USC
UNIVERSIDADE
DE SANTIAGO
DE COMPOSTELA

Escola Técnica Superior de Enxeñaría
Universidade de Santiago de Compostela

Framework para la Construcción y Despliegue de Sistemas de Procesamiento en Tiempo Real

Autor:
Rodrigo Martínez Castaño

Tutores:
Juan Carlos Pichel Campos
David Enrique Losada Carril

Máster Universitario en Tecnoloxías de Análise de Datos Masivos: Big Data

Junio de 2018

[1]`https://creativecommons.org/licenses/by-sa/4.0/legalcode`

Agradecimientos

Me gustaría agradecer a mis tutores Juan Carlos Pichel Campos y David Enrique Losada Carril su gran apoyo y confianza.

Índice general

Capítulo 1

Introducción

En los últimos años se han desarrollado numerosas tecnologías destinadas al procesamiento de datos masivos, muchas de ellas de código abierto y de uso libre. Estas plataformas se centran en la escalabilidad horizontal, lo que implica que para el procesamiento de una mayor cantidad de datos sin grandes distorsiones en el ritmo, no es necesario aumentar o actualizar los recursos de una máquina (escalabilidad vertical), sino que es suficiente con añadir más nodos con similares características a un clúster. La proliferación de este tipo de tecnologías de código abierto han democratizado y condicionado el gran número de aplicaciones que hacen uso de estas plataformas en multitud de ámbitos, tanto profesionales como académicos.

Centrándonos en los *frameworks* de procesamiento, nos encontramos con una importante limitación: los datos han de poder dividirse en grupos independientes, de tal modo que sea posible paralelizar el trabajo en diferentes máquinas aunque existan puntos de procesamiento secuencial. Existen dos grandes tipos de tecnologías de procesamiento de este tipo: procesamiento de lotes (*batch processing*) y procesamiento de flujos (*stream processing*). En el primer caso, los resultados finales se obtienen juntos al finalizar el procesamiento del lote de datos compuesto por una o más etapas. Para definir el trabajo a realizar, se define una topología de procesamiento que indica el flujo de los datos a través de las distintas etapas. Cada nodo (físico o virtual) puede ejecutar una instancia de la topología (aislada del resto de instancias), repartiéndose los datos de forma equitativa entre las instancias existentes. En las tecnologías de procesamiento de flujos, las distintas etapas de una topología son independientes y no pertenecen a una instancia concreta. Por tanto, las distintas etapas pueden ser paralelizadas de forma individual sin aumentar el grado de paralelismo de toda la topología. Estas tecnologías son adecuadas para aplicaciones que obtienen información en tiempo real y deben dar una respuesta inmediata, ya que cuando un dato completa su camino a través de las distintas etapas, el resultado puede obtenerse de forma instantánea. Sin embargo, con el procesamiento de lotes los resultados se obtienen cuando un lote de datos es procesado por completo. Un caso de aplicación de procesamiento en tiempo real es el análisis de contenidos en redes sociales para la detección temprana de riesgos. Este será el objetivo principal de este proyecto.

La sociedad está expuesta a un gran abanico de riesgos y amenazas, muchos de los cuales se exteriorizan en Internet a través de las redes sociales. Algunos de los riesgos son externos, generados por criminales, acosadores, etc. Otros, sin embargo, se originan por los propios individuos. Por ejemplo, la depresión puede llevar a desórdenes alimenticios como la bulimia o la anorexia o incluso al suicidio [7]. La detección temprana de riesgos consiste en descubrir situaciones con probabilidad de evolucionar negativamente hacia escenarios de mayor gravedad. Una detección temprana apropiada puede reducir o minimizar problemas. Algunos tipos de riesgos pueden ser detectados analizando la actividad de usuarios en Internet. Actualmente, las tecnologías existentes son mayormente reactivas: las alertas suelen dispararse cuando los problemas ya han aparecido. La detección temprana de riesgos es un área de estudio cuya importancia está en auge.

Reddit es un sitio web donde los usuarios publican contenidos como textos, imágenes o enlaces, y donde otros usuarios pueden comentar y votar a favor o en contra. La plataforma se subdivide en distintas comunidades centradas en temas concretos. Actualmente, Reddit está posicionado como el sexto sitio web con mayor tráfico del mundo según la clasificación de Alexa[1]. El promedio de usuarios mensuales es superior a los 330 millones y existen más de 138.000 comunidades activas[2].

En este trabajo se presentan tres aportaciones. En primer lugar (1) CATENAE[3], una librería Python que permite construir de forma sencilla sistemas modulares y escalables de procesamiento en tiempo real. La librería fue diseñada para servir de base a (2) REDD[4] (*REddit Depression Detection*) pero con un enfoque amplio, buscando poder convertir rápidamente un prototipo escrito en Python (de cualquier índole) en un sistema escalable de procesamiento en tiempo real. REDD es una plataforma integral para la detección temprana de riesgo de depresión en tiempo real que opera sobre Reddit y en la que los expertos pueden revisar las alertas generadas. Por último, (3) ANCORIS[5] es un gestor de recursos para clústers que, de nuevo, está ideado desde una perspectiva genérica (gestión sencilla de recursos de un clúster a través de contenedores Docker [4]) pero con una aplicación concreta: proveer los recursos necesarios para la ejecución de sistemas creados con CATENAE.

Tanto CATENAE como REDD fueron presentados en dos congresos de investigación: *European Conference on Information Retrieval* (CORE A) y *Spanish Conference in Information Retrieval.* Ambas aportaciones se describen en las publicaciones adjuntas a esta memoria, presentadas en los Capítulos 2 y 3.

[1]`https://www.alexa.com/siteinfo/reddit.com/`
[2]`https://www.redditinc.com/`
[3]Disponible en `https://github.com/catenae`
[4]Disponible en `https://github.com/redd-system`
[5]Disponible en: `https://github.com/ancorisrm`

1.1. Catenae

La librería Catenae se desarrolla ante la necesidad de proporcionar una base para la construcción de topologías para la plataforma Redd en Python. Se escoge este lenguaje por dos razones principales: su popularidad creciente en Ciencia de Datos y la existencia de librerías importantes como scikit-learn [10], empleada además para la construcción del clasificador original de individuos en redes sociales.

Actualmente, el framework más popular para procesamiento de flujos de datos, Apache Storm [2], cuenta con un soporte para Python poco eficiente. Al no ser un lenguaje JVM, Storm se comunica con los procesos Python indirectamente. Además, la gestión de dependencias es un proceso tedioso a la hora de desplegar topologías.

Catenae proporciona un entorno para desarrollar sistemas modulares de procesamiento en tiempo real, delegando el paso de mensajes entre módulos a Apache Kafka [1]. Además, esto permite conectar las topologías creadas con Catenae con otros sistemas de forma sencilla. El objetivo de esta librería es facilitar la escalabilidad horizontal de aplicaciones escritas en Python a través de la división del procesamiento en micromódulos (etapas de transformación o filtrado de una topología) que se ejecutarán dentro de contenedores Docker, proporcionando aislamiento entre módulos. Debido a esta característica, es posible utilizar distintas versiones de una librería en distintos módulos de una misma topología e incluso distintas versiones del intérprete de Python. Algunas características destacables son la serialización transparente y el sistema de priorización de flujos de entrada para los módulos.

1.2. REDD

La plataforma Redd permite la detección temprana de usuarios en riesgo de depresión en tiempo real. El sistema trabaja sobre la red social Reddit con un *crawler* propio y toma como base un clasificador y un conjunto de datos de entrenamiento detallados en [6] para la fase de predicción. La plataforma cuenta con los siguientes módulos: (1) una topología Catenae para la extracción de usuarios y contenidos de Reddit, (2) una segunda topología Catenae para el clasificador de textos de los usuarios, (3) una API HTTP para la gestión de alertas y el acceso a los textos de los usuarios, y (4) una interfaz web para la gestión de alertas por parte de los expertos.

1.3. Ancoris

Para la ejecución de la plataforma Redd, otras topologías creadas con Catenae y, en general, de sistemas formados por módulos empaquetados en contenedores, se desarrolla Ancoris, un gestor de recursos basado en contenedores Docker enfocado en la sencillez de uso y en una gestión de recursos de alta granularidad. El sistema se describe con mayor detalle en el Capítulo 4.

Capítulo 2

A Micromodule Approach for Building Real-Time Systems with Python-Based Models: Application to Early Risk Detection of Depression on Social Media

Este artículo fue publicado en *Advances in Information Retrieval*, parte de las *Lecture Notes in Computer Science* (LNCS) de la editorial Springer en marzo de 2018 tras su presentación en la *40th European Conference on Information Retrieval* (ECIR 2018. Grenoble, Francia) [8] de categoría A según la clasificación CORE (2018)[1].

Resumen. En este trabajo presentamos CATENAE, una nueva librería cuyo objetivo principal es proveer un solución fácil de usar para crear aplicaciones escalables de procesamiento en tiempo real a través de micromódulos Python. Para demostrar su potencial, hemos desarrollado una plataforma que procesa datos de redes sociales y alerta sobre signos tempranos de depresión. La plataforma está compuesta por los siguientes módulos: (1) una topología de seguimiento y extracción de textos, (2) una topología de clasificación para procesar los textos extraídos, (3) una API HTTP para la consulta y gestión de alertas emitidas por el sistema, y (4) una interfaz web.

