An Approximate Gazetteer for GATE based on Levenshtein Distance

Bruno Woltzenlogel Paleo
Technische Universitaet Wien
bruno.wp@gmail.com

Abstract. GATE is an architecture that allows the use and development of useful plugins for typical Natural Language Processing tasks. However, there is currently no plugin capable of annotating a document that contains words that only approximately match words specified in a list of words to be searched. Here we describe the development of such a plugin, based on Levenshtein Edit Distance, and its integration to the GATE environment. This plugin enables GATE to be useful for tasks in which exact matching is not enough.

1 Introduction

GATE (General Architecture for Text Engineering) provides a simple abstraction of a typical Natural Language Processing task ([3],[2]). In this abstraction, each task is defined as a pipeline, consecutively applying different *processing resources* (Part-of-Speech taggers, Sentence Splitters, Gazetteers, ...) to a given *language resource* (a text document or a collection of text documents).The output is a language resource enriched with *annotations*.

Here we are interested in a particular type of processing resource called Gazetteer. In section 2 we explain what is the purpose of a gazetteer, what annotations it produces and how the ***DefaultGazetteer*** provided by GATE works. In Section 3 we briefly describe a few applications in which the ***DefaultGazetteer*** is not useful, because it is not able to cope with noise or errors in the language resource. These applications are the motivation for the implementation of a new gazetteer, here called ***AproximateGazetter***. Section 4 describes its ideas and algorithms and, finally, in Section 5, we evaluate this new gazetteer.

2 Gazetteers

Gazetteers search for words in texts. Whenever one word belonging to one of the gazetteer's lists of words is found in the text, the matching region

Proceedings of the Twelfth ESSLLI Student Session
Ville Nurmi and Dmitry Sustretov (editors)

in the text is annotated with the majorType and minorType associated with the list to which the word belongs. If, for example, one of the lists of the gazetteer is a list of city names, then for each matching city name encountered in the text, the gazetteer will produce an annotation with the following parameters:

- **Start** - the position where the matching word starts.
- **End** - the position where the matching word ends.
- **Features** - a set of features associated with the annotation (for the city example, it would be majorType=location, minorType=city)
- **Lookup** - the set of annotations to which this annotation belongs.

2.1 GATE's DefaultGazetteer

Gate provides, within its plugin ANNIE (A Nearly New Information Extraction System)([1]), the interface ***Gazetteer*** and the class ***AbstractGazetteer***, which specify the minimal requirements of a gazetteer in terms of properties and methods that they should have and implement the basically functionality of some methods. Additionaly it provides the class DefaultGazetter, extending ***AbstractGazetteer*** and implementing Gazetteer into a fully functional gazetteer ([6]).

DefaultGazetteer is based on finite state machines (FSM). When it is loaded, all the words in its lists are read and a FSM is built combining all of them so that their characters are the labels in the transitions between the states, and a state is final whenever it corresponds to the end of a word. When the ***DefaultGazetteer*** is executed on a text, it reads the characters of the text and performs the corresponding transitions in the FSM, adding annotations when it finds final states.

The lists of words to be searched and annotated are simple plaintext files with one word in each line, and the gazetteer becomes aware of these lists by reading another file ("lists.def"), in which each line points to a different a list that should be used and specify its majorType and minorType features. An example of "lists.def" might look like:

```
city.lst:location:city
country.lst:location:country
```

In this case, the gazetteer will search all words specified in the plain-text files "city.lst" and "country.lst" and will annotate the occurrences found with the features majorType=location, minorType=city for words in "city.lst" and majorType=location, minorType=country for words in "country.lst".

3 Applications of Approximate String Matching

Although the ***DefaultGazetteer***, based on FSM as described previously, is very efficient and has a good precision, its recall may be not so good for some applications, because it is only able to detect exact matches of the words. If one character is mistyped in the name of a city in the text, for example, this occurrence will not be found. The application areas briefly described below are examples that require a more flexible Gazetteer, capable of detecting occurrences with noise and errors:

- **Bioinformatics** - one of its common tasks is to align sequences of DNA or Aminoacids in genes or proteins. These sequences can be seen as texts and since genes and proteins may contain mutations and abnormalities, approximate (inexact) matching is necessary for the alignment.

- **Information Retrieval** - if the documents that are being analyzed contain errors and noise, the demanded document might not be found simply because the relevant word contained errors and hence it couldn't be found by exact matching. Noise and errors in documents can come from digitalization via Optical Character Recognition (OCR) or from Speech recognition technologies.

- **Text Classification** - Text classification based on the annotations of a gazetteer may have their accuracy hindered if the gazetteer does not find noisy occurrences of the relevant words.

