Simplifying Text Processing with Grammatically Aware Regular Expressions

Rafal Rzepka † Tyson Roberts ‡ Kenji Araki †
† Graduate School of Information Science and Technology, Hokkaido University
{kabura,araki}@media.eng.hokudai.ac.jp
‡ Google Japan
nallohki@gmail.com

Abstract

In our paper we introduce Grammatically Aware Regular expression (GARE) and describe its usage using examples from moral consequences retrieval task. GARE is an extension to the regular expression concept that overcomes many of the difficulties with traditional regexp by adding Normalization (e.g., searching all grammatical forms with basic form of a verb or adjective is possible) or POS awareness (e.g. searching only for adjectives after "wa" particle is possible). We explain how it works, what makes it more expressive for natural language, and how it solves a number of matching cases that traditional regular expressions cannot solve on their own.

1 Introduction

As the research topic, text mining becomes more and more popular, and this task is an interesting subject when teaching students Natural Language Processing techniques. One of the most common pattern matching and extraction methods, regular expression, is useful for shallow parsing tasks, but its difficulty discourage many undergraduates (See Fig. 1 to see the complexity of traditional regexp). Also, when using in more complicated tasks, regexp lacks flexibility, especially in situations where deeper grammatical parsing is required. As a solution to these problems, we introduce grammatically aware regular expression (GARE) and describe its usage examples from an ongoing task of moral consequences retrieval. GARE utilizes dependency parsing which allows to retrieve related tokens which are impossible to distinguish with surface form information only. By this paper we want to introduce the system to Japanese NLP researchers and illustrate its functions with few examples from our research[1] on retrieving human moral behavior patterns[1].

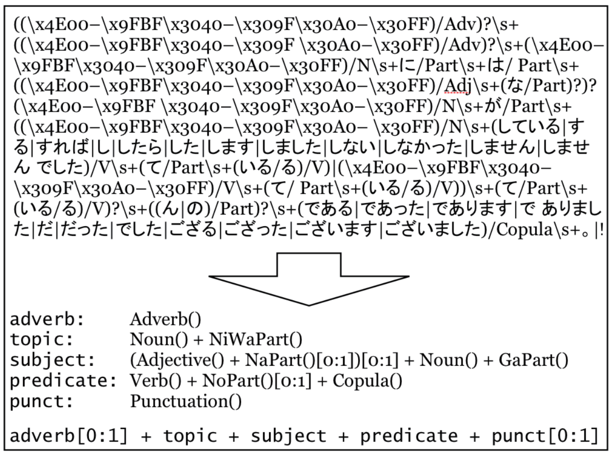
1.1 Classic Regular Expression

Regular expressions (often abbreviated to *regex* or *regexp*) are used by researchers familiar with tools like UNIX grep but for most lay people trying to use it to determine if a particular pattern occurs in a larger text is a difficult task. A regular expression is a formal language used to describe a particular regular grammar that can thereafter be read by a regular expression parser to examine if a portion matching the described grammar exists in a target sequence. It examines it and identifies parts that match the provided specification. Typical examples of such specification are concatenation, alternation, grouping, literal and repetition. One can find a date within a sentence searching for repeated digits between particular words like names of months, etc. However, when dealing with natural languages, affixes, inflections and other linguistic phenomena change surface forms which always makes a user to extend the specification. Linguistic information (e.g. part of speech, inflection information, and any other tagged data) is not available and one have to use external data to perform a successful matching.

1.2 Moral Judgment Retrieval Task

Examples used in this paper come from a task to extract moral consequences of human actions. It is a search task where input is an action as "to kill a man" or "to steal a car" and output is a emotional reaction of bloggers combined with a set of other consequences presented by manually set keywords: prise/punishment, impact on society, possibility to be imprisoned, etc. So if majority of bloggers react negatively to a given action, it is labeled as unmoral. Enhanced regular expression allows for deeper search and refining output as one can easily extend or limit verb forms or calculate data credibility by weighting down expressions showing lower probability (e.g.

