12th Conference on Language Resources and Evaluation, LREC2020, Marseille, 11 - 16 May 2020.

DeepNLPF: A Framework for Integrating Third-Party NLP Tools

Francisco Rødrigues[1], Rinaldo Lima[2], William Domingues[3], Robson Fidalgo[1], Adrian Chifu[3], Bernard Espinasse[3], Sébastien Fournier[3]

[1] Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
[2] Departamento de Computação, Universidade Federal Rural de Pernambuco, Recife, Brazil
[3] LIS UMR CNRS 7020, Aix-Marseille Université/Université de Toulon, France
rinaldo.jose@ufrpe.br, {fasr, rdnf}@cin.ufpe.br
{william.domingues, adrian.chifu, bernard.espinasse, sebastien.fournier}@lis-lab.fr

Abstract

Natural Language Processing (NLP) of textual data is usually broken down into a sequence of several subtasks, where the output of one the subtasks becomes the input to the following one, which constitutes an *NLP pipeline*. Many third-party NLP tools are currently available, each performing distinct NLP subtasks. However, it is difficult to integrate several NLP toolkits into a pipeline due to many problems, including different input/output representations or formats, distinct programming languages, and tokenization issues. This paper presents *DeepNLPF*, a framework that enables easy integration of third-party NLP tools, allowing the user to preprocess natural language texts at lexical, syntactic, and semantic levels. The proposed framework also provides an API for complete pipeline customization including the definition of input/output formats, integration plugin management, transparent multiprocessing execution strategies, corpus-level statistics, and database persistence. Furthermore, the DeepNLPF user-friendly GUI allows its use even by a non-expert NLP user. We conducted runtime performance analysis showing that DeepNLPF not only easily integrates existent NLP toolkits but also reduces significant runtime processing compared to executing the same NLP pipeline in a sequential manner.

Keywords: Natural Language Processing, NLP tools integration, Framework

1. Introduction

The automatic processing of natural language texts has been increasingly employed in many text mining applications, such as information extraction, automatic summarization, sentiment analysis, etc. Usually this processing is broken down into subtasks where the output of one the subtasks becomes the input to the following one, which constitutes a *pipeline*. Many freely available Natural Language Processing (NLP) tools for many languages have already been proposed in the literature. Such tools usually provide distinct and even complementary subtasks that would be of great interest if the user could easily integrate them first. However, the use of these tools, their integration according to a specific pipeline may raise many issues including different input and output representation formats, programming languages, tokenization conflicts, to name a few. Moreover, most of the NLP subtasks at a higher level (semantic parsing) usually require a lot of computation resources, thus it is essential to optimize their execution taking advantage of modern CPU-based architectures allowing multiprocessing or parallelism.

In this paper we present *DeepNLPF*, a framework that promotes an easy integration of third-party NLP tools allowing the preprocessing of natural language texts at lexical, syntactic, and semantic levels.

The proposed framework also provides an API for complete pipeline customization including the definition of input/output formats, integration plugin management, transparent multiprocessing execution strategies, corpus-level statistics, and database persistence. Furthermore, the friendly user interface of DeepNLPF allows its deployment and use even by a non-expert user.

A preliminary evaluation of DeepNLPF has been performed on a reference corpus which showed that DeepNLPF can not only easily integrate existent NLP toolkits, but also reduce a significant amount of runtime processing compared to executing the same NLP pipeline in a sequential manner.

DeepNLPF is freely available and a dedicated wiki (https://deepnlpf.github.io/site) provides all the documentation on how to install, to use, and to customize NLP pipelines.

This paper is structured as follows. Section 2 presents and compares several NLP toolkits. In Section 3 the general architecture of DeepNLPF, as well as its main components are presented. Section 4 discusses strategies for deploying and running DeepNLPF on multiprocessor architectures, while Section 5 reports experiments that evaluate these strategies. Finally, we conclude by presenting some perspectives for future development.

2. Related Work

2.1 GeoTxt

Karimzadeh et al (2019) propose GeoTxt, an NLP toolkit for recognizing and geolocating place names (toponyms) in natural texts. GeoTxt provides six Named Entity Recognition (NER) algorithms and a search engine to index, classify, and retrieve toponyms. It was developed in Java using the Play Framework which enables not only its use as endpoints, but also via an interactive GUI. User applications can query GeoTxt services using HTTP GET or POST requests and receive the answer as a GeoJSON *FeatureCollection* object, created to facilitate data storage, analysis, and visualization. However, Geotxt does not cover other types of typical NLP analysis required by

downstream text mining applications, neither provides a way to integrate other tools into its pipeline. In addition, it seems that GeoTxt does not provide a means to control parallel or multiprocessing tasks.

2.2 CLAMP

Soysal et al (2017) propose the CLAMP[1] (Clinical Language Annotation Modeling and Processing), a Java desktop application specialized in NLP-based analysis of text from the biomedical domain. According to the authors, with existing biomedical NLP systems, such as MetaMap, end users often need to adapt existing systems to their selected tasks, which may require significant NLP skills. That's the reason why CLAMP offers a user-friendly GUI that helps users to quickly build customized NLP pipelines. On the other hand, CLAMP does not provide direct support for integrating external tools developed in other programming languages than Java.

