The rsyntax package

What is rsyntax

Many techniques for automatic content analysis rely on a bag-of-words assumption, meaning that syntactic information and even the order of words is ignored. However, syntax can be crucial for extracting certain types of information from texts. The rsyntax package offers tools for querying dependency trees to extract syntactic information. This information can then also be used to annotate tokens, thus enabling more fine-grained automatic content analysis at the level of quotes and clauses, using the same bag-of-words techniques for the analysis.

Input: dependency parse data

The input data for `rsyntax` is a data.frame with the output of a dependency parser, such as Stanford CoreNLP (English), Alpino (Dutch) or ParZu (German). Short examples are included in `rsyntax`.

rsyntax

First, install the latest version of rsyntax.

```
devtools::install_github('vanatteveldt/rsyntax')
```

```
library(rsyntax)
```

preparing the token index

The input for rsyntax is a data.frame with the output of a dependency parser. There are four mandatory columns:

- *doc_id*: the document ids as numeric values
- *sentence*: the id of the sentence (preferably according to position in the document) as numeric values.
- *token_id*: the id of the token (preferably according to position in the document or sentence) as numeric values. The global token ID is the combination of the document, sentence and token_id, and has to be unique.
- *parent*: the id of the token's parent in the dependency tree. Specifically, parent points to the token_id within the same document-sentence pair.

The data.frame needs to have these columns, and by default uses these names, but this can be customized with the `tokenindex_columns()` function.

```
tokenindex_columns(doc_id = 'doc_id', sentence='sentence',
                   token_id='token_id', parent='parent')  ## these are also the default values
```

We can now create the tokenindex. This verifies whether the mandatory columns are present, converts the data.frame to a data.table (of the data.table package), and creates a key and index for the global token (and parent) IDs. This enables the efficient lookup of parents/children.

```
tokens = data(tokens_corenlp)  ## demo data included in rsyntax
tokens = tokens_corenlp[23:32,] ## we'll use a selection for this demo
```

```r
tokens = as_tokenindex(tokens)
tokens
```

doc_id	token_id	sentence	offset	token	lemma	POS	parent	relation
1	23	4	95	John	John	NNP	24	nsubj
1	24	4	100	loves	love	VBZ	NA	root
1	25	4	106	Mary	Mary	NNP	24	dobj
1	26	4	110	.	.	.	24	punct
1	27	5	112	Mary	Mary	NNP	29	nsubjpass
1	28	5	117	is	be	VBZ	29	auxpass
1	29	5	120	loved	love	VBN	NA	root
1	30	5	126	by	by	IN	31	case
1	31	5	129	John	John	NNP	29	nmod:agent
1	32	5	133	.	.	.	29	punct

Querying the token index

We can now look for nodes in the token index. This can be done directly with the `find_nodes()` function, or we first make a query with `tquery()` and then apply it with `apply_queries()`. The `find_nodes()` function is usefull for single queries or to develop and try out new queries. For now, we will focus on the slightly more verbose combination of `tquery()` and `apply_queries()`, which should be more intuitive.

Using tquery() and apply_queries()

The `tquery()` function is used to create token queries, and the `apply_queries()` function is used to apply one or multiple queries on the tokenindex. In the `tquery()` function, the query terms are given as name-value pairs, in which the name has to correspond to a column in the tokenindex, and the value is a vector with query terms. By default, queries use case sensitive matching, with the option of using common wildcards (* for any number of characters, and ? for a single character). There are more advanced query options, but we'll ignore these for now.

The first argument for `apply_queries()` is the tokenindex (here named `tokens`). Next, any number of queries can be added as (named) arguments. If a named argument is used (recommended) the name will be used in the unique ids for the results, which is convenient later on.

For example, the following query looks for all tokens in which the POS tag is "VB*", where * is a wildcard for any number of undefined characters. Thus, any POS tag that starts with"VB" will be found.

```r
q = tquery(POS = 'VB*')
apply_queries(tokens, q1 = q)
```

doc_id	sentence	.ID
1	4	q1#1
1	5	q1#2
1	5	q1#3

The output contains three columns: doc_id, sentence and .ID. The doc_id and sentence simply indicate the context in which a match is found, and the .ID is a unique id for the match. Each row is a unique match.

What you do not yet see is the actual token that is matched. This is because you will need to explicitly state which of the matched tokens you want to `save` as nodes in the network pattern. (this will make sense once we get to more complex queries). Here we will save the nodes using the name “verb”.

```
q = tquery(POS='VB*', save='verb')
apply_queries(tokens, q1 = q)
```

doc_id	sentence	.ID	verb
1	4	q1#1	24
1	5	q1#2	28
1	5	q1#3	29

The output now contains an additional column named “verb”, that contains the token ids of the tokens that match the query.

