that contains both types of quotes. We give the queries names (*a* for *according* and *s* for *speechverbs*) to see which query is applied where.

```
tokens = spacy_parse("John said he loves Mary, but according to Bob that's a lie.",
                     dependency=T)

tokens %>%
  annotate("quotes", a = according, s = speechverb) %>%
  plot_tree(annotation = "quotes")
```

Figure 5. Quote extraction based on two queries.



The output, as shown in Figure 5, contains two annotations. The first annotation is based on the *speechverb* query, as can also be seen by the s# in the annotation_id (*s#text1.1.2*) that is shown above the box. This annotation distinguishes that "John" (source) "said" (verb) "he loves Mary" (quote), and in addition contains as harmless collateral some punctuation and "but" in the quote. The second annotation, based on the *according* query (*a#text1.1.8*), distinguishes that "that's a lie" (quote) "according to" (verb) "Bob" (source).

Extracting clauses

By extracting clauses, Van Atteveldt et al. (2017) more specifically refer to the extraction of subject-predicate relations. The procedure is described in four steps (p. 211).

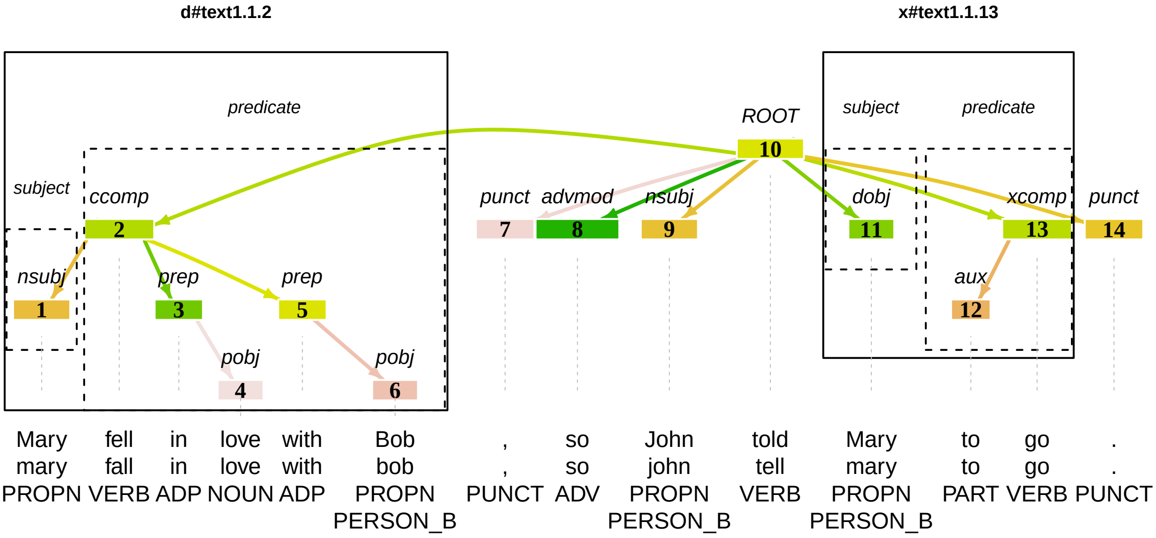
The first step (a) states that "The most common pattern is the straight-forward case where a (non-speech) verb has a subject (or passive agent)". This is essentially the same as the first query for quote extraction, but in this case the verb has to be anything but a speech verb. Here we can use the NOT operator to exclude the list of speech verbs that is used for the quote extraction. The second step (b) describes a query for dealing with causal complements. As an example the sentence "John told Mary to go" is given, in which Mary is the direct object of the verb "told", but also the predicate subject of the verb "(to) go". The queries for step a and b be defined as follows.

```
direct = tquery(pos = "VERB*", NOT(lemma = say_verbs), label = "predicate",
                children(relation = c("nsubj", "agent"), label = "subject"))
xcomp = tquery(pos = "VERB*",
          children(relation = "xcomp", NOT(lemma=say_verbs), label = "predicate"),
          children(relation = "dobj", label = "subject"))
```

To see these queries in action, we annotate the sentence "Mary fell in love with Bob, so John told Mary to go", that contains both types of clauses.

```
tokens = spacy_parse("Mary fell in love with Bob, so John told Mary to go.",
                     dependency=T)

tokens %>%
  annotate("clauses", d=direct, x=xcomp) %>%
  plot_tree(annotation = "clauses")
```