2.3 Jigg

Noji and Miyao (2016) propose the Jigg Framework, an NLP framework for integrating distinct NLP tools. It allows the user to build a pipeline by choosing the tools at each step via a command-line interface. Annotations are made by a Scala XML object. According to the authors, Jigg is a set of NLP components developed by several contributing groups. Jigg is an open NLP system allowing the user to add a new tool to the pipeline by writing a wrapper according to its API. However, Jigg does not allow multiprocessing execution since it only executes one document at a time on a single machine.

2.4 xTAS

De Rooij et al. (2012) offer a set of open source web services called xTAS[2], developed at the University of Amsterdam. The main goal of xTas is to allow users to perform a variety of NLP tasks as quickly as possible, without having to worry about the database, storage or caching the results. It is designed to integrate existing open source and/or proprietary analysis algorithms using a scalable distributed architecture. To use xTAS, the user must communicate with the tool using the web service that can be included in his application as a library written in Python language. xTAS uses MongoDB[3] to store documents and results, and the Celery[4] Framework to distribute analyses to processing nodes, allowing on-demand document processing. xTAS is a robust NLP tool for corpus processing, but it does provide a more advanced semantic-based analysis of the input texts.

2.5 GATE

GATE[5] (General Architecture for Text Engineering) (Cunningham, 2002) is a Java suite of NLP tools developed at the University of Sheffield and used by several communities in NLP. GATE is both a framework and a graphical development environment that allows users to develop and deploy language engineering components and resources. GATE is also considered an ecosystem, because the architecture defines the organization of a language engineering system, the assignment of responsibilities to the various components, and ensures that component integrations meet the system requirements. As a development environment, GATE helps to minimize the time spent for creating or modifying existing systems, providing a development mechanism for new modules. GATE has been employed in many NLP-based downstream applications. However, due to its complexity, and specialized output format based on inline annotations representations, it is difficult for non-NLP expert users.

2.6 FreeLing

FreeLing[6] (Padro and Stanilovsky, 2012) is an open-source language analysis toolbox and was built in C++. It provides APIs in Python and Java for text processing and annotation capabilities to NLP application developers. Its architecture consists of a simple two-layer client-server approach: a linguistic service layer that provides analysis services (morphological analysis, tagging, parsing, among others), and an application layer which, acting as a client, requests the desired services from the analyzers. A major advantage of Freeling is its speed, also providing API library services in Java, Perl, and Python languages. The internal architecture of the system is based on two kinds of objects: linguistic data objects and processing objects. Freeling does not support the integration of other third-party NLP tools in its pipeline.

2.7 Analysis of Related Work

In what follows, we provide a qualitative comparative evaluation of the different systems previously presented according to the following criteria: (i) dependence and independence on the domain covered, (ii) the main NLP tasks supported, (ii) the external NLP tools integrated into the system, (ii) whether the NLP pipeline can be customized by the user, (iv) whether the system has a RESTFul API, (v) whether it provides a GUI, (vi) whether it allows optimized processing strategies (parallelism in a cluster environment), (vii) whether the system has a database, (viii) whether it provides corpus-level statistics, and finally, (ix) the type of system architecture.

According to Table 1, regarding domain-dependency, only the CLAMP system (Soyal et al., 2017) is domain dependent, other systems are domain-independent. Concerning the linguistic analyses supported, few systems support semantic-level analysis.

Regarding the customization of an analysis pipeline, except for GeoTxt and FreeLing systems, the other systems do not allow the integration of new components to its analysis pipeline. For the RESTFul API, CLAMP, Jigg and FreeLing systems, do not provide it, which hinders the development of Web client applications.

The Jigg, xTAS and Freeling systems do not provide a GUI, which makes their use more difficult for some kind of users. Regarding processing strategies, GeoTxt and Jigg systems allow some level of optimization by using the

[1] https://clamp.uth.edu
[2] http://xtas.net
[3] https://www.mongodb.com
[4] http://www.celeryproject.org
[5] https//gate.ac.uk/
[6] http://nlp.lsi.upc.edu/freeling/

parallel computing strategy required to process large collections of texts.

Regarding the use of databases, CLAMP, Jigg, and FreeLing systems do not have secondary storage strategies using the database, which hinders performance.

Finally, concerning the information about corpus statistics, none of the systems selected has such a functionality.

Table 1 summarizes the main characteristics of each studied system and the proposed framework DeepNLPF, according to aforementioned criteria. Besides these comparison criteria, we can also add the criteria of the data output format. Few NLP systems support multiple data output formats. Most of them offer the user with only one alternative. Therefore, when another format is required, the developer must create a parser to convert it into the desired format.

