The bdpar Package: Big Data Pipelining Architecture for R

by Miguel Ferreiro-Díaz, Tomás R. Cotos-Yáñez and David Ruano-Ordás

Abstract In the last years, big data has become a useful paradigm for taking advantage of multiple sources to find relevant knowledge in real domains (such as the design of personalized marketing campaigns or helping to palliate the effects of several mortal diseases). Big data programming tools and methods have evolved over time from a MapReduce to a pipeline-based archetype. Concretely the use of pipelining schemes has become the most reliable way of processing and analysing large amounts of data. To this end, this work introduces **bdpar**, a new highly customizable pipeline-based framework (using the OOP paradigm provided by R6 package) able to execute multiple pre-processing tasks over heterogeneous data sources. Moreover, to increase the flexibility and performance, **bdpar** provides helpful features such as (i) the definition of a novel object-based pipe operator (`%>|%`), (ii) the ability to easily design and deploy new (and customized) input data parsers, tasks and pipelines, (iii) only-once execution which avoids the execution of previously processed information (instances) guaranteeing that only new both input data and pipelines are executed, (iv) the capability to perform serial or parallel operations according to the user needs, (v) the inclusion of a debugging mechanism which allows users to check the status of each instance (and find possible errors) throughout the process.

Introduction

Social networks and instant messaging applications have arguably become an essential part of the human experience. In fact, nowadays more than 60% of the population from industrialized countries use these mechanisms to communicate or share information. This phenomenon emerged due to (i) the declining costs of computers and storage systems by a factor of more than 200 (Engineering, 1984), (ii) an exponential increase in processing speed and computer hardware capabilities (Iansiti and Khansa, 1995), (iii) the emergence of high-throughput and fully-available communication networks (Dorogovtsev and Mendes, 2013), and (iv) certain human needs such as keeping interconnected and having permanent access to the data (Kabeer, 2005).

This scenario has promoted an exponential growth in the amount of data generated and stored in the last decade. Concretely, the latest reports from 2018 showed that around 2.16EB (exabytes) of data are created every day (Domo-Data, 2019; VCloud, 2019), and trends are showing that the growth of available information is four times higher than the world economy (VCloud, 2019). Indeed, 90% of the total world data have been created in the last two years alone (IBM, 2019).

In addition to the availability of unlimited sources and tools to generate, exchange and handle information, the lack of a standardised way of representing data, has led to a massive increase in unstructured information. In fact, approximately 80% of the existing data is unstructured (VCloud, 2019; IBM, 2019). The data obtained from a single source are usually insufficient to carry out a suitable decision making-process. However, the ability to take advantage of the combination of data from multiple (and unstructured) sources requires the execution of pre-processing operations that guarantee a unified data format. The need for facilitating the management and exploitation of vast amounts of heterogeneous data (in terms of data types and formats) within a reasonable elapsed time and cost-effective manner, led to the emergence of the Big Data era (Mervis, 2012; Labrinidis and Jagadish, 2012).

Big Data is an abstract concept used to refer to the use of new programming paradigms able to handle large volumes of information and the execution of data mining tasks by taking advantage of parallel programming schemes over large computer clusters (IBM et al., 2011; Brown et al., 2011). In this context, MapReduce (Wu et al., 2014; Miner and Shook, 2012) is the most popular programming model to develop, execute and deploy Big Data analyses on large clusters. However, its batch-processing nature forces uploading data to the system (cluster) every time is analysed, even when the input data has been previously utilized. This requirement (i) makes this programming paradigm unsuitable when leading with real-time streaming sources and (ii) avoids achieving full use of the computational capabilities and resources since clusters are idle while the data is being loaded. In order to solve these limitations, the utilization of pipelining schemes for big data processing was recently introduced Di Tommaso (2019). This concept (extrapolated from the electronic domain) is focused on dividing the whole data analysis process into a set of computationally simple tasks (O'Donovan et al., 2015) whereby the required information for each task is handled exclusively, which avoids the (pre)loading of unnecessary information. This advantage has prompted the emergence of multiple enterprises

offering cloud pipeline-based data analysis services such as BDB Solutions for Big Data (Solutions, 2019), AWS Amazon Data Pipeline (Amazon, 2019) or Google Cloud Dataflow (Google, 2019). These services allow users to build highly customized pipelines using a simple graphical interface, even if their technical skills are basic. Despite the great advantages of these services, their cloud-oriented nature causes customers to be reluctant to use them due to (i) the full control of the pipeline by the company offering the service, (ii) data privacy and security concerns (since information is executed in a foreign infrastructure), and (iii) the difficulty of assessing and calculating the cost of computational resources required to process data through the defined pipeline.

In order to cope with these problems, several customers decided to change the third-party cloud-based service to a proprietary solution by designing and implementing their own pipelining tools. Meanwhile, multiple open-source offline Big Data Pipeline frameworks emerged from the academic community to make this new paradigm available to everybody (Di Tommaso, 2019). However, as can be observed from the list shown in (Di Tommaso, 2019), despite the great number of available solutions, the majority are developed using Java language (>90%) while only a few belong to the R ecosystems. Among these, two packages should be mentioned, **repo** (Napolitano, 2020) and **drake** (Landau, 2018). The former one is a data-centred pipeline focused on solving bioinformatics data problems (specific-purpose application). Conversely, the latter is a generic pipeline tool implemented focused on the GNU Make utility. As can be seen, despite both applications are focused on the same concept (pipelines), the target, implementation schema, and provided functionalities are quite divergent. In addition, we found some important issues that are not addressed by the actual pipelines tools such as (i) lack of a pure object-oriented (OO) implementation to facilitate the use and reduce the learning curve for people coming from object-oriented environments, (ii) the absence of an application based on the pipelining concept used by the well-known **magrittr** package (Bache and Wickham, 2020) (focused on UNIX pipes), (iii) the use of a black-box implementation which hampers users to easily trace and debug both code and the intermediate results.