Figure 6. Clause extraction queries part one.


d#text1.1.2
x#text1.1.13
predicate
subject
ROOT
subject
predicate
ccomp
punct
advmod
nsubj
dobj
xcomp
punct
nsubj
prep
prep
aux
pobj
pobj
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Mary fell in love with Bob , so John told Mary to go .
mary fall in love with bob , so john tell mary to go .
PROPN VERB ADP NOUN ADP PROPN PUNCT ADV PROPN VERB PROPN PART VERB PUNCT
PERSON_B PERSON_B PERSON_B

The result is shown in Figure 6. The first annotation is based on the *direct* query (*d#text1.1.2*) and the second on the *xcomp* query (*x#text1.1.13*).

Van Atteveldt et al. describe two more steps. Step c describes how to deal with nominal actions, such as “the Israeli invasion”, which is not a verb phrase, but implies that Israel performs the action of invading. In the demo case presented by Van Atteveldt et al., a list of words that describe aggressive actions is used. The query then looks for cases where this action has a child with a *poss* (possession modifier) relation, which implies that the action is in this case a noun that is possessed.

```
agg_action = c(’violence’,’attacks’)  ## abbreviated

nominal = tquery(lemma = agg_action, label = "predicate",
                 children(relation = "poss", label = "subject"))
```

Importantly, this query is used in a special way. It is only applied to tokens that have not yet been labeled by one of the previous clause queries, including the fill children. The rationale is similar to chaining queries, in the sense that it is used to improve the recall by casting a wide net, without harming the precision of the clause queries. It is more rigorous than regular chaining, because regular chaining only blocks labeling of tokens that are directly matched by a previous query, but in this case the fill children are also blocked.

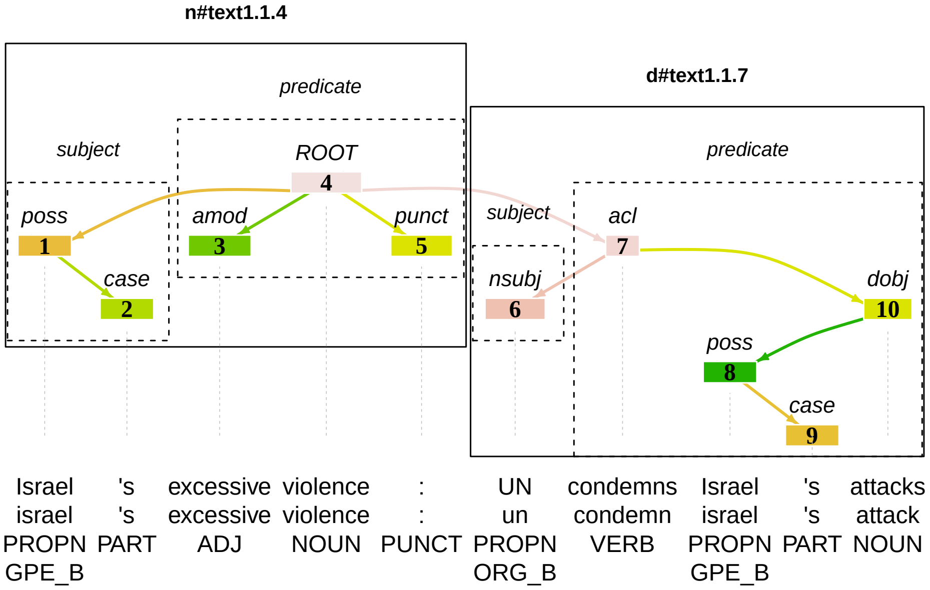
The following example shows a sentence with both an isolated nominal action, and a nominal action that occurs within the fill children of a clause. To achieve the more rigorous form of chaining, we add the annotation for the nominal action with a separate call to the *annotate* function, in which the *block_fill* argument is set to TRUE. This prevents all tokens that have been labeled earlier in the chain, including the fill children, to be preserved.

```
tokens = spacy_parse("Israel’s excessive violence: UN condemns Israel’s attacks",

tokens %>%
  annotate(’clauses’, d = direct, x = xcomp) %>%
  annotate(’clauses’, n = nominal, block_fill = T) %>%
  plot_tree(annotation = ’clauses’)
```

