Accepted at CICLING 2015, Cairo, Egypt, April, 2015

# Arabic Natural Language Processing from Software Engineering to Complex Pipelines

Younes Jaafar, Karim Bouzoubaa

Mohammadia School of Engineers, Mohammed Vth University - Rabat, Morocco.
younes.jfr@gmail.com
karim.bouzoubaa@emi.ac.ma

**Abstract.** Arabic Natural Language Processing (ANLP) has known an important development during the last decade. Nowadays, Several ANLP tools are developed such as morphological analyzers, syntactic parsers, etc. These tools are characterized by their diversity in terms of development languages used, inputs/outputs manipulated, internal and external representations of results, etc. This is mainly due to the lack of models and standards that govern their implementations. This diversity does not favor interoperability between these tools or their reuse in new advanced projects. In this article, we propose APIs and models for three types of tools namely: stemmers, morphological analyzers and syntactic parsers, using SAFAR platform. Our proposal is a step for standardizing all aspects shared by tools of the same type. We review also the issue of interoperability between these tools. Finally, we discuss pipeline processes.

**Keywords:** ANLP; Software engineering; interoperability; pipelines

## 1 Introduction

The Arabic digital world knows a continuous growth in terms of content (texts, videos, images etc.) From 2000 until 2011, this content has known a large growth estimated by 2501% [1]. Thereby, the processing of this content requires the development of appropriate tools dedicated to Arabic Natural Language Processing (ANLP). Nowadays, many tools for ANLP are already developed, morphological analyzers, syntactic parsers, search engines, machine translation systems, etc.

By exploiting these different tools, the researcher in ANLP needs sometimes to call several ones at the same time in the same project, whether morphology, syntax or even semantic. However, these tools are not always encapsulated in homogeneous and interoperable entities. This heterogeneity is mainly due to the lack of standards (APIs) governing the implementations of the different tools. In addition, most researchers do not take into account the software engineering principles while developing their tools, which affects the efficiency, productivity, flexibility, robustness, etc. It is also due to the various prospects of researchers, their main purpose being the result and not the organization, modeling, reusability, etc.

It should be also noted that it is almost difficult to launch pipelines to produce advanced processing tools. This is due to the diversity in terms of inputs and outputs of the various tools. For example running a pipeline such as: [syntactic parsing > tokenization > normalization > named entity recognition > morphology analysis], would be difficult. To remedy this, one solution consists of adjusting all inputs and outputs to meet the needs of this pipeline only, and this may not be reused for another pipeline in another context. Some pipeline solutions for NLP have emerged such as "ITU Turkish Natural Language Processing Pipeline" [2] and U-compare [3]. However, they show some technical limitations (detailed below) and also no direct support of Arabic.

Thus, our objective in this article is to provide solutions to all these issues using SAFAR platform (Software Architecture For Arabic language pRocessing). Our philosophy is to gather all ANLP tools within a single and homogeneous structure that is flexible, extensible and modular. In previous works [4, 5] we presented the architecture of the platform and focused primarily on the morphology layer. In the current work, we propose improvements for this layer and we propose also a model for the syntactic layer. Finally, we give an advanced use of SAFAR combining multiple already integrated tools that simplifies the establishment of pipeline processes. To illustrate it, we expose the pipeline of an automatic summarization system.

The development of a platform such as SAFAR will contribute in solving problems of interoperability/pipelines. It will also contribute in the development of more advanced applications such as question/answering, applications for social media, opinion mining, sentiment analysis, etc. These latters often use text processing tools, morphological or syntactic tools in order to achieve their results. Therefore, a platform as SAFAR which contains a lot of number of basic tools will be of great utility in developing such advanced applications.

The rest of this paper is organized as follows. The next section presents the problem of interoperability between tools. In section 3, we present SAFAR platform briefly. In section 4, we present the software engineering perspective within SAFAR via its interfaces and models. In Section 5, we present processing pipelines within SAFAR. Finally we conclude and give some future works in Section 6.

## 2 Interoperability Between Tools

The major issue of the heterogeneity between the different ANLP tools is interoperability. In some cases, it is very difficult to integrate some already developed tools inside other projects in order to produce more robust applications. It is also difficult to exploit their outputs directly in another process such as a pipeline since there are no standards. This is due to their different architectures and inputs/outputs.

