TeXTracT: a Web-based Tool for Building NLP-enabled Applications

Alejandro Rago[1], Facundo M. Ramos[3], Juan I. Velez[3], J. Andres Diaz-Pace[1], Claudia Marcos[2]

[1] ISISTAN, UNICEN University, CONICET
Campus Universitario, Paraje Arroyo Seco, B7001BBO, Tandil, Bs. As., Argentina
[2] ISISTAN, UNICEN University, CIC
[3] Facultad de Ciencias Exactas, UNICEN University

{arago,cmarcos,adiaz}@exa.unicen.edu.ar -
{ramos.facundo.m,velez.juanignacio}@gmail.com

Abstract. Over the last few years, the software industry has showed an increasing interest for applications with *Natural Language Processing* (NLP) capabilities. Several cloud-based solutions have emerged with the purpose of simplifying and streamlining the integration of NLP techniques via Web services. These NLP techniques cover tasks such as language detection, entity recognition, sentiment analysis, classification, among others. However, the services provided are not always as extensible and configurable as a developer may want, preventing their use in industry-grade developments and limiting their adoption in specialized domains (e.g., for analyzing technical documentation). In this context, we have developed a tool called *TeXTracT* that is designed to be composable, extensible, configurable and accessible. In our tool, NLP techniques can be accessed independently and orchestrated in a pipeline via RESTful Web services. Moreover, the architecture supports the setup and deployment of NLP techniques on demand. The NLP infrastructure is built upon the UIMA framework, which defines communication protocols and uniform service interfaces for text analysis modules. *TeXTracT* has been evaluated in two case-studies to assess its pros and cons.

Keywords: natural language processing, web services, module composition, framework, software development, software as a service

1 Introduction

Nowadays, there is a high demand in the Software Development market for applications featuring text processing and understanding capabilities [14]. Algorithms able to make sense of textual documents written in natural language are often referred to as *Natural Language Processing* (NLP) techniques [7]. In practice, software developers must spend considerable effort and time to codify this kind of characteristics in end products, adjusting the techniques to a particular domain and integrating algorithms implemented by different people. This situation creates the need of reusing existing frameworks and particular solutions to simplify the development and ensure good-quality results [8]. Several industry players have provided NLP techniques as a series of cloud-based micro-services [6]. Essentially, these micro-services are offered to the developers in the form of Software as a Service (SaaS), a business model in which clients are charged with a subscription fee for renting a set of functionalities. This way, they can access a discrete number of well-defined NLP services via a Web-based interface.

Unfortunately, most cloud-based NLP solutions still present limitations that hinder the development of applications. Some disadvantages worth noting are the following. First, there is little room for adapting and tailoring the services, making it difficult to adjust a subset of techniques to specialized domains (e.g., analyzing technical documentation vs. news articles vs. scientific journals) [12]. Second, many NLP services available to developers are often the "last element" of the chain of analysis, preventing the composition of NLP modules as an end-to-end pipeline that incrementally analyzes the text [6]. Third, the services only expose a few customizable parameters of the NLP techniques to developers and the optimization of the algorithms becomes complicated. Finally, only a few of the vendors give developers an opportunity for adding new NLP techniques or updating existing ones, which are in a state of constant evolution. An interesting alternative to address the limitations discussed above is to build a software solution that allows developers: (i) to uniformly incorporate, configure and execute NLP techniques implemented by third-parties, (ii) to compose different sequences of modules capable of performing an end-to-end analysis of the text per application, and (iii) to expose the techniques with technologies that can be consumed from diverse programming languages and OS platforms.

In this context, we have built a prototype tool called *TeXTracT* with the objective of simplifying the integration of NLP technologies in the development of standalone and Web-based applications in multiple platforms. The development of *TeXTracT* was driven by three goals, namely: the interoperability of the tool, the adaptation of text processing pipelines, and the flexibility for incorporating NLP techniques. The tool defines a facade of Web services that allows the developers to run a dynamic discovery of services, the execution of customized pipelines of services and the setup of individual services. Developers can dynamically list the techniques available in *TeXTracT*, enabling the incorporation of new or upgraded NLP modules without having to recompile or restart the tool. NLP services can be sequentially connected through HTTP requests in order to define an analysis pipeline. The main contribution of our work is two-fold. First, we build an extensible but yet easy-to-use tool for developing NLP-enabled applications. Second, we rely on well-known technologies to orchestrate the configuration and execution of NLP modules. We evaluated our tool in two case-studies with promising results. The case-studies consisted in the development of a Web-based grammar checker application and the adaptation of Requirements Engineering application with embedded NLP capabilities. respectively.