To see why this is usefull, let’s create a more complex query that looks for patterns of multiple nodes. In a dependency tree, tokens are nodes in a network, and each node can have parents and children. We can look for the parents or children of nodes by passing the `parents()` and `children()` functions as arguments to the `tquery()` function. The `children()` and `parents()` functions work (mostly) identical to the tquery() function. They can also themselves contain `parents()` and `children()`, to look for more complex patterns.

Here we add `children()` to our former query. As a condition, we say that the `relation` has to match ’nsubj*’. In the current data, the `relation` column contains the dependency relation of a token to its parent. Thus, we are now looking for children of verbs that are the subject of the verb. We save these nodes as “subject”.

```
q = tquery(POS = 'VB*', save='verb',
           children(relation = 'nsubj*', save='subject'))
apply_queries(tokens, q1=q)
```

doc_id	sentence	.ID	verb	subject
1	4	q1#1	24	23
1	5	q1#2	29	27

Now the output has two node columns: “verb” and “subject”. Also note that there are fewer results. This is because all the verbs that did not have a child with a subject relation do not match the query.

We can now annotate the tokens (tokenindex) based on the nodes found with the tquery. This is also a good way to inspect whether the tquery works as intended. For this we use the `annotate_nodes()` function, which has three mandatory arguments: the tokenindex, the nodes, and a name for the column that is added to the tokenindex.

```
nodes = apply_queries(tokens, q1=q)
annotate_nodes(tokens, nodes, column='nodes')
```

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	4	23	95	John	John	NNP	24	nsubj	q1#1	subject
1	4	24	100	loves	love	VBZ	NA	root	q1#1	verb
1	4	25	106	Mary	Mary	NNP	24	dobj	NA	NA
1	4	26	110	.	.	.	24	punct	NA	NA
1	5	27	112	Mary	Mary	NNP	29	nsubjpass	q1#2	subject
1	5	28	117	is	be	VBZ	29	auxpass	NA	NA
1	5	29	120	loved	love	VBN	NA	root	q1#2	verb
1	5	30	126	by	by	IN	31	case	NA	NA
1	5	31	129	John	John	NNP	29	nmod:agent	NA	NA

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	5	32	133	.	.	.	29	punct	NA	NA

This adds two columns to the tokenindex: nodes and nodes_id. The `nodes_id` column contains the id of a pattern of nodes (i.e. a row in the nodes data.table), and the `nodes` column contains the given name of the nodes (i.e. the columns in the nodes data.frame, as given in the tquery with the save parameter).

Note that in the second sentence, "Mary is loved by John", Mary is now considered the subject, but in truth the subject is John, because the sentence is passive. Since language is creative, you will often have to combine multiple tqueries.

Combining multiple queries

Combining queries introduces some new arguments in annotate_nodes, because you will have to deal with overlapping nodes across queries. This can require different settings depending on what you want to do. Here we use an example to introduce the relevant arguments.

Continuing on the previous example, we can combine queries for a *direct* and *passive* subject.

```
passive = tquery(POS = 'VB*', save='verb',
                 children(relation = 'nsubjpass'),
                 children(relation = 'nmod:agent', save='subject'))
direct = tquery(POS = 'VB*', save='verb',
                children(relation = 'nsubj*', save='subject'))

nodes = apply_queries(tokens, pas=passive, dir=direct)
annotate_nodes(tokens, nodes, column='nodes')
```

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	4	23	95	John	John	NNP	24	nsubj	dir#1	subject
1	4	24	100	loves	love	VBZ	NA	root	dir#1	verb
1	4	25	106	Mary	Mary	NNP	24	dobj	NA	NA
1	4	26	110	.	.	.	24	punct	NA	NA
1	5	27	112	Mary	Mary	NNP	29	nsubjpass	NA	NA
1	5	28	117	is	be	VBZ	29	auxpass	NA	NA
1	5	29	120	loved	love	VBN	NA	root	pas#1	verb
1	5	30	126	by	by	IN	31	case	NA	NA
1	5	31	129	John	John	NNP	29	nmod:agent	pas#1	subject
1	5	32	133	.	.	.	29	punct	NA	NA

Now the results are correct, but there is some magic going on that you need to be aware of. The reason that Mary is no longer (wrongly) coded as the subject, is that by default the `annotate_nodes()` function ignores a pattern of nodes (i.e. a row in the nodes data.table) if one of the nodes has been matched in a previous pattern. In this case, the node "loved" was already matched in the *passive* query, and so the *direct* match is ignored.