For example, to develop a machine translation system, one of the approaches requires the use of a morphological analyzer as well as a syntactic parser. To do this, one of the solutions consists of developing both morphological and syntactic analyzers as a first step, then developing the machine translation system based on the outputs of these two tools. Of course, this solution is not recommended since researchers

will have to develop two heavy independent tools before focusing on the automatic translation. Another solution consists of taking advantage of already developed and widely used tools. This will allow researchers to save considerable time and focus on their work. However, even for this second solution, it is not obvious to integrate tools that have heterogeneous architecture and inputs/outputs. Thus, the researcher will waste time once again in order to adapt these tools before being able to use them.

Generally, this problem is avoided by the use of some platforms dedicated to Natural Language Processing such as UIMA [6], GATE [7], NLTK [8], OpenNLP [9], LingPipe [10], NooJ [11] etc. The different tools inside these platforms are homogeneous in their architectures, and thus creating complex processing applications becomes easy. These platforms are more or less mature for the Latin languages. Nevertheless, there is still much work to do for the case of the Arabic language. In addition, these platforms do not meet the needs of the ANLP community because they do not incorporate enough modules and resources for processing the Arabic language. They do not also offer an architecture (API) exploiting the nature of the Arabic language in a clear and effective way. This does not help the integration of ANLP tools within such platforms. This justifies also the lack of processing components of Arabic inside these platforms. A more technical and architectural comprehensive study of the available platforms in regards to the Arabic language and the Arabic NLP can be found in [4] and [5]. Thus, the need for the ANLP community is to have an integrated platform that is modular, flexible and extensible. For this, we have developed SAFAR (Software Architecture for Arabic language pRocessing).

## 3 SAFAR Platform

SAFAR is an integrated platform dedicated to ANLP written in Java. It is cross platform, modular, extensible and provides an integrated development environment (IDE). SAFAR brings together all layers of Arabic Natural Language Processing.

As specified in Figure 1, SAFAR has several layers. The utilities layer includes a set of technical services, the resources layer provides services for consulting language resources such as lexicon, the basic layer contains the three regular layers (morphology, syntax and semantics), the application layer contains high-level applications that use the layers listed above, and finally the client applications layer which interacts with all other layers providing to users web applications, web services, etc.

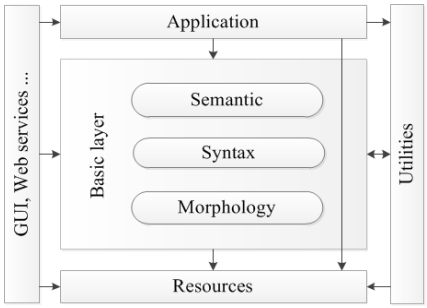

**Fig. 1.** SAFAR platform architecture

Thus, the general idea of SAFAR is to gather the available set of tools that are already developed within a single homogeneous structure, this will promote the interoperability and reusability of this set of tools as long as they will share the same philosophy concerning their architecture. Users can use the tools that suit them with a simple call. They can also add new implementations or modify the existing ones. This gives great flexibility concerning the interoperability and reusability. To our knowledge, there is no similar platform to SAFAR particularly dedicated to the ANLP. So far, SAFAR contains the following most used tools:

**Table 1.** Integrated tools in SAFAR platform

| SAFAR layers | Type | Name of the tool |
|---|---|---|
| Morphology | Morphological analyzers | Alkhalil,BAMA |
| | Stemmers | Khoja, Light10, ISRI,<br>Motaz stemmer, Tashaphyne |
| Syntax | Syntactic parsers | Stanford Parser |
| Utilities | Utilities | SAFAR Normalizer,<br>SAFAR Sentence Splitter,<br>SAFAR Tokenizer,<br>SAFAR Transliterator,<br>SAFAR Benchmark |
| Applications | Applications | Sentence processor,<br>Stem counter,<br>Morphosyntactic processor,<br>Q/A application [12] |

For more information about tools available in SAFAR platform, visit the website: http://sibawayh.emi.ac.ma/safar.

## 4 SAFAR and Software Engineering

Most of Arabic processing tools do not offer structured architectures (APIs) that can facilitate their integration in different projects and also their improvement if necessary. Most of them provide just a graphical interface for interacting with the tool. SAFAR aims to remedy this by providing an API for the set of layers shown in Figure 1 and Table 1. In this way, we guarantee the homogeneity of the set of tools integrated within the platform. In this section, we present the UML models and interfaces for the “Morphology” and “Syntax” layers.

**Stemmers**

Stemming is the process of removing affixes from a word and reducing it to the stem. It should be noted that some stemmers reduce the word to the root instead of the stem [13].

In Figure 2, we present the interface that the stemmers must implement before their integration inside the SAFAR platform.