The results are shown in Figure 7. Here the rationale of the rigorous chaining also becomes more apparent. In the first annotation, the nominal action “Israel’s (excessive) violence” stands on its own, whereas in the second annotation “Israel’s attacks” is the predicate of the clause “UN condemns Israel’s attacks”. It can be argued that in the first annotation the relation between Israel and the act of violence is more clearly made compared to the second annotation, where Israel’s act of violence is treated more as an object in itself. Nevertheless, this decision regarding how to deal with nominal actions clearly requires a judgment call from the researcher, which will also depend on the research question and type of data. The recommended approach for making this call would be to compare the results to human annotations, to measure which approach has the highest validity.

Finally, Van Atteveldt et al. describe step d, which is a solution for dealing with conjunctions and relative clauses. Conjunctions and relative clauses pose a problem for

Figure 7. Clause extraction for nominal actions, using the *block_fill* argument in *annotate*.


n#text1.1.4
predicate
subject
ROOT
4
poss
1
amod
3
punct
5
case
2
d#text1.1.7
predicate
subject
acl
7
nsubj
6
dobj
10
poss
8
case
9
Israel 's excessive violence : UN condemns Israel 's attacks
israel 's excessive violence : un condemn israel 's attack
PROPN PART ADJ NOUN PUNCT PROPN VERB PROPN PART NOUN
GPE_B ORG_B GPE_B

querying the dependency tree, because they introduce certain recursions and complications due to the tendency in human language to drop arguments when they are implied by syntactic conventions (Bender, 2013). Consider the sentence “Hamas fired rockets at Israel, killing 20 civilians”. This sentence contains two clauses, “Hamas fired rockets at Israel”, and “Hamas kill[ed] 20 civilians”. In the second clause, the subject is omitted (i.e. “killing” does not have a child with an nsubj relation), but it is clear that this subject should be “Hamas”.

The approach that Van Atteveldt et al. used to solve this is to write additional, longer queries that match the most common of such cases. In the example, “killing” is a child of “fired”, so the relation between “Hamas” and “killing” can be queried with a longer pattern that passes over “fired”. However, this approach has several limitations. Firstly, it requires the initial patterns to be copied and transformed into special versions, which requires more work and makes the queries more complex. Secondly, it does not deal with recursion, such as conjunctions in conjunctions—though admittedly, these should not occur very often in most types of texts. Thirdly, an inconvenient side effect of this approach is that certain tokens would have to be annotated multiple times, such as Hamas in the example being the subject in two separate annotations. This introduces a number of data issues that would better be avoided.

To deal with these cases more properly, [package] provides several generic tools for *reshaping* the dependency tree. The purpose of these tools is essentially to simplify the dependency tree, by applying certain procedures to deal with nested and recursive structures, and to copy arguments where they are implicitly present.

Simplifying the dependency tree

As introduced at the end of the previous section, a complication for querying dependency trees is that sentences often contain nested sentences, and arguments that are implicitly clear for human readers are often dropped. For example, the sentence “Bob and John ate bread and drank wine” already contains four clauses: “Bob ate bread”, “Bob drank wine”, “John ate bread”, and “John drank wine”. By first reducing text to these simpler components, the extraction of semantic relations becomes much easier.

The transformation from a complex sentence into more simple sentences can be performed by applying certain operations on the dependency tree (Siddharthan & Mandya, 2014, see e.g.,). Techniques for *text simplification* are often applied for the actual purpose of making texts easier to read for people with reduced literacy. For our purpose, the task is actually less complicated, because the goal is not to make text easier to read for people—which also requires thinking about correct and simple syntax—but only to simplify the dependency tree to make it easier to perform queries.

[package] contains a set of tools for simplifying the dependency tree, that build on the same syntax and heuristics for writing queries.

Using the reshape operations

The first step is to develop a *tquery* for selecting certain tokens. Then, the labels assigned to these tokens can be used to perform three general types of operations, using the following five functions:

- **mutate_nodes** can be used to change the node attributes, such as reassigning the parent or removing the parent and relation to make the node a ROOT. All of the nodes in the pattern matched by the query can be used to update each others information.
- **copy_nodes** can create a copy of a node. The fill children of a node can also be copied along, which makes it easy to copy entire branches.
  - **copy_fill** is a special application of *copy_nodes* that only copies the fill children.
- **remove_nodes** can remove nodes from the tree. The fill children are by default also deleted.
  - **remove_fill** is a special application of *remove_nodes*, that only removes the fill children.

