# Contents

1 Overview .................................................. 1
   1.1 Motivation ............................................. 1
   1.2 Tasks .................................................. 1
   1.3 Summary .............................................. 4
   1.4 How to cite ........................................... 5
   1.5 License and contact ................................. 5

2 Install, build and run ..................................... 7
   2.1 Creating a new Knowledge Base version from new Wikidata and Wikipedia dumps ........... 8

3 *entity-fishing* Console .................................. 9

4 *entity-fishing* REST API .................................. 13
   4.1 *entity-fishing* query processing .................. 13
   4.2 Knowledge base concept retrieval .................. 25
   4.3 Term Lookup .......................................... 28
   4.4 Language identification .............................. 28
   4.5 Sentence segmentation ............................... 29
   4.6 Customisation API .................................... 31

5 Evaluation ................................................. 35
   5.1 Datasets for long texts ............................... 35
   5.2 Evaluation commands ................................. 36
   5.3 Generation of pre-annotated training/evaluation data ........................................ 36
   5.4 References ............................................ 36

6 Train and evaluate ........................................ 39
   6.1 Training with Wikipedia ............................... 39
   6.2 Evaluation with Wikipedia ............................. 40
   6.3 Training with an annotated corpus .................... 40
   6.4 Creating entity embeddings ........................... 40

7 License and contact ....................................... 43
1.1 Motivation

One of the backbone of the activities of scientists regarding technical and scientific information at large is the identification and resolution of specialist entities. This could be the identification of scientific terms, of nomenclature-based expressions such as chemical formula, of quantity expressions, etc. It is considered that between 30 to 80% of the content of a technical or scientific document is written in specialist language (Ahmad, 1996). Researchers in Digital Humanities and in Social Sciences are often first of all interested in the identification and resolution of so-called named entities, e.g. person names, places, events, dates, organisation, etc. Entities can be known in advance and present in generalist or specialized knowledge bases. They can also be created based on open nomenclatures and vocabularies and impossible to enumerate in advance.

The entity-fishing services try to automate this recognition and disambiguation task in a generic manner, avoiding as much as possible restrictions of domains, limitations to certain classes of entities or to particular usages.

1.2 Tasks

entity-fishing performs the following tasks:

- entity recognition and disambiguation against Wikidata in a raw text, partially-annotated text segment,
• entity recognition and disambiguation against Wikidata at document level, for example a PDF with layout positioning and structure-aware annotations,

with previous failed attempts at cardiac resynchronization therapy

Anoop K. Shetty1,2, Simon G. Duckett1,2, Julian Bostock3, Eric Rosenthal1, and C. Aldo Rinaldi1,2

1Cardiology Department, Guy’s and St Thomas’ Hospital NHS Foundation Trust, London, UK; and 2King’s College London, Westminster Bridge Road, London SE1 7EH, UK.

Aims
Problems with implenting a ventricle (LV) lead during cardiac resynchronization therapy (CRT) procedures are not uncommon and may occur for a variety of reasons including: intrinsic nerve stimulation (PNS) and high capture thresholds. We aimed to perform successful CRT in patients with previous failed lead problems using the multiple pacing configurations available with the SmartVest® Quartet model 169Q quadripolar LV lead to avoid over-capture or high capture thresholds.

Methods and results
Four patients with previous failed attempts at LV lead implantation underwent a further attempt at CRT using a Quartet lead. In all four cases, successful CRT was achieved using a Quartet lead placed in a branch of the coronary sinus. Problems with PNS or high capture thresholds were seen in all four patients but were successfully overcome. Satisfactory lead parameters were seen at discharge and at short-term follow-up.

Conclusion
The Quartet lead allowed four different pacing vectors to be used and may overcome common pacing problems because of the multiple pacing configurations available. Problems with either PNS or unsatisfactory pacing parameters experienced during CRT may be resolved simply by changing the pacing configuration using this quadripolar lead system.

Keywords
CRT • Quadrupolar lead • PNS • Intrinsic nerve stimulation (PNS) • Failed CRT

 Introduction
Cardiac resynchronization therapy (CRT) improves heart failure symptoms and mortality in patients with heart failure (HF) and left ventricular (LV) dysfunction and is a broad class of therapy aimed at improving cardiac contractility and reducing symptoms. Patients with CRT require LV lead implantation and can be categorized into two groups: those with a history of successful CRT implantation and those with a history of failed CRT implantation. Failed CRT implantation occurs in up to 11% of cases.1 This may be due to an inability to stimulate the coronary sinus (CS) region, an inability to pass the LV lead into a CS branch, unsatisfactory pacing parameters, or other technical limitations. In a study of 190 consecutive patients undergoing CRT, Bri et al. showed that clinically relevant PNS occurred in 10% of patients in CRT patients undergoing CRT and that no recurrence was noted in those patients for whom the LV lead was placed at pacing sites more associated with reverse remodeling. In the aforementioned study, if patients required an LV lead revision or were turned off and reprogrammed (the capability to program both the pacemaker and the CRT device) in a patient with CRT, CRT was described as a possible solution to the problem of CRT. Indeed, other studies2,3 have shown that the availability of multiple pacing configurations may overcome problems with high capture thresholds and ENS.

