Next Improvement towards Linear Named Entity Recognition using Character Gazetteers

Giang Nguyen, Štefan Dlugolinský, Michal Laclavík, Martin Šeleng, Viet Tran

Institute of Informatics, Slovak Academy of Sciences
Dúbravská cesta 9, 845 07 Bratislava, Slovakia
{giang.ui, stefan.dlugolinsky, laclavik.ui, martin.seleng,viet.ui}@savba.sk

Abstract. Natural Language Processing (NLP) is important and interesting area in computer science affecting also other spheres of science; e.g., geographical processing, social statistics, molecular biology. A large amount of textual data is continuously produced in media around us and therefore there is a need of processing it in order to extract required information. One of the most important processing steps in NLP is Named Entity Recognition (NER), which recognizes occurrence of known entities in input texts. Recently, we have already presented our approach for linear NER using gazetteers, namely Hash-map Multi-way Tree (HMT) and first-Child next-Sibling binary Tree (CST) with their strong and weak sides. In this paper, we present Patricia Hash-map Tree (PHT) character gazetteer approach, which shows as the best compromise between the both previous versions according to matching time and memory consumption.

Keywords: gazetteer, named entity recognition, natural language processing

1 Introduction

The task of Named Entity Recognition (NER) is to recognize occurrences of well-known entities in input texts coming from websites, web portals, social media, data dumps, documents, etc. These entities might be of arbitrary type, but the most used types in NER are persons, organizations, locations, times, and quantities. NER is often understood or classified as a subtask of Information Extraction (IE) in Natural Language Processing (NLP **Fig. 1**). One of the commonly used NER techniques is gazetteer – a simple list of well-known entities, which are looked up in input texts. In NER research area, the word "*gazetteer*" is used interchangeably for both the set of entity lists and for the processing resource that uses those lists to find occurrences of named entities in texts.

There are two main approaches of a gazetteer implementation:

- Machine learning techniques: usually known under various rule-based techniques
- Finite State Machines (FSM) techniques.

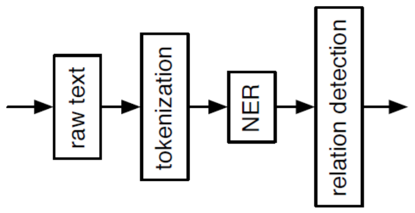

Fig. 1. Natural Language Processing (NLP) and Named Entity Recognition (NER)

An example of a high-level rule-based language for building and customizing NER annotators is described in [5]. Unfortunately, the process of designing the rules themselves is manual and time-consuming, but the rule-based approach performance is comparable to that of machine learning. Pattern-based matching and validation in an unlabeled corpus is described in [6] and [7] respectively. The outcome is that patterns can be quite complex to fulfill different requirements.

Authors in [2] present a gazetteer implemented as an FSM. Gazetteer is built at initialization time starting from the list of phrases that need to be later recognized with contextual dependency. Contextual dependency is also one of the assumptions in [3]. In the work [4], authors deal with automated gazetteer construction as well as with various problems like entity-noun ambiguity, entity-entity ambiguity and entity boundary detection. The overlapping of entities is solved here by selecting and aliasing them.

There are two main approaches that are used to find matches according to the way of handling input text:

- Token-based: input text is split into a sequence of tokens on which a matching is performed
- Character-based: input text is processed character by character. The approach of input handling determines internal representation of the gazetteer list

An example of the token gazetteer is Hash Gazetteer provided by Ontotext[1] in GATE project [1]. It is constructed based on hash tables of words. Authors declare that it takes in average four times less memory and that it works three times faster than GATE's previously optimized character-based FSM implementation. Another example is Large Knowledge Base Gazetteer, a new-generation gazetteer, which provides support for ontology-aware NLP. It allows using of large gazetteer lists and speeds up subsequent loading of data by caching. Linked Data Gazetteer is an experimental gazetteer that uses Linked Open Data for lookups. Besides the Ontotext gazetteers, there is a number of scientific works dealing with NER using comparable approaches. Authors in [2] present an implemented FSM gazetteer, which is built at initialization time by starting from the list of phrases that need to be later recognized with contextual dependency in token level.

