Football Information Extraction System

ESTEVE LLOBERA
Lund University
e6801661@est.fib.upc.es

CIAN DALTON
Lund University
cian.dalton@student.cs.ucc.ie

JULIO ANGULO
Lund University
djjupa@canada.com

Abstract

We are going to make a program able to extract information from sportive texts. We will focus on football texts and the program will extract the winner and loser teams, as well as the final score and the location where the match has been played.

The program will deal with texts in two languages: english and spanish. The user has to tell to the program what language to use to analize the text because it will not try to identify the language.

1 Introduction

This system consists on an information extraction program that extracts relevant information from football related articles. Relevant information is considered to be the name of the participating teams, the resulting score of the match played and the location of the match. In other words, the program basically analyses the contents of the article and returns relevant information about the match.

In order to analyse the article the program follows some stages of development. At each stage the text is analyse in a different way, trying to obtain the correct result at the end. There are five stages of development, tokenizing the text, splitting each sentence, analysing each of these sentences, search for relevant patterns, look for important results, and finally display the results. These stages are explained in detail in this article.

The program will deal with texts in two languages: English and Spanish. The user has to tell to the program what language to use to analyze the text because it will not try to identify the language. The default language is set to English, in case the user fails to choose the language of the article to be analyzed

2 Program Implementation

The project has been made using a design in layers. It has 5 layers (see figure X): tokenizer, sentence splitter, phrase analyzer, pattern searcher and table analyzer. We will make an small overview of all the stages now, and they will be explained in more detail later. Each layer generates an intermediate representation of the analyzed text. The tokenizer returns an array with the tokenized text, the sentence splitter splits the tokenized text in sentences, which are just an array of tokenized words. The phrase analyzer is called for each sentence found, and returns an instance of a class Phrase for each one. The pattern searcher gets the entire Phrase objects and search for sportive patterns that can provide information about a football match. If a sentence contains relevant information, it is stored in a class called Match, which contains winner, loser, score and location. Of course, most of those fields will be filled in empty because a sentence usually doesn't contain all that information. For each sentence with useful information the pattern searcher will create a new Match object with that information. Finally, the pattern searcher returns a table with

all those Match objects, and the table analyzer will try to merge some of those "matches" in order to get all the possible information (score, location, etc.) and to return the real match.

2.1 The Tokenizer

The first step on processing the text is tokenizing it. The program uses Java's tokenizing mechanism for achieving this task easily. The class *StringTokenizer* returns each token that is found in the text. This particular program also considers the following characters as tokens: '!', '/', '?', ':', '-', ';', '¿', '\n', '\t', '.', ',', '(', ')'. Tokenizing the text makes it easier for analysing each word and each sentence in the later stages. This is a very simple step that it is very easily performed with the help of Java classes.

At this stage the program already considers the difference in language. We found that it was useful to recognize the accented letters at the beginning of the text and transformed them into a not accented letter. In Spanish all the vowels can be accented (á, é, í, ó, ú) and this can bring complications at later stages. In general, if there is a word in the text, which contains an accented letter, this letter gets transform to its non-accented equivalent.

2.2 Sentence Splitter

The constructor for this class takes two parameters, the input (String[]) and a Boolean value indicating the language of the input text, either English or Spanish.
The class has two major methods, *thereIsSentence*, which returns a Boolean value indicating whether or not there are any sentences left in the input text, and *nextSentence* which returns the next sentence in the input array. The program uses a variable *startIndex* to indicate the index position where each sentence starts. This index is incremented with each step of the process and was critical to both of the methods.
The method *thereIsSentence()* simply returns true if the length of the input was greater than the start Index, i.e. the *startIndex* has not reached the end of the input array and therefore there is still at least one more sentence to split.

