Advancing NLP via a distributed-messaging approach

Ilaria Bordino, Andrea Ferretti, Marco Firrincieli,
Francesco Gullo, Marcello Paris, Stefano Pascolutti, and Gianluca Sabena
UniCredit, R&D Department, Italy
{*ilaria.bordino, andrea.ferretti2, marco.firrincieli,*
francesco.gullo, marcello.paris, stefano.pascolutti, gianluca.sabena}*@unicredit.eu*

***Abstract*—Natural Language Processing (NLP) constitutes a fundamental module for a plethora of domains where unstructured text is a predominant source. Despite the keen interest of both industry and research community in developing NLP tools, current industrial solutions still suffer from two main cons. First, the architectures underlying existing systems do not satisfy critical requirements of large-scale processing, completeness, and versatility. Second, the algorithms typically employed for entity recognition and disambiguation—a core task common to all modern NLP systems—are still not well-suited for deployment in a real industrial environment, for evident issues of efficiency and result interpretability.**

In this paper we present Hermes, a novel NLP tool that overcomes the two main limitations of existing solutions. By employing an efficient and extendable distributed-messaging architecture, Hermes achieves the critical requirements of large-scale processing, completeness, and versatility. Moreover, our tool includes an entity-disambiguation algorithm enhanced with a two-level hashing-based approximation technique to considerably improve efficiency, as a well as a densest-subgraph-extraction method to increase result interpretability.

I. 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. Hence, dealing with text properly is nowadays a critical problem in a variety of application domains.

Natural Language Processing (*NLP*) is one of the predominant technologies to handle and manipulate unstructured text. NLP is defined as the set of methodologies and techniques for automated generation and analysis of pieces of text written in human (natural) language. NLP tools can solve disparate tasks, from marking up syntactic and semantic elements (e.g., named entity recognizers, part-of-speech (POS) taggers, syntactic parsers, semantic role labelers), to language modeling or sentiment analysis.

Motivations. Due to its wide applicability and usefulness, NLP has attracted great interest by both industry and research community in the last decade. Attention has been devoted to developing real-world practical systems [1]–[5] and algorithms for NLP tasks [6]–[9]. However, current industrial solutions are still unsatisfactory, as they exhibit two major limitations.

First of all, existing solutions 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 monolithic, designed to work on a single machine, thus they are not really suitable for distributed environments and large-scale data processing.

The second main limitation is an algorithmic one. A core task for NLP consists in *entity recognition and disambiguation* (ERD), that is recognizing entity mentions in a piece of unstructured text (entity recognition) and correctly linking them to entities of a knowledge base (entity disambiguation) [9]. This is indeed a fundamental task that has been well studied [10]–[15]. However, some critical points make it not yet amenable to be deployed in a real-world industrial context. The first weakness concerns efficiency, as existing entity-linking methods typically take quadratic time in the number of ⟨mention–candidate entity⟩ pairs in the input text. This is a critical issue, which may easily cause ERD to become the bottleneck of the whole system, thus making the aforementioned effort in designing efficient architectures useless. As a second weakness, most of existing ERD approaches just identify entities and present them to the user. However, sole entities, without any structure and/or semantic organization, do not provide consciousness of the meaning hidden in the underlying text, thus resulting not well-explanatory from a user perspective.

Contributions. In this paper we present Hermes,[1] a novel NLP tool that overcomes the aforementioned state-of-the-art limitations. As a first contribution, Hermes advances existing work with the following architectural features:

- *Capability of large-scale processing*. Hermes is able to work in a distributed environment, deal with huge amounts of text and arbitrarily large resources usually required by NLP tasks (e.g., knowledge bases, vocabularies, ground-truth annotations), and satisfy different demands, being them real-time or batch.
- *Completeness*. We design an integrated, self-contained

[1] In Greek mythology Hermes is a messenger of the gods. This is an allusion to the distributed-messaging architecture at the base of the proposed tool.

toolkit, which handles all phases of a typical NLP application, from fetching of different data sources to producing annotations, storing/indexing the content, and making it available for smart exploration/search.
- *Versatility*. While being complete, the proposed tool has flexibility as another of its main qualities. The tool is indeed designed as a set of independent components that are fully decoupled from each other and can be easily 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. Each phase of the NLP process is assigned to one or more specific components that are oblivious of the rest of the architecture. The components communicate with each other via messages exchanged through a queuing system. Each component may act like a *producer* of messages that are pushed to the queue, and/or a *consumer* that reads and processes messages generated by other consumers. The queues are distributed across different machines, so that the system can handle arbitrarily large message-exchanging rates without compromising efficiency. The underlying 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.

Besides the architectural design novelties, Hermes incorporates novel algorithmic solutions for the core NLP task of entity recognition and disambiguation, thus overcoming the aforementioned issues of efficiency and interpretability.

- To overcome the efficiency issue, we devise a two-level hashing approximate solution that achieves high speed-up (up to two orders of magnitude) with limited accuracy loss with respect to state-of-the-art approaches.
- For result interpretability, we define a densest-subgraph-extraction method to identify semantically-coherent groups of entities, which provide the user with the themes of the input text. We add further explanation to each entity group by including extra-document entities that are well-explanatory of that semantic context.

