RadeX

Developers Manual

Tiran Software

[2008]

TABLE OF CONTENTS

1. OVERVIEW

1.1. About RadeX

RadeX is a product of Tiran Software developed as our senior project. It's capable of extracting useful information out of radiology report and stores them in database such that it's easily searchable on demand.

1.2. Prerequisites

1.2.1. Essential Software

- JRE 1.5 or above
- PostgreSql

1.2.2. Recommended Hardware

- 1.6Ghz and above CPU
- 1Gb Ram
- 500Mb Hard disk space

2. ARCHITECTURE

2.1 PREPROCESSOR

2.1.1. The Task

The main task of the preprocessor was to split reports into groups , groups into sentences and sentences into words. Then we added many capabilities such as simplifying sentences by eliminating some words or word phrases, combining words into word groups and setting some basic semantics for words which contains numbers or other special characters.

2.1.2. Functions

The source code of the preprocessor is located under arac package within Preporcessor.java. Here are some of the important functions.

2.1.2.1. cumleleriAyir()

Input(s): File
Output: List<Cumle>

This function reads the report line by line and tokenizes it. Then for each token it does the following:

- Try to find the root and suffixes of the word using zemberek
- If zemberek fails, it tries to find a similar word using SpellChecker(if enabled).
- Then, it tries again zemberek with the similar word.
- It construct MyKelime object using the word and mark the word as recognized or unrecognized.

2.1.2.2 KisimlariAyir()

Input(s) : List<Cumle>
Output: none

This function splits report into groups, baslik, klinik bilgi, bulgular and sonuclar.

2.1.2.3. HELPER FUNCTIONS

These are the functions that simplify sentences and give some basic semantics to words.

setBasicAnlam() : It gives semantics for the words Yuzde, Aralik , Olcum , Omur, Omur_Araligi, Kacinci, Sayisal. Note that these are the enumaretions in IsimAnlamTipi.

firstTouch() : It does some miscellaneous tasks like converting “ya da” into “veya”. Also it gives semantisc for # cm,mm, and so on.

Birlestir() : It reads gloss_birlestir.txt and combines words into word groups. It adds one single MyKelime object with isKelimeGrubu= true, and deletes the actual words from the sentence.

oncekiyle_birlestir() : It is similar to birlestir(), but it reads oncekiyle_birlestir.txt and combines the words with the previous token.

sonrakiyle_birlestir () : It is similar to oncekiyle_birlestir().

kimGitsin() : It reads kim_gitsin.txt and removes the instances of these word(s) from the report.

kimVeOncesiGitsin() : It’s similar to kimGitsin() but it can remove up to sentence start.

kimVeSonrasiGitsin() : It’s similar to kimVeOncesiGitsin().

deDaIcerisinde() : This function combines the two words X içinde/içerisinde into Xde. (karaciğer içerisinde -> karaciğerde)

2.1.2.4. OTHER

There are two other important functions, namely `setLoggingOn(Boolean) and setSpellChecking(Boolean),` which sets logging and spellchecking on/off respectively.

2.2. Rules package

In our approach to text mining for medical reports, we've seen that most of the sentences are semi-structured and appropriate to be parsed by regular expressions. In order to make use of this fact, we decided to parse each sentence by a different template. These templates are hard to directly code in a imperative language. For this purpose, we implemented our own parser generator. By help of it, we are able to write our templates(rules) in a declarative fashion.

This package holds the class RuleBase, which is our parser generator, that takes a rule source.and generates the corresponding java code.

2.2.1 The format of rules

Here is a sample rule

```
# mediastende ve her iki hiler bölgede kitle saptanmamıştır
@@
.ret dbase.Problem_Bulgusu(dbase.Data)
...
(< ~sonEkDeDaMi() > { $i.bolge_niteleyici += this })*
<sonEkDeDaMi() ~copMu()> { $i.bolge = this.govde() }
( <baglacMi()> { $i++ }
(< ~sonEkDeDaMi() | organMi() > { $i.bolge_niteleyici += this })*
<sonEkDeDaMi() ~copMu()> { $i.bolge = this.govde() }
)+
(<~copMu()> { $A.problem += this })*
<problemMi()> { $A.problem += this.govde() }
( <baglacMi()> { $A.problem += this }
(<~copMu()> { $A.problem += this })*
<problemMi()>{ $A.problem += this.govde() }
)+
<anlamtipi:Y.var> {
        $A.sonuc = this.varlik;
        $A.kesinlik = this
}
```