[1]Due to lack of space, richer set of full matching examples will be presented on the poster this paper describes



```
((\x4E00−\x9FBF\x3040−\x309F\x30A0−\x30FF)/Adv)?\s+
((\x4E00−\x9FBF\x3040−\x309F \x30A0−\x30FF)/Adv)?\s+(\x4E00−
\x9FBF\x3040−\x309F\x30A0−\x30FF)/N\s+に/Part\s+は/ Part\s+
((\x4E00−\x9FBF\x3040−\x309F\x30A0−\x30FF)/Adj\s+(な/Part)?)?
(\x4E00−\x9FBF \x3040−\x309F\x30A0−\x30FF)/N\s+が/Part\s+
((\x4E00−\x9FBF\x3040−\x309F\x30A0− \x30FF)/N\s+(している|す
る|すれば|し|したら|した|します|しました|しない|しなかった|しません|しませ
ん でした)/V\s+(て/Part\s+(いる/る)/V)|(\x4E00−\x9FBF\x3040−
\x309F\x30A0−\x30FF)/V\s+(て/ Part\s+(いる/る)/V))\s+(て/Part\s+
(いる/る)/V)?\s+((ん|の)/Part)?\s+(である|であった|であります|で ありまし
た|だ|だった|でした|ござる|ござった|ございます|ございました)/Copula\s+。|!

adverb:     Adverb()
topic:      Noun() + NiWaPart()
subject:    (Adjective() + NaPart()[0:1])[0:1] + Noun() + GaPart()
predicate:  Verb() + NoPart()[0:1] + Copula()
punct:      Punctuation()

adverb[0:1] + topic + subject + predicate + punct[0:1]
```

Figure 1: Brevity and clearness difference between classic regexp and GARE.

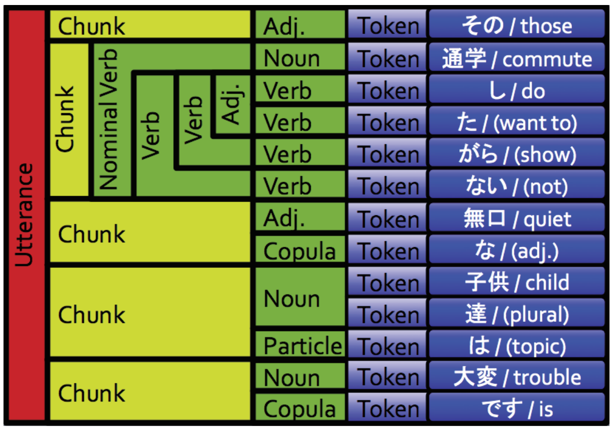


Figure 2: The final tree is of variable depth, and word objects potentially contain other words.

"maybe", "probably", "perhaps", etc.). Below we explain how it is possible.

2 MuCha System

The base of GARE[3] is MuCha system[2], which was developed for Japanese language implementation of MIT Media Lab's ConceptNet[4]. Named in honor of the CaboCha project on which the underlying parsing tree is based, the MuCha is a modular suite written in the Python programming language. Though MuCha is implemented for Japanese, it can be extended to any language that has a working dependency parser. Its interface is inspired by pyparsing[2], a general parsing library, and there are two main modules to the system: the Tree Parser and the Pattern Matcher.

[2]http://pyparsing.wikispaces.com/

2.1 Tree Module

The tree module is an extended version of the tree structure described by the CaboCha dependency structure analyzer. The tree first builds an utterance by copying CaboCha's shallow chunk/node tree. These CaboCha tokens are then abstracted into "Word" nodes. Since Japanese is highly inflected, the idea of a word can be comprised of a variable number of tokens (See Fig. 2 and 3). To achieve an abstraction, each token node is first wrapped in a Word node, and then custom combination rules based on Japanese grammar are applied recursively to build up words. Japanese is generally agglutinative, and inflections can change the part of speech of the entire word each time an affix is encountered. This can happen repeatedly for highly inflected words. Classes in MuCha describe the properties and idiosyncrasies of each type and this allows easy implementation of several major features as normalization of verb and adjective conjunctive forms or enumeration of objects in an utterance by word, token, or chunk. In the Chunk phase, the iterator operates by running rules over each child in a chunk. Initially, the children of each chunk are token objects. The rules themselves work by wrapping the child nodes with appropriate word nodes, sometimes recursively. By the end of this phase, all token objects are wrapped in word objects. The Word phase, which comes next, operates on the precondition that all children of chunks have been wrapped in (possibly recursive) word nodes. This phase is concerned with fixing certain idiosyncrasies of the previous phase and combining complex conjugations. The tree structure is made up of several types of nodes in a strict hierarchy: utterance, chunk, word, and token objects. Each has its own object representation, derived from a common object type, TreeNode. The chunk class is contained within utterance nodes and contains word nodes. They represent a partition of the utterance that happens across linguistic boundaries. Words are branch nodes that are children of chunks and parents of other word nodes (complex words) or tokens (simple words). Lastly, tokens consist of only a single class, and are not overridden. Tokens are the children of word nodes, and contain the basic information about linguistic objects.

2.2 Verb and Adjective Normalization

A simple scheme is implemented: normalized verb conjugations are reported as part of the verb, separated by the swung-dash character ∼, but arbitrary normalization algorithms can be implemented (See Fig. 4). For example:

Kare wo yurushi taku nari masen deshita .

