Hindawi
Scientific Programming
Volume 2020, Article ID 2390941, 13 pages
https://doi.org/10.1155/2020/2390941

Research Article

Using Natural Language Preprocessing Architecture (NLPA) for Big Data Text Sources

María Novo-Lourés,[1,2] Reyes Pavón,[1,2,3] Rosalía Laza,[1,2,3] David Ruano-Ordas,[1,2,3] and Jose R. Méndez[1,2,3]

[1]*Department of Computer Science, University of Vigo, ESEI-Escuela Superior de Ingeniería Informática, Edificio Politécnico, Campus Universitario As Lagoas s/n, Ourense 32004, Spain*
[2]*CINBIO-Biomedical Research Centre, University of Vigo, Campus Universitario Lagoas-Marcosende, Vigo 36310, Spain*
[3]*SING Research Group, Galicia Sur Health Research Institute (IIS Galicia Sur), SERGAS-UVIGO, Vigo 36312, Spain*

Correspondence should be addressed to Jose R. Méndez; moncho.mendez@uvigo.es

Received 25 October 2019; Accepted 30 June 2020; Published 1 August 2020

Academic Editor: Autilia Vitiello

During the last years, big data analysis has become a popular means of taking advantage of multiple (initially valueless) sources to find relevant knowledge about real domains. However, a large number of big data sources provide textual unstructured data. A proper analysis requires tools able to adequately combine big data and text-analysing techniques. Keeping this in mind, we combined a pipelining framework (BDP4J (Big Data Pipelining For Java)) with the implementation of a set of text preprocessing techniques in order to create NLPA (Natural Language Preprocessing Architecture), an extendable open-source plugin implementing preprocessing steps that can be easily combined to create a pipeline. Additionally, NLPA incorporates the possibility of generating datasets using either a classical token-based representation of data or newer synset-based datasets that would be further processed using semantic information (i.e., using ontologies). This work presents a case study of NLPA operation covering the transformation of raw heterogeneous big data into different dataset representations (synsets and tokens) and using the Weka application programming interface (API) to launch two well-known classifiers.

1. Introduction and Motivation

The technological advances reached during the last decade have prompted important improvements in the information technology industry, such as the emergence of social networking, the Internet of Things, or cloud computing. These new technologies, together with the enhancement of data storage and computation capabilities, have stimulated the continuous generation of large volumes of heterogeneous data (structured and unstructured) at an unprecedented speed. A clear example of this situation is shown in the latest Statista report [1] estimating an exponential data growth trend in the world that will reach 175 ZB during 2025 (representing an increase of 530% with respect to 2018). However, the deluge of raw data is neither meaningful nor useful in any aspect of life (people, enterprises, research results, etc.). Big data has emerged as a new paradigm to analyse and extract valuable information from the raw data in real time by applying data mining tasks through the use of parallel programming schemes over large computer clusters [2]. This scenario makes big data a fundamental paradigm to increase competitiveness in all areas of knowledge (business, science, government, healthcare, and so on).

Mining large volumes of raw data has allowed great improvements in different fields [3, 4]. Advantages reached in all these areas have boosted the use of big data and big data analytics that will lead to an economic benefit that the latest reports predict will be US$103B by 2027 [5]. However, despite this positive scenario, only 30% to 40% of existing raw data on average have been analysed [6].

The competitive advantages of the big data paradigm have bolstered the continuous creation of multiple tools and cloud-based services able to handle, manage, and interpret data to offer meaningful information. Additionally, the exponential growth in recent years of social media users and interconnected devices to continuously share information in real-time has prompted the need to perform big data analytics and decision-making processes in the shortest period of time [7]. Consequently, initial batch-mode processing technologies used for processing big data are becoming obsolete and inefficient mainly due to their inability to lead with real-time streaming sources.

During the last few years, new improved solutions based on the pipeline concept have emerged with the specific aim of solving this situation [8, 9]. The concept of data pipelining is based on segmenting a complex task into several simpler and interconnected subtasks (also named pipes) where the output of one subtask becomes the input of the next one [10]. Ever since this divide-and-conquer paradigm emerged in the big data domain, multiple cloud-based pipelining frameworks have been developed by important companies to perform data analysis and processing services [11].

However, the need to execute and store data in external services and infrastructures (managed by third-party enterprises) makes several businesses uneasy with the possibility of private data commingling with other organizations [12, 13]. Additionally, since the cloud infrastructure is entirely owned, managed, and monitored by the service provider, it makes it difficult for customers to have the level of control that they would want over their back-end infrastructure [13, 14]. These drawbacks motivated large and medium enterprises to progressively migrate the cloud-based services to a proprietary solution by designing and implementing their own objective-specific pipelining tools. Meanwhile, in order to bring free alternatives to the community, multiple offline domain-specific Big Data Pipeline tools emerged from both the research and academic environment [15]. This scenario has allowed the widespread creation, development, and use of many big data-oriented technologies focused on extracting relevant information from large volumes of (mainly nonstructured) information.

This context has prompted the appearance of Text Analytics models. This concept is the result of combining big data-oriented technologies over a large amount of nonstructured data sources (such as tweets, blogs, and wikis) and Artificial Intelligence (AI) methods to analyse and extract previously unknown information [16]. In order to successfully apply AI methods to extract valuable information, the raw text should be transformed by applying textual preprocessing techniques into features (usually represented as columns) and instances (represented as rows). Preprocessing methods involve the selection, combination, and execution of different preprocessing tasks able to transform raw input data collected from multiple heterogeneous sources of text into full-featured datasets.

Text Analytics has been widely used by industries and researchers [17, 18] since it can help decision-makers to understand the behaviour of nonstable domains (where sudden changes occur). Despite the promising results achieved by the published works using Text Analytics models, we found an important limitation regarding the extended use of token-based preprocessing techniques. Text Analytics models are based on using statistical methods and regular expressions to extract and detect the most important features from text contents. These (token-based) methods and expressions may have served as an adequate compromise between computational resource consumption and performance during the 1990s, when computational capabilities and technologies were considerably limited. However, technological advances achieved during the last decade in the field of information sciences have allowed the development and deployment of new approaches focused on detecting and extracting concepts according to their semantic relationships (synset-based representation). Despite the fact that these new methods achieve better performance results than token-based methods [19–22], their application in the big data domain is not common. Additionally, the domain-dependant nature of the stages involved in the Text Analytics process (information extraction and text mining) [16] complicates the development of general-purpose Text Analytics frameworks. In order to palliate these issues, we developed NLPA, a Java pipeline-based tool able to extract features from multiple sources by using a flexible combination of tasks. By default, NLPA enables the extraction of both tokens (using token-based methods) and synsets (using semantic-based methods). However, in order to adapt to user needs, it also allows an easy design, development, and deployment of new functionalities (tasks). NLPA has been developed over BDP4J (a Java pipelining framework for processing big data) due to its performance, flexibility, and highly customizable capabilities [23].

The rest of our work is structured as follows. Section 2 presents several well-known NLP Java frameworks used in big data projects and their limitations. Section 3 introduces NLPA, including a brief description regarding its support of text mining tasks and its interaction with the BDP4J framework. Section 4 includes a case study showing the use of NLPA. Section 5 outlines the lessons achieved during the development of NLPA. Finally, Section 6 summarizes the main conclusions and highlights future work.

2. State of the Art

While Text Analytics was being successfully applied to explore texts from different domains, a wide variety of tools (most of them distributed as open-source) emerged and became popular. These tools have been developed in a wide variety of programming languages (including Java, Python, and R). Although the usage of new programming languages (such as Python, Golang, or R) is increasing worldwide, the number of Java developments and developers is clearly greater than other languages [24]. For this reason, this study compiles the following tools implemented in Java: (i) MALLET [25], (ii) GATE [26], (iii) Stanford CoreNLP [27], (iv) OpenNLP [28], (v) UIMA [29], and (vi) DKPro-Core [30] and a study executed by IXA NLP Group using Apache Storm framework for text analysis purposes [31].

MALLET (MAchine Learning for LanguagE Toolkit) is used to perform machine learning over text data. Concretely, it is an integrated collection of Java code to perform statistical natural language processing, document classification, information extraction, clustering, or topic modelling. MALLET implements a flexible system of pipes, which handle distinct tasks such as tokenizing strings, removing stopwords, and converting sequences into count vectors.

