Improving Pipelining Tools for Pre-processing Data

María Novo-Lourés [1,2,3], Yeray Lage[1], Reyes Pavón[1,2,3], Rosalía Laza[1,2,3], David Ruano-Ordás[1,2,3], José Ramón 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, 32004 Ourense (Spain)
[2] CINBIO, University of Vigo, Research Group SI4, Department of Computer Science, 32004 Ourense (Spain)
[3] SING Research Group, Galicia Sur Health Research Institute (IIS Galicia Sur), SERGAS-UVIGO (Spain)

UNIR
LA UNIVERSIDAD
EN INTERNET

Abstract

The last several years have seen the emergence of data mining and its transformation into a powerful tool that adds value to business and research. Data mining makes it possible to explore and find unseen connections between variables and facts observed in different domains, helping us to better understand reality. The programming methods and frameworks used to analyse data have evolved over time. Currently, the use of pipelining schemes is the most reliable way of analysing data and due to this, several important companies are currently offering this kind of services. Moreover, several frameworks compatible with different programming languages are available for the development of computational pipelines and many research studies have addressed the optimization of data processing speed. However, as this study shows, the presence of early error detection techniques and developer support mechanisms is very limited in these frameworks. In this context, this study introduces different improvements, such as the design of different types of constraints for the early detection of errors, the creation of functions to facilitate debugging of concrete tasks included in a pipeline, the invalidation of erroneous instances and/or the introduction of the burst-processing scheme. Adding these functionalities, we developed Big Data Pipelining for Java (BDP4J, https://github.com/sing-group/bdp4j), a fully functional new pipelining framework that shows the potential of these features.

Keywords

I. Introduction and Motivation

Data mining techniques emerged as a set of tools for exploiting heterogeneous and often unstructured data compiled from a wide variety of information sources to improve decision support processes in different domains (healthcare, commercial decisions, stocks market predictions,...) [1]. Under this paradigm for addressing decision support, facts could not be explained with simple and isolated variables from the same domain, but as the combination of a large collection of circumstances (variables) that occur in different and heterogeneous domains [2]. Hence, stocks market predictions, for example, should be modelled by compiling information about the moods of the people (maybe from the news or from social networks), company results (profit and loss), customer satisfaction, the company image (for non-customers), etc. These intuitive and simple ideas became more and more popular and originated the current revolution of big-data.

Tools and programming methods to implement the compilation and pre-processing of data have evolved considerably over time. Currently the most advanced tools to address these issues are included in big-data frameworks, which facilitates the processing of enormous volumes of information over large computer clusters. Particularly, MapReduce [3], [4] is a powerful and earlier programming paradigm, mainly popularized by Google and Hadoop Project, which simplifies the processing of data using hundreds of cluster nodes. Several MapReduce implementations are available including Apache Hadoop [5], Amazon Elastic MapReduce [6], Disco MapReduce [7], [8] and Spark [9], among others. However, MapReduce does not provide a complete solution to easily address all the problems of pre-processing data [3], [4], [10]. Particularly, the most relevant limitation of MapReduce is the processing model (batch) which requires all data to be (up) loaded into the cluster to execute its analysis. Consequently: (*i*) this model is not suitable for processing real-time streaming sources (such as twitter streaming data); and (*ii*) the harnessing of the computational capabilities of the cluster is clearly hampered by the need for loading the data. These limitations led to the introduction of other forms of analysing data, such as the pipelining methods [11]. These methods are based on dividing the data analysis process into a set of small and easy to implement tasks; and orchestrating their sequential/ parallel execution. After the emergence of pipelining methods, many pipelining frameworks were introduced [12] and several important companies started offering effective pipeline-based data analysis services such as AWS Data Pipeline [13], SnapLogic IIP [14] or Alooma Enterprise Data Pipeline [15]. Due to the popularity of pipeline technology, recent studies have addressed the issue of improving the speed of data processing [16]–[20]. However, there are still many areas for improvement of such frameworks, such as the facilities provided to

developers or the implementation of mechanisms capable of detecting errors at an early stage (constraints, datatype issues, etc.).

During the last years, Java has become popular in the development of enterprise software [21]. By getting in touch with IT –Information Technology– professionals and examining job offers, we found that important companies (e.g., Inditex, PSA, CaixaBank) primarily use Java technology in their developments. Hence the use of Java to define pipelines (*i*) enables the possibility of reusing many software components (especially persistence components to access their information); (*ii*) facilitates the search of qualified developers; and (*iii*) allows for the use of a wide variety of software libraries and services that are currently developed in Java (sometimes exclusively) [22], [23].

In this context, this work focuses specifically on the pipelining schemes used by enterprises and, therefore, primarily considers pipelining frameworks supporting the execution of Java tasks. This study identified some limitations of current software implementing pipeline functions such as: (*i*) the definition or checking of constraints; (*ii*) the invalidation inconsistent/error data; or (*iii*) the data sharing between tasks. Particularly, some mechanisms should be introduced to minimize the errors in the entire process (constraints). Additionally, when invalid data are found they should be marked to discard (instance invalidation) and should not be further processed. Finally, sharing data between tasks cannot be safely implemented through external repositories (databases, files, etc.) because some task execution questions should be previously addressed (i.e. how additional data instances -with regard to the initial ones- affect the process). To this end, we have developed BDP4J [24], a new pipelining framework that successfully address all limitations identified for analysed software. In addition to BDP4J software, this work also contributes different processing schemes for resuming or debugging the operation of pipelines, several constraints that should be checked in pipelines, and methods to implement the identified functionalities.

The rest of our work is structured as follows: Section II presents the state of the art in Java frameworks to define computational pipelines together with their limitations. Section III introduces BDP4J as a framework able to solve the limitations that have been identified in analysed software. Finally, section IV summarizes the main outcomes and future work.

II. State of Art

As stated before, several pipeline-based tools for data processing have been successfully introduced in recent years. In order to facilitate the implementation of pipeline-based data solutions in enterprises and reutilize previously developed software components, the use of language-independent, and/or Java pipelining frameworks would probably be the best solution. In this section, we analyse a set of the most adequate frameworks, demonstrating their strengths and weaknesses, in order to facilitate the definition of (big) data studies by enterprises. We analyse the functionalities of some frameworks that are no longer available (such as COMPSs [25], [26] whose URLs and GitHUB repositories have been removed) or currently obsolete (Conan2 [27], Dockerflow [28], [29], Suro [30] and Swift [31]–[33]). Conan2 can only be built using Java 6 which is obsolete, Dockerflow has been abandoned on 2017 (see readme.md in official GitHUB repository), Suro can only be compiled using obsolete Gradle/JDK versions and Swift requires Java endorsed dirs property which is obsolete (see https://docs.oracle.com/javase/8/docs/technotes/guides/standards/). Moreover, we also analyzed other available software including COMPI [34], Cromwell [35], Drake [36], Mallet [37], [38] and ML –Machine Learning– pipeline [9], [39], [40].