The rest of this article is organized as follows. Section 2 discusses existing tools and frameworks for performing NLP analyses remotely. Section 3 introduces the *TeXTracT* tool and describes its architecture. Section 4 reports on the experiences of using our tool in two application contexts: developing a new application, and migrating an existing one. Finally, Section 5 presents the conclusions and futures lines of work.

2 Related Work

A current trend in the development of NLP-based applications is the usage of the cloud to run the underlying algorithms and their models, relying on implementations coded and maintained by third-party developers and service providers [15]. In the cloud, the functionality is normally provided as *Software as a Service* (SaaS), a business model in which software is hosted by an external vendor and made available to clients over the Internet. From the clients' viewpoint [15], this model is attractive because they have hassle-free access to up-to-date and tested functionality, provisioning can be elastically handled for scalability purposes, and subscriptions can be cancelled without paying substantial upfront fees. Multiple vendors have defined NLP solutions in the form of cloud-based micro-services distributed as SaaS [6]. From a cost-benefit perspective, these NLP solutions allow clients to rapidly develop and deploy an NLP-enabled application without requiring much experience in machine learning/linguistics and without implementing/integrating text processing algorithms in the code themselves. Table 1 summarizes the features of these solutions.

An interesting cloud-based solution is Aylien Text Analysis [1]. Aylien is composed of a set of text processing tools, information retrieval and machine learning techniques that can extract knowledge and understand concepts from textual documents. Some services provided by Aylien are: language detection, entity recognition, concept recognition, document classification, summarization, sentiment analysis, and hashtag recommendations. Some disadvantages of Aylien are the following: services cannot be composed to accomplish more complex analyses, developers cannot incorporate new/modified NLP techniques, and techniques cannot be optimized through parameters. Language Tools is an open-source tool for verifying the grammar, orthography and writing style of textual documents [3]. Some techniques provided in Language Tools are: language identification, sentence splitting, token splitting, POS tagging, lemmatization, disambiguation, text segmentation, rule-based search, among others. The rule engine is really interesting, because it allows developers to define patterns for identifying domain-specific information. Unfortunately, such rules cannot be composed, preventing to create rules built on top of the results of another. Additionally, the developers are not allowed to incorporate custom NLP algorithms to the solution.

MeaningCloud is a solution that allows developers to enrich their applications with semantic text analyses in a simple and cost-effective way [4]. Its services can be tailored for specific domains, e.g., by using custom dictionaries and concept taxonomies for improving news and social media categorization. Some

Feature	Aylien	LanguageTools	MeaningCloud	DeveloperCloud
Customizable	✗	✓	✓	✓
Extensible	✗	✗	✗	✗
Composable	✗	✗	✗	✗
Scalable	✗	✗	✓	✓
Technology	REST/HTTP	REST/HTTP	REST/HTTP	REST/HTTP
Pricing	Subscription	Free	Subscription	Subscription
Code Distribution	Closed Source	Open Source	Closed Source	Closed Source
I/O Format	Raw/XML	Raw/XML	Raw/JSON	Raw&Markup/JSON

Table 1. Comparison of Cloud-based NLP platforms

of techniques offered are: language identification, lemmatization, parsing, POS tagging, text classification, topic extraction, sentiment analysis. MeaningCloud is equipped with several classification models for identifying news categories, subject topics, word thesaurus relations, among others. The text classification service is very useful because developers can train the algorithms and create custom prediction models for identifying domain-specific concepts. Unfortunately, this solution does not support adding custom-built algorithms to the solution. Another limitation is that the services have to be invoked individually, preventing the assembly of end-to-end pipelines and forcing the developers to call services one at a time. Another NLP solution built and maintained by IBM is DeveloperCloud [2]. This solution exposes some of the technologies on which the Watson question & answering AI system is implemented. The services offered in DeveloperCloud are: concept expansion, concept perception, translation, personality insights, among others. A nice feature of this solution is that it can scale on demand. Moreover, classification services can be personalized with user-provided datasets. Even though this platform is really powerful, there are still some limitations. For example, developers cannot invoke multiple services in a single request, because they work independently one to another. This also means that the services cannot be composed sequentially to construct more complex pipelines of analysis. Furthermore, DeveloperCloud does not allow developers to incorporate their own algorithms to the infrastructure.