The reshape operations are designed to be applied in a pipe. The first step in this pipe is to use the *select_nodes* function to apply a tquery on the token data. The pipe can then contain any of the reshape operations in any order. The following code shows a toy example.

```
tq = tquery(label = "verb",
            children(relation = "nsubj", label = "subject"))

tokens = select_nodes(tokens, tq) %>%
  mutate_nodes("subject", parent = verb$parent, relation = verb$relation) %>%
  copy_fill("parent", "subject") %>%
  remove_nodes("parent")
```

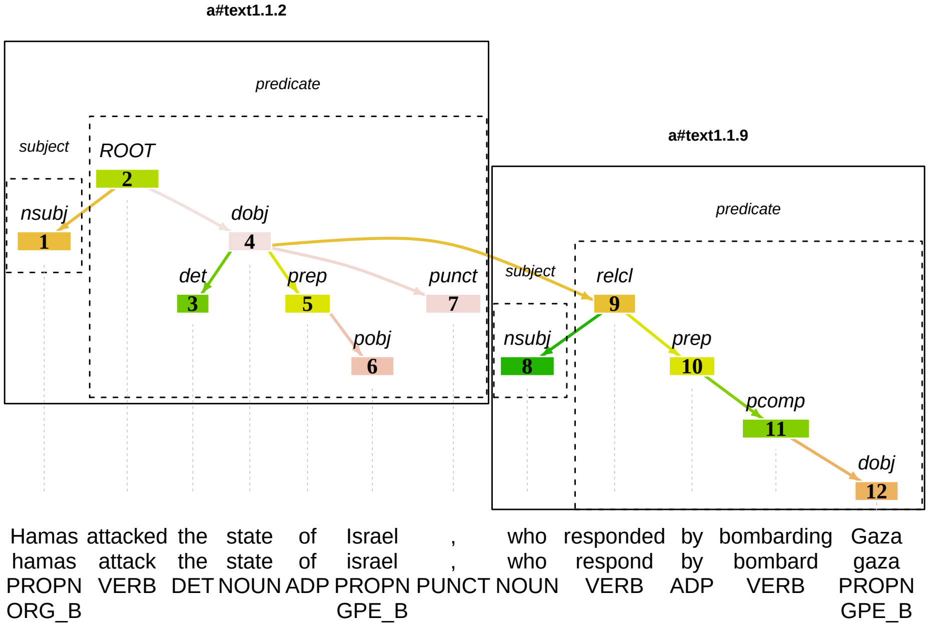
After selecting the nodes as specified in the query, the mutate_nodes function updates the attributes of the "subject" node. Using named arguments, any attribute can be mutated with a given value, and the attributes of other nodes can be used with the syntax *node$attribute*. In this case, the parent and relation attributes is replaced with the parent and relation attributes of the "verb" node. Next, the fill children of the "parent" node are copied to the "subject" node. Finally, the "parent" node is removed, together with its fill children (but not the fill children that have been copied to "subject").

Example application: isolating relative clauses

Consider the following sentence, that is an example used in Van Atteveldt et al. (2017)[11]. For reference, we also apply the clause extraction queries.

```
tokens = spacy_parse('Hamas attacked the state of Israel,
                      who responded by bombarding Gaza', dependency=T)
tokens %>%
  annotate('clauses', d = direct, x = xcomp) %>%
  plot_tree(annotation = 'clauses')
```

Figure 8. An example of a sentence that contains a relative clause modifier.


a#text1.1.2
predicate
subject
ROOT
2
nsubj
1
dobj
4
det
3
prep
5
punct
7
pobj
6
a#text1.1.9
predicate
subject
relcl
9
nsubj
8
prep
10
pcomp
11
dobj
12
Hamas attacked the state of Israel , who responded by bombarding Gaza
hamas attack the state of israel , who respond by bombard gaza
PROPN VERB DET NOUN ADP PROPN PUNCT NOUN VERB ADP VERB PROPN
ORG_B GPE_B GPE_B

This sentence contains a relative clause, which can be considered as a separate sentence. In the dependency tree, as shown in Figure 8, the head of the relative clause ("responded") has a relative clause modifier as a relation. It is a child of the token "state" ("the state of Israel"), which indicates that the relative clause is about this token. The sentence

[11] The sentence has been slightly changed to better demonstrate how *copy_nodes* can copy the fill children

could be split into two sentences: “Hamas attacked the state of Israel”, and “The state of Israel responded by bombarding Gaza”.