- **Multi-language Search** - Some languages are similar in their vocabulary, having words with the same root but with slightly different derivational morphology (e.g. "Algeria, Algrie", "Andorra, Andorre", "Bhoutan, Bhutan, Butao"). Different morphology may also occur as a result of internet slang. A gazetteer with approximate matching would be able to recall all these words, without knowledge of the morphology particularities of each language.

- **Text Correction and Completion** - A system might suggest corrections and completions for a word that has matched only approximately some word in the list of words. This may be useful and feasible for applications with a limited vocabulary.

- **Recovering from Noise in Signal Analysis** - By finding the best match for a noisy signal according to an allowed code, it is possible to recover the intended message.

By programming a simple approximate gazetteer, here called ***AproximateGazetter***, we expect to expand the frontiers of application of GATE

to all these areas. We also note that an improved recall provided by an approximate gazetteer may also benefit already classical areas of application of GATE, since other types of processing resources usually use the results of gazetteers. Sentence-splitters, for example, would benefit from a better detection of abbreviations by a gazetteer.

4 The Classic Levenshtein Distance and Some Improvements

To determine whether a string approximately matches another string, we need a distance function to measure how distant from each other the two strings are. There are different such functions ([8],[5],[4]). For the ***AproximateGazetter***, we chose the Levenshtein distance [7], because it seems to be a good compromise between flexibility to deal with different kinds of errors and efficiency of the algorithms that compute it.

The levenshtein distance between two strings is defined as the minimum number of operations to transform one string into the other, where the operations may be deletion of a character, insertion of a character or substitution of a character. Thus the distance between "aaa" and "aaaa" is 1, because one 'a' must be inserted in the first string to make it equal to the second string. Analogously, the distance between "aba" and "aca" is also 1, because 'b' must be substituted by 'c'. The different operations may have costs that are different from 1 and the cost may even depend on the characters that are inserted, deleted or substituted. This allows us to particularize the distance to specific types of errors and noise.

To compute the minimum distance, a dynamic programming algorithm can be used. We incrementally fill a bi-dimensional array D, where $D[i][j]$ stores the distance between the initial prefix subtrings of length i and j respectively of the first and second string. Clearly, to compute $D[i][j]$, we just need to take $D[i-1][j-1]$, if the characters in position i of the first string and in position j of the second string are equal, or we need to take the minimum of $D[i-1][j-1] + substitutionCost$, $D[i][j-1] + insertionCost$, $D[i-1][j] + deletionCost$. Thus we can fill the array from left to right and from top to bottom, since in this way we guarantee that we always have previously calculated the values that we need to calculate $D[i][j]$. Once the array is finished, we can read the final distance between the two strings in its bottom-right corner. If this distance is below a fixed threshold, we may be willing to consider both strings to match each other. Table 4 shows an example of such a computation.

This algorithm is $O(n.m)$ both in time and space, where n and m are the lengths of the strings.

Table 1.1: Example of computation of Levenshtein distance between the strings "CATO" and "GATE".

		C	**A**	**T**	**O**
	0	1	2	3	4
G	1	1	2	3	4
A	2	2	1	2	3
T	3	3	2	1	2
E	4	4	3	2	***2***

4.1 Finding Several Matches in a Text

The algorithm presented previously decides whether 2 strings match, but it's not able to decide whether 1 of the strings matches a substring of the other string (which may be a much longer string, containing a whole text full of words and sentences). Neither it is able to find the positions where these matches occur.

A very nave approach to solve this problem would be to compute the distance of the string to each possible substring of the text. However, this would be too slow, because a text of length m would have $\frac{m^2+2m+1}{2}$ possible substrings. A slightly less nave approach would be to consider only some of these substring, only those separated by spaces and presumed to be words, for example. But with this approach, we cannot handle errors related to insertion of spaces in the middle of words or removal of spaces between words.

Fortunately there is another solution which is as computationally expensive as solving the problem of simply computing the levenshtein distance between two strings [8]. Let's assume that the text (the bigger string) is horizontal in the distances array (i.e. the array has as many columns as characters in the text plus one). Then by initializing the first row of the array with zeros, we implicitly tell the algorithm that the matching of the string may start at any position in the text. And then we can detect the matches by looking at the last row and seeing which cells contain values that are below the distance threshold.

However this solution (as described in [8]) only finds the end position of a match. To determine its start position, we may compute for each cell of the array not only the distance but also the number of deletions and insertions that were used to yield that distance. And then we can find the start position according to:

$$startposition = endposition - (lengthOfString + \#insertions - \#deletions)$$

Table 1.2: Example of detecting matches in a text with the dynamic programming algorithm. Searching for "CA" in "CATACA" with distance threshold equal to 0. End-positions of the matches are marked by the '0's in the last row, which are emphasized in italic bold.