As further pros, all Hermes components are implemented with open-source technologies, and the tool is inherently multi-lingual. Indeed, Hermes employs automatic language-detection methods and its core algorithmic component adopts a language-independent mechanism to solve ERD.

A beta version of the Hermes tool is available at `http://hermes.rnd.unicredit.it:9603`. [2]

Roadmap. The rest of the paper is organized as follows. Section II provides a short overview of the literature on NLP. Section III describes in details architecture and implementation of the proposed tool. In Section IV we discuss the main algorithms that are currently supported. Section V highlights some typical usage scenarios. Section VI presents an experimental evaluation on the main algorithms implemented in the tool. Finally, Section VII concludes the paper and discusses possible future developments.

[2]Please request access credentials by email.

II. Related work

Entity recognition and disambiguation (ERD) is a well-established NLP task that has received great attention in the last decade [9]. It consists of two sub-tasks: *entity recognition*, i.e., identifying entity mentions in a text, and *entity disambiguation* (also known as *entity linking*), which deals with linking candidate entity mentions to actual entities of a given knowledge base (e.g., Wikipedia or Yago).

Entity recognition can be solved retaining n-grams matching some Wikipedia hyper-link anchor text [2], [14], [16], or resorting to sequential prediction without relying on any knowledge base [17], [18].

Existing approaches to entity disambiguation can be broadly classified into *voting approaches* and *graph-based approaches*. In voting methods such as *Wikify* [16], *Illinois Wikifier* [19], *TagMe* [2], and *WAT* [14], each mention-entity pair *votes* for the correct disambiguation of any other mention, contributing to a disambiguation score that is usually computed by taking into account the semantic relatedness between entities [8], [20], [21]. Each mention is eventually assigned to the entity achieving the highest disambiguation score. All entity-disambiguation voting methods have time complexity (at least) quadratic in the number of mention-entity pairs, as each mention-entity pair contributes to the score calculation of every other pair in the text.

Graph-based entity-disambiguation methods construct a graph whose vertices correspond to all mentions and all candidate entities for each mention, while an edge is drawn between every pair of entities and between a mention and its candidate entities. Mention-entity edges are weighted by the similarity of the context surrounding the mention to the candidate entity, whereas entity-entity edges are weighted by semantic relatedness. The disambiguation task is solved by (i) extracting a densest subgraph maximizing some notion of density (e.g., minimum degree), and (ii) post-processing the densest subgraph so as to prune remaining multiple entities ambiguously assigned to the same mention. This category includes approaches such as *AIDA* [11], *Babelfy* [13], *AGDISTIS* [15], and more [10], [12]. Graph-based disambiguation methods take time (at least) quadratic in the number of entity-mention pairs, given that semantic relatedness has to be computed for every pair of candidate entities to build the initial graph.

Further research includes benchmarking frameworks like the *BAT-framework* [22], *Dexter* [23], or *GERBIL* [24], ERD joint with cross-document co-reference resolution [6], methods for short texts like tweets or web queries [25].

Despite the considerable interest devoted to ERD, state-of-the-art methods still suffer from major limitations con-

cerning efficiency and result interpretability. Indeed, as discussed above, all prominent entity-disambiguation methods have time complexity (at least) quadratic in the number of mention-entity pairs. At the same time, all such methods present the identified entities to the user without any semantic organization. We propose solutions to both these issues.

NLP systems currently fall into three main categories: (*i*) suites defining proper APIs (in some programming languages) for common NLP tasks, (*ii*) tools focusing on single NLP micro-tasks, and (*iii*) approaches aimed to unify existing tools/libraries/services.

The first category includes the popular *Stanford Core* [4] and *Natural Language ToolKit (NLTK)* [3], respectively a Java- and a Python-based framework of tools for processing various natural languages, including libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries and easy-to-use interfaces to lexical resources such as WordNet. Similar in spirit to Stanford Core and NLTK are *Apache Lucene and Solr* (`https://lucene.apache.org`) and *Apache OpenNLP* (`http://opennlp.apache.org`). Although all these suites simplify the life of developers of NLP applications, they are clearly far away from being industrial complete systems capable of handling all phases of the NLP process like our Hermes. They are libraries that can be used to develop a system, rather than a system itself.

Solutions in the second category consist of tools for specific NLP tasks, such as *TagMe/WAT* [2], [14], *AIDA* [5], and *Illinois Wikifier* [19], all focusing on entity recognition and disambiguation.Further examples in this category are *DBpedia Spotlight* [26], a tool for annotating mentions of DBpedia resources, and *LingPipe* (`http://alias-i.com/lingpipe`), which employs computational linguistics for categorizing tweets and annotating them with named entities. All these tools are closed solutions that are designed to solve a single NLP task and run on a single machine. They miss the requirements of large-scale processing, completeness, and versatility that are at the base of Hermes.

The latter category comprises attempts to programmatically unifying existing NLP tools/libraries/services. This is the case of *The Curator (Illinois)* [1], an NLP management framework designed to easily incorporate any third-party components, and distribute components across multiple machines. The Curator is a component management system that defines a common interface to use and aggregate different NLP components. It is very different from a complete industrial system like our Hermes.