This scenario, motivated us to design and implement **bdpar**, a framework capable of unifying and preprocessing heterogeneous data through the development and execution of customizable pipelines. To this end, our package allows automatizing the management of a large amount of information by segmenting data into a sequence of simple and indivisible tasks (divide and conquer paradigm). Specifically, **bdpar** allows to (i) use or develop content extractors (such as SMS, or e-mail parsers), (ii) use and implement new preprocessing tasks (pipes), (iii) define customize pipelines (set of tasks) to achieve the desired (structured) output, (iv) visualize the intermediate results achieved by each instance after being processed by the tasks comprising the pipeline (white-box implementation), (v) prevent the re-execution of previously computed instances and tasks, and finally (vi) execute the pipeline following both a sequential or parallel paradigm.

This paper provides a full description of the main functionalities and resources of the **bdpar** package. The current version is 3.0.0 and an updated list (with the whole collection of resources) is available in the vignette document and the reference manual. The following section provides a complete description of the package structure and functionalities. Then, the use of the package is described and finally, an illustrative case study is provided.

Package structure and operation

In order to exploit the main advantages and strengths of the object-oriented paradigm (such as maintainability, modularity or inheritance), the **bdpar** package was fully developed using `R6` classes (package **R6** (Chang, 2019b)). Particularly bdpar was implemented using R6 classes (package R6 (Chang, 2019)) due to its high performance and ease of use when compared with other alternatives (such as S3 or S4) (Chang (2019a); Wickham (2019)). From an operational point of view, `R6` classes implemented in **bdpar** package are divided into three different categories: (i) data extraction functionalities; (ii) pipe-based operations; and finally (iii) **bdpar** framework configuration utilities.

The first category (data extraction methods) comprises all methods responsible automatically detect the format and parse contents according to the inner structure of data gathered from input sources. The second category encapsulates the functionality of extracting features from the parsed information. By default, the **bdpar** framework provides a complete data pre-processing flow comprising 18 different tasks. Additionally, **bdpar** allows for the easy creation of new customized data flows by combining multiple tasks (object-based pipes). Finally, the third category of methods allows handling the configuration parameters needed for the proper operation of both the **bdpar** framework and some tasks using third-party functions (such as credentials for **rtweet** (Kearney, 2019) or **tuber** (Sood, 2019)).

In order to improve the readability of the code and facilitate the comprehension of each implemented class, a naming convention was adopted. Methods included in the data extraction category are labelled using `Extractor` as a prefix, followed by the type (or structure) of the input source (e.g.

`ExtractorEml` and `ExtractorSms` methods are able to parse text contents from e-mails and SMS, respectively). Finally, pipe-based functionalities are named using the operation name followed by the suffix `Pipe` (e.g. `ToLowerPipe` and `FindHashtagPipe` are tasks designed to convert text characters to lowercase and detect Twitter hashtags from textual contents, respectively).

As stated before, the **bdpar** framework is focused on designing, implementing and deploying customized processing flows for the big data domain. In order to handle the information obtained from the input sources, the package uses a specific structure called `Instance` which is responsible for storing the properties extracted by each pipe comprising the processing flow. To provide an insightful view of our **bdpar** framework, Figure 1 provides a graphical representation of its inner operation, which is divided into two main stages: (i) data loading and (ii) pipeline executing.

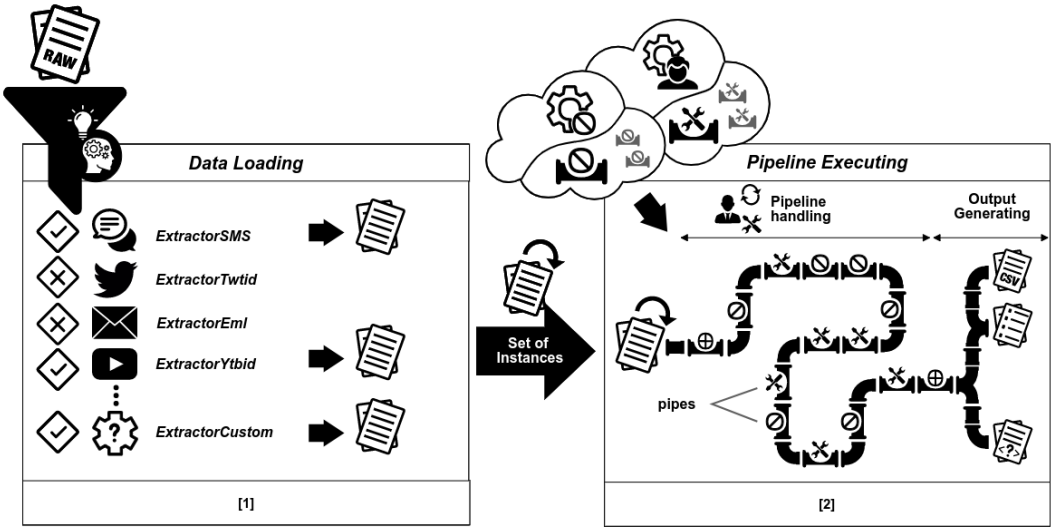

Figure 1: Description of **bdpar** two-stage execution process.

As shown in Figure 1, the first stage comprises the loading of the required extractors according to the type of input data. By default, **bdpar** provides four different types of extractors: (i) `ExtractorSMS` is able to extract the textual contents exchanged through Short Message Service (SMS); (ii) `ExtractorTwtid` is capable to obtain text from Twitter entries (tweets); (iii) `ExtractorEML` can be used to gathering raw content from the body of e-mail messages; and finally (iv) `ExtractorYtbid` allows extracting the comments published on the YouTube platform. Additionally, to increase the compatibility of **bdpar** with other data formats, the framework allows for the easy design and deployment of new customized extractors (by using a simple OOP inheritance relation).