En el artículo adjunto se explica en detalle el funcionamiento de los distintos módulos, en especial de las dos topologías.

[1] `http://portal.core.edu.au/conf-ranks/483/`

A Micromodule Approach for Building Real-Time Systems with Python-Based Models: Application to Early Risk Detection of Depression on Social Media

Rodrigo Martínez-Castaño[1], Juan C. Pichel[1], David E. Losada[1], and Fabio Crestani[2]

[1] Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela, Santiago de Compostela, Spain
{rodrigo.martinez,juancarlos.pichel,david.losada}@usc.es
[2] Faculty of Informatics, Università della Svizzera Italiana (USI), Lugano, Switzerland
fabio.crestani@usi.ch

Abstract. In this work we introduce Catenae, a new library whose main goal is to provide an easy-to-use solution for scalable real-time deployments with Python micromodules. To demonstrate its potential, we have developed an application that processes social media data and alerts about early signs of depression. The architecture has the following modules: (1) a crawler for extracting users and content, (2) a classifier pipeline that processes new user contents, (3) an HTTP API for alert management and access to users' submissions, and (4) a web interface.

1 Introduction

Early risk detection [2] is a challenging and increasingly important research area. People are exposed to a wide range of threats and risks, and many of them exteriorize on social media. Some risks might come from criminals and offenders (e.g., stalkers, or offenders with sexual motivations). Other risks are not originated by external actors, but by the individuals themselves. For example, depression may lead to an eating disorder such as bulimia or anorexia or even to suicide. In this demo, we present the architecture of a system able to massively track online data and support risk assessment. The system is adaptable to multiple scenarios of early risk prediction, but we focus here on the case of depression. Such an application may be useful, for instance, to health agencies seeking a tool for analyzing the impact of depression on society.

Catenae is a Python library for building topologies in the shape of directed acyclic graphs. Graph nodes represent points of data transformation and edges symbolize data flowing between nodes. Nodes can be connected to multiple nodes both to send and receive data. The communication between nodes is managed in

G. Pasi et al. (Eds.): ECIR 2018, LNCS 10772, pp. 801–805, 2018.
https://doi.org/10.1007/978-3-319-76941-7_79

the form of message queues by Apache Kafka,[1] a distributed message broker for high-throughput, low-latency handling of real-time data feeds. Each node can be instantiated multiple times in such a way that if one type of node becomes a bottleneck, it is only necessary to replicate it. In addition, unlike popular batch processing frameworks where resources are assigned to the whole topology, Catenae assigns individually the hardware resources to each node.

We have developed a system for real-time prediction of signs of depression with Catenae. The system uses the social network Reddit as data source. Following the lessons learned in [1], we implemented a dynamic strategy that works with a *depression classifier* (built from the training split detailed in [1]) and incrementally analyzes the stream of texts written by each user. To meet this aim, we defined micromodules as tiny, loosely coupled software modules (nodes of the topology) that can scale horizontally and be deployed independently. The micromodule approach has some advantages over batch processing architectures when dealing with real-time independent events. This is the case in retrieving user texts (posts or comments) from Reddit in real time, where they can be processed in parallel as they are being collected. Our system is oriented to early detection and, thus, it is more reasonable to make the alerts as soon as there is evidence of a potential risk (rather than accumulating cases and making batch processing). The system is explained in detail below.

2 Early Risk Detection of Depression in Real Time

2.1 The Reddit Crawler

In order to maximize the number of tracked users, we have built a web crawler for Reddit following the rules expressed in their `robots.txt` file. The crawler also uses the Catenae library, as it is composed of multiple horizontal-scalable micromodules in a pipeline (see Fig. 1):

- **Submission and comment crawlers.** They retrieve all new submissions and comments and extract their author nicknames.
- **New user filter.** Nicknames will be checked to avoid repeated users. Aerospike[2] is used to deal with this task efficiently as it is a memory-based store. Those users who pass the filter will be sent to a queue of new users.
- **User content crawlers.** In this stage, all texts (submissions/comments) written on Reddit by the users that passed the filter are extracted as far as possible. Collecting all submissions of a user requires n calls, where n is the number of posts to retrieve (with a maximum of 100). On the other hand, retrieving the newest comments made by a user requires a single call. On every iteration, the system only retrieves the new texts available (it stores the identifiers of the last submission and comment for each user). The user content crawlers obtain user identifiers from different queues, ordered by priority.

[1] https://kafka.apache.org/.
[2] https://www.aerospike.com.

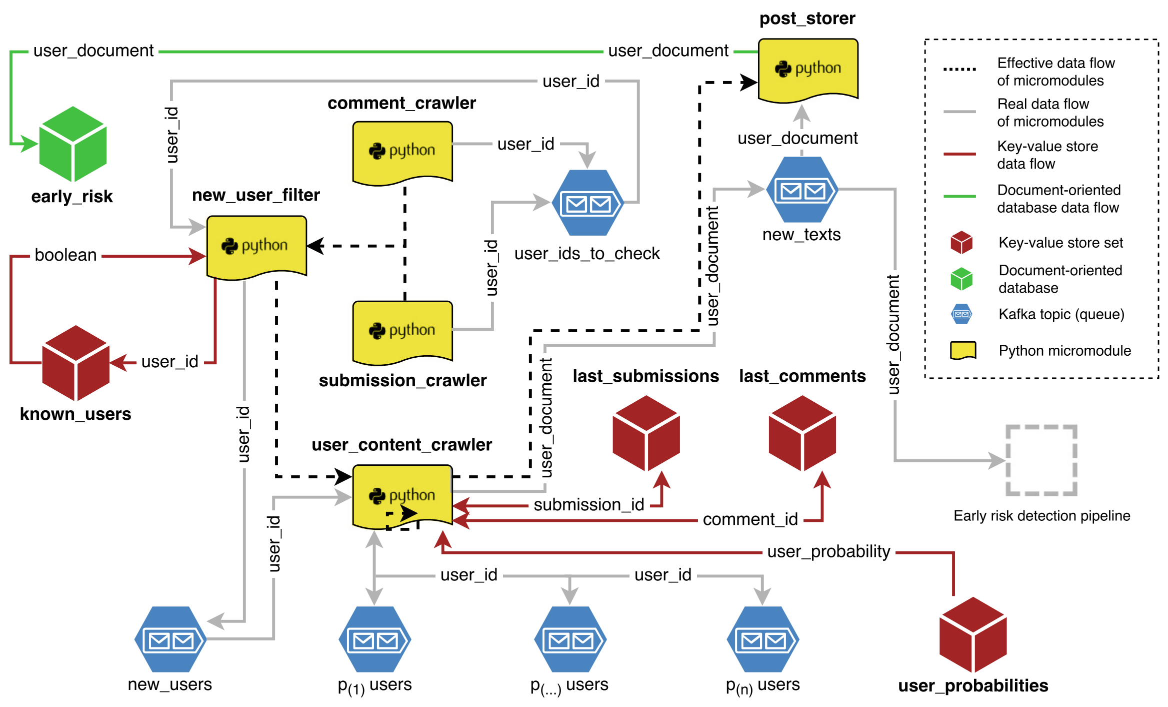


Fig. 1. Architecture diagram of the Reddit crawler.

Based on the estimated probability of risk and the activity since the previous iteration, users can be relocated in different priority queues. A single output queue receives the extracted texts.

– **Post storer.** It is in charge of storing texts in a document-oriented database. In addition, these texts will also feed the early prediction pipeline.

2.2 The Early Risk Detection Pipeline

The early prediction pipeline is a Logistic Regression classifier with L1 regularization, implemented in Python with *scikit-learn*.[3] The classifier is built with a training set of 486 users (83 positive, 403 negative) [1]. Users are represented with a single document, consisting of the concatenation of all their writings. The prediction process has four micromodules (see Fig. 2):

– **Text Vectorizer.** It transforms an input text into a vector of token counts.
– **Aggregator.** We accumulate a vector of token counts that represents all the texts of each user. The aggregator merges the current vector of counts with the vector obtained from any new submission or comment.
– **Tf–idf Transformer.** It transforms the aggregated vector of counts to a normalized tf-idf representation.
– **Model Predictor.** It produces the probability of risk of depression for users given their tf–idf representation. In addition, it produces an individual probability for each document which will be stored by the Post Updater micromodule.

[3] http://scikit-learn.org.