III. Hermes: architecture and implementation

Hermes is based on persisted, distributed message queues, which allow for decoupling the components producing 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.

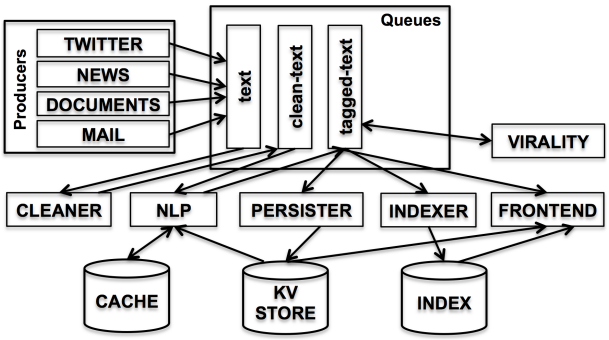

Figure 1: Architecture of Hermes.

Messages are pushed to/read from the queues by a number of independent modules. Each module is a primitive computational entity, which listens to a specific queue, and, upon receiving a message, has the capability of concurrently making local decisions, like invoking some service or determining the actions to be performed when the next message will be received. Each module may act like a *producer* of content to be pushed to the queues, *consumer* of content taken from the queues, or both. Such a queue-based model is characterized by inherent concurrency of computation within and among the various modules, which only interact through direct asynchronous message passing.

Hermes is implemented in Scala, however the architecture at the base of the tool allows for incorporation of components written in any language, as better explained next. All components are implemented by relying on open-source technologies. The main modules of Hermes are depicted in Figure 1 and detailed below.

Queues. We use three distributed message queues, identified by a name recalling the type of handled message: *text*, *clean-text*, *tagged-text*, which respectively refer to raw input text, parsed text (e.g., text resulting from markup removal in HTML pages), and text enriched with extracted entities and other types of semantic annotation. 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 to and pulled from the various queues are encoded as JSON strings.

Message queues in Hermes are implemented as topics on Apache Kafka.[3] We choose Kafka for easy horizontal scalability and minimal setup.

Producers. These are the modules responsible for retrieving the text sources to be analyzed, and feeding them into the system. Each producer fetches information provided by a specific set of sources, performs some minimal processing (if needed), and pushes each piece of information, in a fairly raw content, on the text queue.

[3] `http://kafka.apache.org`

Hermes currently implements producers for the sources listed below, but our flexible architecture allows to plug in any other type of source with minimal effort.

- *Twitter*: a long-running producer, which listens to a Twitter stream and annotates tweets. It uses public API to monitor user-defined accounts and/or hashtags.[4]
- *News articles*: this is a generic article fetcher that downloads news from the Web following a list of RSS feeds (provided by the user), and can be scheduled periodically. To avoid fetching and processing news more than once, news are referenced by URL, and a history of seen URLs is kept in a cache.[5]
- *Documents*: this producer fetches a generic collection of documents from some known local or public file-system directory. It can handle various formats such as Word and PDF, and, similarly to the aforementioned article fetcher, maintains a history of seen documents to avoid unnecessary reprocessing.
- *Mail*: this producer listens to given email accounts and processes all received emails.[6]

Cleaner. This is a consumer module that reads any raw text pushed on the text queue, performs the processing needed to extract the pure textual content from the raw text, and then pushes it onto the clean-text queue to allow for further processing by other components down the line. We implement this module using Goose,[7] a popular Scala article extractor, and Apache Tika,[8] for content extraction and language recognition.

NLP. This module consists of a client and a service. The client listens for incoming texts on the clean-text queue, asks for NLP annotations to the service, and places the result on the tagged-text queue. The service provides APIs to all supported NLP task, from the simplest ones such as tokenization or sentence splitting, to complex operations such as entity recognition and disambiguation, entity explanation, and sentiment analysis. The service is implemented as an Akka application.[9] All APIs can be consumed using other Akka remote actors, or via HTTP by any other application: this testifies once again the versatility of Hermes.

The NLP service constitutes the core algorithmic part of the system. The algorithms supported by this component are detailed in Section IV.

Persister and Indexer. All texts flowing into the system are permanently stored and indexed so as to make them available through search. Persistence and indexing are responsibility of two long-running consumers, which listen to the clean-text and tagged-text queues, and respectively index and persist both texts and tagged texts as soon as they arrive. To keep the architecture horizontally scalable and amenable to large-scale processing, we use a NoSQL distributed key-value store as a storage system, and a distributed multitenant-capable full-text search engine as an indexing service. Specifically, we resort to HBase and Elasticsearch[10] for persistence and indexing, respectively.

Virality. This is a module acting as both consumer and producer. It listens to the tagged-text queue and adds virality information to specific types of tagged texts, such as news or tweets. Currently the tool monitors Twitter, Facebook, and LinkedIn shares, which are retrieved by querying publicly-available APIs.[11] Texts decorated by virality information can be rescheduled, and thus pushed to the tagged-text queue again for further processing by the same module (and by indexer and persister). Rescheduling happens based on rescheduling-frequency and time-to-live parameters that can be set by the user.