Once the content is successfully extracted from the raw sources, the second stage is automatically initiated by **bdpar**. This stage comprises the execution of two steps: (i) pipeline handling; and (ii) output generating. The former is in charge of performing unified data processing by executing a specific set of pipes (also named pipeline) over the previously extracted content. It should be noted that each pipe included in the pipeline is represented as an object (inherited from `GenericPipe` class) responsible for performing a specific operation (task) over the input data. The second step (called output generating) transforms the preprocessed data into a specific output format (e.g. into a CSV structure). Moreover, **bdpar** allows users to develop new specific output-generation methods to achieve the desired output.

As can be realized from the pipeline handling stage shown in Figure 1, the pipe-based structure provides great flexibility and versatility to users since it allows users to easily (i) modify existing pipelines by adding or removing pipes, (ii) develop new pipes implementing additional tasks, or (iii) design and deploy new customized pipelines. To assist users in the creation and deployment of new pipelines, **bdpar** provides a set of 18 combinable pipes implementing basic preprocessing tasks for text sources. As can be seen in Table 1, tasks included in **bdpar** are divided into two different categories: (i) transformer tasks which are able to perform operations that successively alters the original content (such as lowercase conversion, or emoticon finding), and (ii) maintainers which are responsible for executing operations that do not affect the current content (such as storing the extension of the input data).

Pipe name	Pipe type	Name of computed property	Description
GuessDatePipe	Transformer	"date"	Obtains the date and time based on the type and structure of the input data.
File2Pipe	Transformer	"source"	Obtains the source based on the type and structure of the input data.
FindUserNamePipe	Transformer	"userName"	Detects and extracts usernames from textual sources.
FindHashtagPipe	Transformer	"hashtag"	Detects and obtains hashtags from input data.
FindUrlPipe	Transformer	"URLs"	Uses regular expressions to find URLs in text.
FindEmoticonPipe	Transformer	"emoticon"	Identifies and extracts emoticons from textual sources.
FindEmojiPipe	Transformer	"Emojis"	Transforms emojis to its textual representation.
GuessLanguagePipe	Transformer	"language"	Tries to guess the language of a specific text.
ContractionPipe	Transformer	"contractions"	Transforms previously detected contractions.
AbbreviationPipe	Transformer	"abbreviation"	Expands detected abbreviations.
SlangPipe	Transformer	"langpropname"	Identifies slang words to its corresponding formal speech.
ToLowerCasePipe	Transformer	–	Converts the input source to lowercase characters.
InterjectionPipe	Transformer	"interjection"	Detects and extracts interjections from textual sources.
StopWordPipe	Transformer	"stopWord"	Recognizes and obtains stop words from textual sources.
TargetAssigningPipe	Maintainer	"target"	Identifies the target class of the data.
StoreFileExtPipe	Maintainer	"extension"	Guess the extension of the input data.
MeasureLengthPipe	Maintainer	"length"	Computes the length of a given text.
TeeCSVPipe	Maintainer	–	Transforms the final result into a CSV format file.

Table 1: Pipes provided by **bdpar** framework.

Additionally, each pipe provides a property name field where the computed property will be stored. To increase flexibility property names can be specified by users or leave it by default (see names described in Table 1). Finally, pipes definition in Table 1 were sorted to match the execution order defined in the pipeline included by default in **bdpar** (named as `DefaultPipeline`)

Moreover, to increase the reliability of the pipeline, **bdpar** allows defining the execution order of each task comprising the specified pipeline. During the definition of a pipeline, we should specifically take into account the possible interdependence between pipes (e.g. language-dependent tasks should be executed after `GuessLanguagePipe`). To solve this situation and ensure the proper creation and execution of the pipelining process, **bdpar** provides a pipe-orchestration system. This mechanism is automatically invoked when the pipeline is executed and traverses all tasks comprising the pipeline to evaluate two types of interdependencies: (i) always-before (or 'a priori dependence') and (ii) not-after

(or 'a posteriori dependence'). The first constraint is used when a specific pipe requires the previous execution of other tasks to ensure its proper operation (for instance, `AbbreviationPipe` needs to know the text language so it should not be executed before `GuessLanguagePipe`). Conversely, the second type of restriction is used to indicate the tasks that cannot be started after the execution of the current one (for instance, the recognition of contractions should not be run after changing text characters to lowercase or removing punctuation marks). Both restrictions are automatically managed through the `checkCompatibility` method included in the `Instance` class.

Furthermore, in order to ensure the proper execution of tasks implemented through `R6` classes within the pipeline, **bdpar** implements an object-oriented customized operator (denoted as `%>|%`) inspired by the implementation of the primitive forward-pipe operator (`%>%`) provided by the **magrittr** package (Bache and Wickham, 2020). Particularly, the execution of this new operator implies some inner operations such as (i) discarding an Instance (left-side operand) whenever an error occurs during the data processing flow, (ii) automatically manage pipes dependencies between pipes, (iii) simplifying the invocation of the associated pipe task (right-side operand) by hiding its explicit call, (iv) facilitate debugging issues by showing log messages with different levels of granularity, (v) the ability to display the intermediate computation results of each instance throughout the whole preprocessing flow and (vi) the capability to avoid the re-execution of previously processed pipelines. To this end, the operator transparently calls the `pipe` method defined in the pipe object. These functionalities allow improving the processing capabilities (in terms of speed, performance and usability) of the application by preventing the problems derived from the (potential) existence of errors during the pipelining process and shorten pipelining definition by taking advantage of the customized operator capabilities. To increase the customization capabilities **bdpar** allows easy development and deployment of new user-defined pipelines. To ensure full compatibility of new user-defined pipelines **bdpar** provides a reference class called `GenericPipeline`. Additionally, to simplify the use of the framework, **bdpar** provides a predefined pipeline (named `DefaultPipeline`) containing all the pipes included in Table 1.