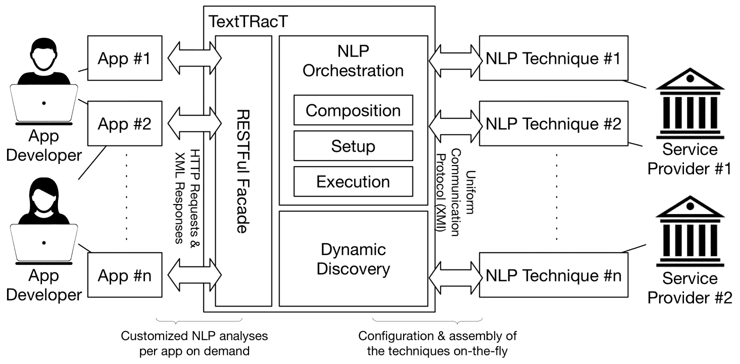
3 *TeXTracT*: a Web-based Tool for NLP

With the purpose of giving developers a more configurable and extensible cloud-based NLP services solution, we built a prototype tool called *TeXTracT*. Essentially, *TeXTracT* aims at simplifying the integration of advanced NLP techniques for the development of NLP-enabled apps by focusing on three aspects: (i) interoperability, (ii) adaptation of processing pipelines, and (iii) flexibility regarding the incorporation of new/upgraded techniques.

3.1 Design of the Tool

Several architectural decisions were made to fulfill interoperability, adaptation and flexibility aspects. Figure 1 depicts the main components of the architecture. Regarding the first aspect, we defined a RESTFUL-FACADE component that is composed of several Web services, addressing each request made by the applications and handling the communication with the NLP techniques needed on demand. Inspired by the Broker architectural pattern, we also defined the DYNAMICDISCOVERY and NLP-ORQUESTRATION components for dynamically searching and composing different textual analysis pipelines on-the-fly. From a design viewpoint, NLP techniques are packaged in containers that operate as standalone applications and are isolated one from another. The containers share a uniform communication protocol and thus can be arranged and executed sequentially for an end-to-end analysis of the text. The containers also expose a standardized interface for customizing NLP techniques for a custom domain. The DYNAMICDISCOVERY allows developers to search for the techniques available, so as to incorporate or interchange an NLP container in runtime. Along this line, each NLP container is discoverable and alternative implementations of an individual technique can be substituted at runtime.

Fig. 1. Conceptual Architecture of TexTRacT



From an application developer's perspective, there are multiple services for interacting with *TeXTracT*. Figure 2 illustrates a typical usage scenario of the tool. Initially, the application (or the developer who is coding it) can ask for the list of NLP services available for consumption (Step #1). This communication is encoded as a HTTP-GET request that is received by *TeXTracT* server. Then, our tool searches for NLP containers loaded in memory. Let us note that there might be multiple variants of the techniques provided by third-parties (e.g., a Tokenizer implemented by the OpenNLP group and another by the Stanford NLP Group). In the example, *TeXTracT* returns that there are four techniques available, namely: Tokenizer (OpenNLP), Tokenizer (Stanford), Part-Of-Speech Tagger (Stanford) and Entity Recognizer (Stanford).

After analyzing the services and choosing those that best suit their objectives, an application can request to process a textual document by using a custom sequence of NLP techniques (Step #2). For instance, the application can ask for processing the text with OpenNLP Tokenizer, use the results as input for Stanford POS Tagger, and fed the output to Stanford Entity Recognizer with the end goal of recognizing relevant people. This service is encoded as a HTTP-POST request, in which the sequence of techniques and their setup is codified in the URL and the text is submitted as POST data. An interesting feature here is that the Entity Recognizer is configured to search for "people", meaning that only that predictive model will be loaded. If developers are searching for another type of entities instead, such as organizations or countries, they just have to modify the configuration of the Entity Recognizer in the request. Once the processing request is received by *TeXTracT*, the tool dissects the URL in order to configure and invoke each particular NLP technique.