By studying the above-mentioned frameworks, we identified a set of relevant features that are not addressed by all of them. Table I summarizes the analysed features and displays a comparative analysis of their presence in the studied frameworks

As shown in Table I, the vast majority of analysed frameworks do not have any checking strategy while orchestrating tasks inside a pipeline (*i*, *ii*). Keeping this in mind, we estimate the utility of type constraints included in strong typing languages (such as Java) as an effective method to prevent development errors (*i*). Inspired by strong typing languages, we found type constraints appropriate to check whether the type of output information generated by a task is consistent with the input information required for the next task executed in sequence. The type of input information for tasks that are executed in parallel should be the same. We are strongly convinced of the need to add some constraints to establish the right execution order of tasks (*ii*) and prevent inconsistent pre-processing (for instance, stripping HTML tags is mandatory before tokenizing contents). Additionally, in our view, tasks can be classified into different categories (*iii*), thus contributing to help users in their selection (if a GUI –Graphical User Interface– is provided) and in the validation of their orchestration. The goal of some tasks in particular is to simply compute instance properties and not to transform the input data. Therefore, the classification of tasks into different categories would provide the possibility of defining constraints for validating tasks and their orchestration as well as to detect the execution errors.

Additionally, taking advantage of parallel processing schemes (*iv*) and/or distributed execution methods (*v*) could contribute to reduce the data processing time required.

The communication between tasks is often not limited to the output-input stream. As an example, in text mining, a dictionary is created after the tokenizing process, and can be used later to build dense vector representations. However, data sharing (*vi*) between tasks in burst-based operation implies taking additional issues into account. For example, in a large number of problems, the task that generates the shared data should complete the processing of all data instances belonging to the current burst before executing the task that consumes the shared data. Additionally, we should also select the fastest sharing mechanisms from those available. In this sense the use of language-internal mechanisms (for example, a singleton-based object) is preferred to that of external mechanisms (a database). However, if only external mechanisms can be used, they should be carefully selected to improve performance. In this sense, distributed memory object caching systems (such as Memcached or Ehcache) are clearly better than (slower) SQL-based database servers.

Loading task orchestration (*vii*) from a file (XML –eXtensible Markup Language– or YAML –YAML Ain't Markup Language–) allows for providing independence between individual task definitions and their orchestration. Additionally, these orchestration definition languages would allow customizing tasks with specific execution parameters and easily modifying the steps of the pipeline.

The dynamic loading of tasks (*viii*) from files included in a directory (for instance .jar files) enables the user to develop customized tasks and facilitates their development (because the tasks can be defined in separate projects).

Once a great amount of data has been compiled, it must be analysed by using ML strategies. Therefore, data mining developers would greatly appreciate the provision of facilities for the integration of ML frameworks (such as Weka) (*ix*).

The most relevant feature that could be provided from a pipeline framework is probably the ability to transparently resume the execution (*x*) of a particular pipeline when the application crashes. To implement this functionality, we should address the persistence of data instances after being processed by each task and check whether

TABLE I. Relevant Features for Pipeline Frameworks

	COMPSs	COMPI	Conan2	Cromwell	Dockerflow (+Google Cloud Dataflow API)	Drake	Mallet	ML Pipeline (Apache Spark)	Suro	Swift
(i) Type check				✓	✓			✓		
(ii) Dependency check	✓	✓		✓		✓				✓
(iii) Different kinds of tasks						✓				
(iv) Parallel processing	✓	✓			✓	✓		✓	✓	✓
(v) Distributed	✓				✓				✓	✓
(vi) Data sharing	✓		✓	✓	✓		✓ (partially)	✓		
(vii) Loading Pipeline from File		✓			✓					
(viii) Dynamic loading of tasks						✓			✓	
(ix) Integration with ML					Cloud Machine Learning		Own ML API	Spark ML		
(x) Resume execution from a particular task		✓		✓	✓	✓		✓	✓	✓
(xi) Orchestration GUI			✓		✓				✓	
(xii) Open Source	✓	✓	✓	✓	✓		✓	✓	✓	✓
(xiii) Language agnosticism	✓	✓			✓	✓				
(xiv) Resource manager	✓			✓						
(xv) Instance invalidation										
(xvi) Last instance notification										
(xvii) Developer mode										

the pipeline and data burst are the same when resuming the execution of the pipeline. The storage of instances should be completely transparent for developers and carried out as quickly as possible to reduce the time requirements for data analysis.

Finally, there are minor features that improve the benefit of using a data pipelining framework including, but not limited to, the existence of a GUI application to support the orchestration of tasks (*xi*), its availability as open source (*xii*), language agnosticism (i.e. the possibility of implementing tasks in any programming language) (xiii), or the inclusion of resource management utilities (*xiv*). These utilities (for resource management) could allow assessing the resources (RAM –Random Access Memory–) required for pre-processing data and/or

their reservation in cloud environments (as COMPSs do), or simply limit the usage of CPU –Central Processing Unit– /RAM (as Cromwell).

As we can deduct from the current analysis, current software implementing pipelining strategies have important limitations that hamper their application to address the pre-processing of data. This fact suggests the need to develop a pipelining framework and implement all the features analysed in this study. Furthermore, we have also identified several interesting features that have not been raised by any of the analysed frameworks. Particularly, we found the advances in the following directions to be suitable: (*xv*) instance invalidation, (*xvi*) last instance notification, and (*xvii*) developer mode.

The instance invalidation (*xv*) is the capability of discarding a data

instance when we detect its invalidity during the pipelining process (for instance when trying to download the contents of a tweet through its Twitter ID that has been previously removed). The invalidation of a data instance could be invoked in any task belonging to the pipeline and implies that it will not be further processed (no other task will be called with the instance). Despite the fact that this functionality seems to be very simple and intuitively required, most pipeline implementations do not incorporate it.

The main advantage of pipeline processing schemes is the processing of streaming data instances (for instance a Twitter stream). As shown in Fig. 1, before the beginning of the pipelining process (b), a set of available data instances are buffered, in the form of burst (a) to be processed when computational resources are available (after processing the last data-instances burst).
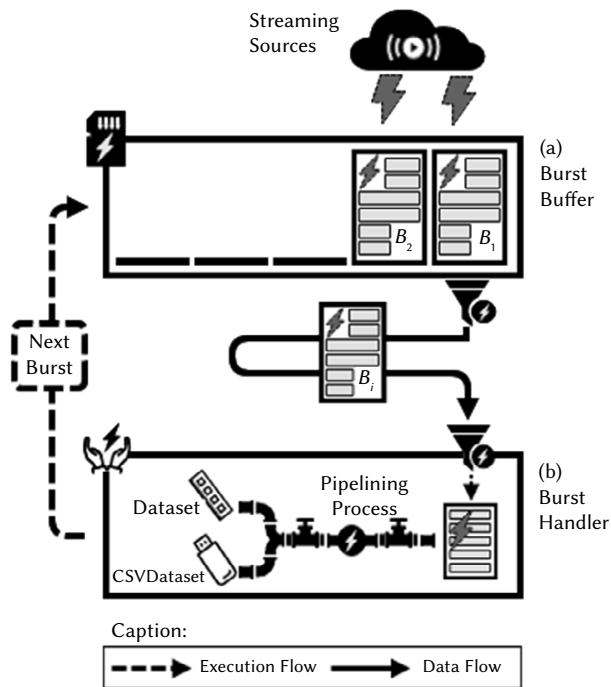

Fig. 1. General burst operation overview.