Finally, once all `Instance` objects are processed, the output generation stage starts. As can be seen from Figure 1, this stage is responsible for storing the results achieved after executing the pipeline process over each (valid) `Instance`. Although this stage allows the use of customized storage and output-representation methods (implemented by user), **bdpar** provides two methods able to (i) save the achieved output into an external CSV file (using `TeeCSVPipe` pipe) or (ii) internally store in memory a set of preprocessed `Instance` objects (default output).

In order to exemplify the structure and operation of an object-based pipeline in **bdpar**, we have included below a code snippet comprising 13 different text processing tasks (implemented as pipe objects).

```
instance %>|%
  TargetAssigningPipe$new() %>|% StoreFileExtPipe$new() %>|%
  GuessDatePipe$new() %>|% File2Pipe$new() %>|%
  MeasureLengthPipe$new("length_before_cleaning_text") %>|% FindUrlPipe$new() %>|%
  FindEmojiPipe$new() %>|% GuessLanguagePipe$new() %>|%
  SlangPipe$new() %>|% ToLowerCasePipe$new() %>|%
  InterjectionPipe$new() %>|% StopWordPipe$new() %>|%
  MeasureLengthPipe$new("length_after_cleaning_text") %>|% TeeCSVPipe$new()
```

To facilitate the understanding of the pipelining process, the code included above assumes that each input data has been successfully loaded and stored in an `Instance` object (denoted as `instance`). As can be seen from the code snippet, each `instance` is processed through all the tasks comprising this pipeline. Specifically, the first 13 ones performs different preprocessing operations over each `instance`, while the latest one stores the achieved results into a CSV file.

Using bdpar package

The package can be installed and attached as described in the code included below (please refer to the 'README' file to access the latest and development versions).

```
install.packages("bdpar")
library(bdpar)
```

Please note that the core functionalities of **bdpar** require the previous installation of six R packages (described in the '`Imports`' field included in the 'DESCRIPTION' file). In addition, some optional tasks (mainly belonging to specific data-processing pipes) used certain packages (indicated on the '`Suggest`' field included in the 'DESCRIPTION' file) and should be also installed in order to ensure its proper operation. It should be taken into account that in case of needing all the dependencies, the argument `dependencies = TRUE` should be included in the command `install.packages`.

Executing bdpar framework

In order to guarantee a high level of flexibility, **bdpar** can be easily executed by using two different ways (i) following an OOP paradigm or (ii) using a classical function call approach. Below is include a code snippet describing both scenarios.

```
bdpar <- Bdpar$new()
bdpar$execute(path, extractors= ExtractorFactory$new(),
              pipeline = DefaultPipeline$new(), cache = TRUE,
              verbose = FALSE, summary = FALSE)
```

a) Executing **bdpar** using OOP paradigm

```
output <- runPipeline(path, extractors = ExtractorFactory$new(),
                      pipeline = DefaultPipeline$new(), cache= TRUE,
                      verbose = FALSE, summary = FALSE)
```

b) Executing **bdpar** following a function-based approach.

As can be depicted, both execution methods require the same three arguments since `runPipeline` is a wrapper function which encapsulates **bdpar** execution using OOP paradigm. The first argument is mandatory since it is used to specify the directory or file(s) path where the raw input data is located. The second parameter indicates the extractors required to parse the input sources. If not defined, **bdpar** automatically invokes the default `ExtractorFactory$new()` object which initializes the four extractors provided by **bdpar** framework. Following, the third argument is used to determine the sequence of preprocessing tasks that should be executed (pipeline) to achieve the desired output (featured dataset). If the argument is not assigned, **bdpar** executes `DefaultPipeline$new()` object which implements an error-safe pipeline comprised of 18 tasks described in Table 1. The fourth one is used to enable (or disable) **bdpar** not-re-execution functionality (defined as `N-RE`). Particularly, this feature is able to detect which instances, tasks and even pipelines were previously executed with a view to avoiding their re-execution. Moreover, the penultimate argument is used to indicate (if needed) the generation of a log output showing different levels of granularity (`DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`). It should be noted that `DEBUG` level allows displaying the intermediate results achieved by each instance after being processed by the tasks comprising the pipeline. This is very useful to detect the location of possible errors. Finally, the latter argument allows showing (if needed) a detailed summary of all the operations and tasks performed during the pipeline execution.

Developing new functionalities

As previously stated, the design of the software architecture of **bdpar** is focused on facilitating the customization of any stage of the process (data loader and pipeline executor). Specifically, **bdpar** allows users (i) defining new types of input data parses, (ii) creating new pipes and (iii) implementing and deploying new pre-processing tasks.

Regarding the first aspect, the development of new content parsers involves two stages: (i) the implementation of a customized extractor by overriding solely the methods of the `Instance` class that are necessary to load the input and (ii) the registration of the created extractor so it can be loaded by the **bdpar** framework. Two code fragments to detail the development and registration of a new customized parsers are included below.

```
ExtractorImage <- R6::R6Class(
  classname = "ExtractorImage",
  inherit = Instance,
  public = list(
    initialize = function(path) {
      super$initialize(path)
    },
    obtainSource = function() {
      source <- imager::load.image(super$getPath())
      super$setSource(source)
      super$setData(source)
    } ) )
```

a) Implementation of new `ExtractorImage` parser.

```
extractors <- ExtractorFactory$new()
extractors$registerExtractor(extension = c("jpeg","png"), extractor = ExtractorImage)
```

b) Dynamic extractor registration operation in **bdpar**.

As can be seen, the first code snippet describes the formal structure of a new extractor. Particularly, `ExtractorImage` is able to load an image into an Instance object. To accomplish this task, the `obtainSource` method loads (by invoking `load.image()` method) an image from the file path received as parameter of class constructor (`super$getPath()`). Then, the loaded image is stored in the source variable of the `Instance` superclass (by invoking `super$setSource()` method) and is assigned to the data variable by calling to `super$setData()` method. The data field is used to store the result of each task comprising the pipeline.