GATE (General Architecture for Text Engineering) is an open-source software toolkit for text analysis and processing. It allows text processing to become a comprehensive task, bringing together software developers, language engineers, and research staff from diverse fields. GATE components are one of three types: (i) LanguageResources (lexicons, corpora, ontologies, etc.), (ii) ProcessingResources (parsers, generators, ngram modellers, tokenizer, POS tagger, sentence splitter, gazetteer, orthomatcher, coreference, etc.), and (iii) VisualResources (visualisation and editing components that participate in GUIs). It supports a wide variety of formats (such as plain text, HTML, SGML, XML, RTF, e-mail, and PDF), provides easy-to-use and extendable facilities for text annotation (ontology), facilitates persistent storage of language resources, and implements multilingual data processing and NLP methods.

Stanford CoreNLP provides a set of human-language technology tools. It integrates many of Stanford's NLP tools, including the POS tagger, the named entity recognizer (NER), the parser, the coreference resolution system, sentiment analysis, bootstrapped pattern learning, and the open information extraction tools. Moreover, Stanford CoreNLP provides a robust annotator for arbitrary texts support for a wide range of (human) languages, and it can be used from the command-line via its original Java programmatic API, the object-oriented simple API, third-party APIs for most major modern programming languages, or a web service. It supports Linux, macOS, and Windows operating system platforms.

The Apache OpenNLP library implements machine learning methods for processing natural language. It supports the most common NLP tasks, such as tokenization, sentence segmentation, POS tagging, named entity extraction, chunking, parsing, language detection, and coreference resolution. These functionalities are structured in such a way that executing one of them could make the other available for further processing. Additionally, it supports train-test evaluations to benchmark different configurations. These facilities are accessible via API or by using the command-line interface provided for programmatically defined experimental protocols.

UIMA (Unstructured Information Management Applications) are software applications designed to analyse large volumes of unstructured information (e.g., text, audio, and video) to discover relevant knowledge. UIMA enables applications to be decomposed into components and provides a large list of components for analysing unstructured information such as whitespace tokenizer, snowball, regular expression, dictionary, and configurable feature extractor. Each UIMA component has been developed by implementing interfaces defined by the framework and provides self-describing metadata via XML descriptor files. The framework implements the mechanisms to exchange information between them (i.e., data flow) and their management.

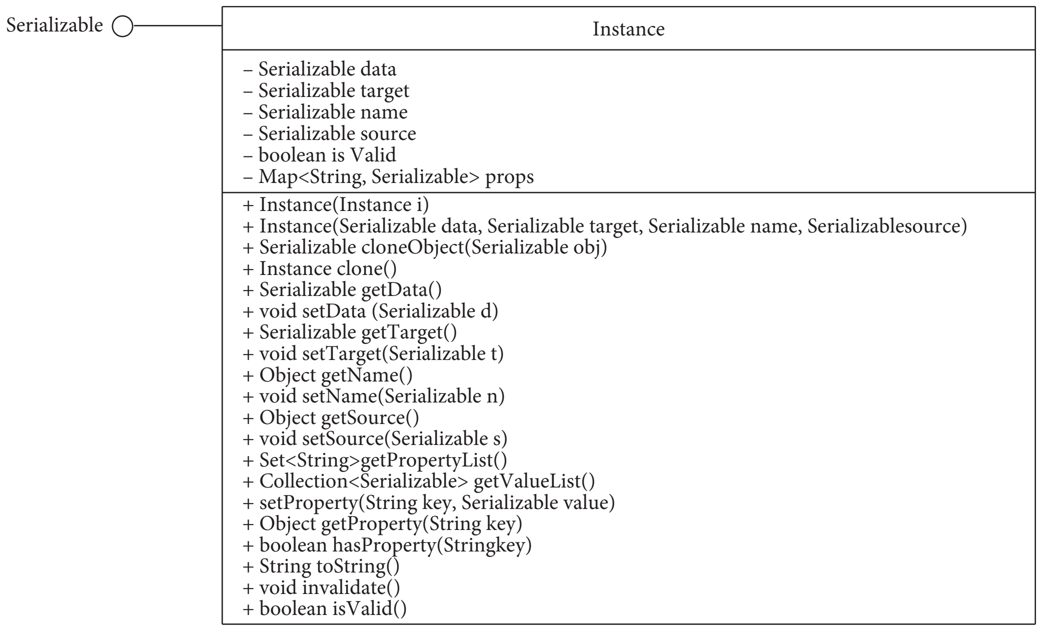
DKPro-Core (Darmstadt Knowledge Processing Core) is a software component library implementing natural language processing (NLP) tasks. It is based on the Apache UIMA framework and integrates many state-of-the-art NLP tools (i.e., uimaFIT components) allowing their combination to build an experiment pipeline. DKPro-Core is primarily focused on the execution of linguistic preprocessing tasks (e.g., part-of-speech taggers, parsers, etc.).

Following the usage of big data approaches for Text Analytics, Agerri et al. [31] designed and test a big data streaming approach for text processing. They take advantage of Apache Storm framework designing *Storm bolts* to execute NLPA operations in a distributed architecture. They developed five text processing operations (tokenizing, POS tagging, NER, constituency parsing, and coreference resolution) and made some experimental analysis mainly focused on the evaluation of the throughput.

Based on the general operation of these frameworks, we find adequate usage of big data approaches to deal with the analysis of the text (unstructured data). Moreover, we find that these frameworks do not provide some interesting functionalities (such as the creation of semantic representations of the texts or the handling of multiple information data sources). Taking into consideration the contributions of these previous works together with our ideas, we developed NLPA, which provides the following additional functionalities: (i) use new semantic-based (synset) representation; (ii) handle different sources of information (tweets, e-mail, messages websites, etc.); and (iii) properly configure and execute many low-level text preprocessing steps to fulfil the requirements of each concrete user, such as the elimination of URLs or abbreviation handling. The next section introduces NLPA and details the operations that can be combined to preprocess text corpora.

3. Introducing NLPA

As previously stated, in order to execute Text Analytics, big data should be transformed into full-featured datasets ready to be processed using AI techniques. Customizing and executing this transformation is the main functionality provided by NLPA. For this purpose, NLPA contains a list of preprocessing task implementations written in Java that can be used as a plugin for the BDP4J framework. These task definitions use synsets and/or tokens to implement efficient processing of the information exchanged through the different protocols and Internet services. NLPA provides more than 30 preprocessing tasks applicable over corpora containing e-mails (RFC 5322 [32]), websites (Web Archive, WARC), tweets, YouTube comments, SMS (Short Message Service), or plain text. The identification of these file types is made by using file extensions (.eml, .warc, .twtid, .ytbid, .tsms, and .txt, respectively). Given the architecture of the BDP4J framework, these preprocessing tasks are applied to *org.bdp4j.types.Instance* objects (see Figure 1).



```
– Serializable data
– Serializable target
– Serializable name
– Serializable source
– boolean is Valid
– Map<String, Serializable> props

+ Instance(Instance i)
+ Instance(Serializable data, Serializable target, Serializable name, Serializablesource)
+ Serializable cloneObject(Serializable obj)
+ Instance clone()
+ Serializable getData()
+ void setData (Serializable d)
+ Serializable getTarget()
+ void setTarget(Serializable t)
+ Object getName()
+ void setName(Serializable n)
+ Object getSource()
+ void setSource(Serializable s)
+ Set<String>getPropertyList()
+ Collection<Serializable> getValueList()
+ setProperty(String key, Serializable value)
+ Object getProperty(String key)
+ boolean hasProperty(Stringkey)
+ String toString()
+ void invalidate()
+ boolean isValid()
```

Figure 1: UML representation of the BDP4J Instance class.

As shown in Figure 1, Instance class brings together four attributes: (i) source, (ii) name, (iii) data, (iv) target, and (v) props. In detail, the *source* attribute stores the information required to access the source of the information compiled (usually a *java.io.File* is enough). The *name* attribute represents any form of unique identification of an instance. The processing of the instance implies a sequence of modifications of the *data* attribute (originally with the same value as the *source* attribute) while preprocessing tasks are applied. Logically, modifications of the *data* attribute could imply the loss of information from the original data and the impossibility of returning to the original state of the data (which could be achieved from *source* attribute). Moreover, the *target* attribute is useful for addressing classification and prediction problems to include in the instance information about the real prediction/classification. Finally, the *props* attribute contains several properties that are computed through preprocessing tasks (such as language and text length).

NLPA preprocessing tasks comprise five input types for *data* attribute: *java.io.File*, *java.lang.StringBuffer*, *org.nlpa.types.SynsetSequence*, *org.nlpa.types.TokenSequence*, and *org.nlpa.types.FeatureVector*. Table 1 compiles the input and output data types of the *data* attribute for all tasks included in our framework.

As shown in Table 1, most tasks are designed to use *java.lang.StringBuffer* (a mutable representation of strings) as input data. Some useful operations, such as the processing of abbreviations, slang, interjections, stopwords, URLs, and references to users, are made from this representation.