Frontend. This component consists of a single-page client that interacts with a REST API exposing the needed textual content. The client home page shows annotated texts ranked by a relevance function described in Section IV. Users can search (either in natural language or using smarter syntax) for texts related to specific information needs. The (JavaScript) single-page application is implemented in ECMAScript 6 using Facebook React; We also make use of d3js to display graphs.[12] The REST API is instead a Play application.[13] Section V provides more details and screenshots of the frontend.

IV. Algorithms

The NLP module of Hermes supports a number of text-processing tasks, whose algorithmic solutions have been designed with a few principles in mind: algorithms should work well for various languages; they should require a minimum amount of supervision; they should be able to cope with short, noisy, unstructured text data that has sheer volume, dynamic nature, and is often only available at query time and thus cannot be preprocessed. The above needs raise a variety of interesting technical challenges, requiring to trade off between efficiency and effectiveness.

A. Entity recognition and disambiguation

The core NLP task in Hermes is *entity recognition and disambiguation* (ERD). This task consists of three steps: (i) *Spotting*, where a set of mentions is detected in the input text, and for each mention a set of candidate entities is retrieved from a given knowledge base; (ii) *Disambiguation*,

[4] https://dev.twitter.com/streaming/public

[5] In our implementation we use Redis (http://redis.io)

[6] We implement this module by using Apache Camel (http://camel.apache.org/mail.html)

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

[8] http://tika.apache.org

[9] http://akka.io

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

[11] http://newsharecounts.com, https://developers.facebook.com/docs/graph-api, https://developer.linkedin.com

[12] http://es6-features.org/, https://facebook.github.io/react, https://d3js.org

[13] https://www.playframework.com

where for each mention associated with multiple candidate entities, a single entity is selected to be linked to it; (*iii*) *Ranking*, where the entities ultimately detected are ranked according to some policy, e.g., annotation confidence.

Basic ERD. The ERD task needs an external knowledge base that acts like an entity catalog. In this work we resort to Wikipedia, which is the go-to resource for most of state-of-the-art ERD approaches, due to its continuously growing size (>5M pages in English), the support for multiple languages, its free availability, and its trading off between a catalog with a rigorous structure but low coverage and a large text collection with unstructured and noisy content.

With Wikipedia as a knowledge base, we resort to the well-established *wikification* approach to ERD, which was first proposed by Miihalcea et al. [16], and then has had a huge success in the NLP community [2], [11], [12], [27]. The main idea of the wikification process is to consider each article in Wikipedia as an entity, and the anchor text of all hyperlinks pointing to that article as the possible mentions for that entity. All entities are organized in a graph structure given by the underlying Wikipedia hyperlink graph, where nodes correspond to entities and an arc from entity e_1 to entity e_2 exists if e_1 contains an hyperlink to e_2 in its body.

In the wikification process spotting is easily performed by generating all n-grams occurring in the input text and looking them up in a table that maps Wikipedia anchor-texts to their possible candidate entities. For disambiguation different options are available (see Section II). In this work we resort to the well-established voting approach of Ferragina et al. [2], dubbed Tagme, which we choose for the ability to deal with short and unstructured text. Disambiguation in Tagme is performed as follows. Given an input text T, let $\mathcal{M}_T$ denote all mentions extracted from T in the spotting step, and, for each mention $m \in \mathcal{M}_T$, let $E(m)$ be the set of candidate entities of m. For a given entity e, let also $in(e)$ denote the in-neighbor entities of e in the Wikipedia hyperlink graph. The main idea is to compute a *disambiguation score* for each ⟨mention–candidate entity⟩ pair $a \mapsto e$ (based on all other ⟨mention–candidate entity⟩ pairs $b \mapsto e'$ within the input text), and ultimately link each mention a to the entity e^* that maximizes that score, i.e., $e^* = \arg\max_e score(a \mapsto e)$. Specifically, a measure of *relatedness* between entities is used. Tagme employs the popular Milne and Witten's measure [8], which computes relatedness as directly proportional to the number of in-neighbors shared by the two entities to be compared:

$$rel(e_1, e_2) = \tag{1}$$
$$= 1 - \frac{\max\{\log|in(e_1)|, \log|in(e_2)|\} - \log|in(e_1) \cap in(e_2)|}{|W| - \min\{\log|in(e_1)|, \log|in(e_2)|\}},$$

where W is the number of Wikipedia articles. The contribution, i.e., the *vote*, of mention b to the ultimate score of a target ⟨mention–candidate entity⟩ pair $a \mapsto e$ is:

$$vote(a \mapsto e \mid b) = \frac{1}{|E(b)|} \sum_{e' \in E(b)} rel(e, e') \Pr(e' \mid b), \tag{2}$$

where $\Pr(e' \mid b)$ is the *commonness* of mapping $e' \mapsto b$, i.e. the ratio between the number of times b' appears in a Wikipedia hyperlink pointing to e' and the number of all Wikipedia hyperlinks pointing to e'. The disambiguation score of $a \mapsto e$ is:

$$score(a \mapsto e) = \sum_{b \in \mathcal{M}_T \setminus \{a\}} vote(a \mapsto e \mid b). \tag{3}$$

Computing $score(a \mapsto e)$ needs to look at all other $b \mapsto e'$ in the input text. Disambiguation thus takes $\mathcal{O}(N^2)$ time, where $N = \sum_{m \in \mathcal{M}_T} |E(m)|$.