Afterward, the techniques are initialized either with default (Steps #3.1 and #3.2) or custom parameters specified in the request (Step #3.3). Finally, the techniques are executed one by one (Steps #4.1, #4.2 and #4.3), interconnecting the outputs of a technique with the input of the next. The input information provided by the application can either be a raw text that is transformed to the special format used for communicating the NLP techniques or the already formatted information to directly feed the techniques (allowing partial invocations of the services). Additionally, a group of NLP experts (or a well-motivated app developer) can implement and incorporate a custom classifier (e.g., trained with Weka[4] [9]) for identifying domain-specific concepts (Step #5). Consequently, the following invocations for enumerating NLP services would automatically list this custom classifier (Step #6).

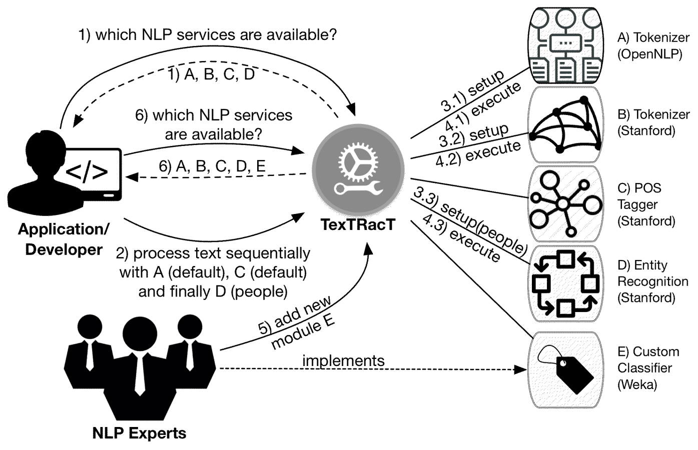
3.2 Technology & Implementation Details

The implementation of *TeXTracT* comprised several technologies and design decisions. In order to expose the NLP services of the tool in the Web, we chose to use the Jersey[5] library for the REST (REpresentational State Transfer) architectural style. The main features of REST are: it differentiates clients and servers, it is stateless (all the data can be held in the requests), it is cacheable (e.g., for scalability), it can be layered (e.g., for load balancing) and it presents a uniform interface (URIs, HTML, XML, JSON, etc.). Jersey allowed us to easily define Web services for listing, composing, setting up and executing NLP modules by using Java annotations on the source code.

For the text processing infrastructure, instead of writing everything from scratch, we relied on the UIMA[6] (Unstructured Information Management) frame-

[4] `http://www.cs.waikato.ac.nz/ml/weka`
[5] `https://jersey.java.net`
[6] `http://uima.apache.org`

Fig. 2. Typical Usage Scenario of TexTRacT



work [8]. UIMA is the industry standard for text analytics and defines components for wrapping, integrating and deploying diverse kinds of NLP techniques. A remarkable feature of UIMA for *TeXTracT* is its data representation, called Common Annotation Schema (CAS). This representation is composed of the actual text that can be processed and analyzed by the techniques and a set of annotations that hold the information produced by the algorithms. An annotation is basically a marker in the text, which delimits a fragment of the text (i.e., begin and end position) and can be decorated with properties (strings, numerical values, etc.) or references to other annotations. The CAS format is serializable and is very useful for concatenating NLP techniques that have dependencies with others (e.g., the POS tagger requires the results of the tokenizer). We adopted the CAS as the communication protocol between the apps and the NLP techniques, encoded as XML information that is sent through the Web.

The Adapter design pattern was also applied for abstracting the NLP algorithms [5]. Basically, we defined an interface that is responsible for the conversion of the input and output of a CAS to the internal representation used by a NLP technique, as well as standardize the setup of the techniques by means of parameters and their execution. Since we used UIMA, the conversion of input and output only required to instantiate the persistent CAS to in-memory object structure and to write it to XML format, respectively. With this design, a developer can codify a custom file reader in order to process Word, PDFs, and others kinds of documents easily. However, other adaptations can be implemented if required. We also relied on reflection mechanisms for retrieving all the instances of the adapter loaded in the JVM to fulfill the discovery of NLP techniques. *TeXTracT* is equipped with several NLP adapters, allowing the application developers to have access to NLP techniques such as sentence splitting, token splitting, POS tagging, stemming and lemmatizing, lexical thesaurus, dependency graphs, syntactic parsing, semantic role labeling, among others. These techniques are UIMA annotators wrapped with an Adapter, and the underlying algorithms were implemented by third-party teams and universities such as OpenNLP, Stanford CoreNLP, Mate-Tools, etc.