Moreover, the second fragment of code exemplifies the registration of the previously created extractor in **bdpar** framework. As can be depicted, this operation is performed by a simple call to the `registerExtractor` method included in `ExtractorFactory` class. Due to the one-to-one dependency between each extractor and the different input formats, **bdpar** requires the definition of a specific extension (or set of extensions) to discern which type of extractor should execute. Also, `ExtractorFactory` provides two additional methods (i) `getAllExtractors` which shows all registered extractors in **bdpar** framework and (ii) `removeExtractor` which deletes a specific data extractor. Finally, to avoid parsing errors, unsupported input contents by registered extractors are automatically ignored by **bdpar**.

For the creation and deployment of new preprocessing tasks, **bdpar** provides an abstract class named `GenericPipe`. This type of class is very common in OOP to ensure all subclasses (pipes) follow the same structure and implement the methods defined in the superclass (`GenericPipe`). Particularly, `GenericPipe` defines two main methods that should be included on each subclass: (i) `initialize` and (ii) `pipe`. The former includes three optional parameters that are `propertyName` (which refers to the specific name to the output value computed in the task, "alwaysBeforeDeps" and "notAfterDeps" which handles two types of dependencies between pipes ("always-before" and "not-after" respectively). Finally, the `pipe` method is used to implement the behaviour of the new task. Below we include a code snippet exemplifying how to develop a basic image-preprocess pipeline. Concretely we design three pipes: (i) `Image2Pipe` responsible for invoking the `obtainSource` method provided in the `ExtractorImage` parser, (ii) `ImageCroppingPipe` in charge of halving the image and (iii) `ImageRotatePipe` which rotates the image 30 degrees clockwise.

```
Image2Pipe <- R6::R6Class(
  name = "Image2Pipe",
  inherit = GenericPipe,
  public = list(
    initialize = function(propertyName = "",
                          alwaysBeforeDeps = list(),
                          notAfterDeps = list()) {
      super$initialize(propertyName, alwaysBeforeDeps, notAfterDeps)
    },
    pipe = function(instance) {
      instance$obtainSource()
      instance
    } ) )

ImageCroppingPipe <- R6::R6Class(
  "ImageCroppingPipe",
  inherit = GenericPipe,
  public = list(
    initialize = function(propertyName = "",
                          alwaysBeforeDeps = list("Image2Pipe"),
                          notAfterDeps = list()) {
      super$initialize(propertyName, alwaysBeforeDeps, notAfterDeps)
    },
    pipe = function(instance) {
    data <- instance$getData()
    data <- imager::imsub(data, x > height/2)
    instance$setData(data)
      instance
    } ) )

ImageResizePipe <- R6::R6Class(
  "ImageResizePipe",
  inherit = GenericPipe,
  public = list(
    initialize = function(propertyName = "",
```

```
                        alwaysBeforeDeps = list("Image2Pipe"),
                        notAfterDeps = list()) {
      super$initialize(propertyName, alwaysBeforeDeps, notAfterDeps)
    },
    pipe = function(instance) {
    data <- instance$getData()
    data <- imager::imrotate(data, 30)
    instance$setData(data)
      instance
    } ) )
```

As can be seen, the `Image2Pipe` class stores the loaded image into a new instance. Following, `ImageCroppingPipe` and `ImageResizePipe` class apply different image manipulation functions (`imager::imsub`, `imager::imrotate`) to halve and rotate the images respectively. Following, the result of each pipe is stored into a specific field of the `Instance` object (by calling the `instance$setData()` method). To ensure proper operation of both tasks, image content should be previously loaded into an instance object (by invoking `Image2Pipe` associated task). To this end, a priori dependence with `Image2Pipe` has been defined in the `initialize` method of both pipes. As mentioned before, to facilitate the development and execution of pipes, dependencies between pipes are automatically managed by the object-oriented pipe operator (%>|%). Finally, in order to develop robust pipelines, `instance` objects should be invalidated when an error occurs or the requirements are not satisfied (such as the storage of empty data or non-identification of textual language).

Finally, **bdpar** allows users to customize existing pipelines and develop new ones from scratch. In order to motivate the usage of **bdpar** regardless of user programming skills, the framework allows to manually or dynamically design new pipelines. The first method requires the creation of a new class (inheriting from `GenericPipeline`) which implements the `execute` method defined in the parent class. Moreover, to ensure proper management of (possible) execution errors (such as invalidated instances) a try-catch function should be included. Additionally, **bdpar** allows customizing the log messages (if needed) by calling the `bdpar.log()` function. Below we include an example showing how a customized pipeline is manually created.

```
TestPipeline <- R6::R6Class(
  classname = "TestPipeline",
  inherit = GenericPipeline,
  public = list(
    initialize = function() ,
    execute = function(instance)
      message("[TestPipeline][execute][Info] ", instance$getPath())
      tryCatch(
         instance %>|% Image2Pipe$new() %>|%
           ImageCroppingPipe$new() %>|% ImageResizePipe$new(),
          error = function(e)
               bdpar.log(message = paste0(instance$getPath()," :", paste(e)),
                 level = "ERROR",
                 className = class(self)[1],
                 methodName = "execute")
                 instance$invalidate()
        )
      return(instance)
     ) )
```

On the other hand, the dynamic method allows the creation of custom pipelines by simply indicating a list containing the pipe objects to be used. To achieve a higher level of flexibility dynamic, the model provides two ways of defining pipelines: (i) during the object instantiation or (ii) by calling the add function. A simple example describing how the previous pipeline is created following the dynamic method is included below.

```
pipeline <- DynamicPipeline$new(pipeline = list(Image2Pipe$new(),
                                                ImageCroppingPipe$new(),
                                                ImageResizePipe$new()))
```

a) Pipeline definition during object instantiation.

```
pipeline <- DynamicPipeline$new()
pipeline$add(list(Image2Pipe$new(), ImageCroppingPipe$new(), ImageResizePipe$new()))
```

b) Pipeline creation using the add method.