The last ERD step, i.e., ranking, is performed by assigning each entity a ranking score defined as a combination (by average) of disambiguation score with another factor dubbed *link probability*, which is the ratio between the number of occurrences of a mention as an anchor text in Wikipedia, and the total number of its occurrences in the whole Wikipedia. Entities with ranking score below a certain threshold are discarded to prune un-meaningful annotations.

Hashing-based ERD. A major weakness of Tagme (and all prominent state-of-the-art ERD methods) is its inefficiency in handling long input documents or real-time constraints. Inefficiency is mainly due to two critical points:

1) Milne and Witten's relatedness measure takes $\mathcal{O}(\min\{|in(e_1)|, |in(e_2)|\})$ time, where e_1 and e_2 are the two entities to be compared (see Equation (1)).
2) The disambiguation score of each ⟨mention–candidate entity⟩ pair has to be computed by looking at all candidate entities of all other mentions in the text. This gives a quadratic time complexity in the number N of ⟨mention–candidate entity⟩ pairs.

We address the above criticalities by introducing two levels of approximation based on hashing.

Issue 1: To speed up the computation of relatedness for a single pair of entities we devise a solution based on *MinHash* [28], which is a principled method for quickly estimating the similarity between two sets. Specifically, let U be a universe of elements, and $h^{(1)}, \ldots, h^{(K)}$ be a set of hash functions, where $h^{(i)} : U \to I \subseteq \mathbb{N}$, $\forall i \in [1..K]$. For any set $S \subseteq U$ and a hash function $h^{(i)}$, let also $h^{(i)}_{min}(S) = \min_{x \in S} h^{(i)}(x)$. The basic MinHash argument consists in estimating the Jaccard similarity $J(A, B)$ between any two sets $A, B \subseteq U$ as:

$$\frac{1}{K} \sum_{i=1}^{K} \mathbb{1}[h^{(i)}_{min}(A) = h^{(i)}_{min}(B)].$$

In our context we first perform an offline step, taking K hash functions $h^{(1)}, \ldots, h^{(K)}$, and computing a *MinHash signature* of each entity e, i.e., a K-dimensional real-valued vector

$$\mathbf{v}_e = \left[h^{(1)}_{min}(in(e)), \ldots h^{(K)}_{min}(in(e))\right].$$

This steps takes $\mathcal{O}(Kn + K\sum_e deg(e)) = \mathcal{O}(K(n+m))$, where n and m denote the number of entities and arcs in the Wikipedia graph.

As far as the operations to be performed online, we estimate (i) $J(in(e_1), in(e_2))$ as $\frac{1}{K}\sum_{i=1}^{K} \mathbb{1}[\mathbf{v}_{e_1}(i) = \mathbf{v}_{e_2}(i)]$, and (ii) $|in(e_1) \cap in(e_2)|$ as $\frac{J}{1+J}(|in(e_1)|+|in(e_2)|)$. The time complexity of the online phase becomes $\mathcal{O}(K)$ (rather than $\mathcal{O}(\min\{deg(e_1), deg(e_2)\})$). This method allows for speeding up all computations where the in-neighborhoods of the involved entities have size $> K$ (clearly, if $\min\{deg(e_1), deg(e_2)\} < K$ the exact method is employed).

Issue 2: To speed-up the quadratic-time disambiguation-score computation we employ a method based on *Locality Sensitive Hashing* (LSH) [29]. LSH is a technique to hash items so that similar items map to the same buckets with high probability. The idea is to apply LSH on the MinHash signatures of the Wikipedia entities so as to bucketize entities into groups exhibiting highly similar in-neighborhoods. This way, the computation of the disambiguation score can be simplified by comparing each candidate entity of a mention with the only entities of other mentions that have at least one bucket in common with it, i.e., the entities that contribute more on the score computation. The entities that have no common buckets with the one taken into consideration are excluded from the voting.

Particularly, we have an offline phase where for each entity e the e's MinHash signature is split into L groups of equal size. For each group $i \in [1..L]$, an LSH bucket $b_i(e)$ is computed. This phase takes $\mathcal{O}(Ln\frac{K}{L}) = \mathcal{O}(Kn)$ time. Given an input text T, the online phase instead requires to (i) retrieve LSH buckets for all entities in T; (ii) compute an inverted index: for each bucket b, $entities(b) = \{e \mid b(e) \in lsh(e)\}$; and (iii) approximate $score(a \mapsto e)$ as:

$$\frac{1}{|E(b)|}\sum_{e' \in buckets(e)} rel(e, e') \Pr(e' \mid b).$$

Note that the aforementioned inverted index refers to only the entities contained in the input text. Thus, it has to be computed online, to avoid wasting time in scanning unnecessary entities included in the buckets. While the worst-case time complexity of the LSH-based disambiguation-score computation remains quadratic, in practice a large benefit is expected, as only comparisons between entities in the same bucket are performed. This is testified by our experiments reported in Section VI.