[1] http://www.ontotext.com/collaborations/gate

2 Character gazetteers

Despite the large number of listed works in the previous section, existing gazetteers are in various implementation states and availabilities. A lot of them are integrated as a part of other software products. Therefore our work in Information Extraction (IE) area concretely on NER comes from the following requirements:

- standalone gazetteer independent of 3rd party libraries
- do not rely on external preprocessing; e.g., tokenization
- linear complexity lookup algorithm, which provides fast and effective processing of input text as a stream, especially for Big Data
- editable data structure; i.e., add/remove Named Entities (NEs) between lookups
- memory efficient data structure; i.e., ability to deal at least with several millions of entities
- robustness; i.e., input texts of any size and in any language

Gazetteers with token-level tokenization usually faces several problems such as the need of multi-travel search and matching, finding word boundaries, chunking or processing of non-trivial strings and characters. These problems lead to longer running time and low processing performance of this kind of gazetteers. Therefore, we have turned to gazetteers based on character-level tokenization, which provides more precise results and much better performance than token-level equivalent despite that character gazetteer is more memory consuming.

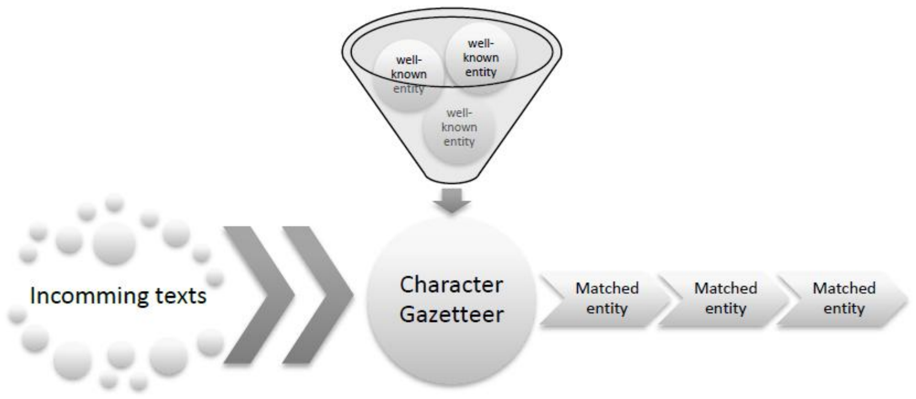

Fig. 2. Character gazetteer

The usage of gazetteers has usually two phases:

- Filling gazetteer structures with datasets of well-known entities: this step is usually performed in the initial phase.
- Matching entity occurrences in continuously incoming input text and reporting these occurrences with their references (URI or MDI, where MID is an ID of an object in Freebase) to the system for next processing steps.

Entity datasets, as the main source of gazetteer lists, are usually large and have tendency to grow; e.g., due to adding of new known entities during the running time. This fact significantly affects memory requirements, running time as well as the tokenization complexity, especially of character gazetteers. Therefore, an efficient data structure is required to effectively fold all the entities at the character level, which also enables a fast linear tokenization of input text and produces required output (references, occurrence quantity and positions in the input text). Tree structure is a powerful way of organizing data hierarchically and suits these considerations very well. Moreover, it provides easy and quick access operations in order to find elements and traverse through the structure.

Possible entity positions in input text are depicted in **Fig. 3**. The basic situations are case 1 - following situation, case 2 - embedded situation and case 3 - overlapping situation. Case 4, where the entities start at the same position and are embedded in the longest entity, is quite well solved and well-known as a problem of finding the longest or the first matched entity. The case 1, case 2 and case 3 can be freely occupied and they can create various combinations in incoming input texts (e.g. case 5). Therefore, matching process is quite complicated to find all entities, especially when entities are embedded, but not from the start boundary and/or they are also overlapped.

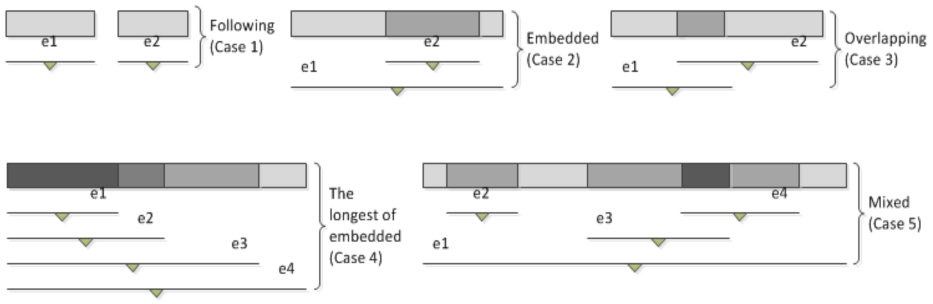

Fig. 3. Possible entity positions in incoming input text (*e* stands for *entity*)

2.1 Hash-map Multi-way Tree (HTM) and Child-Sibling Tree (CST) character gazetteers

In our previous work [9] [10], we presented two representations of the character gazetteer data structure as depicted in **Fig. 4**.