The method *nextSentence()* uses a loop to go through the input array and firstly checks if each character equaled a sentence delimiter (".", "?", ",", "!", ";" or ":") and considered all characters between the two delimiters a sentence by adding the sentence to our list (LinkedList) and incrementing the *startIndex* until it is greater than the index of the delimiter so we can move on and find the next sentence. The process then repeats until the end of the input is reached and thus all sentences are split or separated. Each LinkedList *nextSentence* is converted to an array and returned. The list that is returned, and which is supposed to contain one sentence of the text, is furthered analyzed by the *PhraseAnalyzer* class.

2.3 Phrase Analizer

The phrase analyzer is called for each sentence in the text, so it doesn't have any knowledge about the sentences before. We would have liked to have a good phrase analyzer, but it had increased too much the complexity of this layer and besides, it's not the main objective of this project. So we decided to use a simple approach to a full phrase analyzer. For each language (English and Spanish) we keep a list of verbs, prepositions and conjunctions. The idea is to be reading the sentence until find one of those, that we have called, bounding words. Each group of words between two of those bounding words is a group in the sentence (the subject, the verb, the object or any prepositional sentence). The algorithm is the following: we consider the type of a group depending on the first word of the group. What we find until the first bounding word, is always considered to be the subject of the sentence. Once we find a bounding word, if it is a verb word, this will be the verb of the sentence. In this case, we start reading the words after it. While the word is a verb word, we attach that word to the verb of the sentence. This way we deal with sentences with a composed verb. If the word is a preposition, we consider it a prepositional phrase, and if it's not a preposition, we consider it as the object of the sentence. A conjunction is only used to separate groups.

If we find more than one verb or direct object, only the first one is used. The other ones are ignored.

So, if we analyze the next sentence:

10 November 2003 Blackburn Rovers FC ended a run of five consecutive
Premiership defeats with a nervy 2-1 win against Everton FC at
Ewood Park tonight.

we get:

- Subject:10 November 2003 Blackburn Rovers FC ended a run
- Verb: win
- Object: [NULL]
- PP: of five consecutive Premiership with a nervy 2 - 1 against Everton FC at Ewood Park tonight

ended is not in the verb list so it's not considered a verb.

This information is stored in a Phrase object, which has the following structure:

```
public class Phrase{
  public Group subject;
  public Group verb;
  public Group object;
  public Group[] pp;

        ...
}

class Group{
 public static final int NOUN = 0;
 public static final int VERB = 1;
 public static final int PREP = 2;
 public static final int ADVERB = 3;
 public static final int CONJUN = 4;

 public int groupClass;
 public String[] data;

        ...
}
```

The phrase object will represent the analyzed phrase, and that's what the phrase analyzer gives to the pattern searcher.

2.4 Pattern Search

After the phrase is analysed and divided into subject, verb, object, and prepositions the program searches for relevant patterns on each of these fields. To do this the program uses a super class PatternSearch and two subclasses that extend it PatternSearchEnglish and PatternSearchSpanish in order to deal with the difference in language. These subclasses basically consist of a list of words or sentences that are likely to appear in the different fields, we call them *patterns*. There are patterns related to relevant verbs, to relevant words, to locations, to prepositions, and to scores. For example, some relevant patterns concerning the name of a team were coded in java as an array of patterns elements (i.e. an array of type Pattern):

```
Pattern[] wordPatternSpa = {
 Pattern.compile("[A-Z]\\w*"),
 Pattern.compile("[A-Z]\\w* [A-Z]
                 [A-Z][A-Z]*"),
  . . .
}
```

where the first element of the array will identify all words starting with Capital letter, the second element will identify all words starting with capital letter followed by a space and at least two capital letters (or initials), therefore it will recognize words like ‘Barcelona FC’ or ‘Barcelona FFC’.