4 Preliminary Evaluation

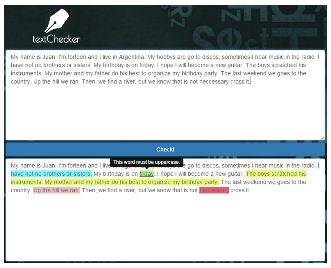
To assess the feasibility of using *TeXTracT*, we conducted two case-studies in which our tool was used to build a new application from scratch and to migrate an existing application with already-embedded NLP capabilities. The objectives of the evaluation are two-fold. First, we wanted to verify if our tool can actually simplify the creation of NLP-based apps and reduce the effort/time of development. Second, we wanted to corroborate that cloud-based NLP services can indeed improve the performance of an existing application, reducing its CPU, memory and heap consumption and shrinking the code footprint and the overall size of the packaged application.

4.1 Development of a New Application

In the first case-study, our experimental hypothesis is that *TeXTracT* can help developers to build and deploy applications that rely on NLP techniques, simplifying their work. The system consisted of a Web-based grammar analyzer called *TextChecker* capable of analyzing a textual document and identifying well-known and common lexical, syntactic and semantic mistakes in English (see Figure 4(a)). We initially plan the development of this application using a traditional approach that would rely on ad-hoc integration of NLP techniques, outlining the tasks to be done in the phases of requirements, analysis and design, implementation, testing and deployment, as well as establishing an estimative duration for each of them. Next, we developed the grammar checker following this plan but taking advantage of the features provided by *TeXTracT* in order to see the improvements over a traditional development lifecycle. To do so, we computed the differences between the estimated and the actual times spent in the phases.

Figure 4(b) illustrates the time savings obtained using *TeXTracT*. During the requirements phase, we did not observed improvements because the investigation of grammar mistakes and the rules to detect them needed to be done regardless if our tool was used or not. These grammar rules include: sentences ending with prepositions, starting with conjunctions, double negatives, noun-pronoun agreements or subject-verb agreements, among others. For the analysis and design, we noticed an important reduction in days because our tool already prescribes a text processing infrastructure, letting developers focus on the user interface design. Furthermore, since most NLP techniques for detecting grammar mistakes are available, developers can pay more attention to the definition

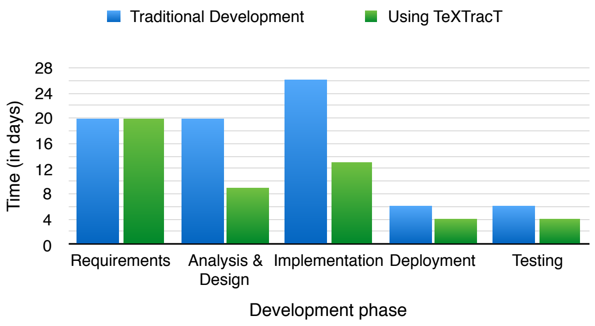
Fig. 3. Case-Study #1: Development of a Grammar Checker



My name is Juan. I'm forteen and I live in Argentina. My hobbys are go to discos, sometimes I hear music in the radio. I have not no brothers or sisters. My birthday is on friday. I hope I will become a new guitar. The boys scratched his instruments. My mother and my father do his best to organize my birthday party. The last weekend we goes to the country. Up the hill we ran. Then, we find a river, but we know that is not neccessary cross it.

My name is Juan. I'm forteen and I live [illegible] e go to discos, sometimes I hear music in the radio. I have not no brothers or sisters. My birthday is on friday. I hope I will become a new guitar. The boys scratched his instruments. My mother and my father do his best to organize my birthday party. The last weekend we goes to the country. Up the hill we ran. Then, we find a river, but we know that is not neccessary cross it.

(a) TextChecker UI



(b) Time Savings Using *TeXTracT*

of text rules. At this design stage, we decided to use RUTA[7] (Rule-based Text Annotation), a powerful language that allows developers to codify pattern-based rules in terms of UIMA annotations [10]. In this way, we can take advantage of the information produced by other NLP services to reveal grammar mistakes. For the implementation, we also observed important cuts in the development times because the composition of the NLP techniques and their invocation via a Web page is straightforward using AJAX calls. We took advantage of our tool extensibility to incorporate RUTA as a new NLP service that can be personalized with rules through parameters. Finally, the last two phases were also shortened with *TeXTracT* because most techniques were already tested and deployed in the Web accordingly.