The following subsections describe the tasks implemented by our framework, classified by the data type required for the *data* attribute of instances being processed. Additionally, for each task, we will provide an example of use.

3.1. Processing java.io.File Data Type. Although streaming sources could be easily used with NLPA, it currently uses locally stored files (i.e., in a local or network file system) represented through *java.io.File* objects as the primary means of reading data instances. Descriptions of each available task for processing instances containing *java.io.File* objects as *data* are included hereinafter. For each task, we also include the changes made in the following instance represented in JSON (JavaScript Object Notation): {props: {}, name: "1", data: "_spam_/ex.tsms", source: "_spam_/ex.tsms", target: null}. In the example, *source* and *data* attributes are instances of *java.io.File* (but represented as *String*) while *name* is really a *String* object.

File2StringBufferPipe is able to transform the *data* attribute of an instance from *java.io.File* to a *java.lang.StringBuffer* type. For this reason, the textual content of the target file (only for supported formats) is stored in the *data* attribute. The modification made by this task in the example is {*data*: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href = \"http://www.xx.com\">here</a>, #xx or @xx"}. After the execution of this task, the *data* attribute is an instance of *java.lang.StringBuffer*.

GuessDateFromFilePipe is able to find the date of the contents (when available) in the input file (interpreting their formats). As a result of this process, the date is stored as a property of each instance (using "date" as default name). Using this task, the attribute *props* of the instance provided as an example is transformed as follows {props: {"date": null}}. As long as the *tsms* (Text of an SMS message) format does not contain the date of the message, the *date* property is filled with null.

StoreFileExtensionPipe creates a property to insert the type of content into the instance properties (using "extension" as the default name). The value of the property is

Table 1: Input and output data types for all tasks.

		Output data				
		File	StringBuffer	SynsetSequence	TokenSequence	FeatureVector
Input data	File	GDFF, SFE, TAFP	F2SB			
	StringBuffer		AFSB, CPFSB, CPTFSB, CFSB, FEjISB, FEtISB, FHISB, FUISB, FUNISB, GLFSB, IFSB, MLFSB, NFSB, SBTLC, SFSB, SHFSB, SWFSB, TCFSB	SB2SS	SB2TS	
	SynsetSequence					SS2FV
	TokenSequence				TSPS, TSSI	TS2FV
	FeatureVector					TCFFV, TDFFV

Available tasks: Input data file: GuessDateFromFilePipe (GDFF), StoreFileExtensionPipe (SFE), TargetAssigningFromPathPipe (TAFP), File2StringBufferPipe (F2SB). Input data StringBuffer: AbbreviationFromStringBufferPipe (AFSB), ComputePolarityFromStringBufferPipe (CPFSB), ComputePolarityTBWSFromStringBufferPipe (CPTFSB), ContractionsFromStringBufferPipe (CFSB), FindEmojiInStringBufferPipe (FEjISB), FindEmoticonInStringBufferPipe (FEtISB), FindHashtagInStringBufferPipe (FHISB), FindUrlInStringBufferPipe (FUISB), FindUserNameInStringBufferPipe (FUNISB), GuessLanguageFromStringBufferPipe (GLFSB), InterjectionFromStringBufferPipe (IFSB), MeasureLengthFromStringBufferPipe (MLFSB), NERFromStringBufferPipe (NFSB), StringBufferToLowerCasePipe (SBTLC), SlangFromStringBufferPipe (SFSB), StripHTMLFromStringBufferPipe (SHFSB), StopWordFromStringBufferPipe (SWFSB), TeeCSVFromStringBufferPipe (TCFSB), StringBuffer2SynsetSequencePipe (SB2SS), StringBuffer2TokenSequencePipe (SB2TS). Input data SynsetSequence: SynsetSequence2FeatureVectorPipe (SS2FV). Input data TokenSequence: TokenSequencePorterStemmerPipe (TSPS), TokenSequenceStemIrregularPipe (TSSI), TokenSequence2FeatureVectorPipe (TS2FV). Input data FeatureVector: TeeCSVFromFeatureVectorPipe (TCFFV), TeeDatasetFromFeatureVectorPipe (TDFFV).

computed as the extension of the file referenced by the *data* attribute of the instance. Using this task, the *props* attribute of our example is changed as follows {props: {"extension": ".tsms"}}.

Finally, *TargetAssigningFromPathPipe* can be applied only for classification purposes. This task searches the path of the File referenced by the *data* attribute of the instance to find a folder matching the class. This task uses a transformation map to connect system folder names with instance categories. By using this task, the *target* attribute of the input example is changed as follows {target: "spam"}.

Once a *java.io.File* is processed into a *java.lang.StringBuffer* (i.e., using File2StringBufferPipe task), a wide variety of operations can be used to preprocess the text. The next subsection presents the tasks that can be used for the processing *StringBuffer* object included as a data instance.

3.2. Processing java.io.StringBuffer Data Type. *StringBuffer* class is used to represent and modify textual contents (issued by mutable property provided by *StringBuffer* Java class). Once the text is extracted, a wide variety of tasks are provided to transform input contents by dropping parts (interjections, stopwords, emoticons, etc.) and/or replacing the text (slang forms, abbreviations, etc.). The descriptions of available tasks for processing this type of data (alphabetically ordered) are provided hereinafter. For this subsection, we will use the following instance as an example: {props: {}, name: "1", data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href = "http://www.xx.com">here</a>, #xx or @xx", source: "_spam_/ex.tsms", target: null}.

AbbreviationFromStringBufferPipe detects abbreviations in text and expands them using dictionaries. Abbreviation dictionaries are implemented for several languages (such as English, Spanish, French, or Russian) using JSON files. In order to properly select the right abbreviations dictionary, a property storing the language of text should previously exist (see *GuessLanguageFromStringBufferPipe*). As a result of preprocessing the example with the pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *AbbreviationFromStringBufferPipe*), the following modifications (*props* and *data* attributes) are made {props: {"language": "EN", "language-reliability": "0.9"}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 December Visit us on here, #xx or @xx"}.

ComputePolarityFromStringBufferPipe adds the polarity of the text as an instance property. Possible results are in the form of the 5-level Likert scale (i.e., 0 to indicate "Very Negative", 1 to "Negative", 2 to "Neutral", 3 to "Positive", and 4 to "Very Positive"). The polarity is computed by taking advantage of the Stanford NLP framework. The change of the instance made by executing this task is {props: {"polarity": "1"}}.

ComputePolarityTBWSFromStringBufferPipe adds the polarity of the text by querying the TextBlob (https://textblob.readthedocs.io) Python library. The polarity score computed by using this library is a float within the range [−1.0, 1.0]. In order to query Python TextBlob library from Java, a REST (Representational State Transfer) web service (TBWS, TextBlob Web Service) was developed to be easily launched as a Docker container using the scripts provided with the tool. By executing this task, the *props* attribute of the example is modified as follows: {props: {"polarityTBWS": "0.0"}}.

ContractionsFromStringBufferPipe replaces contractions in the original text using dictionaries (JSON files). To make the replacements, some language-specific dictionary files are included in NLPA. In order to properly select the right contractions dictionary, a property storing the language of text should be previously computed (see *GuessLanguageFromStringBufferPipe*). By preprocessing the example of instance with the pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *ContractionsFromStringBufferPipe*), the following modifications are made: {props: {"language": "EN", "language-reliability": "0.9"}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we will even give you 19% off online until 31 Dec. Visit us on here, #xx or @xx" }.

FindEmojiInStringBufferPipe finds and removes (if desired) emojis from text and adds them as a property of the instance. By default, the property name is "emoji". The process is made by taking advantage of the emoji-java library (https://github.com/vdurmont/emoji-java). After executing this task, the following fields of the example provided in this subsection are modified: {props: { "emoji": "?" }, data: "December is hre:-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href =\"http://www.xx.com">here</a>, #xx or @xx"}.

FindEmoticonInStringBufferPipe finds and removes (if needed) emoticons from text and creates a new property (named "emoticon" by default) for the instance. Emoticons are found using a complex regular expression over the whole text. One of the main limitations of this task is that *FindHashtagInStringBufferPipe* (see next paragraph) cannot be executed afterwards. After executing this task, the following fields of the example provided in this subsection are modified: {props: {"emoticon": ":-)"}, data: "December is hre, ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href=\"http://www.xx.com">here</a>, #xx or @xx"}.