As result of the pipelining process, a dataset is generated (or grown) and stored in disk or main memory. In a task that saves the data to disk (or any other resource dependent task), the identification of the last data instance of a burst (*xvi*) would help to safely free resources (closing/flushing opened files, database connections, etc.). As a particular example, closing dataset files would make them available to be used for further analysis until the execution of the dataset saving task for the next burst.

As Fig. 1 shows, a framework would also provide facilities to store datasets (disk or memory) and let them grow both in number of rows and columns after processing each burst.

Finally, developers should be able to quickly test the proper operation of their tasks by using a collection of real data. However, to debug a certain task, all previous tasks defined in the orchestration should be executed before debugging the target task. In such a situation, providing mechanisms to achieve more efficiency when testing a new task adds a great value to pipelining frameworks (*xvii*). In particular, if the orchestration and input instances remain invariable for all test executions, the state of instances used as input for a task that is being debugged could be saved to disk (serialized) the first time they are computed, in order to reuse them for later executions.

Given the list of the most adequate features that a pipelining framework should provide, and starting from the simple pipelining support included in Mallet, we designed BDP4J, a new pipelining framework implementing most of the features that have been previously shown. Section III identifies the details of the implementation and use of the framework.

III. Introducing BDP4J

After analysing the pipelining frameworks, which provided a wide list of indispensable features for improving pipelining in enterprises, we developed BDP4J. BDP4J was inspired by the pipelining architecture included in Mallet software. Using this pipeline architecture as a base (and specifically the classes Instance, Pipe and SerialPipes), we transformed the name of the classes to accommodate them to the Java Code Conventions [41] and other non-explicit code rules (i.e. the name of interfaces and abstract classes that usually start with the prefix "Abstract" or "Default"). Specifically, Pipe, SerialPipes and Instance classes from the Mallet framework were transformed into AbstractPipe, SerialPipes and Instance BDP4J classes respectively. Using this architecture as a starting point, we implemented most features compiled in the previous section, achieving a product that is different from the Mallet pipeline implementation.

Similar to Mallet, BDP4J uses the Instance class (see Fig. 2) to represent a specific case (including raw data, properties, and the target solution) and solve a certain problem. As an example, in order to define an e-mail classifier, an Instance would represent the information of a specific message. The raw message contents would be included in source/data attributes; interesting features extracted through pipelining process would then be stored in props attribute, and the class of the message would be represented in target. Moreover, data attribute would be transformed through the pipelining process (from raw data to token list, feature vector…).

BDP4J tasks are represented as simple pipes (by implementing the Pipe interface or, even better, by extending AbstractPipe class). Moreover, the orchestration of pipes is defined through classes SerialPipes and ParallelPipes. These details together with the basic design of BDP4J are shown in Fig. 2. To facilitate readability, only relevant methods/attributes have been included in the classes shown in the diagram. Please refer to the Javadoc documentation (generated through the build process) to obtain a complete list.

As we can deduct from Fig. 2, BDP4J Pipe interface was created to allow developers to implement their tasks by extending, if necessary, from other classes. However, the use of AbstractPipe abstract class provides the implementation of all methods of Pipe interface, except for pipe (which stands for the specific work that should be done), to simplify the development of tasks. SerialPipes class keeps the same functionality that was originally provided in Mallet software while slightly changing the instance processing flow. Finally, BDP4J adds the support for parallel tasks execution through the class ParallelPipes. Configurator class is used for loading pipelines from files and is explained below.

Each pipe or task implemented in BDP4J must implement the methods getInputType and getOutputType to indicate the type of data included in the Instance before and after executing it. This information is used by SerialPipes and ParallelPipes to check data types in the orchestration (feature *i*). Additionally, these types should dynamically be checked with the Instance after executing the task.

When creating a task by extending the AbstractPipe class, the developer must call on its constructor (through super) to specify "Always Before" (alwaysBeforeDeps attribute) and "Not After" (notAfterDeps attribute) task execution constraints (feature *ii*). "Always
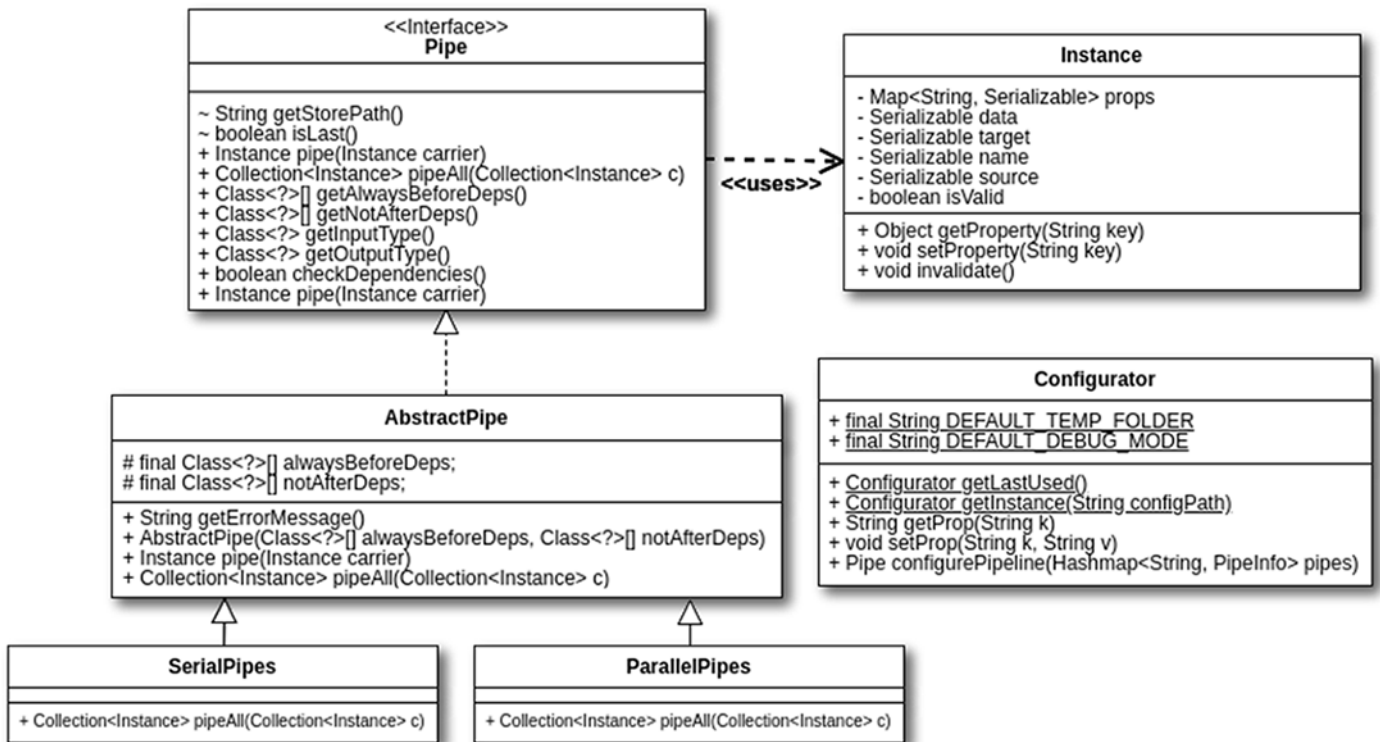
Fig. 2. Main classes comprising BDP4J.