B. Topic extraction, labeling and explanation

The ERD algorithm described above, with the two-level hashing optimization, is able to efficiently tackle the issue of extracting entities from any input text. Once the entities have been extracted, Hermes could just present them to the user as most of existing ERD approaches do. However, displaying only entities without any relational structure and/or semantic organization, often reveals insufficient to help the user gather an immediate comprehension of the semantic meaning hidden in the underlying text. Hermes overcomes this limitation, which affects most of state-of-the-art ERD tools, by enhancing its core entity-linking algorithm with three critical features: (i) *topic extraction*, (ii) *topic labeling*, and (iii) *topic explanation*.

Topic extraction. We devise a densest-subgraph-extraction method [30] to identify semantically-coherent groups of entities, so as to provide the user with the different themes of an input text. We start by building a complete graph of the entities within the document, weighting the arc connecting any two entities by Milne and Witten's relatedness. Next we compute the densest subgraph (maximizing a notion of density based on average weighted degree), remove it from the graph and repeat the procedure until the graph is empty. Each extracted subgraph identifies a topic of the input text.

Topic labeling. Each topic extracted from a document is labeled with a category from the IPTC-SRS ontology, [14] an open-standard hierarchy for news categorization. We consider the top-level categories and manually map each of them onto a set of Wikipedia entities that best represent it. Given a topic, we compute the similarity between each entity in its subgraph and each category, as the average relatedness to the entities representing the category. We assign each entity in the topic subgraph the category achieving maximum similarity. Each entity votes for its category, and each topic is assigned the category obtaining the majority of votes.

Topic explanation. We enrich each entity group by including extra-document entities that are well-explanatory of the semantic context at hand. For each entity in a topic subgraph we include a subset of the entities that are its in- or out-neighbors in the Wikipedia graph, selected as those maximizing the average semantic relatedness to the other entities in the topic. This method was a natural and efficient choice for Hermes, given that our core ERD module relies on a graph representation of Wikipedia. The vast literature about exploratory entity search describes other approaches for building explanatory graphs around query entities, for example including rich semantic relations

[14] https://iptc.org/metadata/

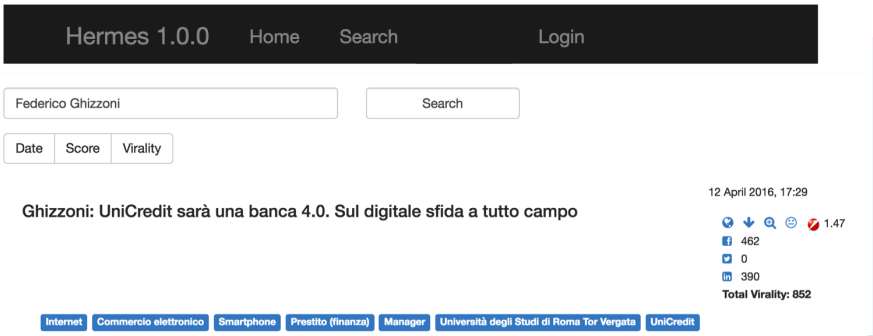

Figure 2: Hermes frontend: search tab

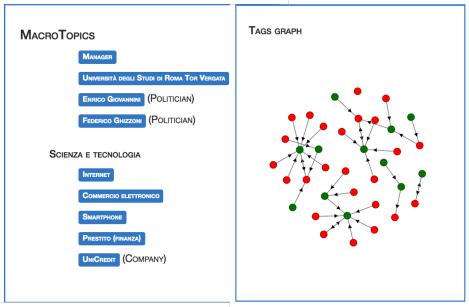

Figure 3: Hermes frontend: text details

derived from domain-specific knowledge bases [31], [32]. In future releases we will investigate whether our method can be improved with ideas suggested by these approaches.

C. Additional features

The NLP module of Hermes supports a few additional features, for which we rely on state-of-the-art solutions: relevance of an article to a target (computed as average semantic relatedness between each entity extracted from the text and the target entity), document summarization based on the work of Mihalcea and Tarau [33], and sentiment analysis performed with the Stanford Core NLP framework [4].

V. Frontend

The frontend of Hermes is comprised of two tabs, *Home* and *Search*. The Home tab shows the most relevant texts that have been collected and annotated so far (the time horizon is customizable). Texts are ranked according to the relevance to a target (see Section IV). In the current implementation the target is UniCredit, but this is customizable too. The Search tab (Figure 2) allows the user to search for documents satisfying specific criteria. By clicking on the down arrow at the right corner of a text (Figure 3), further details are shown, including topics labeled with categories, and an entity explanation subgraph (see Section IV).

VI. Experiments

We tested the performance of our ERD algorithm on three public datasets.

TAGME. The first dataset is one of the datasets used for evaluation in the work of Ferragina et al. [2]. The original dataset is a collection of 186 001 text fragments drawn from a Wikipedia snapshot dating to November 2009. Each fragment is annotated with mentions and the corresponding entities. We use a sample of 500 documents, with on average 4.51 annotations per document.

N3 Collection. The other two datasets that we consider are the two English corpora in the N3 collection [34], dubbed *Reuters-128* and *RSS-500*. The former is a set of 128 economic news articles, with an average of 5.08 mentions per document. The latter is a corpus of 500 texts (with an average of 1.47 mentions per document) gathered by using a list of RSS feeds of major worldwide newspapers.