NLP Toolkits		GeoTxt	CLAMP	Jigg	xTAS	GATE	FreeLing	DeepNLPF
Domain Independent		yes	no	yes	yes	yes	yes	yes
NLP tasks	Preprocessing	text cleaning	no	tokenizer, sentence splitter	tokenizer, stem	tokenizer, sent. split, language identifier	tokenizer sent. split. lang. identifier	tokenizer, Sentence splitter
	Lexical	NER	NER, POS	NER, POS	POS, NER	POS, NER	Number and date detection, NER, POS tagging	POS, NER, lemma, truecase, dcoref, is_alpha, is_title, like num, NER ontonotes
	Syntactic	no	chunker, parsing	parsing	no	chunker, parser	BIO, shallow parsing, depend. parsing	dep. parsing, shallow parsing
	Semantic	no	no	no	no	WSD, ontology, gazetter	WordNet WSD, UKB WSD	frame passing, SRL verb, SRL prep, SRL noum
Integrated NLP tools		CogComp, CoreNLP, GATE, ANNIE, MIT IE, OpenNLP, LingPipe	OpenNLP	CoreNLP, Berkeley	CoreNLP, SEMAFOR, OpenNLP	CoreNLP, LingPipe, OpenNLP	no	CoreNLP, SEMAFOR, PyWSD, SpaCy, CogComp
custom pipeline		no	yes	yes	yes	yes	no	yes
API		no	no	yes	yes	yes	yes	yes
RESTFul API		yes	no	no	yes	yes	no	yes
User GUI		yes	yes	no	no	yes	no	yes
Multiprocessing		no	yes	no	yes	yes	yes	yes
uses database		yes	no	no	yes	yes	no	yes
Corpus statist.		no	no	no	no	no	no	yes
Architecture		Client-server	Eclipse Framework	based on CoreNLP	Client-server, Plugin	Component-based	Client-server	Component-based
Language		Java	Java	Java	Python, Java	Java	C/C++	Python
Data format		JSON	Texto	XML	JSON	TEXT, JSON, XML	CoNLL, TEXT, JSON, XML	JSON

Table 1: NLP toolkits comparison.

DeepNLPF has some advantages compared to majority of related work selected in Tab. 1 including: (i) a richer set of default NLP tools and, consequently, more NLP tasks already available to the user; (ii) a very friendly user GUI allowing non-NLP expert to used it in downstream applications; (iii) it provides corpus statistics and customizable API in Python, an easy to learn and powerful programming language.

3. DeepNLPF Framework

In this section, we present DeepNLPF, a framework for enabling an easy integration of third-party NLP tools. Our main contribution consists in providing both simple API-based and GUI-based natural language processing services concerning several levels of linguistic annotations, including lexical, syntactic, and semantic metadata produced by integrated NLP third-party tools. Furthermore, in its default setting, DeepNLPF already integrates many existing NLP tools including CoreNLP, CogComp, spaCy, SEMAFOR, and PySWD. Another distinguishing DeepNLPF feature is that it provides an easy API for integrating new third-party tools even when such tools are in different programming languages and input and output formats. Finally, DeepNLPF was designed to take profit of the full capacity of user's resources (memory and mainly multi-core CPU architectures) in order to speed up the processing of textual datasets written in English. In the rest of this section, we present the general architecture of DeepNLPF, as well as its main components.

3.1 DeepNLPF Functional Architecture

DeepNLPF is a *grey box* (Jorgensen and Hangos, 1995) type framework, which means that the user does not need to know the details of its implementation in order to exploit it in his/her applications. The DeepNLPF architecture is based on five major components (Fig. 1):

- **Pipeline**: As the backbone of the proposed framework, it ensures the orchestration of all operations needed to execute a customized pipeline analysis of the input dataset. It also provides support for the integration of other components.
- **Plugins**: this component ensures the integration of several NLP tools, and delegates the analysis responsibility to all subcomponents wrappers that access external (third-party) NLP tools. The integration is achieved by combining a set of wrappers, each one in charge of a specific analysis in the entire pipeline.
- **Models**: this component interacts with the data in the internal database. It allows access to the corpus, system logs, annotations, statistics, among other generated metadata.
- **Templates**: this component manages both the schemas that define the output file formats of the dataset annotations.
- **Statistics**: this component performs statistical analyses of the input datasets.

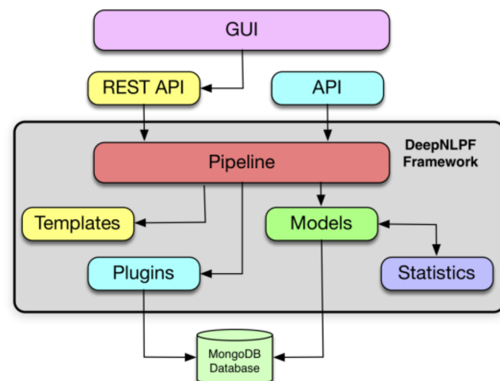

Figure 1: Main DeepNLPF components.

3.2 Pipeline Component

This is the main component in DeepNLPF architecture because it allows both the customization and execution of NLP tasks. It also contains all the control logic to synchronizing multithreads and parallelism of the NLP tools and their individual analysis (lexical, syntactical, and semantic). In addition, this component persists linguistic annotations in the database. Fig. 2 shows the class diagram of the Pipeline component.

The current version of the DeepNLPF already integrates the following third-party NLP tools: Stanford CoreNLP, SpaCy, CogComp, SEMAFOR, and PyWSD. In DeepNLPF, NLP analyses are separated into the following levels lexical, syntactic, and semantic. Fig. 3 illustrates the entire default pipeline.