Before" constraints indicate which tasks must be executed before the current one (for instance, the identification of the text language should probably be done before expanding abbreviations and/or slang terms to select the appropriate dictionary). Moreover, "Not After" dependencies indicate the tasks that cannot be executed after the execution of the current one (for instance, the recognition of acronyms should not be run after changing text to lowercase or removing punctuation marks). checkDependencies and getErrorMessage methods included in AbstractPipe class allow checking these constraints and obtaining information about the specific errors found.

BDP4J has divided the tasks in the following categories (feature *iii*): PropertyComputingPipe used only for computing instance properties; TeePipe used for storing instances when needed by user; TargetAssigningPipe used for assigning the real target class on classification; and TransformationPipe, which transforms the instance data. These task categories allow including additional constraints on tasks. Hence, the input and output data types of a task (getInputType and getOutputType) should be the same for all tasks except for TransformationPipe, which could be different. Moreover, the number of instance properties should be increased after an instance is processed through a PropertyComputingPipe, and the target attribute of an Instance should not be null after the execution of a TargetAssigningPipe. Finally, the number of TargetAssigningPipe included in a pipeline must be zero or one. In the future, the classification of tasks could be used to automatically group the functionalities in GUI pipeline management tools.

Currently, the support of parallel processing (feature *iv*) schemes in BDP4J is limited to the use of ParallelPipes task orchestration class. When several tasks are marked to be executed in parallel, they are performed in separate threads. Although we believe that we can take advantage of load balancing clustering schemes when multiple computers are available (feature *v*), its support has not been implemented yet and should be carefully designed.

Moreover, in order to adequately support burst data processing and data sharing (feature *vi*), the original Mallet processing method for data instances was changed. Mallet instance processing model is based on processing one instance through all tasks included in the pipeline. However sometimes we need all instances included in a data burst to be processed by a certain task in order to fill the shared information (for instance, when a text mining task builds a dictionary that will be used by other tasks to create a dense vector representation or a standard CSV –Comma-Separated Values– file). To cope with these situations, BDP4J executes each task on all available instances of the burst before starting the execution of the next task. This behaviour was implemented in pipeAll methods of orchestration schemes (SerialPipes and ParallelPipes).

BDP4J is able to load pipeline orchestration from XML files (feature *vii*). This functionality is connected with the dynamic loading of tasks from *.jar files (feature *viii*) and cannot be used independently. In order to implement the first functionality, we took advantage of Document Object Model (DOM) Application Programming Interface (API). Configurator class (see Fig. 2) provides the functionality of loading the pipeline configuration using DOM API. The XML file should contain the configuration/general/pluginsFolder parameter, which allows defining the directory where the *.jar files containing task definitions are located. The dynamic loading of tasks from jar files was implemented through the default Java service-provider loading facility (java.util.ServiceLoader<S> included in Java 8). Using this facility, BDP4J searches for Pipe implementations included in the location specified by pluginsFolder parameter. To facilitate the use of standard Java service loader by avoiding the manual creation of file META-INF/services/org.bdp4j.pipe.Pipe, all classes implementing tasks can be annotated with the @AutoService(Pipe.class) [42]. Fig. 3 shows how the orchestration of a pipeline can be easily represented in XML format (Fig. 3a) and loaded for its execution using BDP4J framework (Fig. 3b).

```
<?xml version="1.0"?>
<configuration>
 <!-- General properties -->
 <general>
   <samplesFolder>./samples</samplesFolder>
   <pluginsFolder>./plugins</pluginsFolder>
   <outputDir>./output</outputDir>
   <tempDir>./temp</tempDir>
 </general>
 <!-- the pipeline orchestration -->
 <pipeline resumable="yes" debug="no">
  <serialPipes>
    <pipe>
     <name>File2TargetAssignPipe</name>
    </pipe>
    <pipe>
     <name>File2StringPipe</name>
    </pipe>
    <pipe>
     <name>String2TokenArray</name>
    </pipe>
    <pipe>
     <name>TokenArray2FeatureVector</name>
    </pipe>
    <pipe>
     <name>
        GenerateFeatureVectorOutputPipe
     </name>
     <params>
      <pipeParameter>
        <name>outFile</name>
        <value>out.csv</value>
      </pipeParameter>
     </params>
    </pipe>
  </serialPipes>
 </pipeline>
</configuration>
```
(a)

```
/* Load XML configuration */
Configurator cfg =
   Configurator.getInstance("configuration.xml");

/*Load tasks from jar files*/
PipeProvider pipeProvider =
new PipeProvider(
     cfg.getProp(Configurator.PLUGINS_FOLDER)
   );
HashMap<String, PipeInfo> pipes =
   pipeProvider.getPipes();

/*Load the pipeline orchestration*/
Pipe p =
   Configurator.configurePipeline(pipes);
logger.info("orchest:" + p.toString() + "\n");

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

/*Load and pipe the current burst*/
ArrayList<Instance> burst = …
p.pipeAll(burst);
```
(b)

Fig. 3. XML orchestrating facilities included in BDP4J: a) XML structure used to define task orchestration (configuration.xml) and b) BDP4J source to load task orchestration.

The source code included in Fig. 3b (configurePipeline method) makes it possible to automatically instantiate as many SerialPipes and ParallelPipes as necessary to load the orchestration and configure tasks (because task configuration parameters are also included in XML, Fig. 3a).