The HMT gazetteer is based on multi-way tree and implemented using Java HashMap, which guarantees constant-time performance *O(1)* in average for basic operations. Each node of the HTM tree has a hash map in which it can have arbitrary number of child nodes as its values; i.e. consecutive characters.

The CST gazetteer is based on the first-child next-sibling binary tree and implemented using pure Java simple node object. The CST tree is a binary representation of the HMT tree, where each node refers only to its first child node and its next sibling node. This representation is not as fast as the HMT, but it is efficient in memory consumption – the CST uses in average three to four times less memory in comparison to HMT.

The *filling algorithm* [9] loops over all its characters while traversing and building the tree with inserted entities. Finally, a node representing the last character of the input string is marked as a matching node. The difference between two gazetteers is only in the way of how the tree nodes are physically connected.

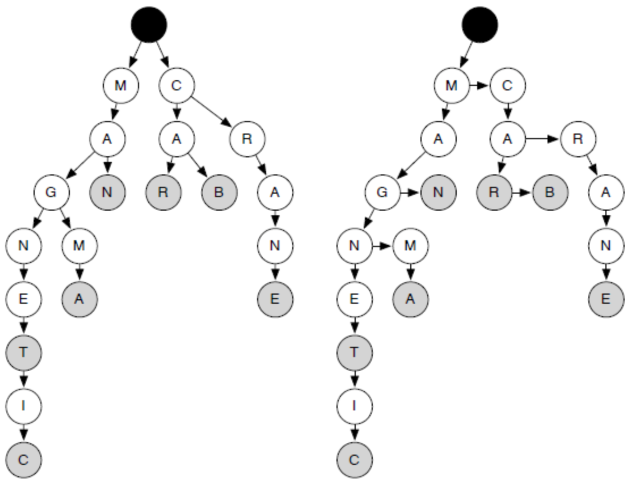

Fig. 4. Multi-way tree structures of HMT gazetteer (left) and first-child next-sibling tree structure of CST gazetteer (right)

After the tree gazetteers are filled with the entities during the initialization phase, they are ready to be used for tokenization and finding of well-known entities in the input text. This process is performed by a matching algorithm [9], which can be briefly described as a travelling of the gazetteer tree from the root node (the black circle in **Fig. 4**) matching entity nodes; i.e., gray circles in **Fig. 4**.. The matching algorithm for both HMT and CST gazetteers has *O(n)* complexity, where *n* is the number of characters in the input text. It means that we need to traverse the input text approximately one time to obtain the results. The matching algorithm is intended to be used on a stream of text of unlimited size.

In paper [10] we present a matching time comparison of existing approaches. In paper [9] we provide an approach of solving situations with embedded and especially with overlapping entities in order to match all the possible well-known entities with negligible increment of matching complexity. This approach of matching algorithm is applied for both HTM and CST gazetteer.

2.2 Patricia Hash-map Tree (PHT) character gazetteer

Although HMT and CST gazetteers work well; i.e., HMT is suitable for machine with more memory and applications that requires very fast processing/matching time; CST is suitable when memory issue is more pushed toward; there is still a gap for a gazetteer, which can work fast and uses less memory. PHT gazetteer (Patricia Hash-map Tree) seems to be a good candidate to fill the gap. It is named after the Patricia tree data structure and is based on Java HashMap implementation, which has an *O(1)* complexity for basic operations. Logical representation of the PHT tree is the same as the HMT's. The only difference is that each PHT tree node is collapsible and can contain more characters instead of only one character. In comparison to HMT and CST, nodes in PHT require more memory, but the number of nodes is significantly reduced and therefore the memory usage is more efficient (**Fig. 5**).