• search query disambiguation (the short text mode) - below disambiguation of the search query “concrete pump sensor” in the service test console,
Concrete is a composite material composed of fine and coarse aggregate bonded together with a fluid cement (cement paste) that hardens (cures) over time. In the past, lime putty, but sometimes with other hydraulic cement, such as a calcium aluminate cement or with Portland cement to form Portland cement concrete (for its visual resemblance to Portland stone). Many other non-cementitious types of concrete exist with other methods of binding aggregate together, including asphalt concrete with a bitumen binder, which is frequently used for road surface, and polymer concrete that use polymers as a binder.

A concrete pump is a machine used for transferring liquid concrete by pump. There are two types of concrete pumps. The first type of concrete pump is attached to a truck or long units are on semi-trailers. It is known as a boom concrete pump because it uses a remote-controlled articulating robot arm (called a boom) to place concrete accurately. Boom pumps are used on most of the larger construction projects as they are capable of pumping at very high volumes and because of the labour saving nature of the placing boom. They are a revolutionary alternative to line-concrete pumps.

A pump is a device that moves fluids (liquid or gas), or sometimes slurry, by mechanical action, typically converted from electrical energy into hydraulic energy. Pumps can be classified into three major groups according to the method they use to move the fluid: direct lift, displacement, and gravity pumps. Pumps operate by some mechanism (typically reciprocating or rotary), and consume energy to perform mechanical work moving the fluid. Pumps operate via many energy sources, including manual operation, electricity, engines, or wind power, and come in many sizes, from microscopic for use in medical applications, to large industrial pumps.

A sensor is a device, module, machine, or subassembly whose purpose is to detect events or changes in its environment and send the information to other electronics, frequently a computer processor. A sensor is always used with other electronics.

- weighted term vector disambiguation (a term being a phrase),
Supervised machine learning is used for the disambiguation, based on Random Forest and Gradient Tree Boosting exploiting various features, including word and entity embeddings. Training is realized exploiting Wikipedia, which offers for each language a wealth of usage data about entity mentions in context. Results include in particular Wikidata.

1.3 Summary

interactive disambiguation in text editing mode.
identifiers and, optionally, statements.

The API also offers the possibility to apply filters based on Wikidata properties and values, allowing to create specialised entity identification and extraction (e.g. extract only taxon entities or only medical entities in a document) relying on the current 37M entities and 154M statements present in Wikidata.

The tool currently supports English, German, French, Spanish and Italian languages (more to come!). For English and French, a Name Entity Recognition based on CRF grobid-ner is used in combination with the disambiguation. For each recognized entity in one language, it is possible to complement the result with crosslingual information in the other languages. A nbest mode is available. Domain information are produced for a large amount of entities in the technical and scientific fields, together with Wikipedia categories and confidence scores.

The tool is developed in Java and has been designed for fast processing (at least for a NERD system, 500-1000 words per second on a medium-profile linux server single thread or one PDF page of a scientific articles in 1-2 seconds), with limited memory (at least for a NERD system, here 3GB of RAM) and to offer relatively close to state-of-the-art accuracy (more to come!). A search query can be disambiguated in 1-10 milliseconds. entity-fishing uses the very fast SMILE ML library for machine learning and a JNI integration of LMDB as embedded database.

### 1.4 How to cite

### 1.5 License and contact

Distributed under Apache 2.0 license. The dependencies used in the project are either themselves also distributed under Apache 2.0 license or distributed under a compatible license.

Main author and contact: Patrice Lopez (&lt;patrice.lopez@science-miner.com&gt;)

*entity-fishing* is developed by SCIENCE-MINER since 2015, with contributions of Inria Paris (2017-2018).

Inria contributors were supported by the H2020 HIRMEOS, IPERION-CH and DESIR EU projects.
Install, build and run

*entity-fishing* requires JDK 1.8 or higher. It supports Linux-64 (preferred) and Mac OS environments. Below, we make available the LMDB binary data for these two architectures, taking into account that only index data for Linux are up-to-date.

Running the service requires at least 3GB of RAM for processing text inputs, but more RAM will be exploited if available for speeding up access to the compiled Wikidata and Wikipedia data (including Wikidata statements associated to entities) and for enabling high rate parallel processing. In case PDF are processed, a minimum of 8GB is required due to additional PDF parsing and structuring requirements. For parallel processing of PDF exploiting multithreading (e.g. 10 parallel threads), 16GB is recommended.

After decompressing all the index data, up to 119 GB of disk space will be used if you wish to use all the supported languages (en, fr, de, it, es) - be sure to have enough free space. For running English language only, you will need around 90 GB. SSD is highly recommended for best performance and experience, in particular with a low amount of available RAM (e.g. RAM < 4GB).

First install GROBID and grobid-ner, see the relative instruction of GROBID and grobid-ner.

