Hermes: A distributed-messaging tool for NLP

Ilaria Bordino, Andrea Ferretti, Marco Firrincieli,
Francesco Gullo, Marcello Paris, and Gianluca Sabena

UniCredit, R&D Department, Italy
{ilaria.bordino, andrea.ferretti2, marco.firrincieli,
francesco.gullo, marcello.paris, gianluca.sabena}@unicredit.eu

Abstract. In this paper we present Hermes, a novel tool for natural language processing. By employing an efficient and extendable distributed-message architecture, Hermes is able to fullfil the requirements of large-scale processing, completeness, and versatility that are currently missed by existing NLP tools.

1 Introduction

Text is everywhere. It fills up our social feeds, clutters our inboxes, and commands our attention like nothing else. Unstructured content, which is almost always text or at least has a text component, makes up a vast majority of the data we encounter. *Natural Language Processing* (*NLP*) is nowadays one of the predominant technologies to handle and manipulate unstructured text. NLP tools can tackle disparate tasks, from marking up syntactic and semantic elements to language modeling or sentiment analysis.

The open-source world provides several high-quality NLP tools and libraries [2–5, 9]. These solutions however are typically stand-alone components aimed at solving specific micro-tasks, rather than complete systems capable of taking care of the whole process of extracting useful content from text data and making it available to the user via proper exploration tools. Moreover, most of them are designed to be executed on a single machine and cannot handle distributed large-scale data processing.

In this paper we present Hermes,[1] a novel tool for NLP that advances existing work thanks to three main features. (i) *Capability of large-scale processing*: Our tool is able to work in a distributed environment, deal with huge amounts of text and arbitrarily large resources usually required by NLP tasks, and satisfy both real-time and batch demands; (ii) *Completeness*: We design an integrated, self-contained toolkit, which handles all phases of a typical NLP text-processing application, from fetching of different data sources to producing proper annotations, storing and indexing the content, and making it available for smart exploration and search; (iii) *Versatility*: While being complete, the tool is extremely flexible, being designed as a set of independent components that are fully decoupled from each other and can easily be replaced or extended.

To accomplish the above features, we design an efficient and extendable architecture, consisting of independent modules that interact asynchronously through a message-passing communication infrastructure. Messages are exchanged via distributed queues to handle arbitrarily large message-exchanging rates without compromising efficiency. The persistent storage system is also distributed, for similar reasons of scalability in managing large amounts of data and making them available to the end user.

[1] In Greek mythology Hermes is the messenger of the gods. This is an allusion to our distributed-messaging architecture.

2 Framework

The architecture of Hermes is based on using persisted message queues to decouple the actors that produce information from those responsible for storing or analyzing data. This choice is aimed to achieve an efficient and highly modular architecture, allowing easy replacement of modules and simplifying partial replays in case of errors.

Queues. Message queues are implemented as topics on Kafka[2] (chosen for easy horizontal scalability and minimal setup). There are at present three queues: news, clean-news and tagged-news. Producers push news in a fairly raw form on the news queue, leaving elaborate processing to dedicated workers down the line. Whilst all present components are written in Scala, we leave the road open for modules written in different languages by choosing a very simple format of interchange: all messages pushed and pulled from the queues are simply encoded as JSON strings.

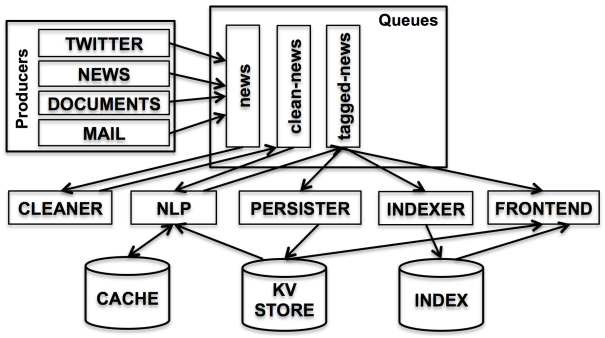

Fig. 1: Depiction of the architecture of Hermes

The main modules of the system are depicted in Figure 1 and described below.

Producers retrieve the text sources to be analyzed, and feed them into the system. Hermes currently implements producers for the following sources. (i) *Twitter*: a long-running producer listening to a Twitter stream and annotating the relative tweets. (ii) *News articles*: this is a generic article fetcher that downloads news from the web following a list of input RSS feeds and can be scheduled periodically. (iii) *Documents*: this producer fetches a generic collection of documents from some known local or public directory. It can handle various formats such as Word and PDF. (iv) *Mail messages*: this producer listens to a given email account, and annotates the received email messages. We use Apache Camel's mail component[3] to process email messages. Each producer performs a minimal pre-processing and pushes the fetched information on the news queue. In the following we call news a generic information item pushed to this queue.

Cleaner. This module consumes the raw news pushed on the news queue, performs the processing needed to extract the pure textual content, and then pushes it onto the clean-news queue for further analysis. The module is a fork of Goose,[4] an article extractor written in Scala.

[2] kafka.apache.org

[3] http://camel.apache.org/mail.html

[4] https://github.com/GravityLabs/goose/wiki

NLP. This module consists of a client and a service. The client listens for incoming news on the clean-news queue, asks for NLP annotations to the service, and places the result on the tagged-news queue. The service is an Akka[5] application providing APIs to many NLP tasks, from the simplest such as tokenization or sentence splitting, to complex operations such as parsing HTML or Creole (the markup language of Wikipedia), entity linking, topic detection, clustering of related news and sentiment analysis. All APIs can be consumed by Akka applications, using remote actors, or via HTTP by other applications. The algorithms we designed for NLP tasks are detailed in Section 3.