FindHashtagInStringBufferPipe searches for hashtags in text and adds them as a property ("hashtag" by default) of the instance. Additionally, the task can be configured to remove the identified hashtags from the original text. Internally, this task uses a regular expression to find hashtags in text. Using this task, the provided example will be modified as {props: {"hashtag": "#xx"}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href=\"http://www.xx.com">here</a>, or @xx"}.

FindUrlInStringBufferPipe finds URLs from the text, adding them as a property ("URLs" by default) of the instance. Additionally, removing URLs from the original text is also possible. This task is carried out by using regular expressions. *FindUserNameInStringBufferPipe* (see next paragraph) task cannot be executed after *FindUrlInStringBufferPipe*. When applied on the provided example, the following changes can be observed {props: {"URLs": "http://www.xx.com"}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href=\"\">here</a>, #xx or @xx"}.

FindUserNameInStringBufferPipe takes advantage of regular expressions to search and optionally remove tokens in the form "@<userName>" from the text. Also, it adds the identified user references as a property of the instance ("@userName" by default). The preprocessing of the example implies the following changes {props: {"@userName": "@xx"}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on <a href=\"http://www.xx.com">here</a>, #xx or "}.

GuessLanguageFromStringBufferPipe determines the language of the text included in the instance. It adds "language" and "language-reliability" properties (by default) to the instance in order to store the result of the process. The *data* of the instance should contain text without HTML tags (call *StripHTMLFromStringBufferPipe* task). To detect both the language and the probability of a successful identification, we take advantage of the language-detector library for Java (https://github.com/optimaize/language-detector), which can distinguish up to 71 languages. By applying the pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe*) on the example, the instance is changed as follows: {props: {"language": "EN", "language-reliability": "0.9"}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on here, #xx or @xx"}.

InterjectionFromStringBufferPipe is able to identify and optionally drop interjections from text using dictionaries (JSON files). It adds them to the "interjection" property of the instance. Interjections are language-dependant and, therefore, the language of the instance should be computed before executing this task (*GuessLanguageFromStringBufferPipe*). This task modifies the following attributes of the provided example: by applying the pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *InterjectionFromStringBufferPipe*) on the example, the instance is changed as follows: {props: {"language": "EN", "language-reliability": "0.9", "interjection": "ho ho ho! -- ho -- here – "}, data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on here, #xx or @xx"}.

MeasureLengthFromStringBufferPipe adds the "length" property (by default) computed by measuring the length of the text included in the *data* attribute of the instance. This task made the following changes to the provided example: {props: {"length": "181"}}.

NERFromStringBufferPipe implements NER by adding all identified entities into instance properties and optionally deletes them from the input text. By default, date (property "NERDATE"), money ("NERMONEY"), number ("NERNUMBER"), address ("NERADDRESS"), and location ("NERLOCATION") are the entities that can be recognized. NER is implemented through the Stanford NLP framework. The changes made on the example when executing this task are {props: {"NERDATE": "December", "NERMONEY": "", "NERNUMBER": "31", "NERADDRESS": "", "NERLOCATION": ""}}.

SlangFromStringBufferPipe detects slang terms in the input text and replaces them with their corresponding formal term taken from dictionaries (JSON files). In order to select the appropriate dictionary, the language of the text should be previously computed (*GuessLanguageFromStringBufferPipe*). Using the pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *SlangFromStringBufferPipe*), the provided example gets modified as follows: {props: {"language":"EN", "language-reliability": "0.9"}, data: "December is here :-), hold on hold on hold on! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on here, #xx or @xx"}.

StripHTMLFromStringBufferPipe removes HTML tags and changes HTML entities by their corresponding characters (i.e., "*ñ*" is converted to "*ñ*"). Using this task, the input example is converted as follows: {data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 Dec. Visit us on here, #xx or @xx"}.

StopWordFromStringBufferPipe drops stopwords from text included in the *data* attribute of an instance. The text language should be previously detected to select the appropriatestopword dictionary. The *AbbreviationFromStringBufferPipe* task cannot be executed after his one.The modifications of the example when executing (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *StopWordFromStringBufferPipe*) are the following ones: {props: {"language": "EN", "language-reliability": "0.9"}, data: "December hre:-), ho ho ho! Beat Christmas days give 19% online 31 Dec. Visit here, #xx @xx"}.

Finally, the *StringBufferToLowerCasePipe* transforms the textual content included in the *data* attribute of an instance into lowercase. This task modifies the provided example as follows: {data: "December is hre :-), ho ho ho! Beat the Christmas days with us and we'll even give you 19% off online until 31 dec. visit us on <a href = \"http://www.xx.com">here</a>, #xx or @xx"}.

Additionally, an instance containing a *StringBuffer* can be transformed into a *SynsetSequence* or a *TokenSequence* (detailed in Sections 3.3 and 3.4). These functionalities are implemented by *StringBuffer2SynsetSequencePipe* and *StringBuffer2TokenSequencePipe*, respectively. The former uses Babelfy API (http://babelfy.org) to recognize synsets of each word included in the text. The latter implements a tokenizing process using a set of characters as word delimiters (tokens are represented in base64 format). Furthermore, *TeeCSVFromStringBufferPipe* stores instances in a Comma Separated Value(s) (CSV) file containing all computed properties together with the respective text and keeps the instance unmodified. The pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *StringBuffer2SynsetSequencePipe*) produces the following modifications of the example provided: {props: {"language": "EN", "language-reliability": "0.9"}, data: [["bn:00025645n", "December"], ["bn:00006898n", "ho"], ["bn:03100869n", "ho ho"], ["bn:00009396n", "Beat"], ["bn:00000086n", "Christmas days"], ["bn:03149538n", "online"], ["bn:00025645n", "Dec"], ["bn:14194518n", "Visit"]]}. Conversely, the pipeline (*StripHTMLFromStringBufferPipe* | *GuessLanguageFromStringBufferPipe* | *StringBufferToLowerCasePipe* | *StringBuffer2TokenSequencePipe*) makes the following modifications: {props: {"language": "EN", "language-reliability": "0.9"}, data: ["tk:ZGVjZW1iZXI = ", "tk:aXM = ", "tk:aHJl", "tk:aG8 = ", "tk:aG8 = ", "tk:aG8 = ", "tk:8J + OhQ == ", "tk:YmVhdA == ", "tk:dGhl", "tk:Y2hyaXN0bWFz", "tk:ZGF5cw == ", "tk:d2l0aA == ", "tk:dXM = ", "tk:YW5k", "tk:d2U = ", "tk:bGw = ", "tk:ZXZlbg == ", "tk:Z2l2ZQ == ", "tk:eW91", "tk:MTk = ", "tk:b2Zm", "tk:b25saW5l", "tk:dW50aWw = ", "tk:MzE = ", "tk:ZGVj", "tk:dmlzaXQ = ", "tk:dXM = ", "tk:b24 = ", "tk:aGVyZQ == ", "tk:eHg = ", "tk:b3I = ", "tk:eHg = "]}.

3.3. *Processing org.nlp.types.SynsetSequence.*

A *SynsetSequence* object brings together a sequence of synsets that are found in the text of an instance. To handle instances with this data type as input, NLPA includes the task *SynsetSequence2FeatureVectorPipe*. It is able to transform a SynsetSequence into a *FeatureVector* (detailed in Section 3.5) which compiles duplicated features and assigns a score to each feature according to a grouping scheme. The grouping scheme can be one of the following: (i) *SequenceGroupingStrategy.COUNT* (default value), which indicates the number of times that a synset is observed in the sequence; (ii) *SequenceGroupingStrategy.BOOLEAN*, which assigns 1 when the synset is included in the content, otherwise 0; and (iii) *SequenceGroupingStrategy.FREQUENCY*, which indicates the frequency of the synset in the text (number of times that the synset is observed divided by the entire number of synsets). By applying this task to the example instance having a *SynsetSequence* in the data attribute (see previous subsection), we achieve the following changes: {data: {"bn:00025645n": 2, "bn:00006898n": 1, "bn:03100869n": 1, "bn:00009396n": 1, "bn:00018836n": 1, "bn:03149538n": 1, "bn:14194518n": 1 }}.

3.4. *Processing org.nlp.types.TokenSequence.*

A *TokenSequence* contains the sequence of tokens that are found in the text of an instance. To handle instances with this data type as input, NLPA includes the tasks described hereinafter. For this subsection, we will use the instance containing a *TokenSequence* in data attribute shown in Section 3.2.