Additionally, the dynamic method provides six methods able to extend the pipeline customization capabilities: (i) `add(pipe,pos=NULL)` which adds new pipe object(s) to the pipeline flow at a certain position (or at the end if not defined), (ii) `removeByPos(pos)` which removes a pipe object at a given position, (iii) `removeByPipe(pipe.name)` responsible for erasing a `pipe` object by name, (iv) `removeAll()` capable of releasing all pipe objects from the pipeline, (v) `get()` returning a list containing the pipe objects comprising the pipeline, and finally, (vi) `print()` which displays the pipes comprised in the pipeline.

As can be realised from both methods, the manual definition of pipelines allows users to have greater control and insight over the pipeline (such as personalizing error-handling methods, or the inclusion of new user-defined object-oriented pipeline operators). Conversely, the dynamic mode enables users to define optimal pipelines without expert knowledge of `R6` and OOP concepts.

Managing bdpar configuration options

`bdpar.Options` included in **bdpar** allows managing configuration parameters to customize the behaviour of available tasks and indicate parameters needed for the proper operation of pipes and/or content extractors (such as path locations for slang dictionaries or credentials required by Twitter or Youtube APIs respectively). Moreover, to easily search and access the configuration, `bdpar.Options` stores parameters following a key-value pair structure. As can be deducted, the key parameter is used to uniquely identify a configuration entry. In order to facilitate the management of configuration parameters, `bdpar.Options` provides four main methods: (i) `bdpar.Options$add(key,value)` which adds a new configuration entry, (ii) `bdpar.Options$set(key,value)` used to modify the value of an existing configuration parameter, (iii) `bdpar.Options$remove(key)` which removes an entry matching a specific name and finally (iv) `bdpar.Options$reset()` used to restore `bdpar.Options` to its initial state (default options). Table 2 describes the configuration options included in **bdpar**.

Type	Key	Assigned by default	Description
API credentials	twitter.consumer.key twitter.consumer.secret twitter.access.token twitter.access.token.secret	x	Set of keys need to connect to Twitter and Youtube API
	youtube.app.id youtube.app.password	x	
API cache	cache.youtube.path cache.twitter.path	x	Path to temporary place extracted data
Pipe parameters	teeCSVPipe.output.path	√	Defines the output file path for TeeCSVPipe
Extractor options	extractorEML.mpaPartSelected	√	Indicates the content-type to parse on multipart emails
N-RE handler	cache.folder	√	Indicates the path to store the intermediate results
Parallel settings	numCores	√	Selects the number of cores used to execute the pipelines
Resource files	resources.abbreviations.path resources.contractions.path resources.interjections.path resources.slangs.path resources.stopwords.path	√	Location for the different language dictionaries (slang, contractions, . . .)

Table 2: Structure of **bdpar** configuration options.

As can be seen from Table 2, configuration options are divided into seven categories: (i) API credentials, (ii) API cache, (iii) pipe parameters (iv) Extractor options, (v) instance cache handler, (vi) parallel settings and (vii) resource files. Is important to take into account that values for API credentials are not provided by default due to the inexistence of publicly available access keys for both Youtube and Twitter (only for private use with prior approval). Following, the optional API cache configuration entries are responsible for designating temporal locations to store information obtained after executing the content extractors. Defining cache paths API ensures that duplicated sources are executed only once (avoids parsing duplicated inputs).

Moreover, pipe parameters and extractor options allow specifying configuration values needed to guarantee the proper execution of pipes and extractors respectively. Particularly, default "extractorEML.mpaPartSelected" entry allows defining which content-type (text/plain or text/html) should be extracted in multipart emails while "teeCSVPipe.output.path" indicates the location to store the CSV file generated by "teeCSVPipe" pipe.

In addition, the not-re-execution handler allows defining the path to store the information required for the proper operation of the not-re-execution functionality. Despite this feature is very useful to reduce both unnecessary computation costs and time consumption requires extra storage space. Therefore, **bdpar** allows the deletion of the intermediate results by invoking the specific `bdpar.Options$cleanCache()` method.

Following, parallel settings category is used to define the configuration values to handle parallelization in **bdpar**. Concretely "numCores" entry allows defining the number of CPU cores to be used when a pipeline is executed. By default, **bdpar** is configured following a sequential paradigm (`numCores = 1`). Additionally, to ensure proper use of CPU resources **bdpar** provides a mechanism to verify whether the number of assigned CPU cores is compatible with the hardware specifications or not. If the assigned CPU cores is not valid, **bdpar** will be executed using the most optimal configuration according to the hardware specifications.

The latter category comprises different dictionaries needed to perform multiple language-dependent operations (such as contraction detection or stopwords removal). Dictionaries provided by default ensures the compatibility of **bdpar** from 8 to 50 different languages (depending on the text-mining operation selected).

A case study

In order to illustrate the functionality of the **bdpar** package from a more realistic perspective, we developed a case study to show the most frequent words from a heterogeneous dataset collection (containing SMS and emails).The dataset comprises 20 emails (eml format) and 20 SMS in plain text from the nutritional and health domain. Moreover, to ensure the straightforward reproducibility (i) all the resources used are included in the package and the pipeline provided by the package (`DefaultPipeline`) was selected to perform the content preprocessing flow and (ii) resulting dataset was stored into a structured CSV file. Once the pipeline was executed, 18 new columns were generated from the outputs acquired after executing some pipeline task (labelled according to the property names described in Table 1). For instance, the execution of `FindEmojiPipe` forces the creation (if not exists) of a new column (named "emojis") containing the emojis found for each instance (or blank if not found).