At the beginning of the pipeline, it is necessary to extract the sentences from the corpus and structure them, e.g., converting a plain text document to a second one containing one sentence per line. The sentences are then cleaned, i.e., symbols and extra spaces are first removed. Then the document is tokenized by default using the Stanford CoreNLP tokenizer. This first step is responsible to unify the sentence representation as a unique (canonical) tokenized representation of the sentence among all the integrated tools in DeepNLPF. As a result, the same number of tokens are passed to the other NLP tools which avoids the issue of having the same sentences with distinct numbers of tokens in distinct NLP tools. Other analysis can be performed just after the tokenization; such as spell checking.

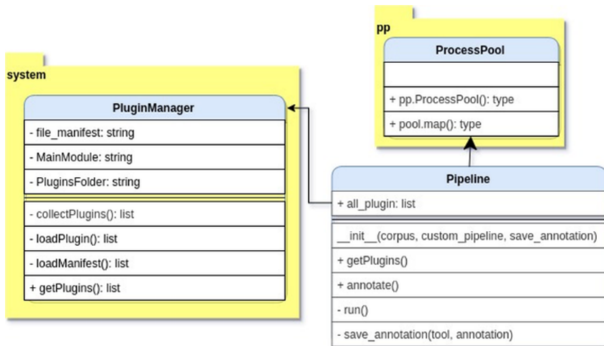

Figure 2: Classes of the pipeline component.

Next, the following NLP subtasks (at three level) are performed by the default DeepNLPF pipeline:

- **Lexical level** which provides the following analyses from (i) *Stanford CoreNLP*: tokenization, sentence splitting, POS tagging, lemmatization, NER and true case; (ii) *SpaCy*: POS tag, word shape, label, is_alpha, is_title, and like_num; (iii) *CogComp*: mapping to NER ontonotes; (iv) *GATE*: mapping to NER with gazetteers; and (v) *Custom*: morph type.
- **Syntactical level** performs the following syntactic analyses using the *Stanford CoreNLP*: constituent parsing, dependency parsing, coreference resolution, and shallow parsing (chunking analysis).
- **Semantic level** is based on the following NLP tools (i) SEMAFOR: frame-based semantic parsing; (ii) CogComp: *Semantic Role Labelling* (SRL): verb, preposition, and Noun; and (iii) *PyWSD* : word sense disambiguation.

For unambiguous WSD words (mainly nouns and verbs), the pipeline also integrates annotations that maps the word *sense id* to external semantic resources including WordNet, WordNet Domain, and aSUMO which provide detailed and more accurate information about the senses of a word.

DeepNLPF also allows the user to deepen the analysis by, for instance, adding a discourse-based analysis, according to the user's project needs.

After completing all the steps described above, each tool produces a document with its annotations, containing its analysis at sentences level. Finally, DeepNLPF retrieves those annotations and integrates them into a single annotation document according to a structured model defined for each linguistic annotation level by the user.

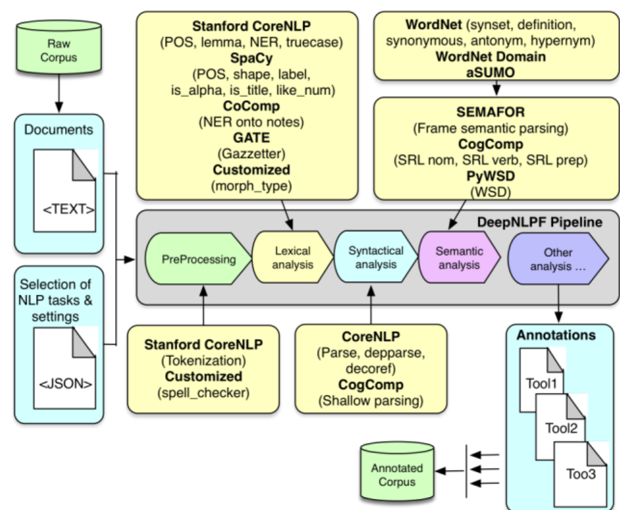

Fig. 3: DeepNLPF default pipeline.

3.3 Plugins Component

In many complex downstream NLP applications (such as Relationship Extraction, Sentiment Analysis, etc.) it is necessary to perform various linguistic analyses from diverse NLP tools, commonly developed in different languages, and distinct settings (input/output formats, parameters, etc). In order to integrate such heterogeneous NLP tools into a single NLP pipeline, one of the most successful strategies is based on the use of plugins (Cunningham, 2002).

In DeepNLPF, the *Plugin component* (Fig. 4) is in charge of such a task. More concretely, all user-built plugins in DeepNLPF must implement all of the *abstract methods* of the *IPlugin* interface. Thus, the integration of a new third-party NLP tool requires a new plugin to be inserted in a new directory named with the name of the NLP tool to be integrated. This label then is used throughout the pipeline in the sequence.

In addition, this directory should also contains the following elements: the file *__init__.py* which is a *wrapper* that provides the functionalities of the NLP tool to be integrated, and the *manifest.json* file which contains all the information necessary for DeepNLPF to manage the new NLP tool to be integrated.