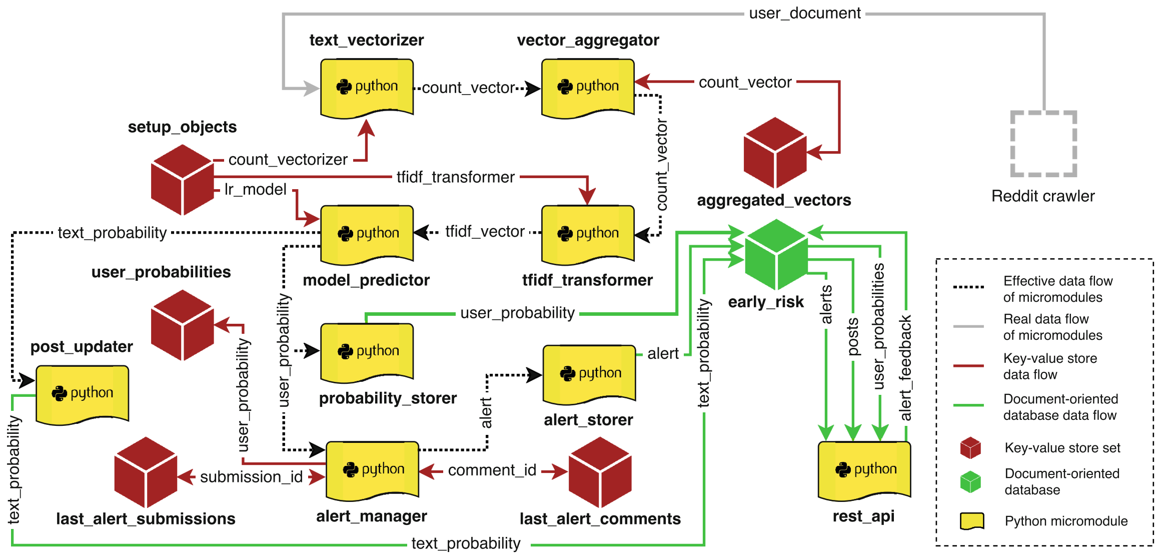


Fig. 2. Architecture diagram of the early risk detection pipeline. Main queues omitted for simplicity.

Both the Probability Storer and the Alert Manager micromodules are fed with user probabilities by the Model Predictor, storing alerts and generating them over a certain threshold, respectively.

The aggregated vectors and the Python objects (count vectorizer, tf–idf transformer, and the classification model) are stored in the Aerospike store. In this way, the vector updates and the micromodule initializations are fast.

2.3 The HTTP API and the User Interface

The user interface is a web application that retrieves information from the Alert Manager HTTP API and provides access to the generated alerts. Among its main functionalities are retrieving and processing those alerts, and retrieving the users' texts and historical records. Each alert has associated a confidence score (as produced by the classifier) and alerts are presented to the user by decreasing recency or in decreasing order of confidence. For each alert, the users' texts are presented ordered by decreasing probability of depression.

Acknowledgements. This work has been supported by MINECO (TIN2014-54565-JIN, TIN2015-64282-R), Xunta de Galicia (ED431G/08) and European Regional Development Fund.

References

1. Losada, D.E., Crestani, F.: A test collection for research on depression and language use. In: Fuhr, N., Quaresma, P., Gonçalves, T., Larsen, B., Balog, K., Macdonald, C., Cappellato, L., Ferro, N. (eds.) CLEF 2016. LNCS, vol. 9822, pp. 28–39. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44564-9_3
2. Losada, D.E., Crestani, F., Parapar, J.: eRISK 2017: CLEF Lab on early risk prediction on the Internet: experimental foundations. In: Jones, G.J.F., Lawless, S., Gonzalo, J., Kelly, L., Goeuriot, L., Mandl, T., Cappellato, L., Ferro, N. (eds.) CLEF 2017. LNCS, vol. 10456, pp. 346–360. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65813-1_30

Capítulo 3

Building Python-Based Topologies for Massive Processing of Social Media Data in Real Time

Este artículo fue publicado en las *International Conference Proceedings Series* (ICPS) de la editorial ACM en junio de 2018 tras su presentación en la *5th Spanish Conference in Information Retrieval* (CERI 2018. Zaragoza, España) [9].

Resumen. En este artículo proponemos una forma de procesar grandes flujos de datos en tiempo real. Catenae es una librería para construir topologías Python de forma sencilla. Las topologías están destinadas a su ejecución en contenedores Docker y, por tanto, la escalabilidad horizontal, asignación de recursos granular y el aislamiento se consiguen fácilmente. Los micromódulos pueden tener sus propias dependencias (incluyendo la versión del intérprete de Python) y los recursos como CPU o memoria pueden ser limitados para cada módulo de forma independiente. Describimos una implementación de un caso de uso compuesto por dos topologías: (1) un *crawler* para el seguimiento de usuarios en redes sociales y (2) un detector de riesgo temprano de depresión. También explicamos cómo las topologías Catenae pueden ser conectadas a otros sistemas escritos en otros lenguajes.

En el artículo adjunto se profundiza en todos estos aspectos, en especial en la librería Catenae.

Building Python-Based Topologies for Massive Processing of Social Media Data in Real Time

Rodrigo Martínez-Castaño
Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela
Santiago de Compostela, Spain
rodrigo.martinez@usc.es

Juan C. Pichel
Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela
Santiago de Compostela, Spain
juancarlos.pichel@usc.es

David E. Losada
Centro de Investigación en Tecnoloxías da Información (CiTIUS), Universidade de Santiago de Compostela
Santiago de Compostela, Spain
david.losada@usc.es

ABSTRACT

In this paper we propose a streaming approach for real-time processing of huge amounts of data. Catenae is a library for easy building and execution of Python topologies (e.g., web crawler, classifier). Topologies are designed for their deployment inside Docker containers and, thus, horizontal scaling, granular resource assignment and isolation can be achieved easily. Furthermore, micromodules can have its own dependencies (including the Python version) and resources such as CPU or memory can be limited for each micromodule. We describe an implementation of a use case composed of two topologies: (1) a crawler for tracking users in social media and (2) an early risk detector of depression. We also explain how Catenae topologies can be connected to non-Python systems.

CCS CONCEPTS

• **Information systems** → *Data extraction and integration*; *Content analysis and feature selection*; • **Computer systems organization** → *Cloud computing*; *Real-time system architecture*;

KEYWORDS

Social Media, Text Mining, Depression, Stream Processing, Real-Time Processing, Docker, Python

ACM Reference Format:
Rodrigo Martínez-Castaño, Juan C. Pichel, and David E. Losada. 2018. Building Python-Based Topologies for Massive Processing of Social Media Data in Real Time. In *CERI '18: 5th Spanish Conference in Information Retrieval, June 2018, Zaragoza, Spain.* ACM, New York, NY, USA, Article 18, 8 pages. https://doi.org/10.1145/3230599.3230618

1 INTRODUCTION

In recent years, numerous technologies have been developed for massive data processing. Many of them are open source and available for free. These platforms focus on horizontal scalability, so they can manage a larger amount of tasks by adding a proportional number of (not necessarily more powerful) nodes to an existing cluster. In order to scale horizontally, it must be possible to partition the data so they can be processed independently by different processes (distributed by the nodes of a cluster), even if at some point partial results have to be merged. Due to the democratization of these technologies that are capable of scaling on commodity hardware, the number of applications that make use of these platforms has increased a lot.

There are two main types of processing technologies of this family: batch and stream processing. In batch processing, results are obtained together at the end of an execution. To perform the computation, a processing topology is defined. A topology is composed by several stages where data are filtered or transformed following a path. Each node of a cluster can execute many instances of the topology, which will receive a fraction of the total input data. Instances are isolated among them and, thus, unbalanced. Therefore, if the input data are not shuffled, bottlenecks may occur. In stream processing, the different stages of a topology are not dependant of a topology instance and they can be individually instantiated multiple times. These processing technologies are suitable for building real-time applications since individual results are reflected instantly once they are obtained in a final stage instance. In addition, the different stage instances are permanently listening for new input data, so the topology does not need to be relaunched as new data are collected. With regards to resource allocation, resources and number of instances can be set at stage level. Since not all the stages of a topology take the same average time to process an input, more resources can be assigned to heavier modules, balancing the execution time per stage.

In Information Retrieval, many tasks can take advantage of stream processing. Many applications must run in real time and have several easily identifiable stages. Data follows a unique or multiple paths through stages forming a topology (a module could serve data to and/or receive from multiple modules). A data extraction stage is always present and usually a final stage to store results. Some examples are Real-Time Filtering (e.g., filtering stage), Real-Time Summarization (e.g., filtering and summarization stages), Real-Time Clustering (e.g., topic extraction), Real-Time User Classification (e.g., user type classifier), Real-Time Trend Detection (e.g., filtering and counting stages), etc.

In this paper, we propose Catenae[1], a new library that facilitates the process of building real-time applications at scale (stream processing). Data are processed through topologies in the shape of directed graphs (see Figure 1). Graph nodes represent points of data transformation and/or filtering and edges symbolize data flowing between nodes. Nodes can be connected to multiple nodes

[1]Publicly available at: https://github.com/catenae

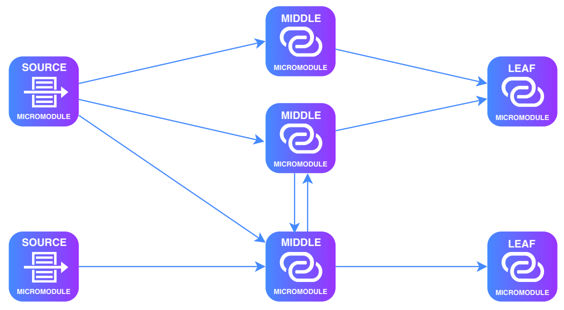


Figure 1. Catenae topology diagram example.

both to send and receive data. Cycles may exist since data can be redirected to previous stages (e.g., if a temporal condition is not met in a certain moment).