		C	**A**	**T**	**A**	**C**	**A**
	0	0	0	0	0	0	0
C	1	0	1	1	1	0	1
A	2	1	***0***	2	1	1	***0***

For example, searching for the word "York" in the text "New York" with distance threshold equal to 1, will first give us two matches, one of them with *endposition* in the 7th character and 1 deletion operation and the other one in the 8th character with 0 deletion operations. Then we compute the *startposition* for both of them and the result is the 5th character.

This algorithm, although sound, is still incomplete. It didn't find the matching of "York" with the substring "ork" in the text, which requires only 1 deletion of the character 'Y'. In order to fix this and obtain a complete algorithm, all we have to do is to consider more matches (and add the corresponding annotations), with increasing values of *startposition*, until the distance threshold is not satisfied anymore. For the above example, after we computed the *startposition* to be the 5th character for the match of "York" with "York" with distance 0, we consider increasing the *startposition* to the 6th character. This corresponds to the deletion of a character and thus the distance of this candidate match would be 1. Since this distance is below the threshold, we annotate this match. Then we consider increasing the *startposition* to the 7th character. This corresponds to 2 deletions and thus to a distance of 2, which is above the threshold. Therefore we don't annotate this candidate match and we stop increasing the *startposition*.

4.2 Avoiding Overlapping Matches

The procedure described above generates several overlapping matches (and their corresponding annotations) on an occurrence of the word, each match with different degree of error within the distance threshold. In most applications, these overlapping annotations are not desirable. The algorithm should return only the best (minimum distance) of such overlapping matches.

To find only the *endposition* of the best matches, it suffices to analyze the last row of the table, considering the sequences in which the distance is below the threshold. For these sequences, the best (non-overlapping) matches are only those that are local minima of distance. Table 4.2 illustrates the avoidance of overlapping matches.

Table 1.3: Avoiding overllapping matches with a distance threshold of 2. Normally, we would obtain 4 matches (shown in italics in the last row), but in order to avoid overlapping, our algorithm may return only those corresponding to local minima (shown in bold in the last row) in the sequences of matches

		T	**C**	**A**	**T**	**G**	**G**
	0	0	0	0	0	0	0
C	1	1	0	1	1	1	1
A	2	2	1	0	1	2	2
T	3	2	2	1	0	1	2
T	4	3	3	*2*	***1***	***1***	*2*

Since, in general, an application may need the overlapping annotations, our implementation allows the user to choose whether he wants to avoid them or not.

4.3 Saving Memory Space

For very long texts, the size of the distance array can be quite big. And it doesn't need to be, if we note that to compute a column, we just need the previous column. Hence, the memory use of the algorithm may be reduced from $O(n.m)$ to $O(n)$. The time complexity continues to be $O(n.m)$.

4.4 Normalization of Distances

Since the size of strings vary, it is not very convenient to deal with absolute distances. One character mismatch in a short word is much more relevant than one character mismatch in a long word. Hence our ***AproximateGazetter*** deals instead with relative distances. The relative distance threshold is specified in a range from 0.0 to 1.0 and then for each word that is searched, an absolute distance threshold is computed by multiplying the relative threshold by the length of the string. Then the algorithm is applied normally with this absolute distance threshold.

5 Evaluation of the ApproximateGazetteer

To evaluate our algorithm, we were primarily concerned with its time performance and its accuracy, in comparison with GATE's ***DefaultGazetteer***.

5.1 Time Performance

It is very clear that ***AproximateGazetter*** is qualitatively much slower than ***DefaultGazetteer***. This happens because ***AproximateGazetter*** has to scan the text once for each word that is searched for, while ***DefaultGazetteer*** compresses all words in a FSM and thus is able to search for all of them simultaneously.

This linearity of ***AproximateGazetter***'s time performance with respect to the number of words that are searched for, constitutes its main disadvantage and restricts its practical usage to small lists of words. This was the price to pay for its extra functionality.

5.2 Accuracy

An interesting way to statistically evaluate the accuracy of ***AproximateGazetter*** would be to measure its recall and precision for a large annotated corpus of noisy text. However we did not have access to such a large corpus and therefore we evaluated our algorithm only in a small domain-specific example, in order to find possible directions for future works and improvements. As language resource to be processed, we used a single XML file ("data.xml") containing several names of cities and their geographical coordinates. A short part of the file can be seen below:

```
<?xml version=‘‘1.0’’ encoding=‘‘ISO-8859-1’’?>
<data>
<countries>
<country>
<name>Argentina</name>
<cities>
    <city>
        <name>Bariloche</name>
        <latitude>-41.150</latitude>
        <longitude>-71.300</longitude>
    </city>
    <city>
        <name>Buenos Aires</name>
        <latitude>-34.613</latitude>
        <longitude>-58.470</longitude>
    </city>
</cities>
</country>
```