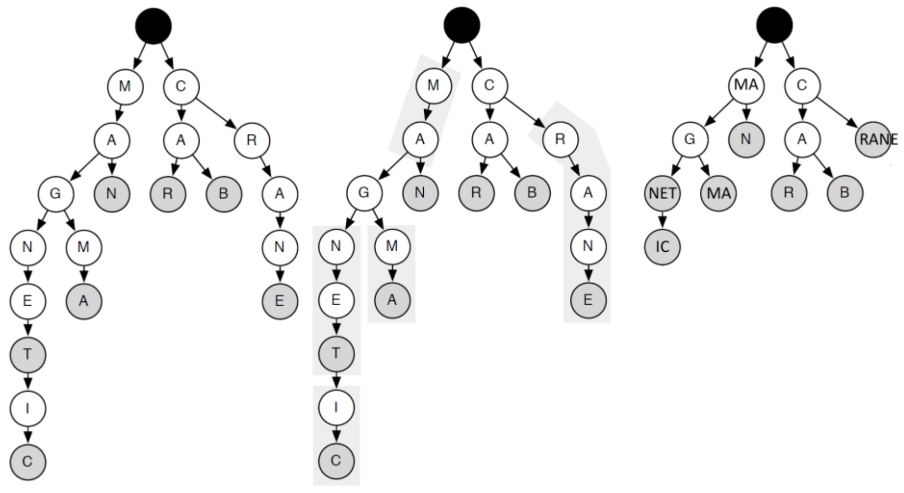

Fig. 5. Tree structure of HMT (left); collapsible nodes (middle) and PHT (right)

Of course the complex collapsible node structure complicates the *filling algorithm*, but except of the usual case, it also deals with cases when:

- the incoming character of the incoming entity splits the existing node into two nodes
- the incoming entity marks new matching/list nodes

In comparison to previous HMT and CST gazetteers, the filling algorithm is modified to deal with the collapsible nodes:

- stores multiple characters in one node when the node does not have a direct child node
- breaks the existing node into two nodes according to incoming character; i.e., makes nodes collapsible

The gazetteer tree is created in a flexible way to utilize the memory. Each node can be expanded when necessary and remains as a single node if the incoming character can be stored on it.

Algorithm: Insert new entity into PHT tree

```
node ← root node
FOR ALL character in entity DO
   IF character is in the right place in node's list THEN
      IF node is not matching node AND node does not have child THEN
         add character to node's list
      ELSE
         IF node has a child with character THEN
            node ← child
         ELSE
            child ← new node with character
            add child to node
            node ← child
         END IF
      END IF
   ELSE
      first child ← new node with node’s sublist from character
      add first child to node
      second child ← new node with character
      add second child to node
      node's list ← sublist before character
      node ← second child
   END IF
END FOR
IF entita is shorter than node's list THEN
   child ← new node with node’s sublist extends entita
   add child to node
   node's list ← entita
END IF
mark node as a matching node
```

Fig. 6. PHT filling algorithm: Insert entity into gazetteer’s tree

The *matching algorithm* is also modified, and in comparison to HMT case is more complex, but it is compensated with the shorter travel path in the matching algorithm.

Algorithm: Matching algorithm of PHT gazetteer

```
buf ← empty
node ← root node
onNode ← 0
WHILE character on input DO
   ch ← next character from input
```

```
    normalize whitespace or skip multiple
    IF ch is mapped in onNode position of node THEN
      add ch to buf
      onNode++
      IF the last mapped character and node is matching node THEN
        found an entity
      END IF
    ELSE IF node has a child node mapped on ch THEN
      add ch to buf
      node ← child node mapped on ch
      onNode ← 1
      IF node has only one character AND node is matching node THEN
        found an entity
      END IF
    ELSE
      IF buf contains boundary characters AND entity was found THEN
        unread characters from buf back to input until the first
        boundary occurrence
      END IF
      buf ← empty
      node ← root node
    END IF
END WHILE
```

Fig. 7. PHT matching algorithm

Although the PHT matching algorithm is a little bit more complicated, the *O(n)* complexity is guaranteed. We need only one travel though the input text to find all the occurrences of entities. Like the HMT and the CST, the PHT matching algorithm deals with all the entity occurrence cases without significant increment of algorithm complexity; e.g., embedded entities and overlapping entities (Fig. **Fig. 5**).