To achieve this, there are two main operations that need to be performed. Firstly, the “who” (more generally the wh-clause) in the relative clause needs to be replaced with the parent of the relative clause. Secondly, the relative clause needs to be isolated by making the head (“responded”) a ROOT. Broken down into individual steps, these operations can be performed as follows:

1. Select the head of the relative clause (“respond”), the parent on which it depends (“state”) and the child that refers to the parent (“who”).
2. Copy the parent including its fill children (“the state of Israel”)
3. Mutate the copy, replacing the parent and relation attributes with those of the reference (making it the subject child of “responded”)
4. Remove the reference, which is now replaced by the copy
5. Mutate the head of the relative clause to make it a ROOT.

In [package] this corresponds to the following code.

```
tq = tquery(relation = "relcl", label = "relcl",
            children(lemma = c("who", "that"), label = "ref"),
            parents(label = "parent"))

tokens = select_nodes(tokens, tq) %>%
  copy_nodes("parent", new = "copy", copy_fill = T) %>%
  mutate_nodes("copy", parent = ref$parent, relation = ref$relation) %>%
  remove_nodes("ref", with_fill = T) %>%
  mutate_nodes("relcl", parent = NA, relation = "ROOT") %>%
```

We can now annotate and plot the token data as usual.

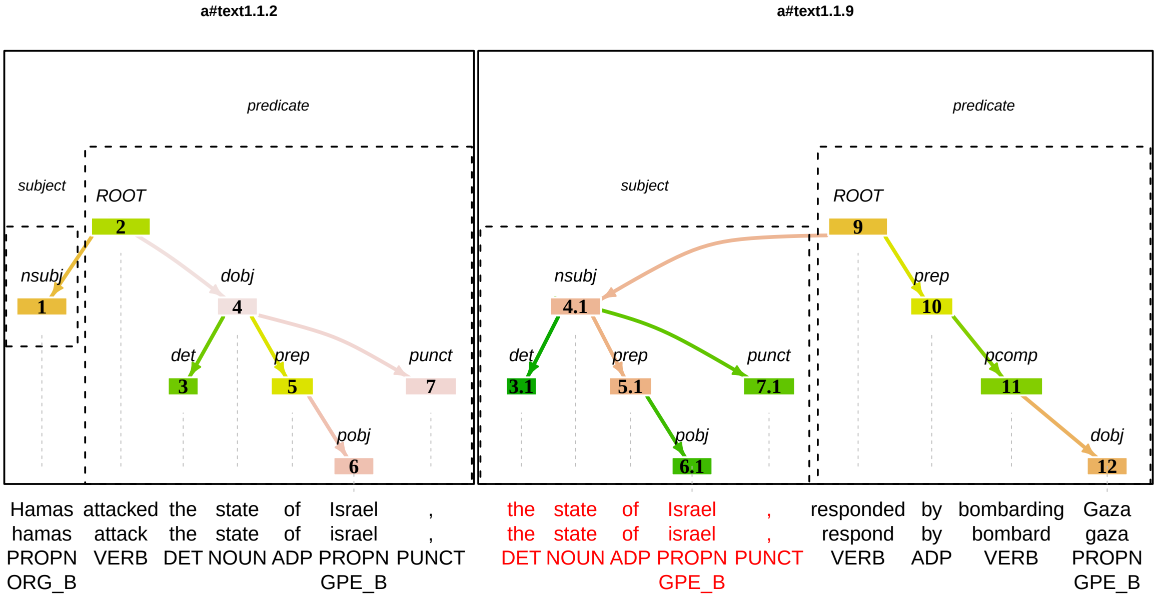
```
tokens %>%
  annotate('clauses', p=passive, a=active, x=xcomp) %>%
  plot_tree(annotation = 'clauses')
```

The output (Figure 9) shows the sentence split into two parts. In the isolated relative clause, “who” has been replaced by “the state of Israel”, which is shown in red by *plot_tree* to indicate that these tokens are copies. Note that the copies also have a unique token_id. In the token data.frame, these tokens have their own unique rows (directly below the original).