The paper is structured as follows: Section 2 describes our Python library to build processing topologies. Section 3 describes a use case where the library is used to build an early risk detection of depression system in real time. Section 4 explains how non-Python systems can be connected to a Catenae topology. In Section 5, related technologies are compared with our library. Finally, Section 6 contains the main conclusions of this study and future work.

2 PYTHON TOPOLOGIES WITH CATENAE

In Catenae, the communication between nodes is managed in the form of message queues by Apache Kafka [4]. Each node (stage) can be instantiated multiple times in such a way that if one type of node becomes a bottleneck, it is only necessary to replicate it. In addition, unlike popular batch processing frameworks where resources are assigned to the whole topology, Catenae assigns the hardware resources individually to each node instance.

In our library, graph nodes are called micromodules, which are tiny, loosely coupled Python scripts packaged inside Docker containers. These modules can scale up and down automatically launching or destroying instances respectively.

There are three types of Catenae micromodules:

- Source links. These modules do not receive data from Kafka but collect them by custom ways such as external APIs connections, custom crawlers or database reads.
- Middle links. All the modules that transform the input data and/or filter them. They consume and emit data within the topology (e.g., filters, classifiers). In Figure 2, a simple filter is implemented with Catenae.
- Leaf links. These nodes consume data from the topology but do not emit back to it. They store the results in a custom way such as database writes or output files (e.g., store alerts based on the output of a classifier).

Running the micromodules inside Docker containers allows the user to scale up a topology by launching more containers. This approach also avoids the Python GIL problem[2] (multiple threads cannot execute Python bytecodes at once) since each instance (container) will have its own CPython interpreter. Furthermore, since

```python
from catenae import Link, Electron


class TokenFilter(Link):

    def setup(self):
        self.allowed_tokens = \
            self.load_object('allowed_tokens')

    def transform(self, electron):
        tokens = electron.value.split()
        if self.allowed_tokens.intersection(tokens):
            return electron

if __name__ == "__main__":
    TokenFilter().start()
```

Figure 2. Micromodule implementation of a token filter (Middle link).

```
# Host
python token_filter.py \
-i <PREVIOUS_MODULE> \
-o <NEXT_MODULE> \
-b <KAFKA_BOOTSTRAP_ADDRESS>:<KAFKA_BOOTSTRAP_PORT>

# Docker container
docker run -d --net=host <DOCKER_IMAGE> \
-i <PREVIOUS_MODULE> \
-o <NEXT_MODULE> \
-b <KAFKA_BOOTSTRAP_ADDRESS>:<KAFKA_BOOTSTRAP_PORT>
```

Figure 3. Example execution of a topology module.

```yaml
---
modules:
  previous_module:
    output: previous_module_output
    instances: 1
  token_filter:
    input: previous_module_output
    output: token_filter_output
    instances: 4
  next_module:
    input: token_filter_output
    instances: 1
conf:
  kafka:
    address: 127.0.0.1
    port: 9092
```

Figure 4. Topology definition file example.

modules are isolated, they can have their own dependencies, avoiding execution problems caused by dependency lacks or conflicts in the cluster nodes. Even the Python version used in each module could differ while the transmitted data between nodes is compatible among them.

At the time of writing these lines, parameters such as input and output modules or the Kafka bootstrap address and port have to be indicated individually for executing each module (See Figure 3). Although it is not supported yet, this process will be managed automatically in future versions following a topology definition file (`topology.yaml`) as described in Figure 4. Also, the required directory structure for future automated building and deployment

[2]https://wiki.python.org/moin/GlobalInterpreterLock

```
|-- topology.yaml
|-- <PREVIOUS_MODULE>
|    |-- <PREVIOUS_MODULE>.py
|    |-- requirements.txt
|
|-- token_filter
|    |-- token_filter.py
|    |-- requirements.txt
|
|-- <NEXT_MODULE>
     |-- <NEXT_MODULE>.py
     |-- requirements.txt
```

Figure 5. Topology directory structure example.

```
FROM catenae/link
COPY token_filter.py /usr/local/bin
ENTRYPOINT ["token_filter.py"]
```

Figure 6. Example of dockerfile script for creating a Docker image of a module.

can be observed in Figure 5. For each module of the topology (i.e., `token_filter`), a directory with its name has to be created containing the following items:

- The main Python script with the same name as the module (i.e. "token_filter.py").
- A file called "requirements.txt" which will contain all the required packages by the module scripts available at the Python Package Index[3].
- Extra custom Python files that the main module will import.

In order to create a Catenae Docker image with our module, the easiest way is to extended our base image as in Figure 6, which contains pre-installed the latest stable release of Python 3, the Kafka client and our library.

Catenae supports multiple inputs for the same node of a topology. There are two implemented modes to manage this situation:

- Parity. With this mode, the modules receive data indistinctly from their inputs.
- Exponential. Modules consume data from their inputs in time windows, assigning a fraction of the window that grows exponentially with the input priority.

The modules of a topology can emit any Python object, which will be serialized and compressed automatically. On the destiny module, the object is also deserialized automatically. Our library uses Pickle[4] to serialize Python objects, which guarantees backwards compatibility among Python interpreters.

The modules are deployed inside containers and thus, resources like CPU or memory can be restricted for each instance of a module.

Finally, Catenae allows the user to load external resources during the initialization of the module: a Python dumped object, for instance. Aerospike [2] is a key-value distributed store that is supported by our library and can be used to store and load data in a fast way during the initialization phase of the topology. It uses RAM memory or SSD disks to store data and supports disk persistence when using memory.

3 USE CASE: EARLY RISK DETECTION OF DEPRESSION

We have developed a system for real-time detection of signs of depression with Catenae [13]. The system uses the social network Reddit as data source. Following the lessons learned in [10], we implemented a dynamic strategy that works with a *depression classifier* (built from the training split detailed in [10]) and incrementally analyses the stream of texts written by each user.

Our system is oriented to early risk detection [11] and, thus, it is more reasonable to fire the alerts as soon as there is evidence of a potential risk (rather than accumulating evidences and making batch processing). Scaling up modules that constitute a bottleneck is easy with this architecture (stream processing). For instance, if a preprocessing module is slower than the predictor, incrementing the number of the preprocessing stage instances would dissolve the bottleneck. In contrast, when launching topologies with batch processing frameworks, resources have to be assigned to the full topology, although only one stage is active at a time. This is a problem in batch processing because any of the stages could not require as many resources as the heavier stage to perform properly.

3.1 The Reddit Crawler

Reddit is a website where users submit content such as text, images or links (submissions) and other users can comment and vote for or against. The platform is subdivided into communities (subreddits) focused on specific topics. It is currently ranked in Alexa [14] as the sixth website with more traffic in the world. The number of average monthly active users is higher than 330 millions and there are more than 138,000 active communities [1].

The first goal of our platform was to maximize the number of tracked users (collecting their new posts periodically). To meet this aim, we have built a web crawler for Reddit following the rules expressed in the `robots.txt` file. The crawler uses the Catenae library, as it is composed of multiple horizontal-scalable micromodules in a pipeline (see Figure 7):

- **Submission and comment crawlers.** They retrieve all new submissions and comments and extract author nicknames.
- **New user filter.** Nicknames are filtered to avoid repeated users. Aerospike is used to deal with this task efficiently as it is a memory-based store. Those users who pass the filter will be sent to a queue of new users.
- **User content crawlers.** In this stage, all texts (submissions/comments) written on Reddit by the users that passed the filter are extracted. Collecting all submissions of a user requires n calls, where n is the number of posts to retrieve (with a maximum of 100). On the other hand, retrieving the newest comments made by a user requires a single call. On every iteration, the system only retrieves the new texts available (it stores the identifiers of the last submission and comment for each user). The user content crawlers obtain user identifiers from different queues, ordered by priority. Based on the confidence score (as produced by the classifier) and the activity since the previous iteration, users can be

[3]https://pypi.python.org/
[4]https://docs.python.org/3/library/pickle.html

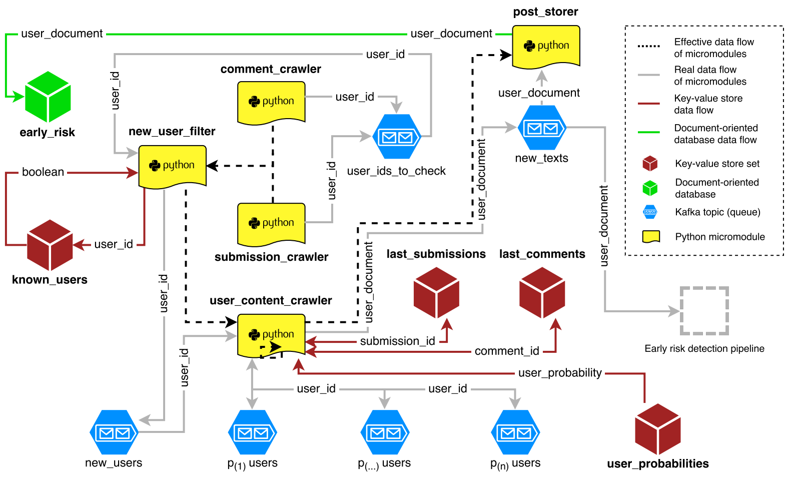


Figure 7. Architecture diagram of the Reddit crawler.

allocated in different priority queues. A single output queue receives the extracted texts.