At this stage the program takes a big step in determining the winner and the loser of a match, as well as the score and the location. To determine the winner and loser it considers five types of verbs that can be relevant to a football article, verbs about winning and losing in the active form, about winning and loosing in the passive form, and about drawing. In general, depending on the type of verb, either the sentence before or after it is considered to contain the name of either the winner or the looser. For example, if the phrase being analyzed at the moment contains a winning verb in the passive (e.g. *has been won*) form then the program assumes that it is likely to find the name of the winner after the verb. In the other hand, if the phrase has a winning verb in the active form (e.g. *triumphed*) then the program considers the winner

to be found before the verb and therefore the user after the verb. This doesn't mean that the final output of the program depends on finding one of these verbs and concluding we can find the appropriate winner and looser, because there are many phrases to be analysed and the results of all of them will be compared at a later stage.

To determine the location of the match, the program considers sentences or groups containing only two prepositions related to location *'at'* and *'in'* (in Spanish there is only one preposition *'en'*. Reading over some corpus we found that it is only after this two words that the location of the match is mentioned in an article. We can encounter sentences as *The match was won by Barcelona in Camp Nou* where it is very simple to detect the location. In the other hand an article might have the sentence *The match was won by Barcelona in September*, for this reason we decided to have a list of words that the program might misinterpret as important, we called this list the unwanted patterns and it includes names of months, days of the week, capitalized prepositions, etc. and it helps the program to get rid of words that are not likely to appear in the final result. A text can also contain the sentence *The team won the game at the beautiful city of Bristol* and again by taking only the relevant patterns the program gets rid of the unwanted *the beautiful city of* to conclude that the relevant location is *Bristol*.

The process of determining the scores is slightly different. First the program tokenizes each of the groups in the phrase to look for a pattern of the form *Number Union Number*, where *Number* is any integer and *Union* is a word or character that appears between the two numbers which is commonly use to display a result, such as '-', 'to', 'against', 'by', ':'. If a pattern of this type is found, then the program considers these two numbers to be a possible score. However, the numbers found can happen to be only a partial result or the result of another match. We found a big difficulty in finding the correct final score when the text presents more than one on these *Number Union Number* patterns. For this reason we implemented the program so that it assumes that the pattern with the highest integer numbers is indeed the final score of the match.

All the results found by the *PatternSearch* class are considered possible results. These results are stores in a list of *Matches*, which is analyzed at the next stage of the development process of the program.

2.5 Table Analyzer

The pattern searcher yields a table that contains many Match objects. As we already know, each Match object represent a match, or part of the information of a match. The objective of this layer is to merge all the matches, so we can get a match with all the information given in the text. The reason for this is because, as we analyze sentence per sentence, each Match object created only contains the information about the match that appears in that sentence. But in most of the cases, the information of a match is given in multiple sentences. For example you can say the team A won team B in one sentence but the score can be in another one. For this reason, the table analyzer will merge all the matches together, in all the possible combinations. Two matches will be merged only if there is not contradiction when merging them.

The table analyzer works in 2 steps. First it gets all the team names it finds on all the matches. Second, it merges the matches considering the team names found.

When the pattern searcher creates a Match object, it doesn't take care about the team names. It only knows that the subject or the object of the sentence may refer to a team name, and fills in the fields with that information. So when we get two matches, for example one with a winner called "The great FC Barcelona team" and another one with a winner "The Barcelona team", we must tell the system that those two teams refer to the same team: Barcelona. That's the objective of the first step in the table analyzer layer. Once we have all the team names that appear in the text, it is easier to know when two matches can be merged using the real team name and not all the expression.

The method we have used to get the team names is quite intuitive: we look for team

patterns in all the winner and loser field of all the Match objects. There are not many teams' patterns, but the problem is that it's very difficult to distinguish team names from football players. For example FC Barcelona is a team, but Javier Saviola is a football player, and both will be considered as team names.

Once we have the team names, the program starts to merge all the possible matches. Two matches will be merged if no contradiction appears when merging them. Contradictions are the winner and the loser would be the same team, different score or location, or different winner or loser. For example if a match with a winner Barcelona and a match with a winner Madrid cannot be merged. But a match with the winner null and the loser Madrid and another match with the winner Barcelona and the loser null can be merged. The resulting match will have winner Barcelona and loser Madrid. This resulting match will be added to the list of matches, so it can be used to be merged again.