3.4 Models Component

This component allows the user to employ a database for corpus storage, language annotations, corpus statistics, and system logs. The main interest of using a database is the speed of access to the data of the corpus to be processed. Moreover, it allows the user to focus on the application under construction, not wasting time creating strategies for data storage (De Rooij et al., 2012). The DeepNLPF database model was implemented in MongoDB[7] (document-oriented) NoSQL system. The schema of the DeepNLPF database is shown in Fig. 5.

To add a new "corpus" document to the database, it is necessary to define a document structure in JSON format, containing the following fields: "name" for the corpus name (mandatory field); "description" which denotes a brief description of the corpus (optional); "sentences" denoting the sentences of the corpus and, finally, *date_time*, denoting date and time when the document was created. This component also includes other functionalities to facilitate parsing texts in many formats, including TXT, XML, and JSON. The Pandas library provides the main services to deal with these types of data storage formats.

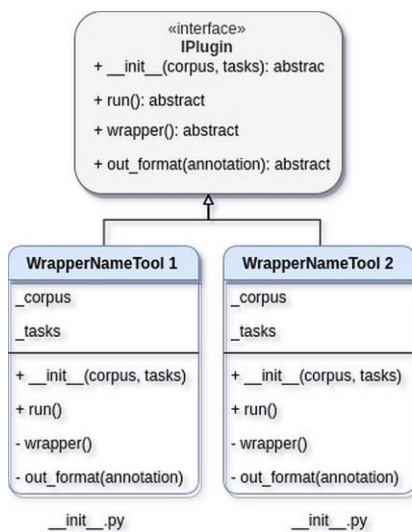

Figure 4: The classes of the plug-in components.

3.5 Templates Component

DeepNLPF offers an annotation scheme based on a hybrid model, capable of encapsulating and organizing annotations by linguistic level (lexical, syntactic, and semantic), in JSON format. All language analysis tasks performed by the NLP tools integrated into DeepNLPF provide JSON annotations by default.

Thus, the Model component retrieves these annotations and maps them with the formats defined by the template model, structuring in this manner all the analyses by linguistic level. When the user needs the annotation in another file format, such as XML, the user needs to indicate that to the Templates component that analyzes the JSON file and converts it to the desired output format.

3.6 Statistics Component

This component generates corpus-level statistics, including the total number of sentences in the corpus, the minimum and maximum number of tokens per sentence, the average number of words per sentence, the word frequency, and the frequency of POS tagging labels. Other customized statistics can be generated according to the user's needs. For example, the user can add a new method in the Statistics component to generate basic statistics concerning the frequency of bigrams and/or trigrams contained in the corpus. This component also allows the user to visualize data statistics interactively using the Python-based PlotLy[8] visualization library including frequency distribution of the number of words per sentence, the frequency distribution of POS tags, and corpus-level word clouds.

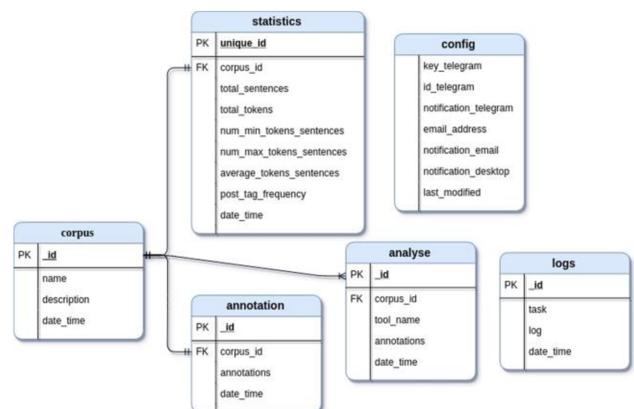

Figure 5: Diagram of the DeepNLPF database.

4. Multi-processing in DeepNLPF

This section presents different strategies for using DeepNLPF which takes advantage of modern multiprocessing and parallel computer architectures. Due to its sequential processing nature, a given NLP pipeline can take a considerable amount of time to process all the input collection of documents. Consequently, it is advisable to use an effective parallelism and/or a multiprocessing strategy.

4.1 Multi-process Strategy

In the case of a complex pipeline defined by the user, i.e., composed of many types of analysis performed by different NLP tools, DeepNLPF takes profit of multi-core computer architecture to process all pipeline analyses in parallel. For that, DeepNLPF automatically identifies the number of cores in the CPU processor(s) and distributes each NLP tool in an asynchronous process, optimizing the execution of the individual pipeline subtasks. DeepNLPF implements its multiprocessing strategy using the Pathos Framework[9], a framework used for heterogeneous computation (McKerns et al., 2011). Pathos mainly

[7] https://www.mongodb.com/

[8] https://plot.ly/

[9] https://pypi.org/project/pathos/

provides the communication mechanisms to configure and initiate parallel calculations via heterogeneous resources. Fig. 6 illustrates the multiprocessing execution of DeepNLPF using the Pathos Framework.