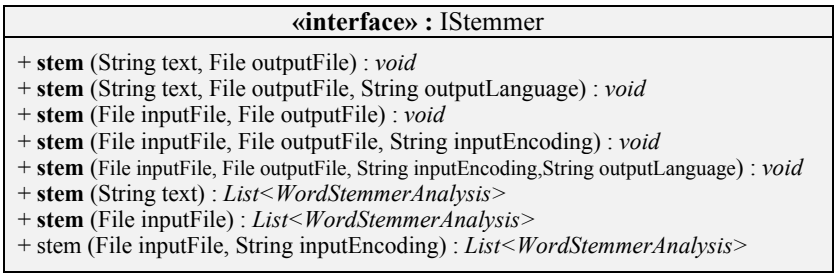


**Fig. 2.** Stemmers interface within SAFAR

The list of parameters of the methods in this interface is defined as follows: *text* is the text to be stemmed, *inputFile* is the file to be stemmed, *outputFile* is the file in which results will be saved, *inputEncoding* is the encoding of the *inputFile, output-Language* is the language of the output file (English or Arabic)

As indicated in Figure 2, we have two types of stem methods: Methods that save stemming results as XML File, and methods that return stemming results as a list of WordStemmerAnalysis memory objects (Figure 3).

Figure 3 shows the model that we propose for stemmers inside SAFAR. The StemmerAnalysis class is used to store one stemming analysis, while WordStemmer-Analysis is used to store all possible stemming analysis (set of StemmerAnalysis) of one word.

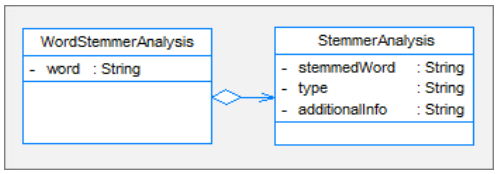


**Fig. 3.** Stemmers model within SAFAR

We propose this architecture because in some cases a word may have several different stems. For example, the word "وهم" will have at least two possible stems according to the meaning of the word. If it is the verb "وَهَمَ " (intent to) then the stem is "هَمَ " (intent to) since the letter "وَ" (and) is a conjunction. Otherwise, if it is the noun "وَهْمٌ " (illusion) then the stem is the word itself "وَهْمٌ " (illusion). Therefore, these special cases should be taken into account in the models in order to be generic.

The WordStemmerAnalysis objects returned by our stem methods can be manipulated directly in more advanced processes such as a pipeline. This gives more strength to our approach since we standardize the inputs and the outputs, this is valid for all methods within SAFAR.

So far, we have integrated the following stemmers in SAFAR platform: ISRI [14], Khoja [15], Light10 [16], Tashaphyne [17], and Motaz stemmer [18]. All these stemmers implement the IStemmer interface of SAFAR. According to [19] and [20], Khoja and Light10 stemmers are the most known within the ANLP community. It should be noted that both ISRI and Tashaphyne are developed in Python, while the others in Java. Despite this, they are used within SAFAR with the same call. Figure 4 gives an example of code calling a stemmer.

```
1  package basic.morphology.stemmer;
2  import ...
3  public class StemmerTest {
4    public void main(String[] args){
5      // Get Khoja stemmer implementation
6      IStemmer stemmer = StemmerFactory.
7              getKhojaImpl();
8      // Text to be stemmed
9      String text = "هنا نص باللغة العربية";
10     // stem the text and save results as xml File
11     stemmer.stem(text, new File("Results.xml"));
12   }
13 }
```

**Fig. 4.** Example of code to call stemmers within SAFAR

At line 6, we call Khoja stemmer implementation. At line 9, we insert an Arabic text to be stemmed. At line 11, we execute the stemmer on the text and save results as an XML file. If the researcher wants to change Khoja stemmer with ISRI, (s)he has only to call the method “getISRIImpl()” instead of “getKhojaImpl()” without changing anything else. This means that a researcher can use any stemmer integrated in SAFAR by changing only the name of the implementation, which will save a lot of time often lost while changing the code of stemmers to adapt them to a given project.

### Morphological analyzers

Morphological analyzers identify the structure of a given word and other linguistic units. They are concerned by the decomposition of words into morphemes and by associating to each morpheme some morphological information such as the stem, root, POS (Part Of Speech), affixes, etc. For example, the word «أحسب» (‘OHsb’) may be analyzed to output the following morphological information: stem = "حسب", pattern = "فَعَلُ ", prefix="أ: هزةفعلاقهم ", root = "حسب". Among the Arabic morphological analyzers we find BAMA [21], MADA+Tokan [22], Alkhalil [23], etc. These analyzers are very important since they are often used in more advanced applications such as syntactic parsers, search engines, machine translation systems, etc.