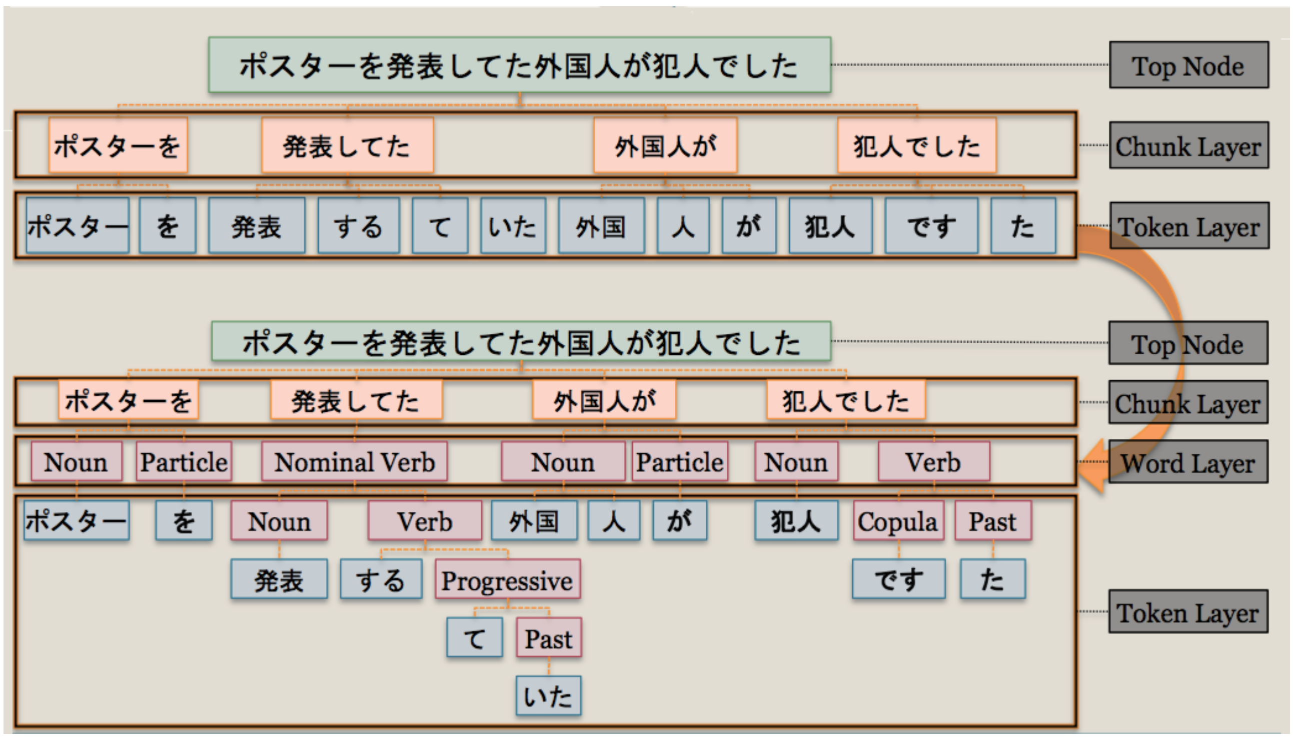


Figure 3: Visualization of MuCha : Tree Module : CaboCha Tree.

I didn't want to forgive him. [Polite]
Kare wo yurushi taku nara na katta .
I didn't want to forgive him. [Plain]
both become:
Kare wo yurusu∼tai∼naru∼nai∼ta

In Japanese, inflections often have different conjunctive surface forms depending on the type root they are attached to. As shown above, polite verb forms in Japanese regularly use conjugations that, while identical in a semantic sense, carry different surface forms from plain verb forms. Normalization helps to solve this problem by creating an extendable regular syntax for representing the Japanese language in a normalized form. The tree nodes that use the normalize functionality are called in the following way. First is to call the *normalize*() function on the node, passing a normalizer object as an argument. Then, if the node is branch, call *join*() on child pairs, or *join_with_type*() on multiple children, which calls *normalize*() on them in turn. If not, the next step is to normalize self as string and return result. In this way, all of MuChas nodes are able to be normalized in a flexible way, and the user is able to write new normalizers if desired.

2.3 MuCha Pattern Matching Module

The Pattern Matching module allows patterns to be written in a way similar to regular expressions,

```
class Normalizer:
  def join(self, a, b):
     h1 = self._handlers
     for c1 in self. get_pair_handler(a):
        if c1 not in h1: continue

        h2 = h1[c1]
        for c2 in self.get_pair_handler(b):
           if c2 not in h2: continue

           return h2[c2](str(a), str(b))

 def join_with_type(self, n1, n2, *arg):
  func = self._handlers[n1][n2]
  args = [x if isinstance(x, str) else
 x.normalized for x in arg]
  return reduce(func, args)
```

Figure 4: A simplified version of the normalization class.