A typical BDP4J pre-processing pipeline would contain one or more TeePipes. As stated before, the mission of TeePipes is to bring together the information of instances to generate datasets in memory (through using Dataset class shown in Fig. 4a) or in disk (through using CSVDatasetWriter class) and continue the execution of the rest of the pipeline. This mission fits with the dataset generation issue shown in Fig. 1. One of the most important features added to CSVDatasetWriter and Dataset classes is the possibility of dynamically growing the number of columns and rows. After processing some data bursts, they can be analysed by using external ML APIs (feature *ix*). Due to the popularity of Weka ML library, we have implemented a feature to transform a BDP4J dataset stored in memory (Dataset class) to Weka dataset (weka.core.Instances) through the getWekaDataset method. Additionally, the CSVDatasetReader class implements the loading of a CSV file to instantiate a BDP4J Dataset (Fig. 4b).
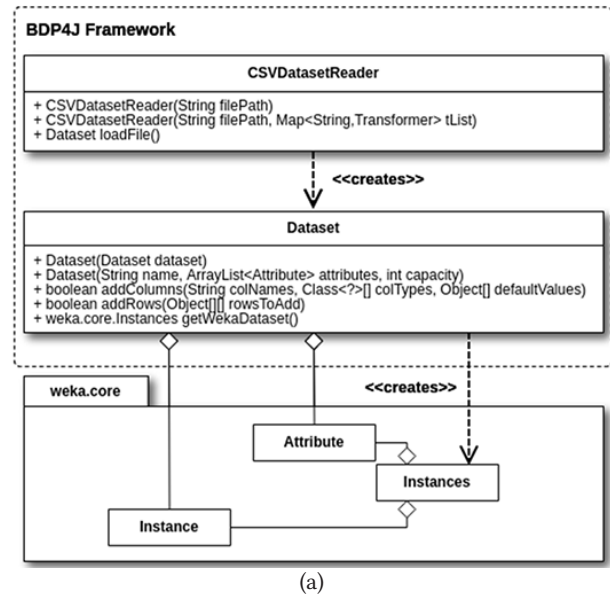
(a)

```
CSVDatasetReader csvdr =
 new CSVDatasetReader("out.csv");
Dataset ds =  csvdr.loadFile();
Instances wekaDS =
 ds.getWekaDataset();

wekaDS.deleteStringAttributes();
wekaDS.setClassIndex(
 wekaDS.numAttributes() - 1
);
int num = wekaDS.numInstances();
int start = (num * 80) / 100;
int end = num - start;

Instances trn =
 new Instances(wekaDS, 0, start);
Instances tst =
 new Instances(wekaDS, start, end);

try {
 Evaluation rfEval = new Evaluation(tst);
 RandomForest rf = new RandomForest();
 rf.buildClassifier(trn);
 rfEval.evaluateModel(rf, tst);
} catch (Exception ex) {}
```
(b)

Fig. 4. Weka integration facilities included in BDP4J: a) BDP4J and Weka interaction architecture and b) BDP4J and Weka interaction snippet.

As we can see from Fig. 4b, source lines highlighted in bold (first three sentences) allow the loading of Weka dataset, while the remaining lines show how we can take advantage of Weka API to run a Random Forest [43] classifier.

Furthermore, following the same orchestration behaviour of SerialPipes and ParallelPipes classes, we implemented ResumableSerialPipes and ResumableParallelPipes classes respectively. These classes support the resumption of the execution of a pipeline (feature *x*) that has been stopped for any reason (an application failure, accidental power down of computer...). The inner operation of the mentioned classes includes saving the state of instances after executing each task to allow resuming the pipeline from the last successfully executed task. One of the most relevant challenges to implement the resuming functionality was its compatibility with the data sharing between tasks. To this end, pipes can be marked by implementing SharedDataConsumer and/or SharedDataProducer interfaces. These interfaces include the methods readFromDisk and writeToDisk, respectively, to allow the programmer defining how to save and read the shared information to make it available when resuming the execution of a pipeline. These details are included in Fig. 5.
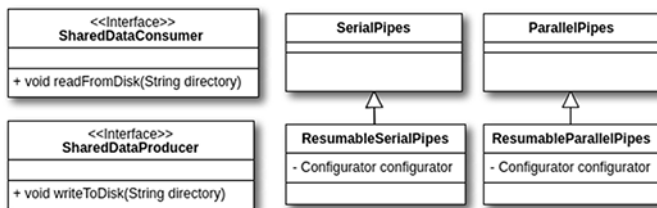

Fig. 5. BDP4J architecture details to support resuming function.

A resumable pipeline could be easily instantiated by defining the orchestration in source code (Fig. 4b) and replacing the use of SerialPipes and ParallelPipes by ResumableSerialPipes and ResumableParallelPipes, respectively. Additionally, the resuming behavior can be achieved with an XML file (Fig. 4a) using resumable and debug modifiers included in pipeline tag. When resumable is set to "yes" ("true" or "1"), the pipeline can be resumed. The debug modifier included in pipeline could be set to "no" ("false" or "0") only if the last complete result of a task remains stored on disk, or "yes" if all partial results are kept on disk (useful to manually drop steps and repeat tasks and only applicable to ResumableSerialPipes).

Finally a GUI application (feature *xi*) to orchestrate tasks was created using JGraphX [44] framework. The GUI is launched when using "gui" as the first parameter for the execution of the main class org.bdp4j.Main and allows visually defining, executing and saving a pipeline orchestration.

BDP4J is released as open source (feature *xii*). Its tasks can only be developed using Java technology (non-Java programming languages are not supported) (feature *xiii*). Moreover, BDP4J does not provide resource management utilities (feature *xiv*) and does not allow optimising the use of resources. In this sense, the limitation of RAM could be easily done by using the -Xmx parameter of Java Virtual Machine (JVM) and the number of cores via the GNU/Linux taskset utility. Therefore, the development of additional resource management functionalities (mainly computing RAM requirements) will be addressed as future works.

The instance invalidation (feature *xv*) was implemented by adding the invalidate method in the Instance class (which can be called from task definition) and the management of invalidated instances in the BDP4J orchestration subsystem (SerialPipes/ResumableSerialPipes and ParallelPipes/ResumableParallelPipes). Methods pipe and pipeAll implemented in the orchestration classes skip the processing of invalidated instances.

Additionally, the detection of the last data instance (feature *xvi*) is provided in the AbstractPipe class by providing a default implementation for isLast method. This implementation marks an instance *i* as last when the instance is processed alone through a call to pipe method or when *i* is the last valid instance of a collection of data (data burst) that is processed by calling the pipeAll method.

Given the support of resuming a pipeline from a certain position, we find it appropriate to take advantage of this feature in order to implement the debugging mode function (feature *xvii*). The debugging mode function allows developers of a task to avoid the processing of all previous required tasks (when they were previously executed) in order to reduce the time required to test whether the new task is operating properly. The code snippets (XML and Java) included in Fig. 6 provide a detailed description of how to take advantage of this functionality when the orchestration is defined in XML (Fig. 6a) or in Java (Fig. 6b). We highlighted in bold the instructions used to select the task that is being debugged.

As we can see from Fig. 6, the orchestrations defined in both columns are exactly the same. Hence, a task can be marked for debug when the orchestration is defined in XML or in Java, providing great flexibility for developers. In the case of using the XML to define the orchestration, the task (pipe) that is being debugged should include a debug tag, and the entire pipeline should be executed in resumable mode (resumable="yes"). Additionally, for source code orchestrations, the debug mode implies the use of ResumableSerialPipes and ResumableParallelPipes classes.

As shown in this section, BDP4J covers important limitations found in current pipelining software. The implemented features provide flexibility to developers, allowing them to analyse and pre-process data obtained from different sources in order to solve different kinds of problems. Subsection A shows a case study of our framework to address the pre-processing and classification of SMS –Short Message Service– spam messages.