4.2 Using DeepNLPF via RESTful API

DeepNLPF can also be used via its RESTFul API. It allows the user to make service requests via Web, GET and POST technologies. This service makes it possible to build NLP applications that will have more flexible Internet services (Masse, 2011). The RESTFul API was built using the *Microframework Flask*[10]. To exploit it, the user has to run the service (python run.py) and then, in a client application, execute the requests via POST or GET, passing the correct settings with the *Postman tool.*

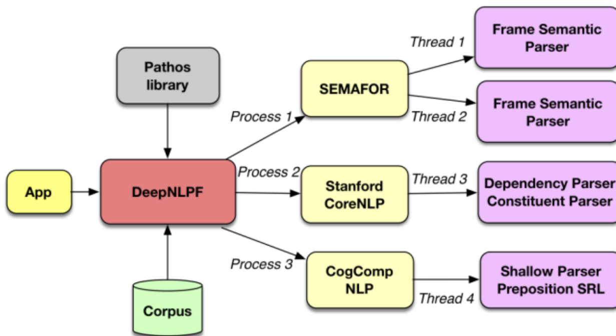

Fig. 6: Deployment and multiprocessing execution of DeepNLPF with Pathos library.

5. Experimental Evaluation

This section discusses the results of two experiments that aim at comparing the runtime performance of the NLP tools and DeepNLPF for performing a pipeline analysis on a single CPU-based multi-core machine. More precisely, both experiments have the objective of answering the following experimental question:

Is there a significant difference in runtime performance when the pipeline analysis performed by the third-party NLP tools are executed individually (by their own) compared to the same pipeline analysis when they are instantiated and executed by DeepNLPF?

To answer the above question, we will consider two distinct assessment scenarios:
Scenario 1: the pipeline analysis is divided into three linguistic levels (lexical, syntactic, semantic) and evaluated at each level for each NLP tool at a time. **Scenario 2**: a full pipeline analysis is performed at once.

5.1 Running Time Evaluation on a Single Multicore Machine

The *Profiling* technique (Eyerman and Eeckhout, 2008) which identifies the most resource-intensive points in a given running application was used for measuring the running execution time of DeepNLPF and the NLP tools evaluated in this section. Furthermore, a dynamic analysis was performed that measures the execution time of the main modules and all its components. This profiling technique has the advantage of indicating the most consuming subroutines for adopting later optimization strategies. It should be mentioned that both the working memory (RAM) and swapping usage were not studied due to the complexity of third-party NLP tools.

For this quantitative evaluation, the SemEval 2010[11] reference dataset was used. This dataset contains 8000 sentences with (Dataset III): 135,886 tokens from sentences composed by at least 4 tokens, and at most 95 tokens (average 17 tokens/per sentence).

We also generated two smaller datasets from the SemEval 2010 dataset containing 1,000 (Dataset I) and 4,000 (dataset II) sentences, respectively. The computer hardware configuration used in all the experiments reported in this section has the following hardware/OS configuration: 16 GB of RAM, Intel Core i5-6200U processor at 2.30 GHz with 4 (4 cores), a 120GB SSD hard disk, running the Linux Ubuntu 19.04 (64 bits) OS.

All the tools were launched from a Python wrapper using the *cProfile* module, performing a dynamic analysis measuring the execution time of the program and all its components.

Scenario 1: Individual NLP tools analysis at a time.

The goal of this experiment is to assess the processing time that each NLP tool at the three linguistic level (lexical, syntactic, and semantic) required to process the three partitions of the SemEval 2010 dataset.

The lexical level is formed by the following two pipelines: *spaCy pipeline* (pos, shape, label, is_alpha, is_title, and like_num), and *CogComp pipeline* (NER ontonotes). Table 2 shows the achieved results.

For the syntactic level, two other pipelines were studied: *Stanford CoreNLP pipeline* (constituent parsing, dependency parsing, and coreference resolution), and *CogComp pipeline* (shallow parsing - chunking).

For the semantic level, three pipelines were evaluated: *SEMAFOR pipeline* (frame-based semantic analysis), *CogComp pipeline* (SRL Noun, SRL Verb and SRL Prep), and *PyWSD pipeline* (WSD). Table 2 summarizes the results obtained on the SemEval 2010 dataset (Dataset I-1000, Dataset II 4000, Dataset III - 8,000).

Tools	Runtime(s)		
	Dataset I (1000)	**Dataset II (4000)**	**Dataset III (8000)**
Lexical Pipeline			
Stanford CoreNLP	73.6	206.0	379.0
spaCy	15.3	46.9	93.3
CogComp	286.0	1,370.0	2,280.0
Syntactic Pipeline			
Stanford CoreNLP	234.0	714.0	1,330.0
CogComp	242.0	1,170.0	2,040.0
Semantic Pipeline			
SEMAFOR	84.1	234.0	466.0
PyWSD	1,170.0	4,390.0	9,210.0
CogComp	1,000.0	4,070.0	8,220.0

Table 2: Runtimes of individual NLP tools.

[10] http://flask.pocoo.or

[11] https://www.cs.york.ac.uk/semeval2010W SI/datasets.html

As it can be seen in Tab. 2, CoreNLP, spaCy, and SEMAFOR pipelines have a notable linear correlation between the number of sentences and the time required to execute them. Not surprisingly, the CogComp and PyWSD tools took the longest running time to process the input dataset. However, both tools were run online and not locally. A local execution requires a lot of computing resources (actually more than 32 GB RAM), due to the size of the models loaded in memory and it could not be evaluated here.

