TOKENSREGEX: Defining cascaded regular expressions over tokens

Angel X. Chang, Christopher D. Manning

Computer Science Department, Stanford University, Stanford, CA, 94305
{angelx, manning}@cs.stanford.edu

We describe TOKENSREGEX, a framework for defining cascaded regular expressions over token sequences. TOKENSREGEX is available as part of the Stanford CoreNLP software package and can be used for various tasks which require reasoning over tokenized text. It has been used to build SUTIME, a state-of-the-art temporal tagger, and can be helpful in a variety of scenarios such as named entity recognition (NER) and information extraction from tokens.

1. Introduction

TOKENSREGEX is a framework for defining cascaded patterns over token sequences. It extends the traditional regular expression language defined over strings to allow working with tokens. In other words, it generalizes from matching over sequences of characters (strings) to matching over sequences of tokens. Furthermore, it uses a multi-stage extraction pipeline that can match multiple regular expressions—a practically more useful scenario than single pattern matching. The multi-stage extraction pipeline also allows for building up patterns in stages, similar to a cascaded, finite-state automaton. As demonstrated by systems such as FASTUS (Hobbs et al., 1997), this approach can be very effective in extracting information from text. Finite-state cascades based parsers are also both fast and robust, and is a feasible alternative to stochastic context-free parsers (Abney, 1996).

We provide an implementation of TOKENSREGEX as a Java library and demonstrate its use for matching over tokens. We also provide two annotators in the Stanford CoreNLP pipeline[1].

Why is a system like TOKENSREGEX needed and why is it useful to be able to define regular expressions over tokens? In NLP applications, text is usually first tokenized and annotated with additional information such as part-of-speech tags. Therefore, it is natural and convenient to specify regular expressions over the tokens. Such token-based regular expressions can be more concise and comprehensible, as well as easier to manipulate and modify than traditional regular expressions over strings.

Regular expressions over tokens also allow matching on additional token-level features, such as part-of-speech annotations and named entity tags. This allows for concise rules at a higher level than just matching against the individual tokens.

In addition, with TOKENSREGEX, we can easily combine the robustness of statistical methods with the control of a rule based system. Typically the best performing part-of-speech annotations and named entity tags come from statistical taggers. However, statistical systems depend on appropriate training data, which is unfortunately not always available. Furthermore, even when training data is available, it is hard to finely control the output of a statistical system. TOKENSREGEX complements supervised learning methods by providing a rule-based system for handling cases with limited training data.

We have used TOKENSREGEX to implement SUTIME (Chang and Manning, 2012), a rule-based temporal tagger. We also demonstrate how we can augment statistical methods with rules and information extraction systems with patterns. We first describe the main components of the system in the following sections.

2. TokensRegex Patterns

Traditional regular expressions over strings have proven to be powerful and useful, with libraries available in most programming languages. However, these implementations are typically limited in that they can only handle regular expressions over strings (i.e. sequences of characters). In TOKENSREGEX, we provide a regular expression implementation that generalizes to sequences of tokens and can also be extended to handle other types as well. These patterns can then be stacked to form finite-state cascades. Our implementation supports many of the features found in modern regular expression libraries while allowing for more expressive power.

[1] nlp.stanford.edu/software/corenlp.shtml

We provide our implementation as a Java library with a similar interface to the Java regular expression library (`java.util.regex`).

We define a syntax for regular expressions over tokens that is similar to the traditional syntax used for regular expressions over strings. The main difference lies in the syntax for matching individual tokens.

2.1. Token Syntax

In NLP applications, text is typically tokenized into units of characters (tokens). Each token is then annotated with various attributes, such as part-of-speech (POS), or named entity type (NER).

For example, given the sentence: *"Reykjavík is the capital of Iceland."*, we have the following tokens:

word	Reykjavík	is	the	capital	of	Iceland	.
pos	NNP	VBZ	DT	NN	IN	NNP	.
ner	LOC	O	O	O	O	LOC	.

In our token syntax we indicate each token by `[ <expression> ]` where `<expression>` specifies token attributes which should be matched as follows. We use the symbol `[]` to indicate any token.

Basic Expressions: describe how a token attribute should be matched.

Example	*Description*
`"abc"`	token text is `abc`
`/abc/`	token text matches regular expression `abc`
`pos:"NNP"`	token POS is `abc`
`pos:/NN.*/`	token POS matches regular expression `NN.*`
`word>30`	text is number and greater than 30. (>=, <, <=, ==, ! = supported)

Compound Expressions: formed by combining basic expressions with boolean operators.

Example	*Description*
`!(pos:/NN.*/)`	POS is not noun
`pos:/NN.*/ | pos:/VB.*/`	POS noun or verb
`word>=1 & word<=10`	text numeric and between 1 and 10

2.2. Regular Expression Syntax

Tokens are combined using similar syntax as regular expressions over strings. TOKENSREGEX supports most features found in regular expression libraries including both greedy and reluctant quantifiers, grouping, capturing, and back references. In addition, TOKENSREGEX also supports features such as named groups, macros, and conjunctions. Table 1 shows a summary of the syntax used.