- **Post storer.** It is in charge of storing texts in a document-oriented database. In addition, these texts will also feed the early prediction pipeline.

There is a clear bottleneck in this crawling topology. Users can be extracted fast since in every iteration multiple users can be reached but the User Content Crawler has to scrape not only the comments and submissions main pages of every user, but a page per submission. This is the only module that requires a high number of instances.

3.2 The Early Risk Detection Pipeline

The early prediction pipeline is a Logistic Regression classifier with L1 regularization, implemented in Python with *scikit-learn*. The classifier is built with a training set of 486 users (83 positive, 403 negative) [10]. Users are represented with a single document, consisting of the concatenation of all their writings. The prediction process has four micromodules (see Figure 8):

- **Text Vectorizer**. It transforms an input text into a vector of token counts.
- **Aggregator**. It accumulates a vector of token counts that represents all submissions and comments of each user. The aggregator merges the current vector of counts with the vector obtained from any new submission or comment.
- **Tf–idf Transformer**. It transforms the aggregated vector of counts to a normalized tf-idf representation.
- **Model Predictor**. It produces the probability of risk of depression for users given their tf–idf representation. In addition, it produces an individual probability for each document which will be stored by the Post Updater micromodule.

The Probability Storer micromodule is fed with user probabilities by the Model Predictor and stores user probabilities. The Alert Manager receives the same input data but stores alerts if the probability is over a certain threshold.

The aggregated vectors and the Python objects (count vectorizer, tf–idf transformer, and the classification model) are stored in the Aerospike store. In this way, the vector updates and the micromodule initializations are fast.

The web interface is composed of three main views that are connected to our HTTP API. In "Alerts" (Figure 9), the fired alerts by the system can be inspected in real time, sorting them by recency or priority (higher confidence first). Alerts correspond to a given user and all the related submissions and comments can be read (ordered by decreasing probability) so the reviewer can tag the alert as a true positive (risk) or false positive (risk free). For each alert, it is also represented the evolution of the user in a line chart (See Figure 11). Each point corresponds to the aggregated probability once a new text (submission or comment) is extracted by the crawler for that user. This view also contains three tabs. The default tab is intended for untagged alerts. The other two tabs contain the tagged alerts: true positives and false positives. The "Datasets" view contains three download options: true positives, false positives and *everything* (See Figure 10). For risk and risk free, the dataset is generated with the users which where tagged at some point with a given tag. In this way, the platform facilitates the creation of labelled collections and benchmarks. Observe also that any classifier can be plugged into the system and, therefore, Catenae supports real-time analysis of users in a number of domains or tasks. Finally, the "Statistics" view shows some information about the running system in real time such as total number of extracted submissions, comments, users and processing speed (See Figure 12).

3.3 Performance Evaluation

In order to test our Catenae use case, we have deployed it in AWS EC2 virtual machines running Amazon Linux 2. AWS gives their users the possibility of running a wide variety of virtual machines in their EC2 infrastructure. In our case, we have used `c5.4xlarge` instances with the following characteristics:

- CPU: Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz with 16 assigned virtual cores.

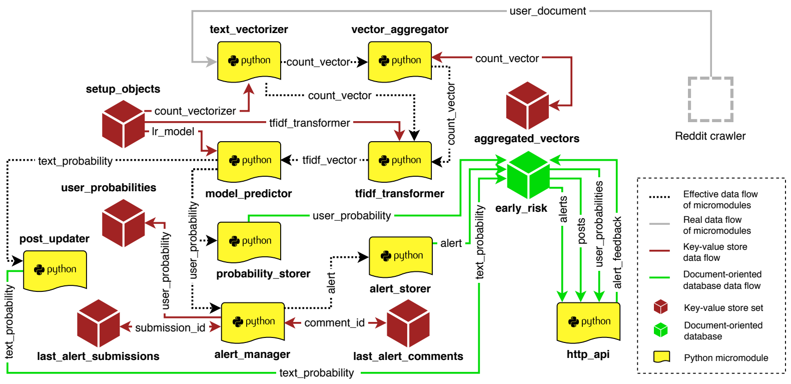


Figure 8. Architecture diagram of the early risk detection pipeline. Main queues omitted for simplicity.

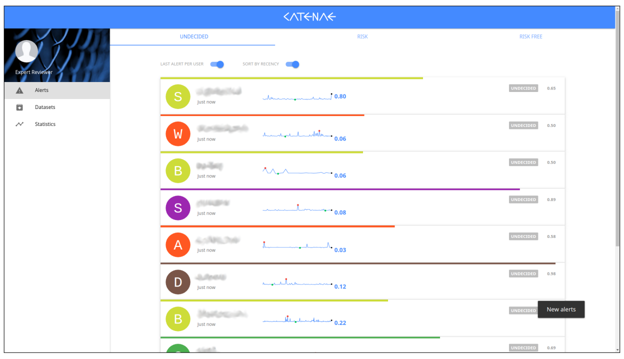


Figure 9. Real-time alert view of web interface (early risk detection of depression).

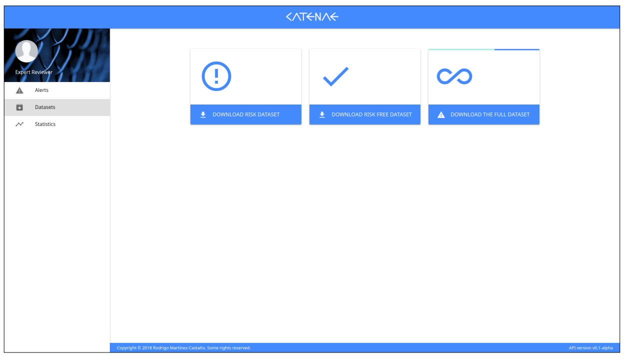


Figure 10. Dataset download view of web interface (early risk detection of depression).

– Memory: 32 GiB.
– Storage: 100 GiB SSD General Purpose volumes (EBS).

According to the specifications provided by AWS, `c5.4xlarge` instances have a dedicated bandwidth up to 2,250Mbps for EBS storage. The general network performance is around 5Gbps.

In these experiments, we consider that the processing is performed in real time when all the extracted texts are classified within seconds, without a growing queue of unprocessed items.

For both topologies, we have defined experimentally the proportion of micromodule instances of each kind in order to remove bottlenecks. In the case of the crawler, the User Content Crawler is the only module that we have considered necessary to scale. For the classifier, the critical micromodules are: Text Vectorizer, Tf-idf Transformer, Vector Aggregator and Model Predictor. We assigned the proportion 2-4-4-1 respectively in order to balance the topology.

The parallelism refers to the number of times that the selected micromodules with the previous proportions are replicated. For instance, a parallelism 2 for the classifier topology would mean 4x Text Vectorizer, 8x Tf-idf Transformer, 8x Vector Aggregator and 2x Model Predictor.

In Figure 13 it can be observed the extracted texts per second for two crawling tests executed in different days. Only one virtual machine was used and the crawler parallelism was configured from 1 to 640. Each configuration was tested for 5 minutes. Both tests behave slightly different due to external factors (Reddit response time varies). Due to this fact, the optimal number of instances of the crawler varies: for higher response times, higher number of instances are needed. It can be observed that for parallelisms higher than 320 units the performance begins to degrade on both tests. However, in Test B, the number of extracted texts recovers due to a reduction in the response time.

In Figure 14, two tests were performed in different days with both the classifier and the crawler. The classifier parallelism was determined depending on fixed crawler parallelisms from 1 to 50 so real-time processing of the retrieved texts was achieved. With higher crawler parallelisms, real-time processing was not possible. Again, each configuration was tested for 5 minutes. The experiment was replicated in Test B with the same parallelism configurations, obtaining a better performance of the crawler while maintaining the processing in real time.