Discussion. Figures 7, 8 and 9 show the runtime results obtained by each NLP tool at the linguistic lexical, syntactic, and semantic levels, respectively.

At the lexical level, the pipelines executed are: CoreNLP "default" pipeline (tokenization, pos, lemma and ner), spaCy "default" pipeline (pos, shape, label, is_alpha, is_title, and like_num), and CogComp "default" pipeline (NER ontonotes). Figure 7 displays the line graphs of the processing time of the NLP tools. More precisely, the dotted lines represent the NLP tools executed individually whereas the solid lines denote the same NLP tools executed individually, but under the control of the DeepNLPF optimization strategies. It can be seen that for spaCy, there is almost an overlap of the lines, which indicates that DeepNLPF has not improved the performance of this tool. This means that spaCy already takes into account optimization aspects in its implementation, such as multithread processing. In this scenario, DeepNLPF was not able to improve spaCy performance. However, for the CoreNLP tool, there is a significant performance improvement when using DeepNLPF. For the CogComp online version, there is still a slight improvement, considering the fact that the online version of CogComp runs in a cluster environment. Nevertheless, the lag for Internet data transfer has to be considered.

At the syntactic level, the pipelines executed were: CoreNLP "default" pipeline (parsing, dependency parsing. and coreference resolution), and CogComp "default" pipeline (shallow analysis). Figure 8 shows another significant improvement in the pipeline performance with DeepNLPF.

At the semantic level, the executed pipelines were: SEMAFOR "default" pipeline (semantic frame analysis), CogComp "defalult" pipeline (SRL Nom, SRL Verb and SRL Prep) and PyWSD "default" pipeline (WSD). As shown in Figure 9, for the SEMAFOR pipeline, there is no performance gain with DeepNLPF, while compared to CogComp pipeline, DeepNLPF gains about 2,500 seconds, and for the PyWSD pipeline there is a significant difference in running time.
These results show that the parallelism processing strategies implemented into DeepNLPF significantly shorten running time of the majority of the integrated third-party NLP tools.

Scenario 2: Full pipeline analysis executed at once. In this scenario, each NLP tool was executed individually, one after the other, and the total of the running times of each tool is calculated. The results presented in Table 3 show that the pipeline processing time of each of the dataset is proportional (almost linear) to the number of sentences in the datasets. Tab. 4 shows the runtime of the same pipelines in Tab. 3 but with those pipelines within DeepNLPF, i.e., the NLP tools in Tab. 3 were integrated into DeepNLPF by means of the proposed plugins.

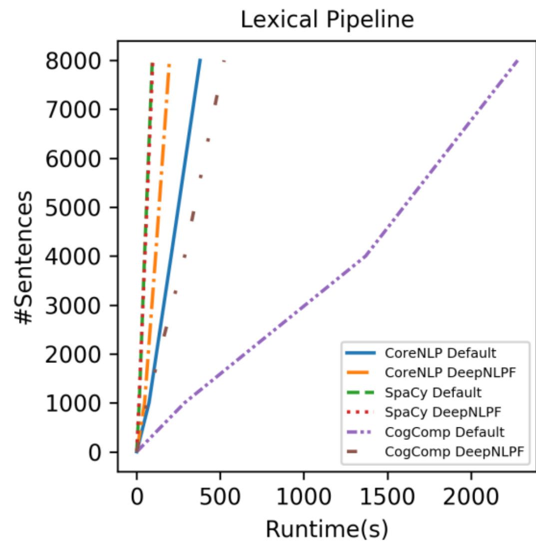

Fig. 7: NLP tools vs DeepNLPF runtime (lexical analysis)

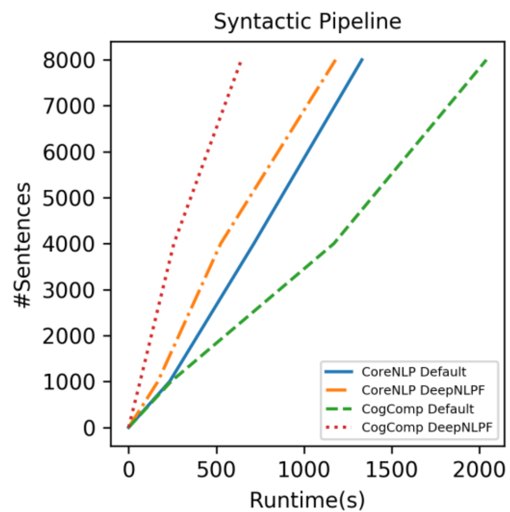

Fig. 8: NLP tools vs DeepNLPF runtime (syntactic analysis).

The objective of this experiment is two-fold: to check whether the integration of third-party tools work correctly through the DeepNLPF plugins, and to verify whether DeepNLPF parallel processing strategies can reduce processing time. Indeed, as the last row of the Tab. 4 shows, DeepNLPF was much more efficient thanks to its multiprocessing/parallel processing strategies, practically reducing processing time by 60%.

To further improve DeepNLPF performance, an implementation of multiprocessing strategies based on the distribution of processes/threads on a cluster environment with several machines is our ongoing work.