4.2 Migration of an Existing Application

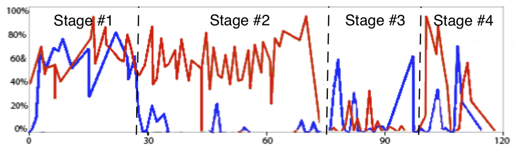
In the second case-study, our experimental hypothesis is that by migrating an existing application with built-in NLP capabilities to *TeXTracT* developers can improve its performance, reducing its code footprint, consuming less memory and running faster. To this end, we made modifications to a Requirements Engineering application called *REAssistant*[8]. The purpose of this application is to process use case specifications for identifying crosscutting concerns (e.g., persistence, security or performance features that cannot be isolated to a single document) by using domain-specific search rules codified in terms of NLP elements [13,11].

REAssistant is implemented as a set of Eclipse plugins, in which one of them is responsible for containing NLP libraries and another one is responsible for the instantiation of the UIMA framework for performing NLP analyses. For this reason, the migration to our cloud-based services was really simple. Basically, the majority of the text analysis techniques could be replaced by those provided

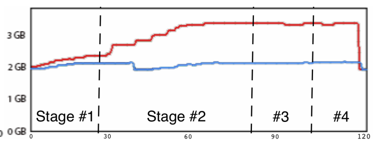
[7] `http://uima.apache.org/ruta.html`
[8] `https://github.com/alejandrorago/reassistant`

in our tool, with the exception of a classifier of "Domain Actions" which was a special addition for processing use cases. This classifier, including its code, the libraries and the prediction model, was therefore moved to *TeXTracT* and wrapped as a new NLP service. Then, we interchanged the execution of the (internal) UIMA pipeline for a remote HTTP request to *TeXTracT*. Finally, we deleted the NLP algorithms and the models from *REAssistant*.

Fig. 4. Case-Study #2: Migration of *REAssistant*



(a) CPU Utilization (percentage over time)



(b) Memory Consumption (GBs over time)

	REAssistant(original)	*REAssistant*(migrated)
TNOC (Total Number of Classes)	2287	511
TLOC (Total Lines of Code)	308511	40027
TNOP (Total Number of Packages)	221	97
SIZE (Size of the Release)	1134029 KB	2353 KB

(c) Structural Improvements

We collected several metrics for determining the advantages of using *TeXTracT*. To analyze runtime improvements, we instrumented the code with Oracle VisualVM for Java and retrieved execution measurements during a typical usage of *REAssistant*, such as CPU usage and memory consumption. The usage scenario consisted of opening the Eclipse distribution and loading the plugins (Stage #1), running the NLP analysis on a set of use cases (Stage #2), creating a file for discovering crosscutting concerns (Stage #3) and executing the search rules (Stage #4). Figures 5(a) and 5(b) show the CPU percentage and the memory consumption of the two *REAssistant* variants (original and migrated) over time, respectively. During stage #1, we observed a similar CPU usage (50 to 80%) and memory consumption (2.0 to 2.2 GB). However, in stage #2 we obtained significant improvements with *TeXTracT*. The original variant exhibited a high demand of processor power for running the NLP techniques and a exceptional increase in memory consumption (from 2GB to 3.25GB), whereas the processor remained idle in the migrated variant (no surprises here) and maintained the same memory demands. Unfortunately, we did not observe large improvements in execution times, meaning that the original and migrated variants took the same in processing the use cases. We speculate that this behavior is related to the overhead introduced by the communication of results of considerable size over the network. In stages #3 and #4, there was not a substantial difference in CPU usage between the variants. Surprisingly, the gap in RAM consumption remain similar to the previous stage until the IDE is closed, meaning that the original variant of *REAssistant* did not release the memory after executing the NLP analyses.

Additionally, we analyzed the structural changes made to the project after migrating the application to *TeXTracT*. Table 5(c) summarizes the differences between the two variants. We observed that 56% of the classes in all the plugins were directly related to NLP algorithms and consequently ~87% of the lines of code were removed as a consequence of the migration. Furthermore, migrating *REAssistant* reduced its disk footprint by a 99.8% (from 1.08GB to 2.3MB), making the distribution of the application easier and allowing to launch the Eclipse IDE much faster than the original variant.