We tested our ERD method on such data, comparing three versions of the algorithm: (i) EXACT, (ii) LSH, and (iii) LSH-MINHASH (LMH). EXACT is the algorithm of Ferragina et al. [2], where no approximation is used. LSH considers only one level of approximation, using LSH to bucketize Wikipedia entities into groups that have highly similar in-neighborhoods. However, the relatedness between any two entities is still computed exactly (no MinHash approximation is employed, see Section IV). LSH-MINHASH (LMH) is our most optimized algorithm, which uses both levels of approximation described in Section IV. We apply LSH to restrict the computation of the voting scheme to the only entities that have at least one bucket in common with a given entity. Furthermore, we replace the exact computation of Milne and Witten's relatedness between two entities with a version that approximates the size of the intersection of their in-neighborhoods by considering their MinHash signatures.

Parameters. We ran LSH and LMH with several parameter configurations. For N, the number of MinHash functions used to compute the signature of each entity, we considered $N = 100$, $N = 200$, and $N = 500$. For S, i.e. the size of each LSH band (meaning that we divided each signature of size N into L bands of size S), we considered $S = 10$, $S = 20$, $S = 50$ and $S = 100$. To avoid building bands of too small size, we used up to $S = 20$ for $N = 100$, and up to $S = 50$ for $N = 200$. In all cases $L = N/S$.

Metrics. For each competing algorithm and for each configuration of parameters, we evaluated performance in terms of both accuracy of ERD (Precision, Recall, F-Measure) and running time needed to perform ERD. For each dataset and parameter configuration the metrics were computed averaging over all the documents in the collection. For robustness, running times of each competing method were averaged over 10 runs. Results for the three datasets are reported in Tables I, II, and III.

Results. The usage of LSH and MinHash signatures does not seem to affect ERD accuracy: precision and recall are always very close to the values obtained with the exact version of the algorithm, with a small drop that is obviously expected. At the same time, we notice that our approximated algorithms achieve a remarkable gain regarding running time: the disambiguation time per document is reduced by

Table I: Results for TAGME Dataset (avg over 10 runs)

		N = 100 S = 10 L = 10		N = 100 S = 20 L = 5		N = 200 S = 10 L = 20		N = 200 S = 20 L = 10		N = 200 S = 50 L = 4		N = 500 S = 10 L = 50		N = 500 S = 20 L = 25		N= 500 S = 50 L = 10		N = 500 S = 100 L = 5	
	Exact	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH
P	.691	.660	.594	**.675**	**.675**	.638	.638	.668	.534	.681	.681	.615	.613	.645	.451	.589	.589	.651	.651
R	.556	.536	.483	**.550**	**.550**	.520	.520	.542	.433	.556	.556	.502	.501	.524	.367	.480	.480	.532	.532
FM	.600	.574	.517	**.589**	**.589**	.557	.557	.581	.465	.595	.595	.536	.534	.562	.393	.514	.514	.569	.569
Td (ms)	663	171	31	**143**	**32**	147	59	155	37	142	36	196	122	173	61	131	50	146	67

Table II: Results for RSS Dataset (avg over 10 runs)

		N = 100 S = 10 L = 10		N = 100 S = 20 L = 5		N = 200 S = 10 L = 20		N = 200 S = 20 L = 10		N = 200 S = 50 L = 4		N = 500 S = 10 L = 50		N = 500 S = 20 L = 25		N= 500 S = 50 L = 10		N = 500 S = 100 L = 5	
	Exact	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH
P	.447	.533	.533	**.542**	**.542**	.542	.542	.538	.484	.546	.437	.517	.465	.517	.517	.533	.533	.488	.542
R	.440	.525	.520	**.533**	**.533**	.533	.533	.529	.476	.538	.430	.508	.458	.508	.508	.525	.525	.480	.533
FM	.442	.528	.528	**.536**	**.536**	.536	.536	.532	.479	.540	.432	.511	.460	.511	.511	.528	.528	.483	.536
Td (ms)	44	87	23	**52**	**21**	58	34	54	25	50	19	71	62	63	48	55	35	50	33

Table III: Results for Reuters Dataset (avg over 10 runs)

		N = 100 S = 10 L = 10		N = 100 S = 20 L = 5		N = 200 S = 10 L = 20		N = 200 S = 20 L = 10		N = 200 S = 50 L = 4		N = 500 S = 10 L = 50		N = 500 S = 20 L = 25		N= 500 S = 50 L = 10		N = 500 S = 100 L = 5	
	Exact	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH	LSH	LMH
P	.565	.524	.472	.513	.461	.483	.434	**.523**	**.523**	.539	.485	.437	.393	.466	.464	.507	.506	.536	.536
R	.467	.431	.388	.421	.379	.394	.354	**.429**	**.429**	.443	.399	.356	.320	.382	.380	.416	.415	.441	.441
FM	.504	.465	.419	.455	.410	.427	.384	**.464**	**.464**	.479	.431	.386	.347	.413	.411	.450	.449	.476	.476
Td (ms)	2923	454	28	308	29	387	55	**304**	**46**	387	34	321	123	310	97	324	69	275	61

2 and 1.5 orders of magnitude respectively in the case of *Reuters* and *TAGME*, and of 4 times in the case of *RSS* (which however consists of very short documents).