TokenSequence2FeatureVectorPipe, similarly to *SynsetSequence2FeatureVectorPipe*, transforms the list of tokens included in the text of the data instance into a *FeatureVector* according to a selected grouping scheme (*SequenceGroupingStrategy*). Using this task, the example instance can achieve the following transformations: {data: ["tk:ZGVjZW1iZXI = ": 1.0, "tk:aXM = ": 1.0, "tk:aHJl": 1.0, "tk:aG8 = ": 3.0, "tk:8J + OhQ == ": 1.0, "tk:YmVhdA == ": 1.0, "tk:dGhl": 1.0, "tk:Y2hyaXN0bWFz": 1.0, "tk:ZGF5cw == ": 1.0, "tk:d2l0aA == ": 1.0, "tk:dXM = ": 2.0, "tk:YW5k": 1.0, "tk:d2U = ": 1.0, "tk:bGw = ": 1.0, "tk:ZXZlbg == ": 1.0, "tk:Z2l2ZQ == ": 1.0, "tk:eW91": 1.0, "tk:MTk = ": 1.0, "tk:b2Zm": 1.0, "tk:b25saW5l": 1.0, "tk:dW50aWw = ": 1.0, "tk:MzE = ": 1.0, "tk:ZGVj": 1.0, "tk:dmlzaXQ = ": 1.0, "tk:b24 = ": 1.0, "tk:aGVyZQ == ": 1.0, "tk:eHg = ": 2.0, "tk:b3I = ":1.0]}.

TokenSequencePorterStemmerPipe applies the Porter stemmer algorithm to the *TokenSequence* included in an instance. This scheme is able to reduce the inflected (or sometimes derived) words to their stem form, using a set of language-dependant rules. The rules are defined by language. Hence, the language of the texts should be previously computed (see *GuessLanguageFromStringBufferPipe* in Section 3.2). By using this task, the example provided is transformed as follows: {data: ["tk:ZGVjZW1i", "tk:aQ == ", "tk:aHJl", "tk:aG8 = ", "tk: aG8 = ", "tk:aG8 = ", "tk:8J + OhQ == ", "tk:YmVhdGU = ", "tk: dGhl", "tk:Y2hyaXN0bWE = ", "tk:ZGF5", "tk:d2l0aA == ", "tk:dQ == ", "tk:YW5k", "tk:d2U = ", "tk:bGw = ", "tk: ZXZlbg == ", "tk:Z2l2ZQ == ", "tk:eW91", "tk:MTk = ", "tk: b2Y = ", "tk:b25saW4 = ", "tk:dW50aWw = ", "tk:MzE = ", "tk: ZGVj", "tk:dmlzaXQ = ", "tk:dQ == ", "tk:b24 = ", "tk: aGVyZQ == ", "tk:eA == ", "tk:b3I = ", "tk:eA == "] }. As an example, the task changes some tokens including "days" that is transformed into "day".

TokenSequenceStemIrregularPipe applies irregular stemming (through language-dependent dictionaries) to tokens with the same purpose as the previous one. The irregular stemming task, if applied, should be executed before *TokenSequencePorterStemmerPipe*, and the language of the text should be computed before its use. In this case, the instance is modified as follows: {data: ["tk: ZGVjZW1iZXI = ", "tk:YmU = ", "tk:aHJl", "tk:aG8 = ", "tk: aG8 = ", "tk:aG8 = ", "tk:8J + OhQ == ", "tk:YmVhdA == ", "tk:dGhl", "tk:Y2hyaXN0bWFz", "tk:ZGF5cw == ", "tk: d2l0aA == ", "tk:dXM = ", "tk:YW5k", "tk:d2U = ", "tk: bGw = ", "tk:ZXZlbg == ", "tk:Z2l2ZQ == ", "tk:eW91", "tk: MTk = ", "tk:b2Zm", "tk:b25saW5l", "tk:dW50aWw = ", "tk: MzE = ", "tk:ZGVj", "tk:dmlzaXQ = ", "tk:dXM = ", "tk: b24 = ", "tk:aGVyZQ == ", "tk:eHg = ", "tk:b3I = ", "tk: eHg = "]}. Concretely, as an example, the second term "is" has been changed to "be".

3.5. Processing org.nlp.types.FeatureVector. FeatureVector compiles a set of features of text properties (synset-based or token-based), identified in the text of an instance, and their values. To handle input instances with this data type, NLPA includes both *TeeCSVFromFeatureVectorPipe* and *TeeDatasetFromFeatureVectorPipe* tasks, which are able to save a dataset into disk (CSV format) or to memory (*org.bdp4j.types.Dataset*) for their subsequent use. These datasets can be easily used to execute experiments in Weka Machine Learning Software (https://www.cs.waikato.ac.nz/ml/weka/) through the functionalities provided by BDP4J framework.

The next section presents a case study in which NLPA is used in order to show the creation and exploitation of a pipeline containing some of the previously described tasks.

4. Using NLPA

NLPA is a plugin for BDP4J that implements a set of natural language processing task definitions. BDP4J is a pipelining Java framework able to combine and orchestrate the execution of preprocessing tasks in sequence or in parallel. The orchestration can be defined in Java source or using XML files. Additionally, BDP4J adds a wide variety of constraint checks that prevent development errors (dependencies and input/output types for tasks). BDP4J also supports the instance invalidation required when an NLPA task fails. It also resumes the execution of pipelining processing after a hardware-software failure. Finally, BDP4J supports the debugging mode that is able to avoid the execution of some tasks by restoring the results they achieved in a previous execution. This functionality was particularly useful during the development of the NLPA plugin.

Figure 2 includes a class diagram and a fragment of source code showing the interaction of BDP4J and NLPA projects. Figure 2(a) specifies some architecture details to facilitate the comprehension of the inner operation of both projects. As we can see, NLPA tasks (three of them have been included as examples) are created as a subclass of *org.bdp4j.pipe.AbstractPipe* BDP4J.

As we can see in Figure 2(b), the pipeline orchestration comprises some tasks (see Section 3) executed in sequence (lines 2–18). In order to check whether the pipeline has been correctly defined, a dependency check is executed (lines 21–25). Additionally, NLPA incorporates mechanisms to automatically load instances from files (lines 28–29). And finally, instances are processed (line 30).

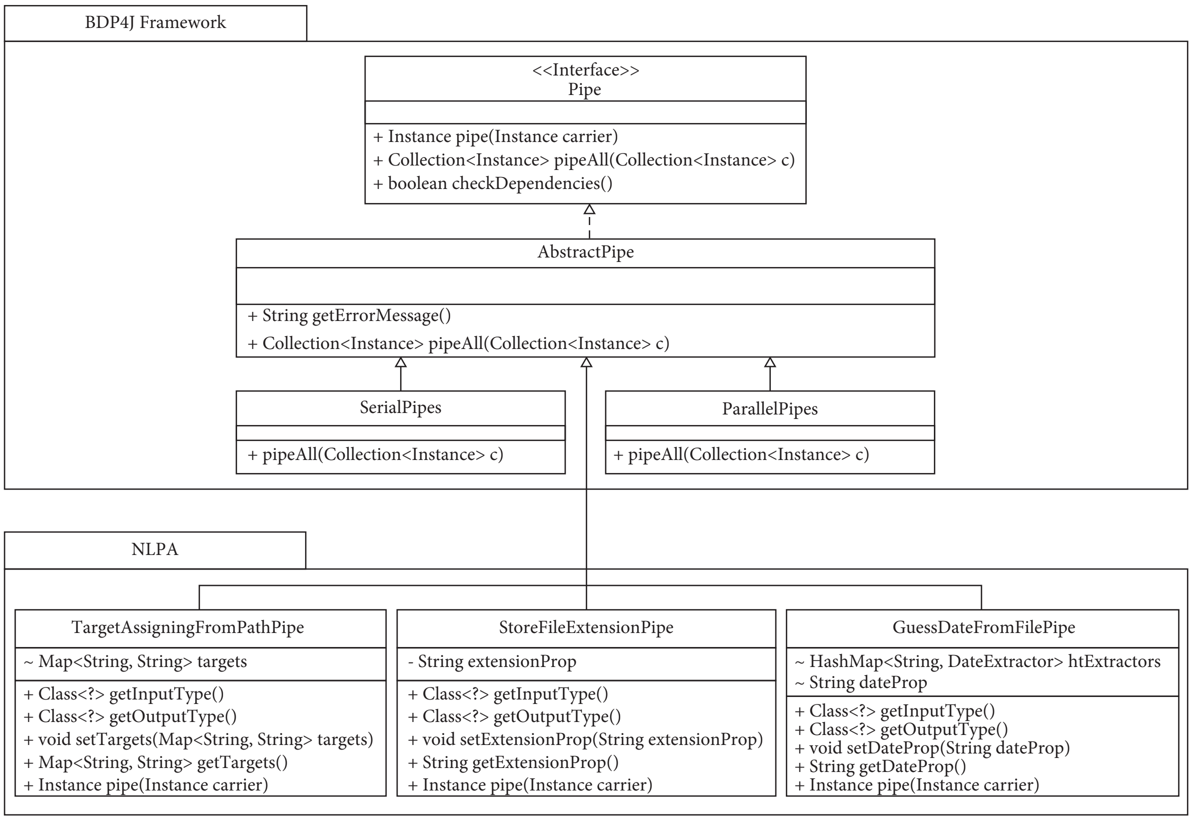
The next subsection introduces a complete case study showing the functionality of the NLPA plugin in greater detail and its interaction with BDP4J to invoke Weka functionalities.