5 Conclusion

In this article, we presented a Web-based tool called *TeXTracT* for consuming NLP services from diverse kinds of applications. Our motivation for building this tool was that existing solutions have limited personalization options and cannot support the composition of the techniques in a customized pipeline. We designed *TeXTracT* for being extensible, configurable, composable and accessible. For the implementation, we relied on well-known technologies such as RESTful Web services and an industry standard architecture for text processing. The tool assembled these technologies by adding discovery and orquestration mechanisms of the NLP services loaded into the tool. From a developers' viewpoint, they can invoke a particular sequence of NLP techniques in a single HTTP request.

We evaluated our tool by carrying out two cases-studies. The first case-study, devoted to the development of an online grammar checker, revealed that several activities of the software development process can be shortened by depending on the NLP services provided by *TeXTracT*. Moreover, we believe that by using our tool developers can potentially reduce the time-to-market, as well as supporting the evolution of the system over time. The second case-study, which comprised the migration of an existing Eclipse application with embedded NLP processing, showed some interesting findings after some minor adaptations to the source code. Basically, the runtime behavior was improved in terms of memory consumption and CPU utilization, though we did not observe advantages regarding the execution time. Another advantage of the migrated application is that its footprint was heavily reduced, cutting the number of classes in half and the drastically shrinking the size of the application of the original variant.

The case-studies also exposed some limitations of our tool: (i) *TeXTracT* is tightly coupled to the UIMA framework, (ii) using XML as communication format is not very efficient in terms of network bandwidth, (iii) it is not prepared to scale under stressful usage scenarios. Some future lines of work include defining a simpler and uniform communication protocol (independent from UIMA), exploring the use of JSON in combination with compression algorithms for transmitting the results, and investigating deployment mechanisms for scaling out to thousand of clients. Another interesting research topic is the automatic assembly of NLP pipelines by means of the analysis of the quality-properties of the techniques, such as accuracy, speed, memory and robustness, among others. Overall, we believe that tools such as *TeXTracT* give developers the chance to rapidly create powerful, rich and lightweight NLP-enabled applications that can be executed from any kind of device or platform.

References

1. Aylien text analysis API. `http://aylien.com/text-api`, accessed: 2016-05-01
2. IBM developer cloud. `https://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud`, accessed: 2016-05-01
3. Language tools API. `https://www.languagetool.org`, accessed: 2016-05-01
4. Meaning cloud. `https://www.meaningcloud.com`, accessed: 2016-05-01
5. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. SEI Series in Software Engineering, Addison-Wesley Professional, 3rd edn. (October 2012)
6. Dale, R.: NLP meets the cloud. Natural Language Engineering 21, 653–659 (2015)
7. Ferilli, S., Ferilli, S.: Natural language processing. In: Automatic Digital Document Processing and Management, pp. 199–222. Advances in Pattern Recognition, Springer London (2011)
8. Ferrucci, D., Lally, A.: UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering 10(3-4), 327–348 (2004)
9. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: An update. ACM SIGKDD Explorations Newsletter 11(1), 10–18 (2009)
10. Kluegl, P., Toepfer, M., Beck, P.D., Fette, G., Puppe, F.: UIMA ruta: Rapid development of rule-based information extraction applications. Natural Language Engineering pp. 1–40 (7 2015), `10.1017/S1351324914000114`
11. Rago, A., Marcos, C., Diaz-Pace, A.: Assisting requirements analysts to find latent concerns with REAssistant. Automated Software Engineering 23(2), 219–252 (2016)
12. Rago, A., Marcos, C., Diaz-Pace, A.: Opportunities for analyzing hardware specifications with NLP techniques. In: 3rd Workshop on Design Automation for Understanding Hardware Designs (DUHDe'16). Design, Automation and Test in Europe Conference and Exhibition (DATE'16), Dresden, Germany (2016)
13. Rago, A., Marcos, C., Diaz-Pace, A.: Text analytics for discovering concerns in requirements documents. In: XIII Argentine Symposium on Software Engineering (ASSE'12). La Plata, Argentina (September 2012)
14. Sateli, B., Angius, E., Rajivelu, S.S., Witte, R.: Can text mining assistants help to improve requirements specifications? In: Mining Unstructured Data (MUD 2012). Canada (2012)
15. Sun, W., Zhang, K., Chen, S.K., Zhang, X., Liang, H.: Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07), chap. Software as a Service: An Integration Perspective, pp. 558–569. Springer Berlin Heidelberg, Vienna, Austria (September 2007)