1. there can be more than one rule in a rule source file, all the rules start with @@
2. in the next line the return value of this rule is stated by .ret,
    1. in the sample rule case dbase.Problem_Bulgusu is the actual class that this rule tries to match.
    2. the class in parantheses(dbase.Data) is the superclass of the actual class.
    3. finally this rule returns List<dbase.Data> on a successful match.
3. Next line is '...' which signifies the start of the regular expression
4. to match a Kelime in a Cumle the conditions between < and > are checked and the actions between { and } are executed.
5. '~' signified NOT, '|' signified OR
6. Whitespace between two conditions make up for AND
7. the condition literals are either
    1. a boolean method in MyKelime class, like sonEkDeDaMi().
        i. So if you add a boolean method in MyKelime class, you can use it for template matching directly
    2. an equality comparison of a field in MyKelime,

        i. equality operator is ':'
        ii. anlamtipi:Y.var is transformed to getAnlamtipi.equals(Y.var)
            1. So to be checked fields must have corresponding getter methods.
        iii. The left hand side of the comparison also might be a method call like getAnlamtipi():Y.var

8. the action literals are as follows
    1. there are three variable references
        i. $i references the dbase.Data that is to be constructed
        ii. $p references the previous one
        iii. $A references all of them.
    2. $i++ signified that template matching should continue with the construction of a new dbase.Data. A new dbase.Data instance is added to the return list.
    3. this keyword refers to the MyKelime that matches with the current conditions.
        i. if it is used solely, getIcerik method of MyKelime is executed, so $i.problem = this means, set the value of problem field of current dbase.Data to the value returned by getIcerik method of the matched MyKelime.
        ii. You can append a method call to this, so you can set the current match to a different value like
            1. $i.problem = this.govde()
9. the regular expression operators are similar to perl and java regular expressions
    1. + --> matches 1 or more occurrences
    2. * --> matches 0 or more occurrences
    3. ? --> matches 0 or 1 occurrence.
10. The template matcher is greedy; it always tries to match the first states as much as possible.

2.2.2 ADDING A NEW TEMPLATE TO RADEX

the template source file should be put in src/rules/finalizerRules/sources
it must have the extension .txt
after the rule file is created main method of RuleBase class has to be executed once as
java rules.RuleBase

2.2.3. THE FORMAT OF GENERATED FILES

The generated java files are nothing but Nondeterministic Finite Automata simulations. All of the generated files extend the rules.NFA class, and have 2 public methods

a) boolean matches(Cumle c)
This function checks if this NFA can match Cumle instance c;

b) List<dbase.Data> action()
This function returns the Data instances that the previous matches call matched.

An example usage for checking whether a template matches a cumle

```
NFA<?> nfa = new rules.finalizerRules.mamo.Rule0();
Cumle c = new Cumle(“sağ akçiğerde lezyon var”);
if(nfa.matches(s)){
        System.out.println(nfa.action);
```

2.3. LuceneUse

2.3.1. The task

This class is used for basically three facilities;

- Indexing and google-like free-text searching reports
- Obtaining similar reports of input report
- Spell-checking for wrong written words

2.3.2. Functions

2.3.2.1. indexReport()

Input(s):

reportID : ID of report that will be indexed

bulgular : extracted information of report (needed for imilarity indexing)

Output: none

Usage: This function can be called without any declaration.

2.3.2.2. similarReports()

Input(s):

reportID: ID of target report

Output: Vector<String> of report Ids that is similar to target report. Extracted information of target report that has “var” or “anormal” property in “sonuc” field, is also indexed and stored as a Lucene.Document.Field called “bulgular”. Report that will be search for its similar according to this field using vector space model.

Usage: This function needs initialization of a LuceneUse object.

2.3.2.3. yakinKelime()

Input(s):

kelime: the word that will be checked

Output: The word in lexicon that most similar to input

Usage: Function does not need any initialization of class.

2.3.2.4. indexSpellWords()

Input(s): none

Output: none

Usage: This function is used to construct an index from gloss text files in order to use in spell-check.

2.3.2.5. Helper Functions

2.3.2.5.1. indexDocs()

Input(s):

writer: IndexWriter object initialized to construct an index for spell-check.

file: gloss file for lexcion to be indexed.

Output: none

Usage: It is used in indexSpellWords function to index multiple gloss files as spell-check index.

2.3.2.5.2. processBulgular()

Input(s):

bulgular: findings that will be formed during analysis phase off application

Output: Regional and problem findings whose values of “sonuc” field are either “var” or “anormal”.

Usage: It is used in indexReport function to index only extracted and meaningful information to enable finding similar report facility

2.4. Compilation

Radex was developed as an eclipse project. Adding it to eclipse as a project is the easiest way to compile all the source files.

To add it as a project to eclipse, follow these steps.

1. If not installed, install eclipse
2. Run eclipse
3. From File menu, click New->Project
4. Select "Java Project"
5. Type "radex" in the Project name field.
6. Select the radio box "Create project from existing source"
7. Click "Browse", point to the radex source main directory, click OK.
8. You're done, wish you a happy development.

2.5. Future Work

Radex's performance of recall and performance is mostly affected by the use of templates. We have currently 30 templates for the finding sections of the reports and 18 templates for the result sections of the reports. To boost its performance, new templates might be added as explained in the previous parts.