4.1. A Case Study. In order to test the functionality of NLPA, we developed a complete case study, which is presented hereinafter. In this context, we propose a spam filtering problem that will be solved using ML binary supervised classification (spam and ham classes). To take advantage of *TargetAssigningFromPathPipe* task, files to classify will be stored into "_ham_" and "_spam_" folders according to their real class. It should be noted that assigning a target is not required for other kinds of problems (i.e., clustering).

For this particular case study, we search for spam corpora containing NLPA supported formats. Table 2 compiles a list of publicly available corpora that were found with this purpose.

For the current work, we selected small corpora that can be quickly analysed even when the available computational resources are low. As a result, we found that *YouTube Spam Collection Dataset* and *SMS-Spam-Collection v.1* are the most adequate corpora for this case study. For preprocessing *YouTube Spam Collection Dataset*, we used the IDs of comments (not the text included in the original source) and redownloaded the original text using the YouTube public API[6] (https://developers.google.com/youtube/v3/). Both corpora were preprocessed using the NLPA pipeline shown in Figure 3. The names of the tasks have been abbreviated according to the nomenclature shown in Table 1.

As shown in Figure 3, pSynsets and pTokens are two examples of SerialPipes comprising a set of preprocessing tasks (identified with a number) to process corpora. The inner processing made by both pipelines is very similar (there are many common steps which are shown in black in Figure 3). However, pSynsets generates a dataset using synsets for representing the text while pTokens generates a token-based representation. When both pipelines are


BDP4J Framework
<<Interface>>
Pipe
+ Instance pipe(Instance carrier)
+ Collection<Instance> pipeAll(Collection<Instance> c)
+ boolean checkDependencies()
AbstractPipe
+ String getErrorMessage()
+ Collection<Instance> pipeAll(Collection<Instance> c)
SerialPipes
+ pipeAll(Collection<Instance> c)
ParallelPipes
+ pipeAll(Collection<Instance> c)
NLPA
TargetAssigningFromPathPipe
~ Map<String, String> targets
+ Class<?> getInputType()
+ Class<?> getOutputType()
+ void setTargets(Map<String, String> targets)
+ Map<String, String> getTargets()
+ Instance pipe(Instance carrier)
StoreFileExtensionPipe
- String extensionProp
+ Class<?> getInputType()
+ Class<?> getOutputType()
+ void setExtensionProp(String extensionProp)
+ String getExtensionProp()
+ Instance pipe(Instance carrier)
GuessDateFromFilePipe
~ HashMap<String, DateExtractor> htExtractors
~ String dateProp
+ Class<?> getInputType()
+ Class<?> getOutputType()
+ void setDateProp(String dateProp)
+ String getDateProp()
+ Instance pipe(Instance carrier)

(a)

```
/* Creating the processing pipe */
AbstractPipe p = new SerialPipes(
  new AbstractPipe[]{
    new TargetAssigningFromPathPipe(),
    new StoreFileExtensionPipe(),
    new GuessDateFromFilePipe(),
    new File2StringBufferPipe(),
    new StripHTMLFromStringBufferPipe(),
    new GuessLanguageFromStringBufferPipe(),
    new SlangFromStringBufferPipe(),
    new StringBufferToLowerCasePipe();
    new StringBuffer2TokenSequencePipe(),
    new TokenSequence2FeatureVectorPipe(),
    new TeeCSVFromFeatureVectorPipe()
  }
);
logger.info("orchestration:" + p.toString() +
" \ n");

/*Check orchestration dependencies*/
if (!p.checkDependencies()) {
     logger.fatal("[CHECK DEPENDENCIES] " +
         AbstractPipe.getErrorMessage());
     System.exit(-1);
}

/*Load and process instances*/
ArrayList<Instance> instances =
     InstanceListUtils.load("instances/")
p.pipeAll(instances);
```

(b)

Figure 2: Example of the use of NLPA. (a) BDP4J framework and NLPA plugin interaction. (b) Example of pipeline orchestration and execution defined in Java.

started, a target class (ham or spam) is assigned to each instance (task 1). Then, the type of input file (task 2) and the text date (task 3) are stored in the “extension” and “date” properties of the instance. After that, the textual content is obtained from the original file and included in the *data* attribute of an instance as a *StringBuffer* object (task 4). From the text of the instance, several properties (using the default name) are extracted: (i) the length of the text (task 5), (ii) user names (task 6), (iii) hashtags (task 7), (iv) URLs (task 8), (v) emoticons (task 9), and (vi) emojis (task 10). Additionally, all text recognized and included in the previously mentioned properties (except *i*) is removed from the original content.

After removing HTML tags and substituting entities by their corresponding character (task 11), and due to the need of identifying the language of instances for applying many text transformations, we use *GuessLanguageFromStringBufferPipe* (task 12). Several dictionary-based tasks are then executed over the text: (i) expand contractions (task 13), (ii) expand abbreviations (task 14), (iii) transforming slang terms into formal terms (task 15), (iv) detect and remove interjections

Table 2: List of available corpora for spam classification.

Dataset name	Language	Type of contents	Size Ham	Size Spam	Available at
CSDMC 2010 Spam Corpus	English	E-mail messages	2,949	1,378	https://github.com/hexgnu/spam_filter/tree/master/data
TREC 2007 Public Corpus	English		25,220	50,199	http://plg.uwaterloo.ca/~gvcormac/treccorpus07/
SpamAssassin	English		4,150	1,897	http://spamassassin.apache.org/old/publiccorpus/
Enron e-mail	English		619,446	0	http://spamassassin.apache.org/old/publiccorpus/
Bruce Guenter spam collection	English		0	>3M	http://untroubled.org/spam/
Ling spam	English		2,412	481	http://csmining.org/index.php/ling-spam-datasets.html
SMS-spam-collection v.1	English	SMS messages	4,827	747	http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/
British English SMS corpora	English		450	425	https://mtaufiqnzz.wordpress.com/british-English-sms-corpora/
Webspam-UK 2007	English	Web pages	105,896,555 —	—	http://chato.cl/webspam/datasets/index.php
Websmap-UK 2011	English		1,769	1,998	https://sites.google.com/site/heiderawahsheh/home/web-spam-2011-datasets/uk-2011-web-spam-dataset
DC 2010/EU 2010	English, French, and German		23M —	—	https://dms.sztaki.hu/en/letoltes/ecmlpkdd—2010—discovery—challenge—data—set
Webb spam 2011	-		0	330,000	http://www.cc.gatech.edu/projects/doi/WebbSpamCorpus.html
ClueWeb 09	Multilingual		1,040M —	—	http://www.lemurproject.org/clueweb09.php/
ClueWeb 12	English		870M —	—	http://www.lemurproject.org/clueweb12.php/
Common Crawl Data	Multilingual		0	9B	http://commoncrawl.org/
YouTube Comments Dataset	Multilingual	YouTube comments	5,950,137	481,334	http://mlg.ucd.ie/yt/
YouTube Spam Collection Dataset	English		951	1,005	https://archive.ics.uci.edu/ml/datasets/YouTube+Spam+Collection
HSpam14.s2	-	Twitter messages	14M —	—	http://www3.ntu.edu.sg/home/axsun/datasets.html

(task 16), and finally (v) drop stopwords (task 17). The text is then converted to lowercase (task 18).

After the 18th task, the operation of pSynsets and pTokens is clearly different. pSynsets (highlighted in green) builds a *SynsetSequence* (i.e., a list of synsets identified in the remaining text) and then transforms the synset sequence into a synset-based *FeatureVector* (using *SequenceGroupingStrategy.COUNT*). Furthermore, pTokens (highlighted in blue) transforms the text into a TokenSequence (task 19). After that, each token is reduced to its root form (tasks 20 and 21 for irregular and regular stemming) and then a token-based *FeatureVector* is built compiling duplicated tokens into a single feature with the number of times the token was found.