Convenience functions that wrap reshape operations

Using the reshape tools, it becomes relatively easy to develop more complex functions for simplifying the dependency tree. Here we demonstrate an example of two such functions. Do note that developing generic simplification operations is not the core purpose

Figure 9. Using the reshape operations to isolate a relative clause.


a#text1.1.2
a#text1.1.9
predicate
subject
ROOT
nsubj
dobj
det
prep
punct
pobj
1
2
3
4
5
6
7
predicate
subject
ROOT
nsubj
det
prep
punct
pobj
pcomp
dobj
3.1
4.1
5.1
6.1
7.1
9
10
11
12
Hamas attacked the state of Israel ,
hamas attack the state of israel ,
PROPN VERB DET NOUN ADP PROPN PUNCT
ORG_B GPE_B
the state of Israel , responded by bombarding Gaza
the state of israel , respond by bombard gaza
DET NOUN ADP PROPN PUNCT VERB ADP VERB PROPN
GPE_B GPE_B

of [package], and these functions should mainly be considered as experimental examples of how to apply the reshape operations.

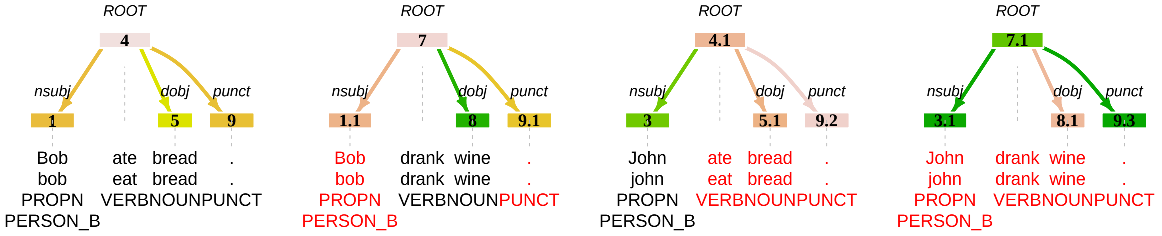
The two functions that we will demonstrate here are *inherit* and *chop*. *Inherit* can be used flatten a tree. It works by having nodes with certain types of relations inherit the parent and relation attributes of their parent, and (optionally) adopt copies of the parents children. This is applied recursively until nodes with the specified type of relation are gone, or the specified recursion depth is reached. By default, all of the parent's fill children are adopted, but only if the direct child has a relation that the node did not already have itself. For example, if the node already has a *dobj* child, it will not adopt another *dobj*. This is a crude but often effective way for dealing with dropped arguments in nested sentences such as conjunctions. The *chop* function is a wrapper around the remove_nodes function, that simply removes a selected node and all its fill children. The selection uses the same syntax as the lookup arguments in *tquery*. here we apply *inherit* to remove conjunctions, and *chop* to remove the tokens that serve as coordinating conjunctions (e.g., “and”, “but”). The output is presented in Figure 10.

```
spacy_parse('Bob and John ate bread and drank wine.', dependency=T) %>%
  inherit('conj') %>%
  chop(relation = 'cc') %>%
  plot_tree()
```

Conclusion

In this paper we demonstrated the [package] R package for querying and reshaping dependency trees. The main goal of this package is to make it easier to develop, test and apply queryies and reshape operations for extracting semantic relations. The current

Figure 10. Application of reshape operations (*inherit* and *chop*) to pull apart conjunctions in the sentence "Bob and John ate bread and drank wine".



version offers the required tools, and is designed to offer as much flexibility as possible by exposing many of the lower level functions, while promoting a transparent workflow by allowing complex procedures to be set up as pipes. Still, while the current version more or less completes the primary features and API, this package will remain in active development, steered by its use in research of the authors, towards establishing more and better applications for communication research.

Several problems have not been addressed in this paper. Firstly, texts can also contain long distance dependencies that span across sentences. Identifying different ways in which a text refers to the same entities is also called *co-reference resolution*. This is a complex problem that requires other approaches than the one used in [package], that often incorporate training data and real-world knowledge[12]. For addressing this problem we recommend using dedicated co-reference resolutions systems. The CoreNLP pipeline (Manning et al., 2014) includes a co-reference resolution system, and for other pipelines there are often decent plugins or post-processing components (though at present still mostly for English).