A. Using BDP4J

In order to show the simplicity of using the BDP4J framework, we developed a case study to apply different text pre-processing techniques over SMS spam messages included in the SMS Spam Corpus v.0.1 [45]. The project has been publicly shared through GitHub (https://github.com/sing-group/bdp4j_sample). This project contains a collection of eight text pre-processing tasks (some of them trivial) implemented in the Java package org.bdp4j.sample.pipe.impl. The implemented pipes (sorted by the right execution order) make it possible to: create a property with the size of the text file (FileSizePipe); load the target attribute from the file (File2TargetAssigningPipe); load SMS text from file (File2StringPipe); create a property with the length of the SMS text (MeasureLengthPipe); add all information generated up to this step to a CSV file (GenerateStringOutputPipe); tokenize the text of each instance (String2TokenArray); create a feature vector for each instance (TokenArray2FeatureVector); and add all information to a CSV File (GenerateFeatureVectorOutputPipe).

The class org.bdp4j.sample.Main (main) creates instances from the files included in the samples directory. In detail during this process, data and source attributes are initialized with an instance of java.io.File representing the disk file containing SMS data. Instances are divided into three groups to simulate a burst operation. Moreover, this class also orchestrates a pipeline with all tasks defined in the project and use it to process the three bursts. After processing the instances, a NaïveBayes [46] classifier from Weka is executed over the generated data and the confusion matrix results (true positives, true negatives, false positives, and false negatives) are printed via system standard

```
<?xml version="1.0"?>
<configuration>
 <!-- General properties -->
 <general>
  <samplesFolder>./samples</samplesFolder>
  <pluginsFolder>./plugins</pluginsFolder>
  <outputDir>./output</outputDir>
  <tempDir>./temp</tempDir>
 </general>
 <!-- the pipeline orchestration -->
 <pipeline resumable="yes" debug="yes">
  <serialPipes>
   <pipe>
    <name>File2TargetAssignPipe</name>
   </pipe>
   <pipe>
    <name>File2StringPipe</name>
   </pipe>
   <pipe>
    <name>String2TokenArray</name>
    <debug/>
   </pipe>
   <pipe>
    <name>
      TokenArray2FeatureVector
    </name>
   </pipe>
   <pipe>
    <name>
      GenerateFeatureVectorOutputPipe
    </name>
    <params>
     <pipeParameter>
      <name>outFile</name>
      <value>out.csv</value>
     </pipeParameter>
    </params>
   </pipe>
  </serialPipes>
 </pipeline>
</configuration>
```
(a)

```
/* Debug String2TokenArray pipe */
String2TokenArray s2ta =
    new String2TokenArray();
s2ta.setDebugging(true);

/* Create the processing pipe */
AbstractPipe p =  new ResumableSerialPipes(
  new AbstractPipe[]{
   new File2TargetAssignPipe(),
   new File2StringPipe(),
   s2ta,
   new TokenArray2FeatureVector(),
   new GenerateFeatureVectorOutputPipe()
  }
 );

logger.info("orchest:" + p.toString() + "\n");

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

/*Load and pipe the current burst*/
ArrayList<Instance> burst = …
p.pipeAll(burst);
```
(b)

Fig. 6. BDP4J debug mode snippet: a) Enabling the debugging for a task in a XML orchestration and b) Enabling the debugging for a task in source code.

output. Additionally, two CSV files are generated on a disk (one per each tee pipe called output.csv and output2.csv) that contains the pre-processing results.

As can be easily found in the source code, two PropertyComputingPipe (FileSizePipe and MeasureLengthPipe), one TargetAssigningPipe (File2TargetAssignPipe), two TeePipe (GenerateFeatureVectorOutputPipe and GenerateStringOutputPipe) and three TransformationPipe (File2StringPipe, String2TokenArray, TokenArray2FeatureVector) tasks are provided as examples. Moreover, TokenArray2FeatureVector and GenerateFeatureVectorOutputPipe share a dictionary (org.bdp4j.sample.types.Dictionary) and hence, they implement the interfaces SharedDataProducer and SharedDataConsumer respectively to allow resuming and debugging functions.

Since Main class is provided and task implementations have been annotated with @AutoService annotation, the example could be executed by launching the generated jar file or by generating an alternative orchestration with BDP4J GUI.

The creation of the bdp4j_sample project provides a simple form for showing the use of the BDP4J pipelining framework to other researchers and developers. During the last months, we have been using BDP4J for pre-processing data in the research activities of our research group. This work has given us some conclusions for usage, summarized in subsection B.

B. Performance Evaluation

We carried out a performance evaluation of BDP4J and other available frameworks introduced in Section II. In order to accurately evaluate the performance of the analysed frameworks we use void tasks (those doing nothing). Fig. 7 represents the benchmarking protocol designed for the evaluation of the frameworks.
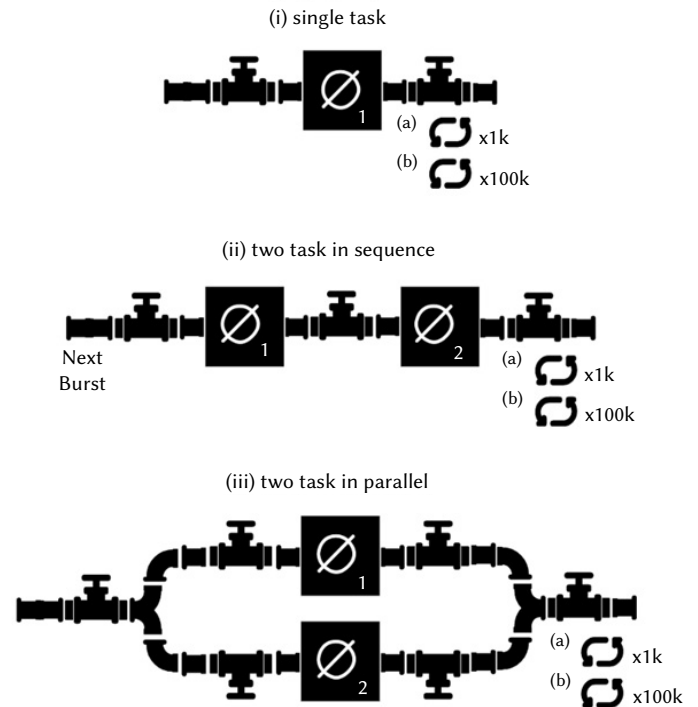

Fig. 7. Benchmarking design.

As shown in Fig. 7 our experimental protocol comprises three simple computational pipelines: (*i*) a pipeline composed exactly by one task, (*ii*) a pipeline that comprises the execution of two tasks in sequence and (*iii*) a pipeline that executes two parallel tasks. These pipelines

were executed 1,000 and 100,000 times in order to compare the impact of the checks made during the execution in the performance of the framework. The source code used for experimentation is available online [47]. The benchmark was executed in a computer with 64 gigabytes of RAM and an Intel i7-6700 processor with four cores (eight threads). Table II shows the evaluation results.