Finally, datasets are generated in memory and disk using *TeeCSVFromFeatureVectorPipe* and *TeeDatasetFromFeatureVectorPipe* (tasks 21 and 22 for pSynsets and 23 and 24 for pTokens). Storing the results of preprocessing in a CSV file is useful to avoid future preprocessing of the same corpora.

After preprocessing (NLPA functionalities), and in order to continue with the classification process, each dataset was split into two stratified groups with 80% and 20% of the original instances to be used for training and testing purposes, respectively. The dimensionalities of these datasets were reduced by applying Weka Information Gain implementation [33] to the training data. Weka implementations of (Sequential Minimal Optimization) SMO and NaïveBayes algorithms were then used to create classifier models using the training data. Finally, each trained model was evaluated using test instances as input.

For comparison purposes, Figure 4 shows a percentage-based evaluation to analyse the number of false positive (FP), false negative (FN) errors, and the hits (OK) together with a detailed comparison using kappa, recall, precision, and f-score evaluations.

As we can see from Figure 4, the differences achieved between token-based and synset-based classification processes are not significant in *SMS-Spam-Collection v.1*. However, we find appreciable differences in *YouTube Spam Collection Dataset*. The poor performance achieved when using synsets representation is due to the reduced number of synsets found in more than one message of each dataset. In this regard, NLPA does not implement generalization schemes, which would allow grouping synset features with similar semantic meanings [21]. For instance, "Viagra",

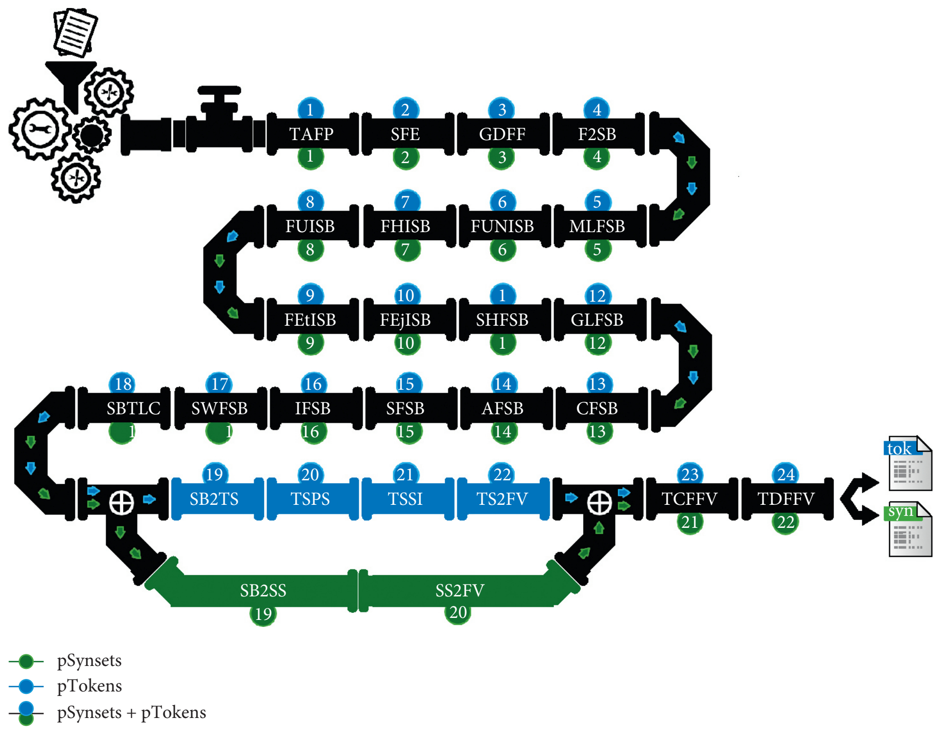
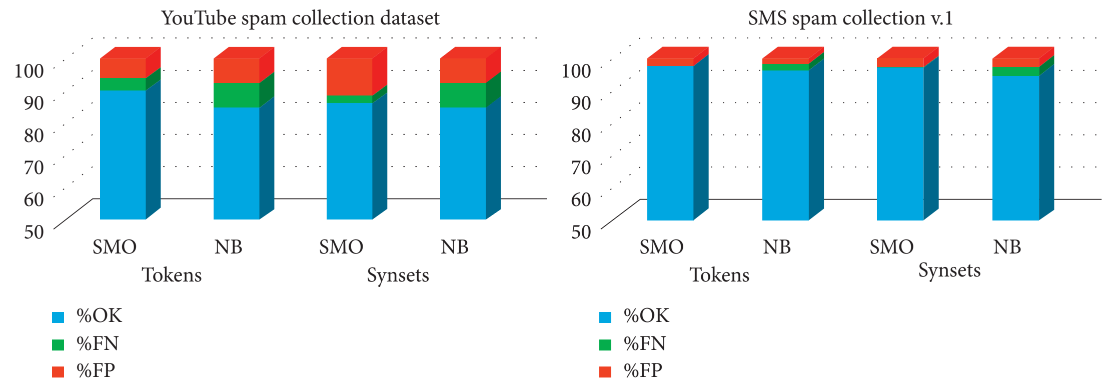


Figure 3: Pipeline used to preprocess corpora.



(a)

	Synsets	Tokens	Synsets	Tokens	Synsets	Tokens	Synsets	Tokens
Kappa	0.539	0.668	0.670	0.779	0.773	0.845	0.867	0.888
Recall	0.623	0.789	0.667	0.829	0.820	0.893	0.807	0.832
Precision	0.754	0.789	0.902	0.887	0.788	0.842	0.976	0.984
f-score	0.683	0.789	0.767	0.857	0.804	0.866	0.883	0.902

(b)

Figure 4: Evaluation of preprocessing configurations. (a) Percentage comparison. (b) Kappa, recall/precision, and f-score evaluations.

"Cialis", and "Xanax" synsets could be grouped into a new "drug" feature. While the use of bigger datasets could partially solve this situation, these solutions are not within the scope of this work.

The source code of this case study is provided for its evaluation as a supplementary file (nlpa_case_study.zip). As shown, NLPA can be easily used to evaluate different text preprocessing and representation configurations. The next section compiles some conclusions achieved during the development of NLPA software.

5. Learned Lessons and Main Outcomes

The development of this work started from the idea of applying big data tools to preprocess the most popular and hard to process sources: text. Many researchers are mining the big streams of data generated by social networks for different purposes (see Introduction section) without using specific big data tools and frameworks. In this work, we create a BDP4J plugin (NLPA), implementing some tasks to be used for preprocessing text sources using pipeline schemes.

The BDP4J framework contributes practical capabilities such as resuming a pipeline execution after unexpected software or hardware failures. However, the most important issue that can be highlighted from BDP4J is the support it provides to invalidate instances anytime during the entire pipelining process (when discovering a problem with the data). In fact, this functionality allows adequately handling situations such as the deletion of contents included in a data source (for instance a tweet) or discarding instances that cannot be processed because their language cannot be determined. Nevertheless, in the case study, the pipeline was defined using Java API and can be graphically (GUI) defined using the command "*java -jar bdp4j.java gui*". Using GUI, it is possible to generate an XML file that can be used to preprocess instances. Finally, the debug mode was particularly useful for saving time during the development of concrete tasks since it allows skipping the execution of previous steps by loading their results from disk.

NLPA makes an extensive use of regular expressions. Regular expressions are the most efficient form of recognizing text patterns (such as URLs and Hashtags). In addition to these common uses, regular expressions were also employed to facilitate the execution of dictionary-based tasks such as stopwords removal, abbreviation expanding, or slang translation, which allowed us to correctly complete the matching process for entire words, and facilitated the development of our proposal. However, regex patterns should be carefully created and invoked in order to prevent speed loss [34, 35]. In particular, the methods included in *java.lang.String* class should not be used, as they imply recompiling regular expressions.

Finally, we utilized software configuration management (SCM) tools and version control systems (VCS), namely, Maven and Git. The former allowed us to efficiently manage software dependencies in order to automatically build, test, and make the software through a repository available. The next section shows conclusions and outlines future work.

6. Conclusions and Future Work

This work has introduced NLPA, a plugin to preprocess big data text contents into full-featured datasets containing different generic text properties (e.g., length, language) together with synset or token columns. To this end, NLPA incorporates a wide variety of small preprocessing tasks (e.g., urban language translation, finding interjections, or abbreviation expanding). NLPA was built using a pipeline-based framework (BDP4J) to facilitate the processing of big data text sources. BDP4J implements the orchestration and execution of a computational pipeline and facilitates the integration of data with the Weka machine learning framework.