Finally, a 3-node cluster was set up with Apache Kafka. During a three-hour experiment (Figure 15) it can be observed huge fluctuations on the crawling capacity between 500 and 1,000 extracted texts per second. Due to these fluctuations, the queue of unclassified texts grows and decreases accordingly. At least with this scenario,

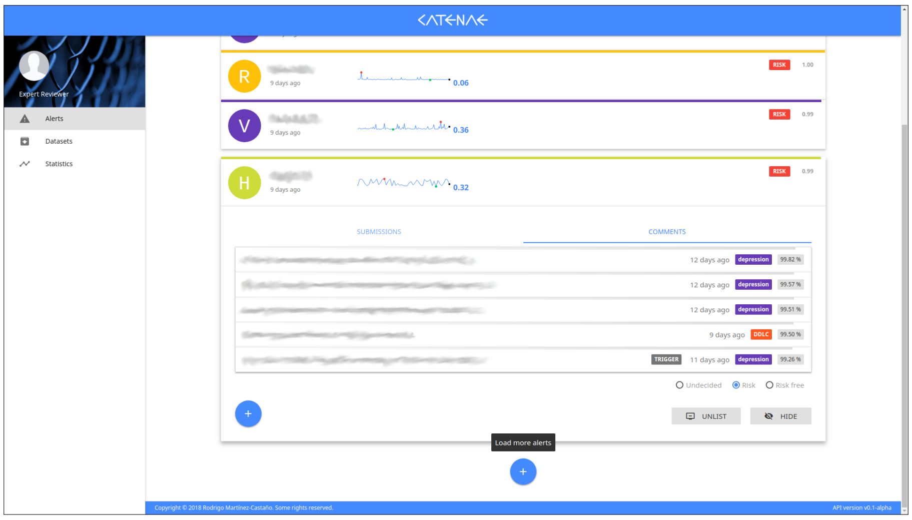
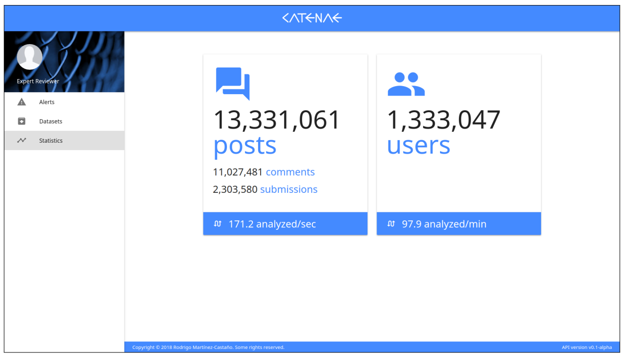


Figure 11. Risk alert view of the web interface (early risk detection of depression).



13,331,061 posts
11,027,481 comments
2,303,580 submissions
171.2 analyzed/sec

1,333,047 users
97.9 analyzed/min

Figure 12. System statistics view of the web interface (early risk detection of depression).

a fixed parallelism for both topologies or even a fixed number of nodes is not an optimal configuration. Topologies should be able to scale up and down automatically for optimal performance and to avoid the waste of resources. Due to this fact, it is necessary to develop a topology controller that automatically balances the parallelism configurations of the running topologies according to their load.

4 COMPATIBILITY WITH OTHER SYSTEMS

Catenae facilitates the creation and deployment of Python topologies. Our library uses Kafka behind the scenes, and, thus, topologies can be connected with other systems using the Kafka client. This connection can be done both for emitting data to the topology (producer) and consuming the output data from it (consumer). It is only necessary to integrate a Kafka producer or consumer in the external system.

Catenae can be connected to other frameworks such as Polypus [12], which is a modular framework that provides the following functionalities: (1) massive text extraction from Twitter, (2) distributed non-relational storage optimized for time range queries, (3) memory-based intermodule buffering, (4) real-time sentiment classification, (5) near real-time keyword sentiment aggregation in time series, (6) a HTTP API to interact with the Polypus cluster and (7) a web interface to analyse results visually. Polypus' main modules are not coded in Python and Polypus does not use Catenae. However, it could be useful to reuse the Twitter crawler as data source for a Catenae topology. The original crawler was connected to two different databases to store the content of the extracted tweets. One of them, Aerospike, was also used to store tweet identifiers in memory. Thereby, checking the existence in the system of a new extracted tweet could be fast while the state is preserved among nodes and between executions. The original outputs for extracted contents were substituted with a Kafka producer (see Figure 16). The Kafka Java API was used to adapt the crawler so

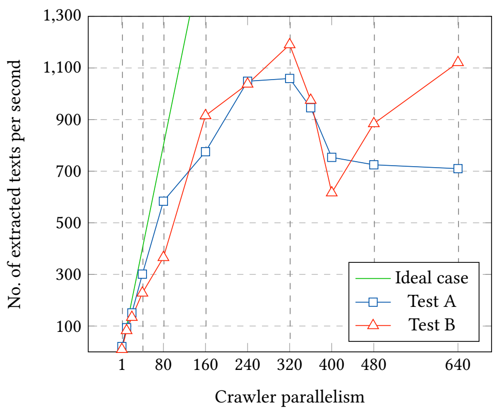


Figure 13. Crawler performance with different parallelism configurations in 1 node.

it can emit tweets to the input topic of a CATENAE topology. Since we are connecting a module written in Java and a Python topology, data should be emitted with basic types. As a matter of fact, we have already taken the first step to connect CATENAE topologies with the POLYPUS's Twitter crawler with a simple Python sentiment analyser.

5 RELATED WORK

MapReduce [7] is a programming model introduced by Google for processing and generating large data sets on a huge number of computing nodes. A MapReduce program execution is divided into two main phases: *map* and *reduce*. The input and output of a MapReduce computation is a list of key-value pairs. Users only need to focus on implementing map and reduce functions. In the map phase, map workers take as input a list of key-value pairs and generate a set of intermediate output key-value pairs, which are stored in the intermediate storage (i.e., files or in-memory buffers). The reduce function processes each intermediate key and its associated list of values to produce a final dataset of key-value pairs. In this way, map tasks achieve data parallelism, while reduce tasks perform parallel reduction. Currently, several processing frameworks support this programming model.

With Apache Spark [16], arbitrary workflows with several processing stages can be defined, so it is a more flexible model than Hadoop MapReduce [3]. It supports several functional programming operations beyond *map* and *reduce*. On the one hand, Spark, written in Scala (JVM language) supports Python, however, non-JVM languages are less efficient since they are not natively executed. Building CATENAE modules requires fewer modifications on the existing code since custom data structures are not needed and the transition is more natural if the modules were already distributed in multiple scripts forming a pipeline. In addition, Python dependency management is not trivial in Spark since libraries must be available in all the nodes of the cluster. With CATENAE, by contrast, each module has its own encapsulated dependencies in a Docker image (including the Python interpreter). The performance and context of a Python module with CATENAE will be the same as when executed locally within a Docker container. A workflow in Spark must be a directed acyclic graph, so data cannot go back to early stages. With our library, as it is focused on real-time applications, cycles could happen (e.g., a module could check a time condition in order to let the data continuing its course).

All these processing technologies require a cluster manager to execute an application (e.g., Apache Hadoop YARN [15], Apache Mesos [9]), whereas CATENAE only requires Docker, a *de facto standard* framework for containers since native clustering functionality is provided with Docker Swarm.

Apache Storm [5] is a framework with the aim of processing streaming data in real time. It requires the definition of *topologies*, which are computational graphs (workflows) where every node represents individual processing tasks. Edges correspond to the flowing data between nodes, which are the responsible of exchange data using *tuples*. Tuples are ordered lists of values, where each value has an assigned name. In particular, nodes exchange non delimited sequences of tuples called *streams*. Every node listens

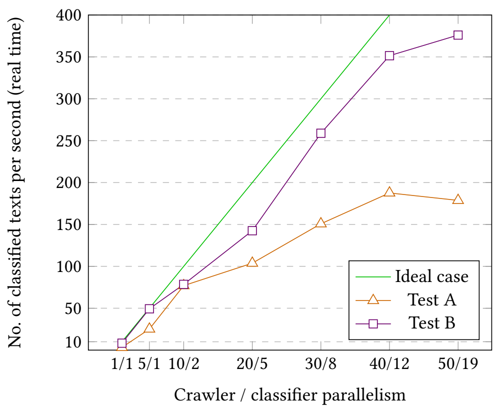


Figure 14. Parallelism configurations to achieve real-time processing in 1 node.

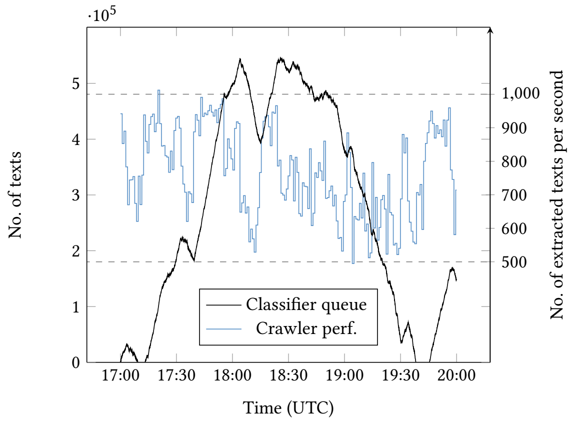


Figure 15. Three-hour experiment on April, 30th 2018 with 3 nodes and 170/0 - 0/50 - 0/50 parallelism (crawler/classifier).