We processed this file with ***AproximateGazetter*** (with "avoidOverlapingAnnotations" set to true and "normalizedDistanceThreshold" set to

Table 1.4: Detailed Performance of the ***DefaultGazetteer*** relative to the performance of the ***AproximateGazetter***

CorrectMatches	PartialMatches	Missing	FalsePositives
265	1	99	0

Table 1.5: Summarized Performance of the ***DefaultGazetteer*** relative to the performance of the ***AproximateGazetter***

Recall	Precision	F-Measure
0.726	0.9962	0.8399

0.15) and with GATE's ***DefaultGazetteer***. Then we compared the annotation sets produced, by using GATE's Annotation Diff Tool. The summarized output of the tool is displayed in tables 5.2 and 5.2:

It is important to note that this test takes the annotations produced by ***AproximateGazetter*** as standard ("key") and measures how well the annotations by ***DefaultGazetteer*** ("response") agree with the key. We can see that ***AproximateGazetter*** produced the same annotations produced by DefaultGazetter ("Correct Matches"=265 and "Partially Correct matches"=1) plus some more ("Missing" = 99). By analyzing the details of the annotations that DefaulGazetteer missed, we can see that:

- Many of the misses were due to the fact that ***DefaultGazetteer*** is not able to produce annotations within annotations. The city "San", for example, was not annotated when it occurred inside a larger city name ("San Pedro de Atacama", "Santiago"). Although there must be reasons for this behaviour of ***DefaultGazetteer***, there may be situations where the inner annotations are also important and shouldn't be ignored. ***AproximateGazetter*** does not ignore them.

- ***AproximateGazetter*** was able to recognize words containing characters with diacritics, which were not detected by ***DefaultGazetteer***: "Münster", "Düsseldorf", "São Paulo".

- it recognized words with extra spaces: "Aguas Calientes" matched "Aguascalientes".

- it handled capital letters well without preprocessing the text to UpperCap: "Aix-en-Provence" matched "Aix-En-Provence".

- it recognized words in different but similar languages: "Sevilla" matched "Seville", "Barcelona" matched "Barcelone", "Brussel" matched "Brus-

sels", "Kopenhagen" matched "Copenhagen" and "Copenhage", "Granada" matched "Grenada", "Cordoba" matched "Cordova", "Iraklio" matched "Iraklion", "Korinth" matched "Corinth", "Hannover" matched "Hanover".

- it had a few false matches: "aracas" (from "Paracas") matched "Caracas", "Argentina" matched "Argentia", "othenburg" (from "Rothenburg") matched "Gothenburg".

6 Conclusions

The goal of this project, to implement an approximate gazetteer for GATE, was successfully achieved and the source code may be downloaded from [9]. Additionaly, some theoretical modifications and improvements of Levenshtein's Edit Distance were discussed. One of its possible uses was qualitatively demonstrated by processing a resource with names of cities in possibly different languages. ***AproximateGazetter*** opens a new area of application for GATE, which is that of processing language resources with noise and errors. However, it is just a first step and much future work has to be done. Among the possible directions for future work, we would like to mention:

- Statistically test the algorithm with large degraded and noisy corpus.
- Find ways of improving the speed of the algorithm, possibly trying to combine ideas of FSM into the dynamic programming algorithms. We note, for example, that for two words with the same prefix of length n, the first n rows of the dynamic table are identical. Therefore we could try to devise a better data structure to allow us to save this kind of repeated computations for prefixes.
- Extend our algorithm to deal with ontologies, as the ***DefaultGazetteer*** was extended to the ***OntoGazetteer***.
- Give more options to the user. For example, how spaces in words or between words should be handled or whether subwords in the text can be matched to the words in the lists of the gazetteer.
- Experiment with other string distances, different from Levenshtein's.

Bibliography

[1] GATE - http://gate.ac.uk/.

[2] H. Cunningham, D. Maynard, V. Tablan, C. Ursu, and K. Bontcheva. Developing language processing components with gate, 2001.

[3] H. Cunningham, Y. Wilks, and R. Gaizauskas. GATE – a general architecture for text engineering.

[4] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[5] Patrick A. V. Hall and Geoff R. Dowling. Approximate string matching. *ACM Comput. Surv.*, 12(4):381–402, 1980.

[6] Tom Kenter and Diana Maynard. Using GATE as an Annotation Tool, Jan 2005.

[7] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[8] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[9] Bruno Woltzenlogel Paleo. Bruno woltzenlogel paleo's website: Software section (http://bruno-wp.blogspot.com/2007/03/software.html or http://paginas.terra.com.br/arte/lua/software/bwpgazetteer.html).