As in the case of stemmers, morphological analyzers are highly heterogeneous concerning their internal architecture, they do not offer any API, and their outputs are mainly intended for visualization and not for eventual automatic processing or pipelining. This decreases their interoperability with other projects.

To overcome this, we proposed an API and a model to be followed in order to standardize all aspects shared by these morphological analyzers. It should be noted that our proposal takes into account the proposal of ALECSO (Arab League Educational, Cultural and Scientific Organization) [24] concerning information that should be returned by a given morphological analyzer. ALECSO made the effort to standardize the outputs of morphological analyzers in Arabic countries. Figure 5 shows our interface and Figure 6 shows the model that is actually an improvement of the old model [5].

| **«interface» :** IMorphologyAnalyzer |
|---|
| + **analyze** (String text, File outputFile) : *void*<br>+ **analyze** (String text, File outputFile, String outputLanguage) : *void*<br>+ **analyze** (File inputFile, File outputFile) : *void*<br>+ **analyze** (File inputFile, File outputFile, String inputEncoding) : *void*<br>+ **analyze** (File inputFile, File outputFile, String inputEncoding, String outputLanguage) : *void*<br>+ **analyze** (String text) : *List<WordMorphologyAnalysis>*<br>+ **analyze** (File inputFile) : *List<WordMorphologyAnalysis>*<br>+ **analyze** (File inputFile, String inputEncoding) : *List<WordMorphologyAnalysis>* |

**Fig. 5.** Morphological analyzers interface within SAFAR

Researchers have the choice of using either analyze() methods that save results in an output file, or using analyze() methods that return results as memory objects that is "WordMorphologyAnalysis".

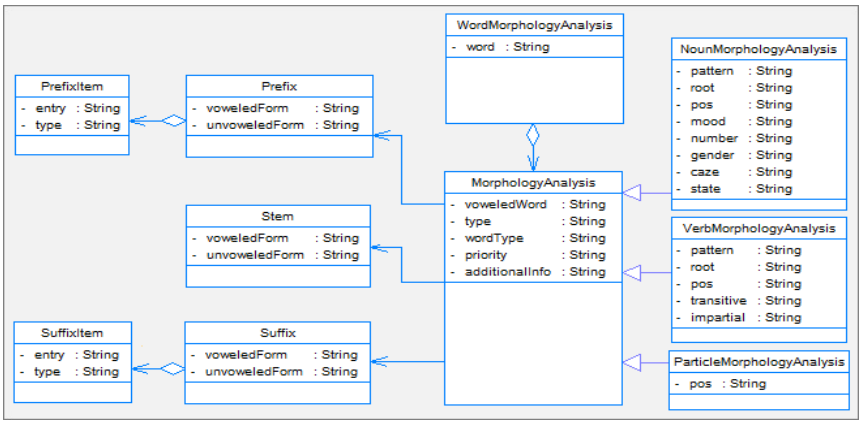


**Fig. 6.** Morphological analyzers model within SAFAR

As indicated in Figure 6, WordMorphologyAnalysis is the main class used to store all morphological analyses of one word. The MorphologyAnalysis class is used to store just one morphological analysis concerning a given word. All information inside the

MorphologyAnalysis class are common to all words types namely verbs, nouns and particles. These three types are represented respectively by VerbMorphologyAnalysis, NounMorphologyAnalysis and ParticleMorphologyAnalysis classes. They have common inherited characteristics from MorphologyAnalysis class, and in addition to that, each one of them has its own unique features. For example, for the type "noun" in addition to the prefix, suffix and stem, it defines other morphological attributes namely pattern, POS and root (which are not available for the type "particle").

The three units namely prefix, stem and suffix are respectively represented in our model by Prefix, Stem and Suffix classes. The prefix and suffix also consist of units respectively represented by the PrefixItem and SuffixItem classes. For example, the prefix "وي" of the word "ويرحم" (to grant mercy) is composed of two PrefixItem units that are, "و: حرف العطف" and "ي: حرف المضارعة". Thus, the researcher can get all the morphological units separately, this will allow better data access and manipulation.