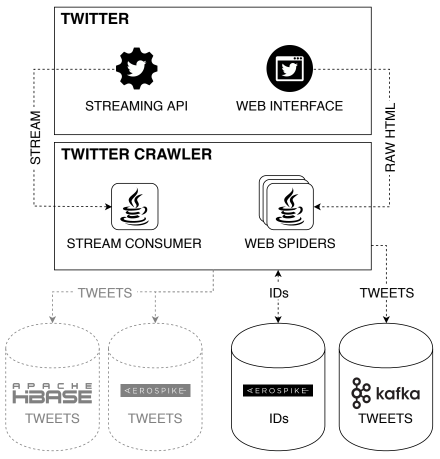


Figure 16. POLYPUS' Twitter crawler.

to one or more streams as input. In the Storm terminology, *spouts* are the sources of a stream within a topology, which usually read data from an external source. Finally, *bolts* are the consumers of the streams and they perform calculus and transformation tasks on the received data. Bolts can emit none, one or more tuples to the output streams. When a topology starts, it stays in execution waiting for new data to process. Storm clusters are composed by two types of nodes: master (Nimbus) and workers (supervisors). Storm uses Apache Thrift [6] and since it can be used in any language, topologies could be defined and submitted from any language. However, non-JVM languages interact with Storm through a JSON-based protocol over stdin/stdout[5] (not very efficient). Again, there is a problem with Python dependencies which will have to be installed in the cluster nodes and the management could be tedious since different topologies could need different versions of the same package. Despite there are libraries such as StreamParse[6] that automatize the installation of dependencies, the problem persists if different modules in the same topology require incompatible dependencies among them.

Kafka is a distributed message broker for high-throughput, low-latency handling of real-time data feeds. Kafka is used as the mechanism to distribute messages between the modules of a Catenae topology. Using Kafka directly requires to manage Kafka producers and consumers for each node of a topology. There exists an official Kafka library for building these kind of real-time topologies: Kafka Streams. However, this library is only available for Java and Scala. During the development of Catenae, a unofficial Kafka Streams library was developed for Python. Our library makes much easier to develop Python topologies since Kafka Streams is a lower level library. Features such as inputs with configurable priority are already implemented and not available in Storm or Kafka. Resource assignment can be easily achieved encapsulating the modules inside Docker containers. Catenae abstracts the developer from Kafka offering a context of minimum intrusion with the existing code.

Catenae topologies can be deployed with Docker [8] containers, which allow us to obtain the benefits of virtualization (isolation, flexibility, portability, agility, etc.) without penalizing the I/O performance considerably. Docker makes use of resource isolation characteristics of the Linux kernel, so independent containers can be executed on the same host. Containers supply a virtual environment with their own space of processes and networks. The containers are built with stacked layers. When a container is in execution, a new writeable layer is created over a set of read-only layers which define a Docker image. The Docker images are always built from a base image, ultimately the *Scratch* image. These images can be easily distributed via the official Docker registry[7], with our own registry or with *tarballs*. Images can be built with a custom scripting language (*dockerfiles*) or by saving the state of a running container.

6 CONCLUSIONS AND FUTURE WORK

Catenae is a Python library which allows the user to build scalable real-time streaming applications easily. Data can flow through the topology in any direction and even backwards. Dependency problems such as library versions or lack on the target cluster are avoided through the use of Docker containers. Furthermore, the Python's GIL problem is bypassed since scalability is achieved by launching more containers for the same module (different processes). The isolation property also allows the user to run scripts written for different Python versions (e.g., Python 2.7, Python 3.6). Basic input data prioritization politics are implemented and they have not to be manually coded. Finally, serialization and deserialization of Python objects is performed automatically. In order to serialize Python objects, our library uses Pickle, which guarantees backwards compatibility across Python releases. There are base containers for building Catenae modules and deploying Kafka.

As future work, new queue handlers will be implemented among the possibility of use custom ones. With Docker as single dependency, deployment of topologies will be handled automatically. Moreover, it will exist the possibility to deploy a disposable Kafka cluster with the topology. The topology manager, in addition to deploying topologies, will let the user to manage a running topology (scaling it up and down). It is usual for real-time systems to have serious changes on their load during its execution, so we have also considered adding support for automated scaling.

ACKNOWLEDGMENTS

This work has been supported by MINECO (TIN2014-54565-JIN, TIN2015-64282-R), Xunta de Galicia (ED431G/08) and European Regional Development Fund.

REFERENCES

[1] About Reddit. 2018. https://www.redditinc.com/. [Online; accessed April, 2018].
[2] Aerospike. 2018. https://www.aerospike.com/. [Online; accessed April, 2018].
[3] Apache Hadoop. 2018. https://hadoop.apache.org/. [Online; accessed April, 2018].
[4] Apache Kafka. 2018. https://kafka.apache.org/. [Online; accessed April, 2018].
[5] Apache Storm. 2018. https://storm.apache.org/. [Online; accessed April, 2018].
[6] Apache Thrift. 2018. https://thrift.apache.org/. [Online; accessed April, 2018].
[7] J. Dean and S. Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation.* 10–10.
[8] Docker. 2018. http://www.docker.com/. [Online; accessed April, 2018].
[9] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation.* USENIX Association, 295–308.
[10] D. Losada and F. Crestani. 2016. A Test Collection for Research on Depression and Language Use. In *Proc. of CLEF.* 28–39.
[11] D. Losada, F. Crestani, and J. Parapar. 2017. eRISK 2017: CLEF Lab on Early Risk Prediction on the Internet: Experimental Foundations. In *Proc. of CLEF.* 346–360.
[12] R. Martínez-Castaño, J. C. Pichel, and P. Gamallo. 2018. Polypus: a Big Data Self-Deployable Architecture for Microblogging Text Extraction and Real-Time Sentiment Analysis. *CoRR* abs/1801.03710 (2018). arXiv:1801.03710
[13] R. Martínez-Castaño, J. C. Pichel, D. E. Losada, and F. Crestani. 2018. A Micromodule Approach for Building Real-Time Systems with Python-Based Models: Application to Early Risk Detection of Depression on Social Media. In *Advances in Information Retrieval.* Springer International Publishing, 801–805.
[14] Reddit on Alexa. 2018. https://www.alexa.com/siteinfo/reddit.com/. [Online; accessed April, 2018].
[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of the 4th Annual Symposium on Cloud Computing (SOCC).* 5:1–5:16.
[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing (HotCloud).* 10–10.

[5] https://storm.apache.org/about/multi-language.html
[6] https://github.com/parsely/streamparse
[7] https://hub.docker.com/

Capítulo 4

Ancoris

Ancoris permite gestionar los recursos disponibles en un clúster y asignarlos a contenedores Docker. El propio gestor se ejecuta dentro de contenedores Docker y está compuesto por dos tipos de procesos que pueden coexistir en la misma máquina: *masters* y *workers*. Los *masters* gestionan los recursos disponibles en el clúster y son los responsables de lanzar contenedores con recursos asignados a través de la API de los *workers*. Los *workers* exponen los recursos disponibles de su *host* cuando son iniciados por primera vez. Tanto las peticiones externas de un cliente como las internas entre *masters* y *workers* se realizan a través de sus APIs HTTP.

El cliente interacciona con el gestor de recursos a través de un nodo *master*. En la Figura 4.1 se puede observar el cuerpo de un ejemplo de solicitud de tarea para un módulo de una topología Catenae. El proceso *master* es responsable de comprobar las solicitudes y encontrar un nodo con los recursos necesarios. Si la operación es exitosa, el proceso *master* interactuará con el proceso *worker* del nodo escogido para lanzar la tarea con los recursos solicitados.

Consul [3] es un sistema distribuido de alta disponibilidad que proporciona un entorno para el descubrimiento y configuración de servicios en un clúster. Entre sus funcionalidades destacan el descubrimiento de servicios (encontrar nuevos proveedores de un servicio dado), *health checking* (comprobar el estado de salud de los nodos y servicios registrados) y un almacenamiento jerárquico de tipo clave-valor. En la Figura 4.2 se puede observar la comunicación básica entre Consul, los procesos *master* y los procesos *worker*.

Actualmente es posible lanzar y destruir tareas, consultar los *logs* de las mismas y consultar los recursos disponibles y estado de las tareas en Consul. Más funcionalidades se añadirán en el futuro como la pausa de tareas y actualización de recursos de tareas en ejecución. Además, una topología Catenae puede ser desplegada mediante ficheros de definición como el de la Figura 4.3 para la topología de *crawling* en Reddit.

```json
{
  "image": "redd/content-crawler",
  "resources": {
    "cores": 1,
    "memory": "128 MiB",
    "swap": "0",
    "volumes": [],
    "devices": [],
    "ports": []
  },
  "opts": {
    "prefered_hosts": ["ancoris1", "ancoris2"],
    "swappiness": 0,
    "network_mode": "host"
  },
  "events": {
    "on_exit": {
      "restart": true,
      "destroy": false
    }
  },
  "args": ["-b", "node1:9092", "-a", "node1:3000", "-p",
           "aerospike:test:setup_objects", "-i",
           "new_users,p1_users,p2_users", "-o", "new_texts"]
}
```

Figura 4.1: Ejemplo de cuerpo JSON empleado para realizar una solicitud a Ancoris.