This is usefull, because it makes it possible to chain queries together in a particular order. Very specific queries can be performed first, and very broad queries at the end. This way, the broad queries do not need to account for many possible exceptions, given that these exections are addressed before. (In this example, it would have been easy to make *direct* more specific by using "nsubj" instead of "nsubj*" as a relation, but it is often more complex)

Be aware that the order of queries thus matters. See, for example, what happens if we first use the *direct* query and then the *passive* query.

```
nodes = apply_queries(tokens, dir=direct, pas=passive)
annotate_nodes(tokens, nodes, column='nodes')
```

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	4	23	95	John	John	NNP	24	nsubj	dir#1	subject
1	4	24	100	loves	love	VBZ	NA	root	dir#1	verb
1	4	25	106	Mary	Mary	NNP	24	dobj	NA	NA
1	4	26	110	.	.	.	24	punct	NA	NA
1	5	27	112	Mary	Mary	NNP	29	nsubjpass	dir#2	subject
1	5	28	117	is	be	VBZ	29	auxpass	NA	NA
1	5	29	120	loved	love	VBN	NA	root	dir#2	verb
1	5	30	126	by	by	IN	31	case	NA	NA
1	5	31	129	John	John	NNP	29	nmod:agent	NA	NA
1	5	32	133	.	.	.	29	punct	NA	NA

Now only the direct queries are used (note that the query that is used can be seen in nodes_id).

In some cases, you might want to use all the nodes, including duplicates. For this, you can set the rm_dup (remove duplicates) argument to FALSE. The duplicate nodes are than concatenated (if concat_dup is TRUE) or the rows in the tokenindex are duplicated (if concat_dup is FALSE)

```
annotate_nodes(tokens, nodes, column='nodes', rm_dup=F)
```

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	4	23	95	John	John	NNP	24	nsubj	dir#1	subject
1	4	24	100	loves	love	VBZ	NA	root	dir#1	verb
1	4	25	106	Mary	Mary	NNP	24	dobj	NA	NA
1	4	26	110	.	.	.	24	punct	NA	NA
1	5	27	112	Mary	Mary	NNP	29	nsubjpass	dir#2	subject
1	5	28	117	is	be	VBZ	29	auxpass	NA	NA
1	5	29	120	loved	love	VBN	NA	root	dir#2,pas#1	verb,verb
1	5	30	126	by	by	IN	31	case	NA	NA
1	5	31	129	John	John	NNP	29	nmod:agent	pas#1	subject
1	5	32	133	.	.	.	29	punct	NA	NA

```
annotate_nodes(tokens, nodes, column='nodes', rm_dup=F, concat_dup=F)
```

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	4	23	95	John	John	NNP	24	nsubj	dir#1	subject
1	4	24	100	loves	love	VBZ	NA	root	dir#1	verb
1	4	25	106	Mary	Mary	NNP	24	dobj	NA	NA
1	4	26	110	.	.	.	24	punct	NA	NA
1	5	27	112	Mary	Mary	NNP	29	nsubjpass	dir#2	subject
1	5	28	117	is	be	VBZ	29	auxpass	NA	NA
1	5	29	120	loved	love	VBN	NA	root	dir#2	verb
1	5	29	120	loved	love	VBN	NA	root	pas#1	verb
1	5	30	126	by	by	IN	31	case	NA	NA
1	5	31	129	John	John	NNP	29	nmod:agent	pas#1	subject
1	5	32	133	.	.	.	29	punct	NA	NA

Using the fill argument

Till now, annotate_nodes() coded only the specific nodes, as matched by the tquery. But often it is usefull to code all the children of these nodes. In the previous example, we might want to extract predicates. A convenient and often accurate way of doing this is by taking everything 'under' the verb in the dependency tree. For this, we can use the *fill* argument.

```
nodes = apply_queries(tokens, pas=passive, dir=direct)
annotate_nodes(tokens, nodes, column='nodes', fill = T)
```

doc_id	sentence	token_id	offset	token	lemma	POS	parent	relation	nodes_id	nodes
1	4	23	95	John	John	NNP	24	nsubj	dir#1	subject
1	4	24	100	loves	love	VBZ	NA	root	dir#1	verb
1	4	25	106	Mary	Mary	NNP	24	dobj	dir#1	verb
1	4	26	110	.	.	.	24	punct	dir#1	verb
1	5	27	112	Mary	Mary	NNP	29	nsubjpass	pas#1	verb
1	5	28	117	is	be	VBZ	29	auxpass	pas#1	verb
1	5	29	120	loved	love	VBN	NA	root	pas#1	verb
1	5	30	126	by	by	IN	31	case	pas#1	subject
1	5	31	129	John	John	NNP	29	nmod:agent	pas#1	subject
1	5	32	133	.	.	.	29	punct	pas#1	verb

Now, every node under "verb" (children, grandchildren, etc.) is coded as "verb", and every node under "subject" is coded as "subject". The exception is nodes that are already coded. For instance, "John" is also a child of "loves", but since the "John" node is already coded as subject, it will not be filled as "verb".

Using fill makes it possible to easily extract usefull sub-sentences, without having to write many elaborate queries for matching all possible patterns. The tradeoff is that it is crude, and should thus only be seen as a heuristic.