For certain types of long distance dependencies [package] does offer solutions. The *add_span_quotes* function enables dealing with long quotes, surrounded in quotation marks, than span multiple sentences, with several options for searching for the source within one or more sentences leading up to the quote. One of the development goals of [package] is to create intuitive and generic tools for dealing with these types of issues, rather than having to rely on ad-hoc solutions.

Another limitation of the rule-based approach for extracting semantic relations is that it requires a dependency tree, but good quality dependency parsers are not available for many languages. The performance of the method also relies directly on the performance of the dependency parser. Furthermore, there will always be many innovative ways in which people express meaning in language, which will be difficult to capture with manually crafted rules, and will also cause mistakes in dependency trees.

one of the future goals of the [package] package is to facilitate a combination with supervised machine learning approaches, as a potential way of addressing some of these issues. To this end, one of the future features of [package] will be an integrated annotation tool that can be used to efficiently check and correct rule-based annotations, as a computer-assisted method for creating training data for extracting general semantic relations, as well as making it easier to develop training data for more specific use cases and contexts. The long-term goal is to contribute to the establishment of good rules and training sets for en-

[12] There are rule-based approaches for dealing with certain common cases of co-references, that we might implement in the future.

hancing the extraction of semantic relations, with a focus on applications in communication research.

References

Bache, S. M., & Wickham, H. (2014). magrittr: A forward-pipe operator for r [Computer software manual]. Retrieved from `https://CRAN.R-project.org/package=magrittr` (R package version 1.5)

Barberá, P., Boydstun, A., Linn, S., McMahon, R., & Nagler, J. (2016). Methodological challenges in estimating tone: Application to news coverage of the us economy. In *Meeting of the midwest political science association, chicago, il.*

Bender, E. M. (2013). Linguistic fundamentals for natural language processing: 100 essentials from morphology and syntax. *Synthesis lectures on human language technologies*, *6*(3), 1–184.

Benoit, K., & Laver, M. (2003). Estimating irish party policy positions using computer wordscoring: The 2002 election–a research note. *Irish Political Studies*, *18*(1), 97–107.

Benoit, K., & Matsuo, A. (2017). spacyr: R wrapper to the spacy nlp library [Computer software manual]. Retrieved from `https://CRAN.R-project.org/package=spacyr` (R package version 0.9.0)

Blei, D. M., Ng, A. Y., & Jordan, M. I. (2003). Latent Dirichlet Allocation. *the Journal of machine Learning research*, *3*, 993–1022.

Boumans, J. W., & Trilling, D. (2016). Taking stock of the toolkit: An overview of relevant automated content analysis approaches and techniques for digital journalism scholars. *Digital Journalism*, *4*(1), 8–23.

Burscher, B., Vliegenthart, R., & De Vreese, C. H. (2015). Using supervised machine learning to code policy issues: Can classifiers generalize across contexts? *The Annals of the American Academy of Political and Social Science*, *659*(1), 122–131.

Cambria, E., Schuller, B., Xia, Y., & Havasi, C. (2013). New avenues in opinion mining and sentiment analysis. *IEEE Intelligent Systems*, *28*(2), 15–21.

Carreras, X., & Màrquez, L. (2005). Introduction to the conll-2005 shared task: Semantic role labeling. In *Proceedings of the ninth conference on computational natural language learning* (pp. 152–164).

Chen, D., & Manning, C. (2014). A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (emnlp)* (pp. 740–750).

Dowle, M., & Srinivasan, A. (2017). data.table: Extension of ‘data.frame‘ [Computer software manual]. Retrieved from `https://CRAN.R-project.org/package=data.table`

Goldberg, Y. (2017). Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies*, *10*(1), 1–309.

Grimmer, J., & Stewart, B. M. (2013). Text as data: The promise and pitfalls of automatic content analysis methods for political texts. *Political Analysis*, *21*(3), 267–297. doi: 10.1093/pan/mps028

He, L., Lee, K., Lewis, M., & Zettlemoyer, L. (2017). Deep semantic role labeling: What works and what’s next. In *Proceedings of the 55th annual meeting of the association for computational linguistics (volume 1: Long papers)* (Vol. 1, pp. 473–483).