The path to grobid-home shall indicated in the file *data/config/mention.yaml*, for instance:

```
# path to the GROBID home (for grobid-ner, grobid, etc.)
grobidHome: ../grobid/grobid-home/
```

Install *entity-fishing*:

```
$ git clone https://github.com/kermitt2/entity-fishing
```

Then install the compiled indexed data:

1. Download the zipped data files corresponding to your environment. The knowledge-base (Wikidata, *db-kb.zip*) and the English Wikipedia data (*db-en.zip*) must always be installed as minimal set-up. You can then add your languages of choice at the following links. Total is around 36 GB compressed, and 119 GB uncompressed. The data for this version 0.0.4 correspond to the Wikidata and Wikipedia dumps from 20.05.2020. The Knowledge Base part contains around 87 million entities and 1.1 billion statements and has considerably grown in the three last years.

   Linux
For MacOS we still make available the data version 0.0.3 corresponding to the Wikidata and Wikipedia dumps from mid-2018. Although outdated, they are still compatible with the latest entity-fishing version 0.0.4 and could be used for test/development.

MacOS

1. Unzip the db archives files under data/db/.
   This will install several sub-directories, one per language, plus wikidata (db-kb): data/db/db-XY/, with XY equal to fr, en, it, es, en. The full uncompressed data is about 119 GB.

2. Build the project, under the entity-fishing project repository.

   $ ./gradlew clean build

   You should be now ready to run the service.

3. Run the service:

   $ ./gradlew appRun

   The test console is available at port 8090 by opening in your browser: http://localhost:8090

For more information, see the next section on the entity-fishing Console.

2.1 Creating a new Knowledge Base version from new Wikidata and Wikipedia dumps

The knowledge base used by entity-fishing can be updated with new versions of Wikidata and Wikipedia using the pre-processing from the library GRISP, see https://github.com/kermitt2/grisp.
The entity-fishing console is a **graphical web interface**, part of the entity-fishing project, providing means to discover and test the service. With the console, it is possible to process chunks of text (typically a paragraph), PDF files and to verify which entities are recognised and how they are disambiguated.

The console is also a **reference implementation** in javascript (with JQuery) of a web application using the entity-fishing API service. As such, it illustrates how to call the services with mainstream Ajax queries, how to parse JSON results with **vulgus JQuery** and how to dynamically annotate a PDF with PDF.js and a dynamic HTML layer.

The console is available at the root address of the server (e.g. for Tomcat at http://<server instance name>/<root context name>, or http://localhost:8090 for jetty deployed instance).

The About page provides licence (Open Source Apache 2 licence for the entire tool including used dependencies) and contact information.

The web page Services allows to test the different REST requests.
A free text form allows the analysis of any text. On the right side of the input form, samples of text can be found, from scientific articles, news and historical documents.

In the lower part, entities are recognised in the provided text and displayed using different colors, based on the entity type and domain. On the lower right side, an infobox is displaying information provided by the service about the disambiguated Wikidata/Wikipedia entity.

In this example the text box is used to disambiguate a search query:
The console allows to test all the different services provided by *entity-fishing*, e.g. it's possible to visualise the various sentences identified by the sentence segmentation service (more details on this specific service in the REST API documentation).
In addition, it is possible to view the service raw response (in JSON format) for helping the integration phase:

More details about the response in the next section.
As RESTful web services, *entity-fishing* is defined by a certain number of stateless transformations of data made available to “consumers” via its interface.

All these RESTful services are available through Cross-origin resource sharing (CORS), allowing clients, such as web browser and server to interact in a flexible manner with cross-origin request.

4.1 *entity-fishing* query processing

The *entity-fishing* query processing service takes as input a JSON structured query and returns the JSON query enriched with a list of identified and, when possible, disambiguated entities.

The *entity-fishing* service can be applied on 4 types of input content:

- **text**, provided as JSON string value, for example one or several paragraphs of natural language,
- **search query**, provided as JSON string value, corresponding to several search terms used together and which can possibly be disambiguated when associated,
- **weighted vector of terms**, provided by a structured JSON array, where each term will be disambiguated, when possible, in the context of the complete vector - weighted vector of term is a very common structure used in information retrieval, clustering and classification.
- **PDF document**, provided as multipart data with the JSON query string.

One and only one input type is mandatory in a query, otherwise an HTTP error 400 is returned (see response status codes below). Combining multiple inputs in a single request is currently not supported.

4.1.1 Supported languages

In the current version English, French, German, Spanish and Italian are supported. We plan to extend the support in future releases, as long the volume of the Wikipedia corpus for a new language is sufficient.

The service returns an HTTP error 406 if the language of the text to be processed is not supported, see below.
4.1.2 Response status codes

In the following table are listed the status codes returned by this entry point.