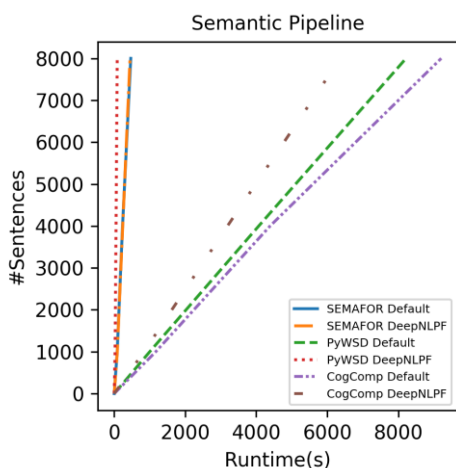

Fig. 9: NLP tools vs. DeepNLPF runtime (semantic analysis).

Full Pipeline				
Tools	**Pipeline**	**Dataset I**	**Dataset II**	**Dataset III**
Stanford CoreNLP	tokenize, ssplit, pos, lemma, ner, parse, depparse, truecase, coref	270.0	776.0	1,630.0
SpaCy	pos, shape, label, is_alpha, is_title, like_num	12.3	46.3	91.2
SEMAFOR	frame-based semantic parsing	87.3	243.0	448.0
CogComp	srl nom, verb, prep, shallow parse, ner, ontonotes	1,050.0	5,110.0	9,350.0
PyWSD	wsd	1,020.0	3,760.0	7,360.0
Total Runtime(s)		**2.439.6**	**9,935.3**	**18,879.2**

Table 3: Full pipeline runtime on the 3 datasets.

Full Pipeline			
Tools	**Runtime (s)**		
	Dataset I	**Dataset II**	**Dataset III**
Stanford CoreNLP, SpaCy, SEMAFOR, PyWSD	2,439.6	9,935.3	18,879.2
Stanford CoreNLP, SpaCy, SEMAFOR, PyWSD **(DeepNLPF)**	824.0	3,250.0	6,320.0

Table 4. Comparison of the sequential processing time of the NLP tools with the parallel processing in DeepNLPF.

6. Conclusion

We presented DeepNLPF, a grey box type framework for integrating third-party NLP tools. It provides a default set of integrated state-of-the-art NLP tools that annotates natural language texts with lexical, syntactic, and semantic linguistic-based metadata.

DeepNLPF provides a user-friendly interface, allowing non-NLP experts users to easily specify their NLP-based pipelines. Furthermore, DeepNLPF provides an optimized version of typical NLP pipelines that can be executed in multi-core processor machines, taking advantage of the available hardware resources for processing large datasets in less time.

DeepNLPF organizes its operations around a database to store both data and metadata: dataset information, analyses, annotations, dataset statistics, and system logs.

A first evaluation of DeepNLPF has been performed on a dataset adopting distinct multiprocessing/parallelism strategies. The obtained results are promising. We are currently testing an improved DeepNLPF version on a cluster computing architecture using larger datasets.

DeepNLPF is freely available and a dedicated wiki (https://deepnlpf.github.io/site) provides all the documentation on how to install, use, and customize NLP pipelines.

7. Acknowledgements

The authors would like to thank the Coordination for the Improvement of Higher Education Personnel (CAPES/Brazil) for financial support.

8. References

Cunningham H. (2002). GATE, a General Architecture for Text Engineering. Comp. and the Humanities 36: 223–254, 2002.

De Rooij O., Vishneuski A., De Rijke M. (2012). TAS_Text Analysis in a Timely Manner. DIR2012 2012 Gent, Belgium

Eyerman S., Eeckhout L. (2008). System-level performance metrics for multiprogram workloads. IEEE micro, [S.l.], v.28, n.3, p.42–53, 2008.

Jorgensen, S. Bay; Hangos, Katalin M. Grey box modelling for control: qualitative models as a unifying framework. International Journal of adaptive control and signal processing, v. 9, n. 6, p. 547-562, 1995

Karimzadeh M., Huang W., Banerjee S., Wallgrün J.O, Hardisty F., Pezanowski S., Mitra P. and MacEachren A.M. (2013). GeoTxt: A Web API to Leverage Place References in Text . ACM- GIR'13, November 05 2013, Orlando, FL, USA

Masse M. (2011). REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces (English Edition). O'Reilly Media.

McKerns and Aivazis M.A.G. (2010). Pathos: a framework for heterogeneous computing. See http://trac. mystic. cacr. caltech. edu/project/pathos, [S.l.], 2010.

McKerns M.M., Strand L., Sullivan T., Fang A., Aivazis M.A.G. (2011). Building a Framework for Predictive Science, Proc. Of the 10th Python In Science Conference (SCIPY 2011).

Noji H., Miyao Y. (2016). Jigg_A Framework for an Easy Natural Language Processing Pipeline. Proceedings of the 54th Annual Meeting of the Assoc. for Comp. Linguistics—System Demonstrations, pages 103–108,

Padro L., Stanilovsky E. (2012). FreeLing 3.0: Towards Wider Multilinguality. Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)

Soysal E., Wang J., Jiang M., Wu Y., Pakhomov S. Liu H.,3 and Xu H. (2017). CLAMP – a toolkit for efficiently building customized clinical natural language processing pipelines. Journal of the American Medical Informatics Association, 25(3), 2018, 331–336.