So far, we have integrated in SAFAR the two morphological analyzers that are the best-known within the ANLP community, namely BAMA and Alkhalil. We have adapted them to match the API of SAFAR, and thus guarantee a better homogeneity and interoperability. Figure 7 gives an example of code in order to call morphological analyzers within SAFAR.

```
1   package basic.morphology.analyzer;
2   import ...
3   public class AnalyzersTest {
4     public void main(String[] args){
5       // Get Alkhalil analyzer implementation
6       IMorphologyAnalyzer analyzer =
7        MorphologyAnalyzerFactory.getAlkhalilImpl();
8       // Text to be analyzed
9       String text = "هنا نص باللغة العربية";
10      // Analyze the text and save results as xml File
11      analyzer.analyze(text,new File("Results.xml"));
12    }
13  }
```

**Fig. 7.** Code to call morphological analyzers within SAFAR

At line 6, we call Alkhalil analyzer implementation. At line 9, we insert an Arabic text to be analyzed. At line 11, we execute the analyzer on the text and save results as an XML file. As for stemmers, if a researcher wants to change Alkhalil analyzer with BAMA, (s)he has only to call the method "getBAMAImpl()" instead of "getAlkhalilImpl()".

### Syntactic parsers

The syntax level describes how to organize tokens in order to build statements. It is the scientific study of sentence construction. StanfordParser [25] is one of the most used syntactic parsers of the Arabic language within the ANLP community, it is written in Java and can be run either through a graphical interface or using a command line.

As for the morphology layer, we have implemented an API and a model for the syntax layer to standardize it within the SAFAR platform. Figure 8 shows the interface to be implemented by the syntactic parsers. We have overloaded the parse method in order to cover several types of inputs: simple sentences, an array of sentences or a file containing one or a set of sentences. Thus, users will be able to call the method that suits their needs.

| **«interface» :** IParser |
| --- |
| + **parse** (String sentence, File outputFile, String outputTreeType) : *void*<br>+ **parse** (String [] sentences, File outputFile, String outputTreeType) : *void*<br>+ **parse** (File inputSentences, File outputFile, String outputTreeType) : *void*<br>+ **parse** (File inputSentences, File outputFile, String outputTreeType, String inputEncoding) : *void*<br>+ **parse** (String sentence) : *List<SentenceParsingAnalysis>*<br>+ **parse** (String [] sentences) : *List<SentenceParsingAnalysis>*<br>+ **parse** (File sentences) *: List<SentenceParsingAnalysis>*<br>+ **parse** (File inputSentences, String fileEncoding) : *List<SentenceParsingAnalysis>* |

**Fig. 8.** Syntactic parsers interface within SAFAR

Figure 9 shows the model containing the returned objects after the parsing. Our model takes into account the syntactic tree and also the dependencies between the different units of the sentence in order to bring together all information in one model.

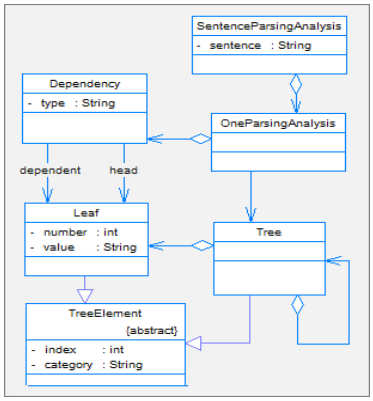


**Fig. 9.** Syntactic parsers model within SAFAR

After the syntactic processing, the parse method returns the results as a list of objects of the SentenceParsingAnalysis class. Each object concerns an input sentence and contains one or more syntactic analysis (OneParsingAnalysis), which justifies the aggregation between the two classes: SentenceParsingAnalysis and OneParsingAnalysis. Thus, our model will cover all the possible cases where we could end up with a sentence that has several different parsing analyses. A syntactic analysis OneParsingAnalysis contains a syntactic tree that is a Tree object and a list of dependencies represented by the Dependency class. Every dependency has a “head” and a “dependent” that are actually Leaf objects. In a syntactic tree, any element that is

not a Leaf then it is a Tree or a sub Tree, and every Tree element may contain a set of sub Tree and a set of Leaf recursively.

We have implemented some methods within the SentenceParsingAnalysis class in order to retrieve and display a set of information that are: the syntactic tree, dependencies and chunks. These information can be outputted in different formats: text, XML file, etc. The OneParsingAnalysis class is also provided with a set of methods allowing the manipulation of the trees and dependencies.