<table>
<thead>
<tr>
<th>HTTP Status code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>400</td>
<td>Wrong request, missing parameters, missing header</td>
</tr>
<tr>
<td>404</td>
<td>Indicates property was not found</td>
</tr>
<tr>
<td>406</td>
<td>The language is not supported</td>
</tr>
<tr>
<td>500</td>
<td>Indicate an internal service error</td>
</tr>
</tbody>
</table>

4.1.3 REST query

POST /disambiguate

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required</td>
<td>query</td>
<td>multipart/form-data</td>
<td>Query to be processed in JSON UTF-8</td>
</tr>
<tr>
<td>optional</td>
<td>file</td>
<td>multipart/form-data</td>
<td>PDF file (as multipart)</td>
</tr>
</tbody>
</table>

NOTE: To process the text query only (no PDF), is also possible to send it as normal application/json raw data.

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

4.1.4 Query format description

The entity-fishing query processing service always consumes a parameter which is a JSON string representing a query, and optionally a PDF file. The service thus follows a Query DSL approach (like, for instance, ElasticSearch) to express queries instead of multiples HTTP parameters. This approach allows queries which are much richer, flexible and simple to express, but also interactive scenarios (where output of the services can be used easily as input after some changes from the user, as for instance in an interactive text editing task).

The JSON query indicates what is the textual content to process, the various (optional) parameters to consider when processing it, optionally some already existing disambiguated entities (already disambiguated by a user or via a particular workflow), and an optional customisation to provide more context to the disambiguation process.

The JSON query is similar to the response of the entity-fishing service, so that a entity-fishing service response can be sent as query after light modifications in an interactive usage scenario, or to be able to process easily already partially annotated text.

When annotations are present in the query, the entity-fishing system will consider them certain and:

- ensure that the user annotations will be present in the output response without inconsistencies with other annotations,
- exploit the user annotations to improve the context for identifying and disambiguating the other possible entities.

Similarly,
• if no language is indicated (usual scenario), the entity-fishing service will use a language identifier to detect the correct language and the language resources to use. However, the query can also optionally specify a language for the text to be processed. This will force the service to process the text with the corresponding particular language resources.

• It is possible also to pass an existing sentence segmentation to the entity-fishing service via the JSON query, in order that the service provides back identified entities following the given sentence segmentation.

The client must respect the JSON format of the entity-fishing response as new query, as described below:

**Generic format**

The JSON format for the query parameter to be sent to the service is identical to a response of the service:

```json
{
    "text": "The text to be processed.",
    "shortText": "term1 term2 ...",
    "termVector": [
        {
            "term": "term1",
            "score": 0.3
        },
        {
            "term": "term2",
            "score": 0.1
        }
    ],
    "language": {
        "lang": "en"
    },
    "entities": [],
    "mentions": ["ner","wikipedia"],
    "nbest": 0,
    "sentence": false,
    "customisation": "generic",
    "processSentence": [],
    "structure": "grobid"
}
```

One and only one of the 4 possible input type - JSON field text, shortText, termVector or a PDF file - must be provided in a query to be valid. Using multiple input type in the same query is not supported in the version of the API described here.

(1) **text**

Provides a text to be processed (e.g. one or several paragraphs). The text have be greater than 5 character or 406 is returned.

(2) **shortText**

Provides a search query to be processed.
(3) termVector

Provides a list of terms, each term being associated to a weight indicating the importance of the term as compared to the other terms.

(4) language

When the source language (parameters language) is pre-set the language is considered certain, and the language identifier is not used.

(5) mentions

Provides the methods to be used to identify mentions to be disambiguated, by default mentions are identified with an NER (the mentions are all Named Entity found in the input text to be processed) and with all the labels of Wikipedia for the appropriate language (all the anchors and titles used to refer to a Wikipedia page).

(6) entities

In the input example above, the list entities can be used to provide predefined entities (typically pre-annotated by a user). Having an already annotated entity helps the disambiguation service to resolve entity mentions by offering an important contribution to the global context. When the entities attribute is not present or empty there are simply no predefined annotations.

For example having a text with the mention “Washington” and manually providing its referring entity (e.g. the city Washington DC) provides an important contribution to the correct disambiguation of the other entity mentions in the text.

Below an example of how the pre-annotated entity can be provided. The algorithm would naturally disambiguate German Army with German Army (Wehrmacht) (wikipediaId: 12354993) because the text is contextualised on the First World War. The users can alter this result, by forcing the term to be the German Army of the Second World War (wikipediaId: 11702744). In the response the entity should be returned with confidence 1.0 (as it has been manually provided).

In order to get the wikipedia information for a term, check the term lookup documentation.

NOTE: At the moment the entity is taken in account only when the wikipediaExternalRef is provided:

```json
{
  "text": "Austria invaded and fought the Serbian army at the Battle of Cer and Battle of Kolubara beginning on 12 August.",
  "language": {
    "lang": "en"
  },
  "entities": [
    {
      "rawName": "German Army",
      "offsetStart": 1107,
      "offsetEnd": 1118,
      "wikipediaExternalRef": 11702744,
      "wikidataId": "Q701923"
    }
  ]
}
```
In a typical interactive scenario, an application client first sends a text to be processed via the /disambiguate service, and receives a JSON response with some entities.