NLPA provides several important features such as multiple data source support (Twitter, e-mail, YouTube comments, websites, and SMS) and the implementation of the semantic representation of text (synsets). Additionally, only minimal time is required to learn how to operate NLPA and build a pipeline configuration in comparison with other tools such as GATE or UIMA. Moreover, the design of new preprocessing tasks to extend NLPA can be made by simply extending *org.bdp4j.pipe.AbstractPipe* class. Finally, NLPA is distributed as open-source software using GPL v.3 license and can be downloaded from GitHub [36].

In this work, we have incorporated a complete case study of the use of NLPA to classify two publicly available spam datasets. The provided case study implements a basic analysis of the capabilities of the plugin that highlights its applicability for developing and testing different solutions in the Text Analytics domain.

The development of NLPA and other similar tools became essential in order to take advantage of current hardware advances for analysing big data and, in particular, a vast number of textual data sources. Future work comprises an extension of the plugin by implementing new preprocessing tasks (such as POS tagging or ngram token feature support) and an application for analysing big data from different domains to achieve relevant information.

Data Availability

The source code of the NLPA plugin described in this study has been deposited in the GitHub repository (https://github.com/sing-group/nlpa, DOI: 10.5281/zenodo.3356589). The whole source used developed for the case study included in the work has been included as a supplementary file (nlpa_case_study.zip). The SMS-Spam-Collection v.1 dataset used to run the case study included in this work is available for download at http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/ and https://archive.ics.uci.edu/ml/datasets/sms+spam+collection. A formatted version of this corpus (ready for its usage on NLPA) is included in the sms-spam-collection folder in the provided supplementary file (nlpa_case_study.zip).

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

D. Ruano-Ordás was supported by a postdoctoral fellowship from Xunta de Galicia (ED481B 2017/018). Additionally, this work was funded by the project Semantic Knowledge Integration for Content-Based Spam Filtering (TIN2017-84658-C2-1-R) from the Spanish Ministry of Economy, Industry and Competitiveness (SMEIC), State Research Agency (SRA), and the European Regional Development Fund (ERDF). SING group thanks CITI (*Centro de Investigación*, *Transferencia e Innovación*) from the University of Vigo for hosting its IT infrastructure.

Supplementary Materials

The whole source used developed for the case study included in the work has been included as a supplementary file (nlpa_case_study.zip). A formatted version of this corpus (ready for its usage on NLPA) is included in the sms-spam-collection folder in the provided supplementary file (nlpa_case_study.zip). (*Supplementary Materials*)

References

[1] M. Armstrong, "Global data creation is about to explode," 2019, https://www.statista.com/chart/17727/global-data-creation-forecasts/.

[2] K. Slavakis, G. B. Giannakis, and G. Mateos, "Modeling and Optimization for Big Data Analytics: (Statistical) learning tools for our era of data deluge," *IEEE Signal Processing Magazine*, vol. 31, no. 5, pp. 18–31, 2014.

[3] M. Chen, S. Mao, and Y. Liu, "Big data: a survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[4] C. L. Philip Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: a survey on Big Data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[5] L. Shanhong, "Big data market revenue forecast worldwide 2011-2027," 2019, https://www.statista.com/statistics/254266/global-big-data-market-forecast/.

[6] M. Gualtieri, "Hadoop is data's darling for a reason," 2016, https://go.forrester.com/blogs/hadoop-is-datas-darling-for-a-reason/.

[7] S. Kemp, "Digital 2018: global digital overview," 2018.

[8] J. Leipzig, "A Review of Bioinformatic Pipeline Frameworks," *Briefings in Bioinformatics*, vol. 18, no. 3, Article ID bbw020, 2016.

[9] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in *Proceedings of the 18th International Database Engineering & Applications Symposium on-IDEAS '14*, pp. 356–361, ACM Press, New York, New York, USA, 2014.

[10] P. O'Donovan, K. Leahy, K. Bruton, and D. T. J. O'Sullivan, "An industrial big data pipeline for data-driven analytics maintenance applications in large-scale smart manufacturing facilities," *Jornal of Big Data.* vol. 2, p. 25, 2015.

[11] S. Saif and S. Wazir, "Performance analysis of big data and cloud computing techniques: a survey," *Procedia Computer Science*, vol. 132, pp. 118–127, 2018.

[12] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[13] Z. Zheng, J. Zhu, and M. R. Lyu, "Service-generated big data and big data-as-a-service: an overview," in *Proceedings of the 2013 IEEE International Congress on Big Data*, pp. 403–410, Santa Clara, CA, USA, June 2013.

[14] K. Gai and S. Li, "Towards cloud computing: a literature review on cloud computing and its development trends," in *Proceedings of the 2012 Fourth International Conference on Multimedia Information Networking and Security*, pp. 142–146, Nanjing, China, November 2012.

[15] P. Di Tommaso, "Awesome-pipeline," 2014, https://github.com/pditommaso/awesome-pipeline.

[16] A. Moreno and T. Redondo, "Text analytics: the convergence of big data and artificial intelligence," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 3, no. 6, p. 57, 2016.

[17] M. Shastri, S. Roy, and M. Mittal, "Stock price prediction using artificial neural model: an application of big data," *ICST Transactions on Scalable Information Systems*, 2018.

[18] P. Ducange, R. Pecori, and P. Mezzina, "A glimpse on big data analytics in the framework of marketing strategies," *Soft Computing*, vol. 22, no. 1, pp. 325–342, 2018.

[19] S. Bloehdorn and A. Hotho, "Boosting for text classification with semantic features," in *Advances in Web Mining and Web Usage Analysis*, pp. 149–166, Springer, Berlin, Germany, 2006.

[20] A. Ben Abacha and P. Zweigenbaum, "Automatic extraction of semantic relations between medical entities: a rule based approach," *Journal of Biomedical Semantics*, vol. 2, p. S4, 2011.

[21] J. R. Méndez, T. R. Cotos-Yañez, and D. Ruano-Ordás, "A new semantic-based feature selection method for spam filtering," *Applied Soft Computing*, vol. 76, pp. 89–104, 2019.

[22] S. A. Babar and P. D. Patil, "Improving performance of text summarization," *Procedia Computer Science*, vol. 46, pp. 354–363, 2015.

[23] M. Novo-Lourés, J. R. Méndez, Y. Lage, R. Laza, R. Pavón, and M. Reboiro, "BDP4J: Big Data Preprocessing for Java," 2019, https://github.com/sing-group/bdp4j.

[24] B. Putano, *A Look At 5 of the Most Popular Programming Languages of 2019*.

[25] A. K. McCallum, "MALLET: a machine learning for language toolkit," 2002, http://mallet.cs.umass.edu.

[26] H. Cunningham, V. Tablan, A. Roberts, and K. Bontcheva, "Getting more out of biomedical documents with GATE's full lifecycle open source text analytics," *PLoS Computational Biology*, vol. 9, no. 2, Article ID e1002854, 2013.

[27] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky, "The Stanford CoreNLP natural language processing toolkit," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pp. 55–60, https://stanfordnlp.github.io/CoreNLP/, Baltimore, MD, USA, June 2014.

[28] Apache Software Foundation, "Open NLP," 2017, http://opennlp.apache.org.

[29] D. Ferrucci, A. Lally, K. Verspoor, and E. Nyberg, "UIMA: unstructured information management architecture," 2009, https://uima.apache.org.

[30] I. Gurevych, M. Mühlhäuser, J. Steimle, M. Weimer, and T. Zesch, "Darmstadt knowledge processing repository based on UIMA," in *Proceedings of the First Workshop on Unstructured Information Management Architecture at Biannual Conference of the Society for Computational Linguistics and Language Technology*, Tubingen, Germany, 2007, https://fileserver.tk.informatik.tu-darmstadt.de/Publications/2007/gldv-uima-ukp.pdf.

[31] R. Agerri, X. Artola, Z. Beloki, G. Rigau, and A. Soroa, "Big data for natural language processing: a streaming approach," *Knowledge-Based Systems*, vol. 79, pp. 36–42, 2015.

[32] P. Resnick, "Internet message format," 2008, https://tools.ietf.org/html/rfc5322.

[33] J. R. Méndez, I. Cid, D. Glez-Peña, M. Rocha, and F. Fdez-Riverola, "A comparative impact study of attribute selection techniques on naïve bayes spam filters," in *Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*, pp. 213–227, Springer, Berlin, Heidelberg, 2008.

[34] Baeldung, "An overview of regular expressions performance in java," 2019, https://www.baeldung.com/java-regex-performance.

[35] C. Mocanu, "Optimizing regular expressions in java," 2007, https://www.javaworld.com/article/2077757/optimizing-regular-expressions-in-java.html.

[36] J. R. Méndez, M. Novo, R. Pavón et al., "NLPA: natural language pre-processing architecture," 2019.