TABLE II. Performance Comparison of Analyzed Frameworks

Pipeline	(i)	(i)	(iii)
x 1k			
COMPI	2s, 554ms	3s, 266ms	3s, 149ms
Cromwell	50s, 076ms	1m, 24s, 113ms	1m, 28s, 917s
Drake	31s, 133ms	30s, 946ms	95s 90ms
Mallet	1.05ms	1.09ms	unsupported
MLPipeline	2s, 227ms	2s, 321ms	unsupported
BDP4J	1.03 ms	4.42ms	56.34ms
x 100k			
COMPI	48s 928ms	1m 37s 83ms	1m, 33s, 436ms
Cromwell	unsupported	unsupported	unsupported
Drake	51m, 53s	51m, 34s	2h, 38m 29s
Mallet	10 ms	9 ms	unsupported
MLPipeline	2s, 475ms	2s, 494ms	unsupported
BDP4J	12.93 ms	15.24 ms	801.77 ms

As shown in Table II, Drake (which is written in R but allows executing tasks written in Java) fairly achieves the worst performance while Mallet is the most efficient framework. However, the performances achieved in Mallet and BDP4J when executing pipelines (*i*) and (*ii*) are very similar (a few milliseconds). The increase of time required by BDP4J for the execution of pipelines containing tasks executed in parallel is caused by the need to clone the instances in order to alleviate the programmer of the issues arising from concurrent programming. The impact on performance of this decision is significative, however the time required for the development of tasks is considerably reduced. Additionally, Cromwell has some issues with memory when processing large pipelines (repeating 100000 times the pipeline is not possible with 64GB of RAM). Keeping in mind the benefits of our proposal, we really believe BDP4J is a reliable solution for analysing data using Java technology.

C. Learned Lessons and Main Outcomes

Our use of BDP4J for the development and execution of different pre-processing tasks over the last several months have led us to important findings. One of the most important findings is the great amount of human error that occurs during the development of applications. A significant number of these errors could be detected by using constraints, type checks, and dependencies implemented by BDP4J. People often wrongly think that constraints, type checks, and dependencies included in different programming models are merely time-consuming issues that negatively affect development because any IT professional can develop software with no errors. However, our experience has shown us that the existence of these elements really helps developers to avoid software errors and cause a low impact on the time required for defining pre-processing tasks.

Additionally, the use of debug mode has significantly sped up the development and debugging of tasks, given that the whole execution process is executed only once. In fact, when a pipeline is executed a second time, the task marked to be debugged will be the first executed. As long as the tasks that will be debugged can be quickly defined, the feature is very useful for developers.

Moreover, we have developed an extremely simple mechanism for defining tasks. When extending AbstractPipe class, only the methods getInputType, getOutputType and pipe (execute the task) should be defined. The BDP4J framework adds many sources to check constraints, types, and operation with no effort on the part of the developer. Most developers who tried BDP4J really appreciate the effort to minimize the source they should write for developing tasks.

Finally, we would like to highlight the use of Apache Maven as a software management and building tool that facilitates compilation, packaging, and execution tasks. This finding, together with those previous mentioned, allow us to understand that we are working in the right direction to define the BDP4J framework. The next section provides detailed conclusions and future work to improve this framework.

IV. Conclusions and Future Work

During the last years, the exploitation of data has been increasingly used as a method to solve many problems and understand the inner details of real life. The main challenge to address in data exploitation is the existence of effective programming models able to take advantage of available data. In the current context, popular developing paradigms such as MapReduce are being abandoned while pipelining schemes are becoming more and more popular. Our intention in this work was to design and provide new and interesting functionalities for pipelining software. Particularly, pipelining frameworks should provide simple ways for defining tasks and incorporate mechanisms for early detecting errors.

We designed functionalities to avoid the processing limitations found in MapReduce schemes and other pipelining frameworks. Particularly, we introduce the burst-processing scheme that can be easily applied to the pre-processing of streaming sources of data (such as Twitter streams), while still allowing for the old batch processing schemes. Burst processing combined with the detection of the last data instance in any task (for flushing/closing files) makes it possible to analyse data after the execution of some bursts, thus avoiding the need for processing all input data before they are analysed.

Finally, it should be noted that a great amount of technology and new processing techniques have been made available to data mining developers. As main contributions of this work (processing techniques), we would highlight the number of techniques used to detect errors (constraints, type checks and task dependencies), the resuming (and debug) capabilities of pipelines, and the instance invalidation (never seen before). These technologies can also be applied to improve other frameworks (especially those developed in other programming languages).

Future work comprises the design of execution schemes to distribute computational requirements in a cluster of computers. To this end, we believe that a careful examination of available load balancing schemes would provide effective load distribution mechanisms to implement this functionality. Additionally, this feature will be complemented with the assessment of the amount of RAM necessary to execute a pipeline over a burst with certain size (resource management).

Acknowledgment

References

[1] I. M. Dunham, "Big Data: A Revolution That Will Transform How We Live, Work, and Think", *The AAG Review of Books*, vol. 3, no. 1, pp. 19–21, Jan. 2015.

[2] Q. Qi, F. Tao, "Digital Twin and Big Data Towards Smart Manufacturing and Industry 4.0: 360 Degree Comparison", *IEEE Access*, vol. 6, pp. 3585–3593, 2018.

[3] V. Kalavri, V. Vlassov, "MapReduce: Limitations, Optimizations and Open Issues," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, pp. 1031–1038.

[4] D. Miner, A. Shook, *Mapreduce Design Patterns Building Effective Algorithms and Analytics for Hadoop and Other Systems*. Oreilly & Associates Inc, 2012.

[5] Apache Software Foundation, "Apache Hadoop." 2018.

[6] Amazon, "Amazon Elastic MapReduce." 2019.

[7] Disco Project, "DisCo MapReduce." 2014.

[8] S. Papadimitriou, J. Sun, "DisCo: Distributed Co-Clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 512–521.

[9] Apache Software Foundation, "Apache Spark - Unified Analytics Engine for Big Data." 2018.

[10] J. Zeng, B. Plale, "Data Pipeline in MapReduce," in *2013 IEEE 9th International Conference on e-Science*, 2013, pp. 164–171.

[11] P. O'Donovan, K. Leahy, K. Bruton, D. T. J. O'Sullivan, "An Industrial Big Data Pipeline for Data-Driven Analytics Maintenance Applications in Large-Scale Smart Manufacturing Facilities", *Journal of Big Data*, vol. 2, no. 1, p. 25, Dec. 2015.

[12] P. Di Tommaso, "Awesome Pipeline: A Curated List of Awesome Pipeline Toolkits." 2018.

[13] Amazon, "AWS Data Pipeline." 2019.

[14] Snaplogic, "SnapLogic Intelligent Integration Platform," 2019. [Online]. Available: https://www.snaplogic.com/products/intelligent-integration-platform. [Accessed: 21-Jun-2020].

[15] Alooma, "Alooma Enterprise Data Pipeline." 2019.

[16] S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir, "Optimization of Data-Intensive Workflows in Stream-Based Data Processing Models", *The Journal of Supercomputing*, vol. 73, no. 9, pp. 3901–3923, Sep. 2017.

[17] G. Kougka, A. Gounaris, A. Simitsis, "The Many Faces of Data-Centric Workflow Optimization: A Survey", *International Journal of Data Science and Analytics*, vol. 6, no. 2, pp. 81–107, Sep. 2018.

[18] J. Leipzig, "A Review of Bioinformatic Pipeline Frameworks", *Briefings in Bioinformatics*, p. bbw020, Mar. 2016.