Honnibal, M., & Johnson, M. (2015, September). An improved non-monotonic transition system for dependency parsing. In *Proceedings of the 2015 conference on empirical methods in natural language processing* (pp. 1373–1378). Lisbon, Portugal: Association for Computational Linguistics. Retrieved from `https://aclweb.org/anthology/D/D15/D15-1162`

Kok, S., & Domingos, P. (2008). Extracting semantic networks from text via relational clustering. In *Joint european conference on machine learning and knowledge discovery in databases* (pp. 624–639).

Kübler, S., McDonald, R., & Nivre, J. (2009). Dependency parsing. *Synthesis Lectures on Human Language Technologies*, *1*(1), 1–127.

Liu, B. (2012). Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies*, *5*(1), 1–167.

Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., & McClosky, D. (2014). The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations* (pp. 55–60).

Nakov, P., Ritter, A., Rosenthal, S., Sebastiani, F., & Stoyanov, V. (2016). Semeval-2016 task 4: Sentiment analysis in twitter. In *Proceedings of the 10th international workshop on semantic evaluation (semeval-2016)* (pp. 1–18).

Nivre, J., De Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajic, J., Manning, C. D., ... others (2016). Universal dependencies v1: A multilingual treebank collection. In *Lrec.*

Palmer, M., Gildea, D., & Xue, N. (2010). Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, *3*(1), 1–103.

Pradhan, S., Moschitti, A., Xue, N., Ng, H. T., Björkelund, A., Uryupina, O., ... Zhong, Z. (2013). Towards robust linguistic analysis using ontonotes. In *Proceedings of the seventeenth conference on computational natural language learning* (pp. 143–152).

Roberts, M. E., Stewart, B. M., Tingley, D., Lucas, C., Leder-Luis, J., Gadarian, S. K., ... Rand, D. G. (2014). Structural topic models for open-ended survey responses. *American Journal of Political Science*, *58*(4), 1064–1082.

Santorini, B. (1990). Part-of-speech tagging guidelines for the penn treebank project (3rd revision). *Technical Reports (CIS)*, 570.

Sennrich, R., Volk, M., & Schneider, G. (2013). Exploiting synergies between open resources for german dependency parsing, pos-tagging, and morphological analysis. In *Proceedings of the international conference recent advances in natural language processing ranlp 2013* (pp. 601–609).

Siddharthan, A., & Mandya, A. A. (2014). Hybrid text simplification using synchronous dependency grammars with hand-written and automatically harvested rules. In *Proceedings of the 14th conference of the european chapter of the association for computational linguistics (eacl 2014).*

Slapin, J. B., & Proksch, S.-O. (2008). A scaling model for estimating time-series party positions from texts. *American Journal of Political Science*, *52*(3), 705–722.

Tang, D., Qin, B., & Liu, T. (2015). Document modeling with gated recurrent neural network for sentiment classification. In *Proceedings of the 2015 conference on empirical methods in natural language processing* (pp. 1422–1432).

Van Atteveldt, W., Sheafer, T., Shenhav, S. R., & Fogel-Dror, Y. (2017). Clause analysis: using syntactic information to automatically extract source, subject, and predicate from texts with an application to the 2008–2009 gaza war. *Political Analysis*, 1–16.

Van Atteveldt, W., Kleinnijenhuis, J., & Ruigrok, N. (2008). Parsing, semantic networks, and political authority using syntactic analysis to extract semantic relations from dutch newspaper articles. *Political Analysis*, *16*(4), 428–446.

van Atteveldt, W., & Peng, T.-Q. (2018). When communication meets computation: Opportunities, challenges, and pitfalls in computational communication science. *Communication Methods and Measures*, *12*(2-3), 81–92.

Van der Beek, L., Bouma, G., Malouf, R., & Van Noord, G. (2002). The alpino dependency treebank. *Language and Computers*, *45*, 8–22.

Yates, A., Cafarella, M., Banko, M., Etzioni, O., Broadhead, M., & Soderland, S. (2007). Textrunner: open information extraction on the web. In *Proceedings of human language technologies: The annual conference of the north american chapter of the association for computational linguistics: Demonstrations* (pp. 25–26).

Zhou, J., & Xu, W. (2015). End-to-end learning of semantic role labeling using recurrent neural networks. In *Proceedings of the 53rd annual meeting of the association for computational* *linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers)* (Vol. 1, pp. 1127–1137).