Persister and Indexer. In order to retrieve articles, we need two pieces of infrastructure: an index service and a key-value store. For horizontal scalability we respectively choose ElasticSearch and HBase.[6] Two long-running Akka applications listen to the clean-news and tagged-news queues, and respectively index and persist (on a table on HBase) news as soon as they arrive, in their raw or decorated form.

Frontend. The frontend consists of a JavaScript single-page client (written in CoffeeScript using Facebook React) that interacts with a Play application exposing the content of news. The client home page shows annotated news ranked by a relevance function that combines various metrics (see Section 3), but users can also search (either in natural language or using smarter syntax) for news related to specific information needs. The Play application retrieves news from the ElasticSearch index and enriches them with content from the persisted key-value store. It is also connected to the tagged-news queue to be able to stream incoming news to a client using WebSockets.

Monitoring and statistics. A few long-running jobs using Spark Streaming[7] connect to the queues and keep updated statistics on the queue sizes (for monitoring purposes) and the hottest topics and news per author, journal, or topic. Statistics are persisted for further analysis and can be displayed in the front end for summary purposes.

3 Algorithms

In this section we describe the algorithms implemented by the Hermes NLP module.

Entity Linking. The core NLP task in our Hermes is *Entity Linking*, which consists in automatically extracting relevant entities from an unstructured document.

Our entity-linking algorithm implements the TagMe approach [3]. TagMe employs Wikipedia as a knowledge base. Each Wikipedia article corresponds to an *entity*, and the anchor texts of the hyperlinks pointing to that article are the *mentions* of that entity. Given an input text, we first generate all the n-grams occurring in the text and look them up in a table mapping Wikipedia anchortexts to their possible entities. The n-grams mapping to existing anchortexts are retained as mentions. Next, the algorithm solves the disambiguation step for mentions with multiple candidate entities, by computing a disambiguation score for each possible mention-sense mapping and then associating each mention with the sense yielding maximum score. The scoring function attempts to maximize the collective agreement between each candidate sense of a mention and all the other mentions in the text. The relatedness between two senses is computed as Milne and Witten's measure[7], which depends on the number of common in-neighbors.

[5] `akka.io`

[6] `https://www.elastic.co/, https://hbase.apache.org/`

[7] `http://spark.apache.org/streaming/`

Relevance of an article to a target. We use entities extracted from an article to evaluate the relevance of the article to a given target (represented as a Wikipedia entity too). To this end, we compute semantic relatedness between each entity in the text and the target, then we assign the text a relevance score given by the weighted average of the relatedness of all entities. We use a p-average, with $p = 1.8$, so that the contribution of relevant topics gives a boost, but non-relevant topics do not decrease the score much.

Related news. Every few hours (time window customizable), a clustering algorithm is run on the recent news to determine groups of related news. We use the K-means implementation provided by Apache Spark MLlib.[8] The feature vector of an article is given by the first ten entities extracted, weighted by their disambiguation scores.

Document categorization. We categorize news based on the IPTC-SRS ontology.[9] Given a document, we compute the semantic relatedness between each extracted entity and each category (by mapping each category onto a set of editorially chosen Wikipedia entities that best represent them), and assign each entity the category achieving highest similarity. The ultimate category assigned to the article is chosen by majority voting.

Document summarization. Following [6], we build a graph of sentences, where edges between two sentences are weighted by the number of co-occurring words, and rank sentences by Pagerank centrality. The highest ranked sentence is elected as a summary.

Sentiment Analysis. We follow the general idea [1, 8] to have a list of positive and negative words (typically adjectives), and then average the score of the words in the text. The novelty here is that the list of positive and negative words is not fixed. Instead, a list of adjectives is extracted from Wiktionary, and a graph is formed among them, where edges are given by synonyms. Two opposite adjectives are then chosen as polar forms –say good and bad– and for each other adjective a measure of positivity is derived from the distance of an adjective from the two polars.

References

1. L. Chen, W. Wang, M. Nagarajan, S. Wang, and A. Sheth. Extracting Diverse Sentiment Expressions with Target-Dependent Polarity from Twitter. In *ICWSM*, 2012.
2. J. Clarke, V. Srikumar, M. Sammons, and D. Roth. An NLP Curator (or: How I Learned to Stop Worrying and Love NLP Pipelines). In *LREC*, 2012.
3. P. Ferragina and U. Scaiella. TAGME: on-the-fly annotation of short text fragments (by wikipedia entities). In *CIKM*, 2010.
4. E. Loper and S. Bird. NLTK: The Natural Language Toolkit. In *ACL ETMTNLP*, 2002.
5. C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *ACL*, 2014.
6. R. Mihalcea and P. Tarau. Textrank: Bringing order into text. In *EMNLP*, 2004.
7. D. Milne and I. H. Witten. Learning to Link with Wikipedia. In *CIKM*, 2008.
8. M. Thelwall, K. Buckley, and G. Paltoglou. Sentiment strength detection for the social web. *JASIS&T*, 63(1), 2012.
9. M. A. Yosef, J. Hoffart, I. Bordino, M. Spaniol, and G. Weikum. AIDA: an online tool for accurate disambiguation of named entities in text and tables. *PVLDB*, 4(12), 2011.

[8] `http://spark.apache.org/mllib/`

[9] `https://iptc.org/metadata/`