3 Experiments and evaluation

There are various sources of named entities available; e.g. Google Data Dumps[2] of FreeBase[3], DBpedia[4]. Therefore, it is not a problem to build up gazetteer lists for miscellaneous domains and testing purposes. Concretely, we use the FreeBase person dataset for the memory consumption testing and for comparisons. The Freebase person dataset has 2,614,401 unique entities of size 163 MB. Matching time was measured in milliseconds by executing the gazetteers over a set of 2,586 documents, 5,107 documents, 7,564 documents and 9,909 documents acquired from CoNLL-2003 da-

[2] https://developers.google.com/freebase/data
[3] http://www.freebase.com/
[4] http://dbpedia.org/

tasets [8] with approximately 7 MB, 15 MB, 22 MB and 29 MB of pure text. There is no limitation of the language in the input text for all three versions (HMT, CST and PHT) of our character gazetteers. The test was repeated several times and the measured values were averaged. Results are depicted in **Fig. 8**.

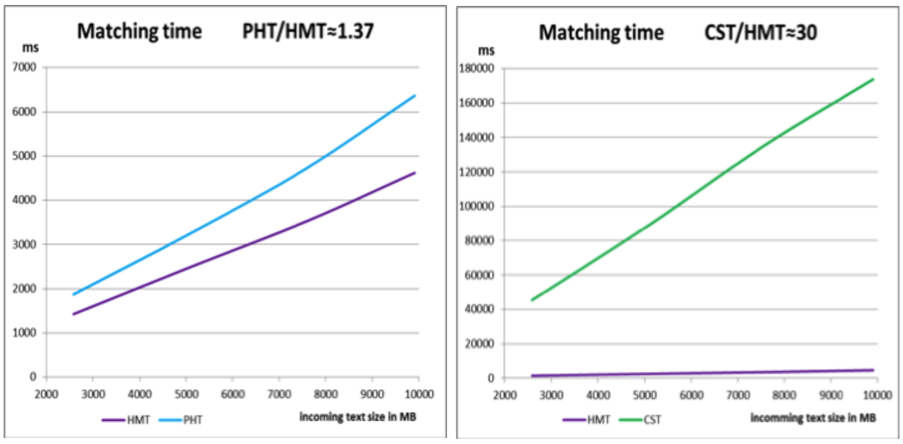

Fig. 8. Matching time comparison: PHT vs. HTM and CST vs. HMT

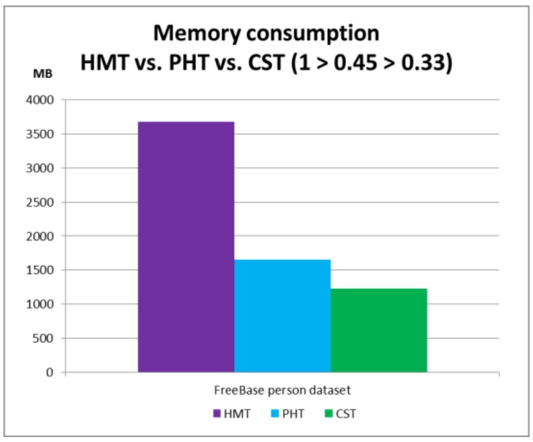

Fig. 9. Matching time comparison between PHT and CST gazetteer

In general, the weakness of the character gazetteer is in its memory consumption, especially for the HMT implementation, where each parent node requires additional memory for its hash map. Also the number of nodes is quite big. Theoretically, each tree node can have so many child nodes as the number of the characters in the charset set, thus the tree structure can be very wide. Fortunately, the tree space is limited by human language and all the combinations will never be covered, thus the management of the gazetteer tree in machine memory is possible also on commodity hardware

The CST version deals the best with the memory issue CST/HMT=0.33 (**Fig. 9**). That means the CST gazetteer uses only one thirds memory amount in comparison with HMT gazetteer. On the second hand, the matching time of the CST is significantly increased CST/HMT=30 (**Fig. 8**), that means it is 30 times slower than the HMT gazetteer. This result is not very bad when the matching time is still linear (*O(n)* matching complexity), but we wanted a better rate.

The PHT gazetteers uses slightly more memory than the CST; i.e., PHT/HMT=0.45 (**Fig. 9**), that means it uses less than haft of memory amount in comparison with the HMT gazetteer when the CST uses nearly one thirds of memory amount in comparison with the HMT gazetteer. The PHT matching time is linear, a little increased in quite low level PHT/HMT=1.37 (**Fig. 8**).