It should be noted that StanfordParser is the only syntactic parser that has been integrated within SAFAR till now. Others such as SYNTAX [26] and MASPAR [27] do not offer the source code and cannot therefore be integrated. Figure 10 gives an example of code in order to call syntactic parsers within SAFAR.

```
1  package basic.syntax;
2  import ...
3  public class SyntaxTest {
4    public void main(String[] args){
5      // Get StanfordParser implementation
6      IParser parser = ParserFactory.
7              getStanfordParserImpl();
8      // Sentence to be parsed
9      String sentence = "هنا جملة باللغة العربية";
10     // Parse sentence and save results in XML file
11     parser.parse(sentence, new File("Results.xml"),
12             ParserOutput.XML_TREE);
13   }
14 }
```

**Fig. 10.** Code to call syntactic parsers within SAFAR

At line 6, we call StanfordParser implementation. At line 9, we insert an Arabic sentence to be parsed. At line 11, we execute the parser on the sentence and save the result tree as an XML file. A researcher can use another parser by changing the method "getStanfordParserImpl()" with the appropriate one.

## 5 Pipelines

In the previous sections, we discussed only one side of Arabic NLP which concerns processing tools separately. However, researchers in ANLP have sometimes to call two or more tools in a sequential way, that is to say, the output of one of them will be the input of the other and so on. This technique saves a lot of time for researchers. For example, if a researcher has an Arabic text with diacritics that (s)he wants to process with a morphological analyzer, then the best solution is to use a tool to automatically remove the diacritics and output a text which will be taken as input by the morphological analyzer.

Some pipeline solutions for NLP have emerged such as "ITU Turkish Natural Language Processing Pipeline" [2] which is dedicated to the Turkish language. This pipeline has several limitations namely: It is fixed as follows: *Input* > *Tokenizer* > *Normalizer* > *Morphological Tagger* > *Named Entity Recognizer* > *Dependency Parse* > *Output*, this means that it follows one path and the user can not intervene to remove a

tool or add another. In addition, this pipeline can be run only through a web interface, which limits its use in offline projects.

There is also another pipeline solution called U-compare [3]. It provides access to a collection of ready-to-use, interoperable, natural language processing components. U-Compare allows users to build NLP workflows from these components via a drag-and-drop interface. Since U-compare is based on UIMA, it has the same limitations concerning the support of Arabic. Furthermore, the user is limited to the use of the GUI, which prevents from developing a customizable code using an API.

SAFAR platform takes into account all these limitations. We modeled our interfaces for the different tools in a way to cover a set of inputs and outputs as presented in the previous section. Thus, a researcher in ANLP would have a large choice when calling methods. He could easily create customizable pipelines directly or with very little manipulations.

**Automatic summarization as example of pipeline**

Arabic Automatic summarization is a complex process which involves the use of several ANLP tools such as text processing tools, stemmers, morphological analyzers, etc. Figure 11 shows an example of a simple possible pipeline that extracts the most significant sentence in a given text.

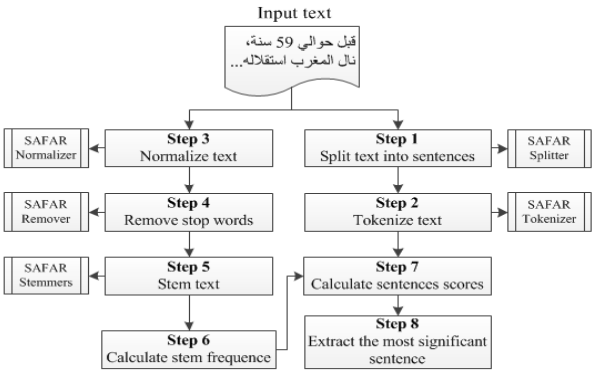


**Fig. 11.** Example of pipeline within SAFAR: case of automatic summarization

The pipeline shows the establishment of a simple Arabic text summarizer system (inspired from [28]) based on the use of different modules and resources of SAFAR platform (the list of available tools are listed in table 1). The summary consists of returning the most significant sentence in the text. This process is done in eight main steps as follows:

1. Segmentation: In this step, we identify each sentence of the text using the "SAFAR Sentence Splitter" tool. This tool takes a text as input and returns an array of sentences as output. It can be called simply as follows:

```
SAFARSentenceSplitter sp = new SAFARSentenceSplitter();
String[] result = sp.split("text");
```

2. Tokenization: we tokenize each sentence of the step1 into a sequence of words using the "SAFAR Tokenizer" tool. This step allows calculating the frequency of words in the text. This tool can be called as follows:

```
SAFARTokenizer tok = new SAFARTokenizer();
String[] tokens = tok.tokenize("text");
```

3. Normalization: The normalization is performed by the "SAFAR Normalizer" tool. This step aims to transform the text into a standard format by: deleting all special characters and non-Arabic letters, deleting all words that are composed of digits only, deleting all words that are composed of both digits and letters, deleting all words that are composed of only one character.