4.1. Recursos

Cuando un *worker* es inicializado, los recursos configurados en su máquina *host* son registrados en el almacenamiento clave-valor proporcionado por Consul.

Los recursos se definen de forma granular de tal modo que, por ejemplo, un cliente pueda solicitar distintos tipos de almacenamiento (SSD, HDD) e incluso escoger entre varios dispositivos específicos.

Un atributo importante referente a la CPU es el factor de normalización, cuyo propósito es representar el rendimiento por núcleo físico/*hyperthread* en los distintos nodos de un clúster heterogéneo. Por ejemplo, se podría establecer para una CPU menos potente un factor 1, mientras que para otro nodo con una CPU con el doble de potencia se especificaría 2. De este modo, el segundo nodo doblaría el número de núcleos virtuales ofertados.

Aunque la configuración de recursos disponibles se realiza actualmente de forma manual en un fichero de configuración, en próximas versiones los recursos serán detectados de forma automática, incluyendo el factor de normalización a través de un *benchmark* de CPU. Además, será posible solicitar características específicas de memoria y CPU (p. ej. frecuencia de memoria, latencia de memoria, tecnología de la memoria, arquitectura de CPU o modelo de CPU) al lanzar un contenedor.

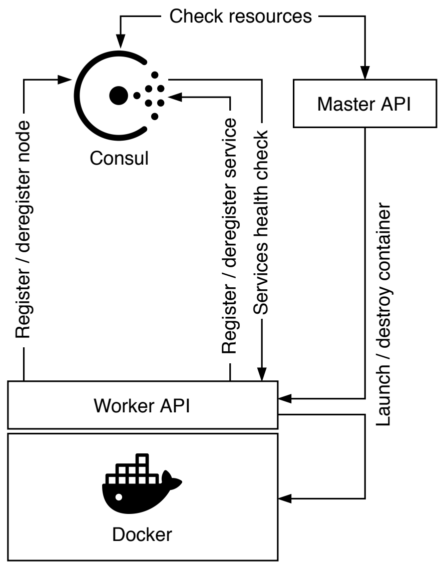

Check resources
Consul
Master API
Register / deregister node
Register / deregister service
Services health check
Launch / destroy container
Worker API
Docker

Figura 4.2: Comunicaciones principales entre ANCORIS y Consul.

4.2. Tareas

Las tareas representan contenedores con recursos asignados. Por defecto un nuevo grupo de tareas se crea de forma automática para cada nueva tarea si un grupo existente no se especifica. Esta característica permitirá al usuario ejecutar acciones en todas las tareas que componen un grupo al mismo tiempo. Cuando una tarea se lanza, los siguientes parámetros deben de ser proporcionados, aunque algunos cuentan con valores por defecto: número de núcleos virtuales, memoria, imagen Docker, argumentos de arranque de la imagen, comportamiento ante la caída del contenedor, puertos a abrir y volúmenes. Es posible suministrar también una lista de preferencia en la que se indican nodos en orden descendiente. Siguiendo esta lista de preferencia y una planificación *round-robin*, se escogerá el primer nodo con los recursos solicitados disponibles. En Consul, las tareas se registran como servicios y se provee un *endpoint* para consultar el estado de las mismas. De este modo es posible consultar el estado de las tareas desde las interfaces de Consul.

Todos los contenedores reciben la IP configurada de la máquina anfitriona. Los puertos solicitados se enlazan con puertos aleatorios disponibles del *host*. También es posible asignar dispositivos más específicos como GPUs a las tareas, aunque esta funcionalidad no está totalmente operativa actualmente.

4.3. Volúmenes

ANCORIS soporta tres tipos de volúmenes: locales, en memoria y distribuidos. Los volúmenes locales son imágenes de disco montadas como dispositivos *loop* y almacenadas en discos locales. Cuando los volúmenes no están en uso, se almacenan comprimidos. Los volúmenes en memoria son volúmenes locales cuyo contenido es copiado a un sistema de archivos temporal que reside en memoria (*tmpfs*), de tal modo que es posible acceder a los datos del volumen con una latencia mucho menor. Los volúmenes distribuidos son enlaces simbólicos a directorios en el interior de un sistema de ficheros distribuido: GlusterFS [5], con el que es posible definir cuotas de uso por directorio.

Los volúmenes pueden crearse con permisos (ro, rw) para la tarea que los monta en caso de estar desmontados y, opcionalmente, para las tareas que lo intenten montar directamente a continuación o que tengan montados volúmenes que forman parte del mismo grupo de volúmenes. Los volúmenes locales solo estarán disponible para una tarea con volúmenes del mismo grupo de volúmenes si (a) el volumen permite ser montado varias veces y (b) el volumen se encuentra montado en el mismo *host* que la tarea en cuestión.

```yaml
---
modules:
  submission_crawler:
    image: catenae/rut-links
    output:
      - user_ids_to_check
    instances: 1

  comment_crawler:
    image: catenae/rut-links
    output:
      - user_ids_to_check
    instances: 1

  new_user_filter:
    image: catenae/rut-links
    input:
      - user_ids_to_check
    output:
      - new_users
    aerospike:
      namespace: test
      set: setup_objects
    instances: 1

  user_content_crawler:
    image: catenae/rut-links
    input:
      - new_users
      - p1_users
      - p2_users
      - p3_users
      - p4_users
      - p5_users
      - p6_users
    output:
      - new_texts
    aerospike:
      namespace: test
      set: setup_objects
    instances: 12

  post_storer:
    image: catenae/rut-links
    input:
      - new_texts

conf:
  kafka:
    address: node1
    port: 9092
  aerospike:
    address: node1
    port: 3000
  ancoris:
    address: node1
    port: 40100
    base_url: /api/v1.0
```

Figura 4.3: Archivo de definición de la topología CATENAE de *crawling* de REDD.

Capítulo 5

Conclusiones

Catenae es una librería Python que se desarrolla para dar soporte a un caso de uso particular desde una perspectiva mucho más amplia, buscando su reutilización en otros proyectos de cualquier índole con necesidades de escalabilidad y procesamiento en tiempo real. Esta librería es particularmente interesante en el campo de la Ciencia de Datos, permitiendo a investigadores escalar sus pruebas de concepto sin desperdiciar una parte importante del proceso en cuestiones tecnológicas tangenciales dada su sencillez. Para dar soporte a esas necesidades de escalabilidad y siguiendo la misma filosofía, se desarrolla Ancoris, un gestor de recursos para clústers con definición granular de los recursos y que permite ejecutar contenedores Docker.

La plataforma Redd para la detección temprana de depresión en Reddit constituye un marco de trabajo completo y totalmente operativo en el que expertos en el área pueden validar o invalidar las alertas emitidas por el sistema. El estado del arte en detección de depresión no permite un proceso totalmente automatizado. Por una parte, el etiquetado de alertas como correctas e incorrectas (falsos positivos) permite mejorar el clasificador de signos de depresión. Dada la magnitud de Reddit, es inviable analizar a todos los usuarios que participan en las distintas comunidades. Por ello, las alertas también funcionan como filtro, siendo posible alzar la voz de alerta ante los casos más serios en tiempo real. El sistema se centra, como se ha comentado, en la detección temprana. Así, se otorga especial relevancia a la rapidez, aún comprometiendo la precisión del sistema a través de la reducción del umbral mínimo de probabilidad para desencadenar una alerta. Además de la gestión de alertas, es posible realizar la descarga de los conjuntos de datos generados a través del etiquetado y consultar estadísticas sobre el desempeño del sistema en tiempo real.

Bibliografía

[1] Apache Kafka. `https://kafka.apache.org/`, 2018. [Online; accessed June, 2018].

[2] Apache Storm. `https://storm.apache.org/`, 2018. [Online; accessed June, 2018].

[3] Consul. `https://www.consul.io/`, 2018. [Online; accessed June, 2018].

[4] Docker. `http://www.docker.com/`, 2018. [Online; accessed June, 2018].

[5] GlusterFS. `https://www.gluster.org/`, 2018. [Online; accessed June, 2018].

[6] D. Losada and F. Crestani. A Test Collection for Research on Depression and Language Use. In *Proc. of CLEF*, pages 28–39, 2016.

[7] D. Losada, F. Crestani, and J. Parapar. eRISK 2017: CLEF Lab on Early Risk Prediction on the Internet: Experimental Foundations. In *Proc. of CLEF*, pages 346–360, 2017.

[8] R. Martínez-Castaño, J. C. Pichel, D. E. Losada, and F. Crestani. A Micromodule Approach for Building Real-Time Systems with Python-Based Models: Application to Early Risk Detection of Depression on Social Media. In *Advances in Information Retrieval*, pages 801–805. Springer, 2018.

[9] R. Martínez-Castaño, J. C. Pichel, D. E. Losada, and F. Crestani. Building python-based topologies for massive processing of social media data in real time. In *5th Spanish Conference in Information Retrieval*. ACM, 2018.

[10] Scikit-Learn. `http://scikit-learn.org/`, 2018. [Online; accessed June, 2018].