The annotated text is displayed to a user which might correct some invalid annotations.

The client updates the modified annotations in the first JSON response and can send it back to the service now as new query via the /disambiguate.

The corrected annotations will then be exploited by the entity-fishing system to possibly improve the other annotations and disambiguations.

(7) processSentence

The processSentence parameter is introduced to support interactive text editing scenarios. For instance, a user starts writing a text and wants to use the entity-fishing service to annotate dynamically the text with entities as it is typed.

To avoid having the server reprocessing several time the same chunk of text and slowing down a processing time which has to be almost real time, the client can simply indicate a sentence - the one that has just been changed - to be processed.

The goal is to be able to process around two requests per second, even if the typed text is very long, so that the annotations can be locally refreshed smoothly, even considering the fastest keystroke rates that a human can realize.

The processSentence parameter is followed by a list of notations (only numbers in integer, e.g. [1, 7]) - note that the index starts from 0) corresponding to the sentence index will limit the disambiguation to the selected sentences, while considering the entire text and the previous annotations.

In this example only the second sentence will be processed by entity-fishing:

```
{
   "text": "The army, led by general Paul von Hindenburg defeated Russia in a series of battles collectively known as the First Battle of Tannenberg. But the failed Russian invasion, causing the fresh German troops to move to the east, allowed the tactical Allied victory at the First Battle of the Marne.",
   "processSentence": [1]
}
```

When processSentence is set, the sentence segmentation is triggered anyway and the value of the attribute sentence is ignored:

```
{
   "text": "The army, led by general Paul von Hindenburg defeated Russia in a series of battles collectively known as the First Battle of Tannenberg. But the failed Russian invasion, causing the fresh German troops to move to the east, allowed the tactical Allied victory at the First Battle of the Marne.",
   "processSentence": [1],
   "sentences": [
   {
      "offsetStart": 0,
      "offsetEnd": 138
    },
    {
      "offsetStart": 138,
      "offsetEnd": 293
    }
  ]
}
```

(continues on next page)
Example using CURL (using the query above):

curl 'http://cloud.science-miner.com/nerd/service/disambiguate' -X POST -F "query={ "text": 'The army, led by general Paul von Hindenburg defeated Russia in a series of battles collectively known as the First Battle of Tannenberg. But the failed Russian invasion, causing the fresh German troops to move to the east, allowed the tactical Allied victory at the First Battle of the Marne.', 'processSentence': [1], 'sentences': [ { 'offsetStart': 0, 'offsetEnd': 138 }, { 'offsetStart': 138, 'offsetEnd': 293 } ], 'entities': [ { 'rawName': 'Russian', 'type': 'NATIONAL', 'offsetStart': 153, 'offsetEnd': 160 } ] }" -F "file=@PATH_FILENAME.pdf"

PDF input

This service is processing a PDF provided as input after extracting and structuring its raw content. Structuration is currently specialized to scientific and technical articles. Processing a PDF not corresponding to scientific articles is currently not recommended.

In addition to the query, it accepts a PDF file via `multipart/form-data`.

The JSON format for the query parameter to be sent to the service is identical to a response of the service:
An additional parameter related to the processing of the structure of the PDF is available, called `structure`. For processing scientific and technical documents, in particular scholar papers, the value should be `grobid` which is a state of the art tool for structure the body of a scientific paper - it will avoid labelling bibliographical information, foot and head notes, figure content, will identify the useful areas (header, paragraphs, captions, etc.) handling multiple columns, hyphen, etc. and it will apply custom processnig based on the identified structure.

If you wish to process the whole document without specific structure analysis (this is advised for non-scientific papers), use the value `full` for the parameter `structure`.

**Example using CURL** (using the query above):

```bash
curl 'http://cloud.science-miner.com/nerd/service/disambiguate' -X POST -F "query={  "language": {'lang':'en'}}, 'entities': [], 'nbest': false, 'sentence': false,  "structure": 'grobid'}" -F "file=@PATH_FILENAME.pdf"
```

### Weighted term disambiguation

Process a weighted vector of terms. Each term will be disambiguated - when possible - in the context of the complete vector.

Example request

```json
{
  "termVector": [
    {
      "term": "computer science",
      "score": 0.3
    },
    {
      "term": "engine",
      "score": 0.1
    }
  ],
  "language": {
    "lang": "en"
  },
  "nbest": 0
}
```

The `termVector` field is required for having a well-formed query.

**Example using CURL** (using the query above):

```bash
curl 'http://cloud.science-miner.com/nerd/service/disambiguate' -X POST -F "query={  "termVector": [ { "term": 'computer science', 'score': 0.3 }, { "term": 'engine', 'score': 0.1 } ], "language": { 'lang': 'en' }, 'resultLanguages': ['$de'], 'nbest': false}" -F "file=@PATH_FILENAME.pdf"
```
Search query disambiguation

This functionality provides disambiguation for a search query expressed as a “short text”.

The input is the list of terms that are typically provided in the search bar of a search engine, and response time are optimized to remain very low (1-10ms).