This tool can be called as follows:

```
SAFARNormalizer norm = new SAFARNormalizer();
String result = norm.normalize("text");
```

4. Removing stop words: This step consists of removing all insignificant words from the text that may influence the weights of the significant words. This step is performed by the "SAFAR Stop Word Remover" tool. It can be called using the following code:

```
String result = Remover.removeStopWords("text");
```

5. Stemming: This step aims to reduce the words to their stems in order to preserve the global sense of the different words and calculate their frequency. The stemming process is performed by one of the integrated stemmers of SAFAR. For this example we use the Light10 stemmer. A stemmer can be called using the following code:

```
IStemmer st = StemmerFactory.getLight10Implementation();
st.stem("text", outputFile);
```

6. Calculating stems frequency: In this step, we calculate the number of occurrences of the same stem in the text in order to attribute it a weight.
7. Calculating sentences scores: This step aims to calculate the score of each sentence. It is calculated in general using statistical formulas.
8. Extracting sentences: In this step, the system returns the most significant sentence in the text based on its score.

This pipeline is a simple example that provides an overview of the interoperability between five integrated tools within SAFAR namely, the sentence splitter, tokenizer, normalizer, remover and stemmers. Thus, researchers have a wide collection of homogeneous tools integrated within a single platform that can be called easily with

small codes, allowing more flexibility and promoting the creation of advanced applications.

More advanced pipelines can be established using, syntactic parsers, generators, etc. For example, Dridi in [29] used SAFAR for event detection from Twitter using stemmers available within the platform. Abouenour has also used SAFAR to develop a question answering system [12]. For this, it was necessary to use a stemmer, a morphological analyzer, a syntactic parser and also text processing tools of SAFAR.

The existence of SAFAR platform, which is dedicated especially to the ANLP community, is an important step towards standardization, resolution of interoperability issues, pipelines, reusability and integration of all development efforts in this field.

## 6 Conclusion and Future Works

In this paper, we addressed the Arabic Natural Language Processing as a software engineering issue. We presented a set of UML models and interfaces for tools that are: stemmers, morphological analyzers and syntactic parsers, and which are available within SAFAR platform. These models represent for us a way to standardize the various aspects shared by the tools of the same type. This will promote interoperability between these tools and allow researchers to call several implementations without any adaptation or modification of the code of tools. This is an important step to move from a set of unstructured tools to structured ones. It will also promote the development of more advanced applications such as question/answering, applications for social media, opinion mining, sentiment analysis, etc.

We presented also the pipelines processes that allow a researcher to combine several tools where the output of one of them is the input of another and so on. These pipelines are usually difficult to achieve because of the heterogeneous inputs/outputs of the different tools. SAFAR solves this problem as long as all inputs and outputs are standardized inside the platform. We presented the case of Automatic summarization as an example of pipeline within SAFAR.

As future work, we plan to develop new models for new tools such as morphological generators within SAFAR platform. We also intend to further enhance our existing models by adding more methods to our classes, which will improve the manipulation of objects. We intend to set up an online/offline Graphical User Interface so that researchers can create ANLP pipelines without the need to go through the code. Thereby, a tool can be accessed directly through a dedicated GUI or through an API. Finally, to allow calling from any coding language, we also intend to implement web services for all the integrated tools.

### References


[1] M. M. Group, "Internet World Stats," 2001. [Online]. Available: http://www.internetworldstats.com. [Accessed 1 February 2015].

[2] G. Eryiğit, "ITU Turkish NLP Web Service," in *European Chapter of the Association for Computational Linguistics*, Sweden, 2014.

[3] Y. Kano, M. Miwa, K. Cohen, L. Hunter, S. Ananiadou and J. Tsujii, "U-Compare: a modular NLP workflow construction and evaluation system," *IBM Journal of Research and Development,* vol. 55, no. 3, pp. 1-11, 2011.

[4] S. Sidrine, Y. Souteh, K. Bouzoubaa and T. Loukili, "SAFAR: vers une plateforme ouverte pour le traitement automatique de la langue Arabe," in *The 6th Intelligent Systems: Theory and Applications SITA*, Rabat, 2010.

[5] Y. Souteh and K. Bouzoubaa, "SAFAR platform and its morphological layer," in *Eleventh Conference on Language Engineering ESOLEC'2011*, Cairo, 2011.

[6] D. Ferrucci and A. Lally, "UIMA : an Architectural Approach to Unstructured Information Processing in the Corporate Research Environment," *Natural Language Engineering,* vol. 10, no. 3-4, pp. 327-348, September 2004.