[19] P. A. Ewels *et al.*, "The Nf-Core Framework for Community-Curated Bioinformatics Pipelines", *Nature Biotechnology*, vol. 38, no. 3, pp. 276–278, Mar. 2020.

[20] M. Bourgey *et al.*, "GenPipes: An Open-Source Framework for Distributed and Scalable Genomic Analyses", *GigaScience*, vol. 8, no. 6, Jun. 2019.

[21] D. Swersky, "Top 43 Programming Languages: When and How to Use Them," 2018. [Online]. Available: https://raygun.com/blog/programming-languages/. [Accessed: 21-Jun-2020].

[22] E. Frank, M. A. Hall, I. H. Witte, *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques,"* Fourth Edi. Morgan Kaufmann Publishers Inc., 2016.

[23] A. Moro, R. Navigli, "Babelfy." 2014.

[24] Y. Lage, J. R. Méndez, M. Novo-Lourés, "Big Data Pre-Processing For Java (BDP4J)." 2018.

[25] F. Lordan *et al.*, "ServiceSs: An Interoperable Programming Framework for the Cloud", *Journal of Grid Computing*, vol. 12, no. 1, pp. 67–91, Mar. 2014.

[26] R. M. Badia *et al.*, "COMP Superscalar: An Interoperable Programming Framework", *SoftwareX*, vol. 3–4, pp. 32–36, Dec. 2015.

[27] T. Burdett, N. Kurbatova, D. Hastings, Emma Faulconbridge, Adam Mapleson, R. Davey, "Conan2 Lightweight Workflow Manager." 2019.

[28] J. Bingham, S. Davis, N. Deflaux, "Dockerflow: A Workflow Runner That Uses Dataflow to Run a Series of Tasks in Docker with the Pipelines API," 2017. [Online]. Available: https://github.com/googlegenomics/dockerflow.

[29] Google Inc, "Cloud Dataflow Documentation," 2019. [Online]. Available: https://cloud.google.com/dataflow/docs/?hl=es-419. [Accessed: 21-Jun-2020].

[30] Netflix, "Suro: Netflix Distributed Data Pipeline." 2012.

[31] J. M. Wozniak, M. Wilde, I. T. Foster, "Language Features for Scalable Distributed-Memory Dataflow Computing," in *Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*, 2014, pp. 50–53.

[32] J. M. Wozniak, M. Wilde, I. T. Foster, "Swift Tutorial for Running on Localhost," 2014. [Online]. Available: http://swift-lang.org/tutorials/localhost/tutorial.html. [Accessed: 21-Jun-2019].

[33] M. Hategan *et al.*, "Swift-Lang, Swift-K," 2019. [Online]. Available: https://github.com/swift-lang/swift-k. [Accessed: 21-Jun-2019].

[34] H. López-Fernández, O. Graña-Castro, A. Nogueira-Rodríguez, M. Reboiro-Jato, D. Glez-Peña, "Compi: A Framework for Portable and Reproducible Pipelines", *PeerJ Computer Science*, vol. 7, p. e593, Jun. 2021.

[35] Broad Institute, "Cromwell: Workflow Management System Geared towards Scientific Workflows." 2019.

[36] A. Malloy *et al.*, "Drake." 2015.

[37] S. Fong, Y. Zhuang, J. Li, R. Khoury, "Sentiment Analysis of Online News Using MALLET," in *2013 International Symposium on Computational and Business Intelligence*, 2013, pp. 301–304.

[38] A. K. McCallum, "MALLET: A Machine Learning for Language Toolkit." 2002.

[39] Apache Software Foundation, "Apache Spark: ML Pipelines," 2018. [Online]. Available: https://spark.apache.org/docs/latest/ml-pipeline.html. [Accessed: 21-Jun-2020].

[40] A. Liu, *Apache Spark Machine Learning Blueprints*, First. Birmingham, UK: PACKT Publishing Ltd., 2016.

[41] D. S. F. Long, D. Mohindra, R.C. Seacord, D.F. Sutherland, "Svoboda, Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs", *Addison-Wesley*, 2013.

[42] Google LLC, "AutoService: A Collection of Source Code Generatos for Java." 2013.

[43] L. Breiman, "Random Forests", *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[44] M. R. G. Alder, D. Benson, "Jgraph/Jgraphx." 2014.

[45] E. P. S. J.M. Gómez Hidalgo, "SMS Spam Corpus v.0.1," 2011.

[46] A. Pérez, P. Larrañaga, I. Inza, "Bayesian Classifiers Based on Kernel Density Estimation: Flexible Classifiers", *International Journal of Approximate Reasoning*, vol. 50, no. 2, pp. 341–362, Feb. 2009.

[47] M. Novo-Lourés, Y. Lage, R. Pavón, R. Laza, D. Ruano-Ordás, J. R. Mendez, "Benchmarking Code for Pipeline-Based Frameworks." 2021.

María Novo

She was born in Galicia (Spain) in 1983. She graduated from the University of Vigo on Computer Science Engineering and she holds an MSc in E-Commerce from the University of Salamanca. At present, she is a researcher on the SING research group, where she is developing her PhD in spam filtering using Big Data and Machine Learning techniques.

Yeray Lage

He was born in Galicia (Spain) in 1995. He recently finished his Computer He was born in Galicia (Spain) in 1995. He recently finished his Computer Science Degree making his end-of-degree project collaborating with SING group. Currently, he is working as Full-Stack Developer in a company specialized in Web Development.

Reyes Pavón

She is a PhD from the University of Vigo and a member of SING research group. Currently, she is Associate Professor in the Computer Science Department of the University of Vigo. Her main research interests are Artificial Intelligence, classification methods and spam filtering. She is joint author of several articles published in international prestigious journals.

Rosalía Laza

She received a PhD in Computer Science from the University of Vigo in 2003. At present, she is member of SING research group and she is Associate Professor in the Computer Science Department of the University of Vigo. She is co-author of several books, book chapters, and articles published in international prestigious journals; most of these present practical and theoretical achievements of case-based reasoning systems, intelligent artificial paradigm, classification methods and spam filtering.

David Ruano-Ordás

He was born in Galicia (Spain) in 1985 and received his PhD in Computer Science from the University of Vigo (Spain) in 2015. He is computer science engineer with high experience on Linux administration and software development under the ANSI/C standard. He collaborates as researcher with the SING group belonging to the University of Vigo. Regarding to the research experience he is mainly focused in the Artificial Intelligence area (automatic learning, or evolutionary algorithms) applied to spam filtering and drugs-discovery domain. Finally, he has participated in some regional research projects and has been co-author of several articles published in journals belonging recognized editorials such as Hindawi, Springer or Elsevier. (http://www.drordas.info/).

José R. Méndez

He was born in Galicia (Spain) in 1977. Currently, he works at the computer science department of University of Vigo. He worked as a system administrator, software developer, and IT (Information Technology) consultant in civil services and industry during 10 years. He is an active researcher belonging to SING group and, although collaborates in different applications machine learning, his main interests are the development and improvement of anti-spam filters. (http://sing-group.org/).