For example, let’s consider the search query: “concrete pump sensor”. From this association of search terms, it is clear that the sense corresponding to concrete is the material, the entity is the device called concrete pump, and it has nothing to do with concrete as the antonym of abstract.

Processing this kind of input permits to implement semantic search (search based on concept matching) and semantic-based ranking (ranking of documents based on semantic proximity with a query, for instance exploiting classifications, domain information, etc.) in a search engine.

Search query disambiguation uses a special model optimized for a small number of non-strictly ordered terms and trained with search queries.

The difference between standard text and short text is similar to the one of the ERD 2014 challenge.

Example request:

```json
{
    "shortText": "concrete pump sensor",
    "language": {
        "lang": "en"
    },
    "nbest": 0
}
```

Example using CURL (using the query above):

```
curl 'http://cloud.science-miner.com/nerd/service/disambiguate' -X POST -F "query={
    'shortText': 'concrete pump sensor','language': { 'lang': 'en' },'nbest': 0}"
```

4.1.5 Response

The response returned by the entity-fishing query processing service is basically the same JSON as the JSON query, enriched by the list of identified and, when possible, disambiguated entities, together with a server runtime information.

If the textual content to be processed is provided in the query as a string, the identified entities will be associated to offset positions in the input string, so that the client can associate precisely the textual mention and the entity “annotation”.

If the textual content to be processed is provided as a PDF document, the identified entities will be associated to coordinates positions in the input PDF, so that the client can associate precisely the textual mention in the PDF via a bounding box and makes possible dynamic PDF annotations.

Response when processing a text

```json
{
    "runtime": 223,
    "nbest": false,
    "text": "Austria was attaching Serbia."
}
```

(continues on next page)
In the example above, the root layer of JSON values correspond to:

- **runtime**: the amount of time in milliseconds to process the request on server side,
- **nbest**: as provided in the query - when false or 0 returns only the best disambiguated result, otherwise indicates to return up to the specified number of concurrent entities for each disambiguated mention,
- **text**: input text as provided in the query, all the offset position information are based on the text in this field,
- **language**: language detected in the text and his confidence score (if the language is provided in the query, conf is equal to 1.0),
- **entities**: list of entities recognised in the text (with possibly entities provided in the query, considered then as certain),
- **global_categories**: provides a weighted list of Wikipedia categories, in order of relevance that are representing the context of the whole text in input.

For each entity the following information are provided:

- **rawName**: string realizing the entity as it appears in the text
- **offsetStart, offsetEnd**: the position offset of where the entity starts and ends in the text element in characters (JSON UTF-8 characters)
- **nerd_score**: disambiguation confidence score, indicates the score of the entity against the other entity candidates for the text mention,
- **nerd_selection_score**: selection confidence score, indicates how certain the disambiguated entity is actually valid for the text mention,
- **wikipediaExternalRef**: id of the wikipedia page. This id can be used to retrieve the original page from wikipedia3 or to retrieve all the information associated to the concept in the knowledge base (definition, synonyms, categories, etc. - see the section “Knowledge base concept retrieval”),
- **type**: NER class of the entity (see table of the 27 NER classes below under “2. Named entity types”).

4.1. **entity-fishing** query processing

---

21
• **sense**: NER sense mapped on Wordnet synset - senses are provided to improve the disambiguation process, but they are currently not very reliable.

The type of recognised entities are restricted to a set of 27 classes of named entities (see GROBID NER documentation). Entities not covered by the knowledge bases (the identified entities unknown by Wikipedia) will be characterized only by an entity class, a word sense estimation and a confidence score, without any reference to a Wikipedia article or domain information.

**Response when processing a search query**

```
{
  "runtime": 146,
  "nbest": false,
  "shortText": "concrete pump sensor",
  "language": {
    "lang": "en",
    "conf": 0.0
  },
  "global_categories": [
    {
      "weight": 0.08448995135780164,
      "source": "wikipedia-en",
      "category": "Construction equipment",
      "page_id": 24719865
    },
    [...]
  ],
  "entities": [
    {
      "rawName": "concrete",
      "offsetStart": 0,
      "offsetEnd": 8,
      "nerd_score": "0.3416037625644609",
      "nerd_selection_score": "0.9793831523036264",
      "wikipediaExternalRef": "5371",
      "domains": [
        "Mechanics",
        "Engineering",
        "Architecture"
      ]
    },
    {
      "rawName": "concrete pump",
      "offsetStart": 0,
      "offsetEnd": 13,
      "nerd_score": "0.695783745626837",
      "nerd_selection_score": "0.9576960838921623",
      "wikipediaExternalRef": "7088907",
      "domains": [
        "Mechanics",
        "Engineering"
      ]
    },
    {
      "rawName": "pump",
      "offsetStart": 9,
      "offsetEnd": 13,
      "nerd_score": "0.33995668143945024",
      "domains": [
        "Mechanics",
        "Engineering"
      ]
    }
  ]
}
```
Response when processing a weighted vector of terms

```
{
  "runtime": 870,
  "nbest": false,
  "termVector": [
    {
      "term": "computer science", "score": 0.3,
      "entities": [
        {
          "rawName": "computer science",
          "preferredTerm": "Computer science",
          "nerd_score": "0.5238665311593967",
          "nerd_selection_score": "0.0",
          "wikipediaExternalRef": "5323",
          "definitions": [
            {
              "definition": "Computer science blablabla.",
              "source": "wikipedia-en",
              "lang": "en"
            }
          ],
          "categories": [
            {
              "source": "wikipedia-en",
              "category": "Computer science",
              "page_id": 691117
            }
          ],
          "multilingual": [
            {
              "lang": "de",
              "term": "Informatik",
              "page_id": 2335
            }
          ]
        }
      ]
    }
  ]
}
```