[7] H. Cunningham, D. Maynard, K. Bontcheva and V. Tablan, "GATE: an Architecture for Development of Robust HLT Applications," in *The 40th Annual Meeting on Association for Computational Linguistics (ACL'2002)*, 2002.

[8] E. Loper and S. Bird, "NLTK: The natural language toolkit," in *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*, 2002.

[9] The Apache Software Foundation, "OpenNLP," [Online]. Available: http://opennlp.apache.org/. [Accessed 1 February 2015].

[10] B. Carpenter and B. Baldwin, Text Analysis with LingPipe 4, New York, 2011.

[11] M. Silberztein, T. VÁRADI and M. TADI, "Open source multi-platform NooJ for NLP," in *Proceedings of COLING 2012: Demonstration Papers*, Mumbai, 2012.

[12] L. Abouenour, "Three level approach for Arabic Q/A," Rabat, 2014.

[13] S. Khoja, "APT: Arabic part-of-speech tagger," in *Proceedings of the Student Workshop at NAACL*, 2001.

[14] H. Algasaier, "The ISRI Arabic Stemmer," 2005. [Online]. Available: http://www.nltk.org/_modules/nltk/stem/isri.html. [Accessed 1 February 2015].

[15] S. Khoja, "Khoja stemmer," 2002. [Online]. Available: http://zeus.cs.pacificu.edu/shereen/research.htm#stemming. [Accessed 1 February 2015].

[16] L. S. Larkey, L. Ballesteros and M. E. Connell, "Light Stemming for Arabic Information Retrieval," *Arabic computational morphology,* pp. 221-243, 2007.

[17] T. Zerrouki, "Tashaphyne 0.2," 2012. [Online]. Available: https://pypi.python.org/pypi/Tashaphyne. [Accessed 1 February 2015].

[18] S. Motaz, "Arabic Computational Linguistics," 2013. [Online]. Available: http://sourceforge.net/projects/ar-text-mining/. [Accessed 1 february 2015].

[19] N. Y. Habash, Introduction to Arabic Natural Language Processing, Morgan & Claypool, 2010.

[20] K. Darwish and W. Magdy, "Arabic Information Retrieval," *Foundations and Trends® in Information Retrieval,* vol. 7, no. 4, pp. 239-342, 2014.

[21] T. Buckwalter, "ARABIC MORPHOLOGY ANALYSIS," 2002. [Online]. Available:

http://www.qamus.org/morphology.htm. [Accessed 27 December 2013].

[22] N. Habash, O. Rambow and R. Roth, "Mada+ tokan: A toolkit for arabic tokenization, diacritization, morphological disambiguation, pos tagging, stemming and lemmatization," in *Proceedings of the 2nd International Conference on Arabic Language Resources and Tools (MEDAR)*, Cairo, 2009.

[23] A. Boudlal, A. Lakhouaja, A. Mazroui, A. Meziane, M. Ould Abdallahi Ould Bebah and M. Shoul, "Alkhalil Morpho Sys: A Morphosyntactic analysis System for Arabic texts," in *Proceedings of ACIT'2010*, 2011.

[24] ALECSO, "مواصفات نظام التحليل الصرفي في اللغة العربية," [Online]. Available: http://www.alecso.org.tn/images/stories/OULOUM/MOHALLILAT%20SARFIA_DAMAS_2009/022%202%20SPECIFICATIONS.pdf. [Accessed 1 December 2014].

[25] S. Green and C. D. Manning, "Better Arabic parsing: baselines, evaluations, and analysis," in *The 23rd International Conference on Computational Linguistics (COLING '10)*, Beijing, 2010.

[26] L. Belguith, A. Ben Hamadou and C. Aloulou, "Vers un système d'analyse syntaxique robuste pour l'arabe," in *7ème Conférence annuelle sur le Traitement Automatique du Langage Naturel (TALN'2000)*, Lausanne, 2000.

[27] L. H. Belguith, C. Aloulou and A. Ben Hamadou, "MASPAR : De la segmentation à l'analyse syntaxique de textes arabes," *Information Interaction Intelligence I3,* vol. 7, no. 2, pp. 9-36, 2008.

[28] F. S. Douzidia and G. Lapalme, "Lakhas, an Arabic summarization system," in *Proceedings of DUC'04*, Boston, 2004.

[29] H. E. Dridi, "Détection d'évènements à partir de Twitter (Ph.D. Thesis)," Montréal, 2014.