but it is much more powerful. The parser itself is a recursive descent parser which is capable of parsing context-sensitive grammars (allows full backtracking), but written to do as much lazy calculation as possible. Patterns are written in straight Python using an interface inspired by the pyparsing library mentioned before. The tokens on which the module operates are not strings, but the enumerated words from the original parse tree and this makes its pattern matching different from classic approaches. For example

Kitto kare ni ha kakoku na batsu ga youi sareteru n darou .

```
leading_adverb = Adverb()[0:1]
topic_marker   = Particle().set(filter=Surface('は')) |
                 Particle()[2].set(filter=Surface('には'))
topic          = Noun() + topic_marker
subject_marker = Particle().set(filter=Surface('が'))
subject        = Adjective() + Copula()[0:1] + Noun() +
subject_marker
predicate      = Token()[:]
punctuation    = Punctuation().set(filter=Surface('。'))

pattern =
leading_adverb + topic + subject + predicate + punctuation
```

Figure 5: Simple example of matching pattern for the sentence *Kitto kare ni ha kakoku na batsu ga youi sareteru n darou.*

(Surely a severe punishment has been prepared for him.)
Would operate on an object stream:
<Adverb: Kitto> <Noun: kare> <Particle: ni> <Particle: ha> <SimpleAdjective: kakoku> <Da: na> <Noun: batsu> <Particle: ga> <NominalVerb: youi sareteru> <Noun: n> <ComplexVerb: darou> <Punctuation: .>
This implementation differs from a traditional regular expression because the grammar is represented in program code in the Python language as opposed to a separately coded string. Elements (tokens) and operators are represented by Python objects directly. Thanks to such approach, implementation is simpler than creating an interpreter for a regular expression engine based on a string-based grammar and arbitrary Python code can be used for filtering. Non-linear matching and sub-matching are also possible; the readability of matching grammars are improved and can be easily built step by step. However, if traditional regular expression semantics need to be preserved – user can override Python operators.

3 Conclusions and Future Work

In this paper we have briefly described GARE, an improved text matching technology using a novel method of matching. It improves Regular Expression in Natural Language tasks by allowing grammatical constructs and other features to be used, which is currently impossible in regexp systems. It is developed to be easier to use, re-use and to learn as writing patterns is far less time consuming and requires less code. It should be suitable for use in education and research - we are currently working on an online matching patterns tester. Because MuCha[3] is simple to extend and modify (by exploiting the Python object inheritance model), a user can define his or her own patterns. We are planning to use it not only for consequences retrieval but also to extract concepts fitting the OpenMind CommonSense (OMCS[5]) entries which are laboriously input by human volunteers. For instance "AtLocation" node candidates can be matched by
Subject() + ComplexNoun() + NiPart() + (Iru() — Aru()).
To use MuCha's extendability even further, we are also planning to add semantic categories (by using WordNet[6] data, for example) to allow limiting ComplexNoun() to one kind of noun:
Subject() + PLACE() + NiPart() + (Iru() — Aru()).
We also need to work on optimization, as the system currently concentrates on complicated matches, not on the matching speed (on 2008 MacBook Pro average time was approximately 68ms per match).

[3]MuCha 1.0.6 is available for download and testing here: http://arakilab.media.eng.hokudai.ac.jp/local/local/Links_files/mucha_package.1.0.6.tar.gz (your feedback is greatly appreciated).

References

[1] Komuda, R., Ptaszynski, M., Momouchi, Y., Rzepka, R. and Araki, K.: "Machine Moral Development: Moral Reasoning Agent Based on Wisdom of Web-Crowd and Emotions", International Journal of Computational Linguistics Research, Volume: 1 , Issue: 3, pp: 155-163 (2010)

[2] Roberts, T,. Rzepka, R. and Araki, K.: "A Japanese Natural Language Toolset Implementation for ConceptNet", the Proceedings of Commonsense Knowledge in the AAAI 2010 Fall Symposium (Technical Report FS-10-02), pp. 88-89, Arlington, USA (2010)

[3] Roberts, T., Rzepka, R. and Araki, K.: "Introducing Grammatically Aware Regular Expressions". In Proceedings of the Pacific Association For Computational Linguistics, paper No 12 (2011)

[4] Havasi, C., Speer, R. and Alonso, J.: "ConceptNet 3: a Flexible, Multilingual Semantic Network for Common Sense Knowledge", in Proceedings of Recent Advances in Natural Languages Processing, pp. 277-293 (2007)

[5] Singh, P., Lin, T., Mueller, E. T., Lim, G., Perkins, T., & Zhu, W.: "Open Mind Common Sense: Knowledge acquisition from the general public". In Proceedings of ODBASE'02. Lecture Notes in Computer Science. Heidelberg: Springer-Verlag. (2002).

[6] Fellbaum, C.: "WordNet: An Electronic Lexical Database", Cambridge, MA: MIT Press (1998)