When we have merged all the matches, we need to select the match the text is talking about. For that reason, each match object is augmented with a counter. When the pattern searcher creates a match, if the fields winner and loser are both null, the counter is 1, if either the winner or the loser is not null, the counter is 2, and if both winner and loser are not null, the counter is 3. When two matches are merged, the counter of the resulting match will be the addition of the two original counters. At the end, we return the match with the higher counter.

3 Measures of Performance

We were confident that the precision and recall of the program in general were fairly good. We tested the system with approximately 30 Internet articles narrating football matches and we estimated a recall and precision between approximately 40% and 60%. In the case when one article mentioned more that one match this percentage of recall and precision dropped considerably. Also, when the text was extremely large, the displayed result was usually not the correct one, and therefore precision and recall also were reduced by approximately 30%, which is a big decrease in performance.

Error handling was not developed in detailed, however we assure that the program never terminates abruptly for any reason. If the input is not correct and the language is selected badly the program might display merely garbage (we said earlier that the program is not concern with checking the language of the input text), but it will not create exceptions, freeze or terminate. Instead, the user is given a chance to clear the text area and try analyzing the text again.

4 Conclusions, Observations and Future improvements

From developing this program we found many interesting facts and difficulties about processing languages with machine instructions. We also realized about the differences and similarities in processing two distinct languages, but fortunately we found it easy to separate the implementation of the two.

There are many things that could have improved in the program. The following is a list of some important observations and future improvements that we were aware of, but because lack of time couldn't be implemented in this version.

1. A better phrase analyzer. A bad one accumulates too much garbage for the *TableAnalyzer* to analyze. Statistically the names of the teams the match talks about are more likely to appear more times in the text, therefore the counter should be enough to get the correct name of the teams, but in the real life it's not so. This could definitely be improved by removing the garbage generated due to the bad phrase analyzer.
2. Optimally the analyzed text only talks about one match, however texts relating multiple matches can also be analyzed. The match with more references made on the text should be the final result, although the program will not always return this as the correct result. In general

the program performs much better if the article only talks about one football, both precision and recall are higher in this case.

3. An improvement in the user interface is the liberty for the user to choose a text file located in his machine, instead of having to copy/paste the article into the text field.
4. The order of the patterns in the array of patterns could be sorted in a way that the most likely pattern appears in the first index of the array, the second most important in the second index and so on. In this way the performance of the system will improve since it doesn't have to go through many elements of the array. In other words, we could have carried out a statistical study to find out the likelihood of occurrence of each of the patterns. In this way we can place the most likely occurring patterns at the beginning of the array of patterns so that the program doesn't have to go through the whole array.
5. Java was a useful tool in the development of the program, since it has build in classes that are very useful for language processing, such as string and stream tokenizer and a Pearl like pattern functionality.
6. The differences in the language were handled by having subclasses for each language that extended a superclass with the majority of the functionality. The subclasses included mostly the relevant vocabulary that can be found in the two different languages.
7. The handling of errors was not done with too much care. Handling all exceptions and possible technical errors could make the system more robust and trustful. However, we made sure that the program never crashes, as mention above.

Like every software development process, there are still some things that we could have done better, but because lack of time, tools and some knowledge we were not able to implement it as good as we would like to. However we were very satisfied with the final result of the system since the accuracy and precision were beyond our expectations

References

Pierre Nugues, 2003. *Assignment #2: Information Extraction.* http://www.cs.lth.se/Education/Courses/EDA171/cw2.html

Pierre Nugues, 2003. *Corpus Processing Tools*

Erik Lindvall and Johan Nilsson. *Extracting information from Sport Articles in Swedish using Pattern Recognition*. Lund University