To carry out the case of study, word frequencies were computed over the text content generated after executing the whole pipeline tasks (stored in the data column). Additionally, some previous text-cleaning operations were performed over the preprocessed text prior to executing the computation of the word frequencies (using word-cloud plots). Concretely, words were reduced to their stem form (stemDocument), punctuation marks were deleted and numbers were removed (using the **tm** package (Feinerer et al., 2008)). Finally, for comparison purposes, frequencies were calculated both individually and jointly (see Figures 2 and 3)

```
#Execute bdpar framework
bdpar::runPipeline(path = system.file(file.path("example"), package = "bdpar"),
                   cache = FALSE)
#Load CSV generated after executing bdpar
dataset <- read.csv(file = bdpar.Options$get("teeCSVPipe.output.path"),
                    sep = ";", stringsAsFactors = FALSE )

#Separate instances by type
sms <- dataset[dataset$extension == "tsms", ]
eml <- dataset[dataset$extension == "eml", ]
# Function to clean text and compute frequencies
word.frec <- function(data) {
  corpus <- tm::VCorpus(VectorSource(data))
  corpus <- tm::tm_map(corpus, removePunctuation)
  corpus <- tm::tm_map(corpus, removeNumbers)
  corpus <- tm::tm_map(corpus, stemDocument)
  sorted <- sort(rowSums(as.matrix(tm::TermDocumentMatrix(corpus)), decreasing = TRUE)
  return(data.frame(word = names(sorted), freq = sorted))
}
sms.words <- word.frec(sms$data)
eml.words <- word.frec(eml$data)
all.words <- word.frec(dataset$data)
# Wordcloud for sms and e-mail
par(mfrow=c(1,2))
wordcloud::wordcloud( words = sms.words$word, freq = sms.words$freq, min.freq = 1,
                      max.words = 100, random.order = FALSE, rot.per = .5,
                      colors = RColorBrewer::brewer.pal(8, "Dark2") )
wordcloud::wordcloud( words = eml.words$word, freq = eml.words$freq,
                      min.freq = 1, max.words = 100, random.order = FALSE,
                      rot.per = .5, colors = RColorBrewer::brewer.pal(8, "Dark2") )
par(mfrow=c(1,1))
#Wordcloud for all instances (sms and e-mail)
wordcloud::wordcloud( words = dataset.words$word, freq = all.words$freq, min.freq = 1,
                      max.words = 100, random.order = FALSE,
                      rot.per = .5, colors = RColorBrewer::brewer.pal(8, "Dark2") )
```

Figure 2 graphically represents the results achieved after performing an individualized analysis of each dataset (SMS and e-mails). Conversely, Figure 3 represents the frequencies achieved when

analysing both datasets together. Observing Figure 2, we see that frequencies of words included in SMS messages are lower than those included in emails. This is mainly due to the limited length of SMS messages (up to 160 characters). Therefore, the word frequency in Figure 3 has increased considerably mainly owing the joint evaluation of both datasets.

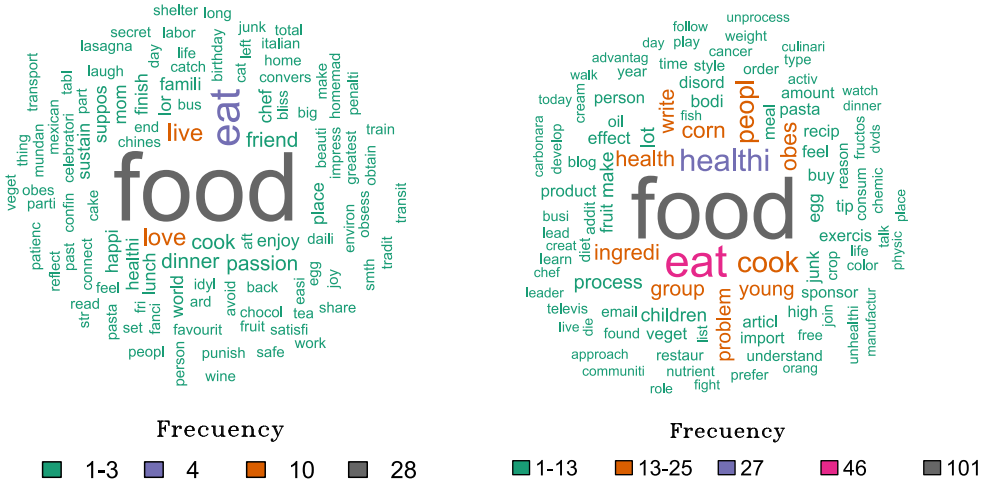

Figure 2: Wordcloud for SMS (left) and e-mail (right).

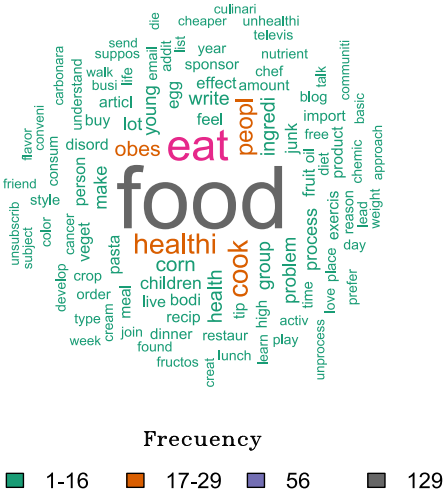

Figure 3: Wordcloud for SMS and e-mail jointly.

Keeping in mind the functionality provided by **bdpar** (demonstrated through the current case study), it is easy to deduce that the application of data mining techniques over unstructured data could be easily addressed by taking advantage of the functionality of our framework. Some big data operations that could be addressed by taking advantage of **bdpar** are the clustering of documents using token features, the classification of documents, or the retrieval of documents for specific queries.

Conclusions and future work

In this work, we introduced **bdpar**, a pipe-based R framework to facilitate the creation of unified datasets from heterogeneous sources. Our framework allows users to (i) define new content extractors (data parsers), (ii) develop and deploy new pre.processing tasks (pipes) and (iii) define and build customized interconnected task flows (pipelines). Additionally, to save computational resources and increase execution speed, **bdpar** provides an optimized pipe operator (noted as `%>I%`) capable of aborting the processing of an instance if an error was detected. Finally, a case study was developed

to demonstrate the capability of the framework to pre-process and unify heterogeneous data into a single CSV file.