Syntax	*Description*
`X Y`	`X` followed by `Y`
`X | Y`	`X` or `Y`
`X & Y`	`X` and `Y`
	Grouping
`(X)`	`X` as a capturing group
`(?name X)`	`X` as capturing group with name *name*
`(?: X)`	`X` as a non capturing group
	Quantifiers (greedy/reluctant)
`X?, X??`	`X`, once or not at all
`X*, X*?`	`X`, zero or more times
`X+, X+?`	`X`, one or more times

Table 1: TOKENSREGEX Regular expression syntax

Below, we give some examples of TOKENSREGEX regular expressions.

Sequences: in a token sequence, individual tokens are bracketed by `[]`, which can be omitted when pattern matching against the text field. Quantification is marked using the standard symbols `*,+,?`.

Match: *Picasso is an artist*
```
[ner:PERSON]+ [pos:VBZ] /an?/
/artist|painter/
```
Match: *five thousand kilometers*
```
([ner:NUMBER]+) /km|kilometers?/
```

Groups: parentheses `()` are used for grouping. By default, groups are captured and accessed using `$n` where the group number n is obtained by counting the number of opening parentheses (from left to right). Group 0 is the entire matched expression. Non capturing groups `(?...)` do not count toward the overall number of groups. For convenience, groups can also be named `(?name ...)`.

Match: *50 kilometers*
```
(?:quant [ner:NUMER]+ ) /km|kilometers?/
```
The *50* corresponds to group 1 and can be referred to by using `$quant` or `$1`.

Macros: to improve regular expression readability, TOKENSREGEX supports definition of macros that can be used in later regular expressions.
```
$UNIT = /km/kilometers?/
[ner:NUMBER]+) $UNIT
```

3. Matching multiple regular expressions

Often it is useful to match not just one, but many regular expressions. TOKENSREGEX provides an extraction pipeline for matching against multiple regular expressions in stages. The pipeline is similar to a cascade of finite automata. During each stage, a series

Syntax	*Description*
`$n`	Matched tokens for capture group n
`$n[i]`	ith token for capture group n
`$n[i].key`	Attribute key (for above token)
`$$n.value`	Value for capture group n
`$$n.text`	Text for capture group n

Table 2: Syntax for accessing capture groups

of extraction rules are applied, and expressions are matched based on the specified pattern of each rule. If multiple rules can be matched, a rule is selected based on the priority of the rule, then the length of the sequence matched, and finally the order in which the rule is specified.

When an expression is matched, additional attributes can be added to the matched tokens. The matched expression can also be treated as an aggregate token (with its own attributes) over which the extraction rules can be reapplied, giving a finite-state cascade.

We use a domain specific language (DSL) for defining the extraction rules and how the expressions should be matched, as described below.

3.1. Extraction Rule Syntax

Extraction rules are specified with a JSON-like syntax.

```
{ ruleType: "tokens",
  pattern: (([ner:PERSON]) /was/ /born/
           /on/ ([ner:DATE])),
  result: "DATE_OF_BIRTH" }
```

Associated with each rule is the `ruleType` and the `pattern` to match against. The `ruleType` specifies how the `pattern` should be used and is described in the next section. Optionally, the rule can have a `priority` and a `stage`. If these are not specified, then all rules have the same priority and are grouped into one stage. The `result` and `action` fields describe what should happen when the rule is matched. With each matched expression, we can optionally associate a *value*. The *value* can be used to create new annotation for the matched expression. The `result` field indicates how this *value* should be derived. The DSL allows for referring back to the captured groups (see Table 2).

3.2. Extraction Rule Types

TOKENSREGEX has four types of extraction rules:

1. **Text**: applied on raw text, match against regular expressions over strings (the tokenization is ignored)

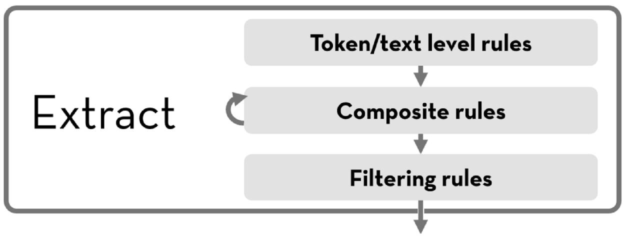

Figure 1: TOKENSREGEX extraction pipeline.

2. **Tokens**: applied on tokens, match against regular expressions over tokens
3. **Compositional**: applied on previously matched expressions (text, tokens, or previous composite rules), and repeatedly applied until no new matches
4. **Filtering**: applied on previously matched expressions, matches are filtered out and not returned

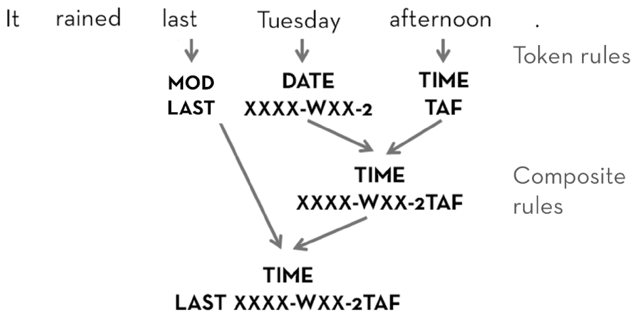

Figure 2: Parsing of a temporal expression using the TOKENSREGEX pipeline.