4 Conclusion

In this paper, we present a Patricia Hash-map Tree character gazetteer (PHT) and compare it with our two previous Hash-map Multi-way Tree (HMT) and first-Child next-Sibling binary Tree (CST). In comparison to HMT gazetteer, PHT gazetteer benefits from memory saving with slightly acceptable slowdown of the matching time. The PHT fulfils our requirements for improving the tree data structure memory consumption and for more efficient traversing as it is in HTM and in CST respectively.

All the presented implementations of our character gazetteer are available online [5].

Acknowledgment: this work is supported by projects VEGA 2/0185/13, KC INTELINSYS ESF ITMS 26240220072, CLAN APVV-0809-11 and EGI-InSPIRE EU FP7-261323 RI.

References

1. H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, and W. Peters, Text Processing with GATE (Version 6), 2011. [Online]. Available: http://tinyurl.com/gatebook
2. D. Maynard, V. Tablan, C. Ursu, H. Cunningham, and Y. Wilks, “Named entity recognition from diverse text types,” in Recent Advances in Natural Language Processing 2001 Conference, 2001, pp. 257–274.
3. X. Liu, S. Zhang, F. Wei, and M. Zhou, “Recognizing named entities in tweets,” in Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1, ser. HLT ’11. Stroudsburg, PA, USA: Association for Computational Linguistics, 2011, pp. 359–367. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002472.2002519
4. D. Nadeau, P. Turney, and S. Matwin, “Unsupervised name-identity recognition: Generating gazetteers and resolving ambiguity,” in Advances in Artificial Intelligence, ser. Lec-

[5] http://ikt.ui.sav.sk/gazetteer

ture Notes in Computer Science, L. Lamontagne and M. Marchand, Eds. Springer Berlin Heidelberg, 2006, vol. 4013, pp. 266–277. [Online]. Available: http://dx.doi.org/10.1007/11766247 23

5. L. Chiticariu, R. Krishnamurthy, Y. Li, F. Reiss, and S. Vaithyanathan, "Domain adaptation of rulebased annotators for named-entity recognition tasks," in Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, ser. EMNLP'10. Stroudsburg, PA, USA: Association for Computational Linguistics, 2010, pp. 1002–1012. [Online]. Available:http://dl.acm.org/citation.cfm?id=1870658.1870756
6. M. Laclavik, L. Hluchy, M. Seleng, and M. Ciglan, "Ontea: Platform for pattern based automated semantic annotation," Computing and Informatics, vol. 28, no. 4, pp. 555–579, 2009.
7. Z. Kozareva, "Bootstrapping named entity recognition with automatically generated gazetteer lists," in Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop, ser. EACL '06. Stroudsburg, PA, USA: Association for Computational Linguistics, 2006, pp.15–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1609039.1609041
8. E. F. Tjong Kim Sang and F. De Meulder, "Introduction to the conll-2003 shared task: language-independent named entity recognition," in Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003 – Volume 4, ser. CONLL '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 142–147. [Online]. Available: http://dx.doi.org/10.3115/1119176.1119195
9. Štefan Dlugolinský, Giang Nguyen, Michal Laclavík, Martin Šeleng: Character Gazetteer for Named Entity Recognition with Linear Matching Complexity, In 3rd World Congress on Information and Communication Technologies, WICT'2013, pp. 364-368, IEEE Catalog Number: CFP1395H-ART, ISBN: 978-1-4799-3230-6
10. Giang Nguyen, Štefan Dlugolinský, Michal Laclavík, Martin Šeleng: Token gazetteer and character gazetteer for named entity recognition. In 8th Workshop on Intelligent and Knowledge Oriented Technologies: WIKT 2013 proceedings. Eds. Babič, F., Paralič, J. - Košice : Centre for Information Technologies, Technical University in Košice, 2013, p. 1-6. ISBN 978-80-8143-128-9. [Online]. Available: http://web.tuke.sk/fei-cit/wikt2013/wikt%202013.pdf
11. Štefan Dlugolinský, Peter Krammer, Marek Ciglan, Michal Laclavík: MSM2013 IE Challenge: Annotowatch. In Proceedings of the Concept Extraction Challenge at the Workshop on Making Sense of Microposts co-located with the 22nd International World Wide Web Conference (WWW'13) Rio de Janeiro, Brazil, May 13, 2013, ISSN: 1613-0073, Vol-1019, pages 21-26, 2013