VII. Conclusion and future work

We have presented Hermes, a novel NLP tool that overcomes the main limitations of existing industrial solutions. Based on an efficient and extendable message-passing architecture, Hermes achieves large-scale processing, completeness, and versatility.

A number of improvements to the current system can be considered. We plan to strengthen the architecture by adding a dedicated queue to handle communication between producers. We intend to enrich the front-end with statistics about hot topics and authors, and to incorporate feedback from the users about tagging accuracy.

References

[1] 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.

[2] P. Ferragina and U. Scaiella, "TAGME: on-the-fly annotation of short text fragments (by wikipedia entities)," in *CIKM*, 2010.

[3] E. Loper and S. Bird, "NLTK: The Natural Language Toolkit," in *ACL ETMTNLP*, 2002.

[4] 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.

[5] 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*, vol. 4, no. 12, 2011.

[6] S. Dutta and G. Weikum, "C3EL: A joint model for cross-document co-reference resolution and entity linking," in *EMNLP*, 2015.

[7] S. Dutta and G. Weikum, "Cross-document co-reference resolution using sample-based clustering with knowledge enrichment," *TACL*, vol. 3, 2015.

[8] D. Milne and I. H. Witten, "Learning to Link with Wikipedia," in *CIKM*, 2008.

[9] W. Shen, J. Wang, and J. Han, "Entity linking with a knowledge base: Issues, techniques, and solutions," *TKDE*, vol. 27, no. 2, 2015.

[10] X. Han, L. Sun, and J. Zhao, "Collective entity linking in web text: A graph-based method," in *SIGIR*, 2011.

[11] J. Hoffart, M. A. Yosef, I. Bordino, H. Fürstenau, M. Pinkal, M. Spaniol, B. Taneva, S. Thater, and G. Weikum, "Robust disambiguation of named entities in text," in *EMNLP*, 2011.

[12] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti, "Collective annotation of wikipedia entities in web text," in *KDD*, 2009.

[13] A. Moro, A. Raganato, and R. Navigli, "Entity linking meets word sense disambiguation: a unified approach," *TACL*, vol. 2, 2014.

[14] F. Piccinno and P. Ferragina, "From tagme to wat: A new entity annotator," in *ERD*, 2014.

[15] R. Usbeck, A.-C. Ngonga Ngomo, R. Michael, S. Auer, D. Gerber, and A. Both, "AGDISTIS - agnostic disambiguation of named entities using linked open data," in *ECAI*, 2014.

[16] R. Mihalcea and A. Csomai, "Wikify!: Linking documents to encyclopedic knowledge," in *CIKM*, 2007.

[17] "Stanford NER, `nlp.stanford.edu/software/crf-ner.shtml`." [Online]. Available: nlp.stanford.edu/software/CRF-NER.shtml

[18] L. Ratinov and D. Roth, "Design challenges and misconceptions in named entity recognition," in *CoNLL*, 2009.

[19] L. Ratinov, D. Roth, D. Downey, and M. Anderson, "Local and global algorithms for disambiguation to wikipedia," in *ACL*, 2011.

[20] D. Ceccarelli, C. Lucchese, S. Orlando, R. Perego, and S. Trani, "Learning relatedness measures for entity linking," in *CIKM*, 2013.

[21] J. Hoffart, S. Seufert, D. B. Nguyen, M. Theobald, and G. Weikum, "Kore: keyphrase overlap relatedness for entity disambiguation," in *CIKM*, 2012.

[22] M. Cornolti, P. Ferragina, and M. Ciaramita, "A framework for benchmarking entity-annotation systems," in *WWW*, 2013.

[23] D. Ceccarelli, C. Lucchese, S. Orlando, R. Perego, and S. Trani, "Dexter: An open source framework for entity linking," in *ESAIR*. ACM, 2013.

[24] R. Usbeck et al., "GERBIL – general entity annotation benchmark framework," in *WWW*, 2015.

[25] R. Blanco, G. Ottaviano, and E. Meij, "Fast and space-efficient entity linking for queries," in *WSDM*, 2015.

[26] J. Daiber, M. Jakob, C. Hokamp, and P. N. Mendes, "Improving efficiency and accuracy in multilingual entity extraction," in *I-Semantics*, 2013.

[27] S. Cucerzan, "Large-scale named entity disambiguation based on wikipedia data," in *EMNLP-CoNLL*, 2007.

[28] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *JCSS*, vol. 60, no. 3, 2000.

[29] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998.

[30] M. Charikar, "Greedy approximation algorithms for finding dense components in a graph," in *APPROX*. Springer-Verlag, 2000.

[31] G. Kasneci, S. Elbassuoni, and G. Weikum, "Ming: Mining informative entity relationship subgraphs," in *CIKM*. ACM, 2009, pp. 1653–1656.

[32] S. Yogev, H. Roitman, D. Carmel, and N. Zwerdling, "Towards expressive exploratory search over entity-relationship data," in *WWW Companion*. ACM, 2012.

[33] R. Mihalcea and P. Tarau, "Textrank: Bringing order into text," in *EMNLP*, 2004.

[34] M. Röder, R. Usbeck, S. Hellmann, D. Gerber, and A. Both, "N^3 - A collection of datasets for named entity recognition and disambiguation in the NLP interchange format," in *LREC*, 2014.