Response description when processing PDF

```
{
  "runtime": 2823,
  "nbest": false,
  "file": "filename.pdf",
  "pages": 10,
  "language": {
    "lang": "en",
    [...]
  }
}
```
As apparent in the above example, for PDF the offset position of the entities are replaced by coordinates information introduced by the JSON attribute pos. These coordinates refer to the PDF that has been processed and permit to identify the chunk of annotated text by the way of a list of bounding boxes.

In addition, an attribute pages is used to indicate the size of each page of the PDF document which is a necessary information to position correctly annotations.

The next section further specifies the coordinates information provided by the service (see GROBID).

**PDF Coordinates**

The PDF coordinates system has three main characteristics:

- contrary to usage, the origin of a document is at the upper left corner. The x-axis extends to the right and the y-axis extends downward,
- all locations and sizes are stored in an abstract value called a PDF unit,
- PDF documents do not have a resolution: to convert a PDF unit to a physical value such as pixels, an external
value must be provided for the resolution.

In addition, contrary to usage in computer science, the index associated to the first page is 1 (not 0).

The response of the processing of a PDF document by the entity-fishing service contains two specific structures for positioning entity annotations in the PDF:

• the list of page size, introduced by the JSON attribute pages. The dimension of each page is given successively by two attributes page_height and page_height.

• for each entity, a json attribute pos introduces a list of bounding boxes to identify the area of the annotation corresponding to the entity. Several bounding boxes might be necessary because a textual mention does not need to be a rectangle, but the union of rectangles (a union of bounding boxes), for instance when a mention to be annotated is on several lines.

A bounding box is defined by the following attributes:

• p: the number of the page (beware, in the PDF world the first page has index 1!),

• x: the x-axis coordinate of the upper-left point of the bounding box,

• y: the y-axis coordinate of the upper-left point of the bounding box (beware, in the PDF world the y-axis extends downward!),

• h: the height of the bounding box,

• w: the width of the bounding box.

As a PDF document expresses value in abstract PDF unit and do not have resolution, the coordinates have to be converted into the scale of the PDF layout used by the client (usually in pixels). This is why the dimension of the pages are necessary for the correct scaling, taking into account that, in a PDF document, pages can be of different size.

The entity-fishing console offers a reference implementation with PDF.js for dynamically positioning entity annotations on a processed PDF.

### 4.2 Knowledge base concept retrieval

This service returns the knowledge base concept information. In our case case, language-independent information from Wikidata will be provided (Wikidata identifier, statements), together with language-dependent information (all the Wikipedia information: Wikipedia categories, definitions, translingual information, etc.). This service is typically used in pair with the main entity-fishing query processing service in order to retrieve a full description of an identified entity.

The service supports the following identifiers:

• wikidata identifier (starting with Q, e.g. Q61)

• wikipedia identifier

The entity-fishing content processing service returns the identifiers of the resulting entities with some position offset information. Then, if the client wants, for instance, to display an infobox for this entity, it will send a second call to this service and retrieve the full information for this particular entity. Adding all the associated information for each entity in the response of the entity-fishing query processing service would result in a very large response which would slow a lot the client, such as a web browser for instance. Using such separate queries allows efficient asynchronous calls which will never block a browser and permits to make only one call per entity, even if the same entity has been found in several places in the same text.

The entity-fishing console offers an efficient reference implementation with Javascript and Ajax queries through the combination of the main entity-fishing query processing service and the Knowledge base concept retrieval.
4.2.1 Response status codes

In the following table are listed the status codes returned by this entry point.