Future work is focused on two main aspects: (i) the development of semantic-based tasks able to explode the semantic relationships between synsets and; (ii) the capability to represent using a graph-based visualization the pipes comprising each pipeline and (iii) the analysis of textual polarity and sentiment analysis..

Acknowledgements

The second author's work has been partially supported by the Grant IN2017-84658-C2-1-R of the Spanish Ministry of Industry. David Ruano-Ordás has been supported by a post-doctoral 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. Finally, we thank the reviewers for their deep appropriate suggestions to improve the quality of the manuscript.

Bibliography

A. Amazon. AWS Amazon Data Pipeline. https://aws.amazon.com/es/datapipeline, 2019. Accesed: 2019-05-6. [p]

S. M. Bache and H. Wickham. *magrittr: A Forward-Pipe Operator for R*, 2020. URL https://CRAN.R-project.org/package=magrittr. R package version 2.0.1. [p]

Brown, Brad, M. Chui, and J. Manyika. Are you ready for the era of 'big data'. *McKinsey Quarterly*, 4 (1):24–35, 2011. [p]

W. Chang. *R6 and Reference Class performance tests*, 2019a. URL https://r6.r-lib.org/articles/Performance.html. R6 package version 2.4.1. [p]

W. Chang. *R6: Encapsulated Classes with Reference Semantics*, 2019b. URL https://CRAN.R-project.org/package=R6. R package version 2.4.0. [p]

P. Di Tommaso. Awesome pipeline. https://github.com/pditommaso/awesome-pipeline, 2019. [p]

Domo-Data. Domo-data never sleeps 6.0. https://www.domo.com/solution/data-never-sleeps-6, 2019. Accesed: 2019-04-05. [p]

S. Dorogovtsev and J. Mendes. *Evolution of Networks: From Biological Nets to the Internet and WWW*. OUP Oxford, 2013. ISBN 9780191004407. URL https://books.google.es/books?id=FFL1AgAAQBAJ. [p]

N. A. o. Engineering. *Cutting Edge Technologies*. The National Academies Press, Washington, DC, 1984. ISBN 978-0-309-03489-0. doi: 10.17226/286. URL https://www.nap.edu/catalog/286/cutting-edge-technologies. [p]

I. Feinerer, K. Hornik, and D. Meyer. Text mining infrastructure in R. *Journal of Statistical Software*, 25 (5):1–54, March 2008. URL http://www.jstatsoft.org/v25/i05/. [p]

Google. Google cloud dataflow. https://cloud.google.com/solutions/big-data, 2019. Accesed: 2019-05-6. [p]

M. Iansiti and T. Khansa. Technological Evolution, System Architecture and the Obsolescence of Firm Capabilities. *Industrial and Corporate Change*, 4(2):333–361, 03 1995. ISSN 0960-6491. doi: 10.1093/icc/4.2.333. URL https://doi.org/10.1093/icc/4.2.333. [p]

IBM. IBM Big Data Success Stories. http://public.dhe.ibm.com/software/data/sw-library/big-data/ibm-big-data-success.pdf, 2019. Accesed: 2019-04-8. [p]

IBM, P. Zikopoulos, and C. Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011. ISBN 0071790535, 9780071790536. [p]

N. Kabeer. Introduction: The search for inclusive citizenship: Meanings and expressions in an interconnected world. 2005. [p]

M. W. Kearney. *rtweet: Collecting Twitter Data*, 2019. URL https://cran.r-project.org/package=rtweet. R package version 0.6.9. [p]

A. Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proc. VLDB Endow.*, 5 (12):2032–2033, Aug. 2012. ISSN 2150-8097. doi: 10.14778/2367502.2367572. URL http://dx.doi.org/10.14778/2367502.2367572. [p]

W. M. Landau. The drake R package: a pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software*, 3(21), 2018. URL https://doi.org/10.21105/joss.00550. [p]

J. Mervis. Agencies rally to tackle big data. *Science*, 336(6077):22–22, 2012. ISSN 0036-8075. doi: 10.1126/science.336.6077.22. URL https://science.sciencemag.org/content/336/6077/22. [p]

D. Miner and A. Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 1st edition, 2012. ISBN 1449327176, 9781449327170. [p]

F. Napolitano. *repo: A Data-Centered Data Flow Manager*, 2020. URL https://CRAN.R-project.org/package=repo. R package version 2.1.5. [p]

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. *Journal of Big Data*, 2(1):25, 2015. ISSN 2196-1115. doi: 10.1186/s40537-015-0034-z. URL https://doi.org/10.1186/s40537-015-0034-z. [p]

B. Solutions. BDB Solutions for Big Data. https://www.bdb.ai/big-Data-Pipeline, 2019. Accesed: 2019-05-6. [p]

G. Sood. *tuber: Access YouTube from R*, 2019. URL https://CRAN.R-project.org/package=tuber. R package version 0.9.8. [p]

VCloud. Vcloud news. http://www.vcloudnews.com/every-day-big-data-statistics-2-5-quintillion-bytes-of-data-created-daily/, 2019. Accesed: 2019-04-17. [p]

H. Wickham. *Advanced R, Second Edition*. CRC Press, 2019. ISBN 9780815384571. URL https://adv-r.hadley.nz/. [p]

X. Wu, X. Zhu, G. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):97–107, Jan 2014. ISSN 1041-4347. doi: 10.1109/TKDE.2013.109. [p]

Miguel Ferreiro-Díaz
Department of Computer Science
University of Vigo
Spain
miguel.ferreiro.diaz@uvigo.com

Tomás R. Cotos-Yáñez
Department of Statistics and Operations Research
University of Vigo
Spain
cotos@uvigo.es

David Ruano-Ordás
SING Research Group, Galicia Sur Health Research Institute (IIS Galicia Sur)
SERGAS – University of Vigo
Spain
ORCiD: 0000-0002-6050-373X
drordas@uvigo.es