Extraction rules are grouped into stages. Figure 1 shows how the rules are applied for each stage. Figure 2 shows an example of how TOKENSREGEX extraction rules are used to parse temporal expressions in SUTIME.

Initially, rules over **text** and **tokens** are matched. For instance, in Figure 2, **token** rules are used to match *Tuesday* to `DATE XXXX-WXX-2` and *afternoon* to `TIME TAF`.

Next, **composite** rules are applied. Tokens from matched expressions are combined to form an aggregate token, and composite rules are applied recursively until no more changes to the matched expressions are detected. Only expressions with an associated value, indicated by `result`, are kept. By adding the `result` to the aggregate token as an annotation, it can be matched against. An example of a composite rule specification for SUTIME is given below. For SUTIME, the `result` values are temporal objects that can be composed and operated on.

```
{ ruleType: "composite",
  pattern: ( ( [ temporal::IS_TIMEX_DATE ] )
           /at/?  ( [ temporal::IS_TIMEX_TIME ] ) ),
  result:  TemporalCompose(INTERSECT,
                           $0[0].temporal,
                           $0[-1].temporal) }
```

At the end of each stage, there is a filtering phase in which the **filter** rules are applied and invalid expressions are filtered out. For instance, in SUTIME filtering rules are used to filter out ambiguous words such as *fall*. If a potential temporal expression is a single ambiguous word and the part of speech tag is not a noun, then it is not resolved to a temporal object.

```
{ ruleType: "filter",
  pattern: ( [ word:/fall|spring|second|march|may/
           & !(tag:/NN.*/)] ) }
```

This process is repeated for each stage of rules.

4. Applications

We have seen in Figure 2 how TOKENSREGEX was used in the temporal tagging scenario to implement SUTime. The multi-stage extraction pipeline allowed for temporal expressions to be built up from mappings of simple tokens (e.g. *Tuesday* to `XXXX-WXX-2`), to more complex patterns involving already recognized time expressions. SUTIME followed the multi-stage strategy of: (i) building up patterns over individual words to find numerical expressions; then (ii) using patterns over words and numerical expressions to find simple temporal expressions; and finally (iii) forming composite patterns over the discovered temporal expressions.

TOKENSREGEX can also be used to augment the output of statistical systems which require specialized training data. For instance, to augment the named entity types recognized by a NER system, it is easier to make a list of entities to be marked (e.g. University of X, list of shoe brands) than to manually gather the required training data. This can be easily achieved using the `TokensRegexNERAnnotator`. Specification of a gazetteer for shoe brands, or more complex regular expressions for recognizing URLs and email addresses is straight-forward:

```
Nike                                    SHOE_BRAND
Reebok                                  SHOE_BRAND
http://.*                               URL
<?\w+@[A-Z0-9.-]+\.[A-Z]{2,4}>?         EMAIL
( [ner:CITY]+ /High/ /School/ )         HIGH_SCHOOL
```

Another application for TOKENSREGEX is in specifying patterns for relation extraction. Patterns have been show to be effective for relation extraction so many current systems use a combination of patterns and machine learning approaches. We have used TOKENSREGEX to recognize relations for the TAC KBP slot filling task (Ji et al., 2010). Some sample rules for identifying potential relations between the entity and a slot value are given below.

```
{ result: "per:children",
  pattern: ( $SLOT_VALUE /,/ /son|daughter|child/
                         /of/ $ENTITY ) }
{ result: "per:cause_of_death",
  pattern: ( $ENTITY /died/ /of|from/ $SLOT_VALUE ) }
```

Beyond these specific applications, TOKENSREGEX can be used to help the development of other NLP systems. One important benefit of TOKENSREGEX is that it is easily extensible. Although we have only discussed regular expressions over tokens, the TOKENSREGEX API allows extensions to matching over other types as well.

5. Conclusion

We have presented TOKENSREGEX, a framework that brings the power of cascaded regular expressions to tokenized text. TOKENSREGEX fills an important gap by providing a system for handling patterns over tokens. We hope that it will be a useful tool for the community and that it will help future research dealing with tokenized text.

References

S. Abney. 1996. Partial parsing via finite-state cascades. *Natural Language Engineering*, 2(4):337–344.

A. X. Chang and C. D. Manning. 2012. SUTIME: A library for recognizing and normalizing time expressions. In *8th International Conference on Language Resources and Evaluation (LREC 2012)*, May.

J. R. Hobbs, D. E. Appelt, J. Bear, D. Israel, M. Kameyama, M. Stickel, and M. Tyson. 1997. Fastus: A cascaded finite-state transducer for extracting information from natural-language text. pages 383–406.

H. Ji, R. Grishman, H. T. Dang, K. Griffitt, and J. Ellis. 2010. Overview of the tac 2010 knowledge base population track. In *Third Text Analysis Conference (TAC 2010)*.