<table>
<thead>
<tr>
<th>HTTP Status code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>400</td>
<td>Wrong request, missing parameters, missing header</td>
</tr>
<tr>
<td>404</td>
<td>Indicates property was not found</td>
</tr>
<tr>
<td>500</td>
<td>Indicate an internal service error</td>
</tr>
</tbody>
</table>

GET /kb/concept/{id}

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required</td>
<td>id</td>
<td>String</td>
<td>ID of the concept to be retrieved (wikipedia, wikidata id (starting with Q) or property (starting with P)).</td>
</tr>
<tr>
<td>optional</td>
<td>lang</td>
<td>String</td>
<td>(valid only for wikipedia IDs) The language knowledge base where to fetch the concept from. Default: en.</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

(3) Example response

```json
{
    "rawName": "Austria",
    "preferredTerm": "Austria",
    "nerd_score": "0.0",
    "nerd_selection_score": "0.0",
    "wikipediaExternalRef": "26964606",
    "wikidataId": "Q1234"
    "definitions": [
      {
        "definition": ""Austria'', officially the '''Republic of Austria''",
        "source": "wikipedia-en",
        "lang": "en"
      }
    ],
    "categories": [
      {
        "source": "wikipedia-en",
        "category": "Austria",
        "page_id": 707451
      },
      {
        "lang": "de",
        "source": "wikipedia-en",
        "category": "Erasmus Prize winners",
        "page_id": 1665997
      }
    ]
}
```

(continues on next page)
The elements present in this response are:

- **rawName**: The term name
- **preferredTerm**: The normalised term name
- **nerd_score**: always 0.0 because no disambiguation took place in a KB access
- **nerd_selection_score**: always 0.0 because no disambiguation took place in a KB access
- **wikipediaExternalRef**: unique identifier of the concept in wikipedia
- **wikidataId**: unique identifier of the concept in wikidata
- **definitions**: list of wikipedia definitions (usually in wikipedia a concept contains one and only one definition). Each definition is characterized by three properties:
  - **definition**: The text of the definition
  - **source**: The knowledge base from which the definition comes from (in this case can be wikipedia-en, wikipedia-de and wikipedia-fr)
  - **lang**: the language of the definition
- **categories**: This provides a list of Wikipedia categories directly coming from the wikipedia page of the disambiguated entity. Each category is characterised by the following properties:
  - **category**: The category name
  - **source**: The knowledge base from which the definition comes from.
  - **pageId**: the Id of the page describing the category
- **domains**: For each entry, Wikipedia provides a huge set of categories, that are not always well curated (1 million categories in the whole wikipedia). Domains are generic classification of concepts, they are mapped from the wikipedia categories.
- **multilingual**: provides references to multi-languages resources referring to the same entity. E.g. the entity country called Austria is Österreich in German wikipedia and Autriche in French wikipedia. The page_id provided here relates to the language-specific Wikipedia (e.g. in the above example the page_id for the country Autriche in the French Wikipedia is 15).

4.2. Knowledge base concept retrieval
4.3 Term Lookup

This service is used to search terms in the knowledge base. This service is useful to verify how many ambiguity a certain term can generate.

4.3.1 Response status codes

In the following table are listed the status codes returned by this entry point.

GET /kb/term/{term}

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>term</td>
<td>String</td>
<td>The term to be retrieved</td>
</tr>
<tr>
<td>optional</td>
<td>lang</td>
<td>String</td>
<td>The language knowledge base where to fetch the term from. Default: en.</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

4.4 Language identification

Identify the language of a provided text, associated to a confidence score.

4.4.1 Response status codes

In the following table are listed the status codes returned by this entry point.

<table>
<thead>
<tr>
<th>HTTP Status code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>400</td>
<td>Wrong request, missing parameters, missing header</td>
</tr>
<tr>
<td>404</td>
<td>Indicates property was not found</td>
</tr>
<tr>
<td>500</td>
<td>Indicate an internal service error</td>
</tr>
</tbody>
</table>

POST /language

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>text</td>
<td>String</td>
<td>The text whose language needs to be identified</td>
</tr>
</tbody>
</table>

(2) Request header
**entity-fishing documentation, Release 0.0.3**

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
<tr>
<td>optional</td>
<td>Content-Type</td>
<td>multipart/form-data</td>
<td>Define the format of the posted property</td>
</tr>
</tbody>
</table>

(3) Example response (ISO 639-1)

Here a sample of the response

```
{
  "lang": "en",
  "conf": 0.9
}
```

**GET /language?text=(text)**

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required</td>
<td>text</td>
<td>String</td>
<td>The text whose language needs to be identified</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

(3) Example response (ISO 639-1)

Here a sample of the response

```
{
  "lang": "en",
  "conf": 0.9
}
```

### 4.5 Sentence segmentation

This service segments a text into sentences. It is useful in particular for the interactive mode for indicating that only certain sentences need to be processed for a given query.

Beginning and end of each sentence are indicated with offset positions with respect to the input text.

#### 4.5.1 Response status codes

In the following table are listed the status codes returned by this entry point.

**POST /segmentation**

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required</td>
<td>text</td>
<td>String</td>
<td>The text to be segmented into sentences</td>
</tr>
</tbody>
</table>
(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
<tr>
<td>optional</td>
<td>Content-Type</td>
<td>multipart/form-data</td>
<td>Define the format of the posted property</td>
</tr>
</tbody>
</table>

(3) Example response

Here a sample of the response:

```json
{
    "sentences": [
        {
            "offsetStart": 0,
            "offsetEnd": 7
        },
        {
            "offsetStart": 6,
            "offsetEnd": 21
        }
    ]
}
```

GET /segmentation?text={text}

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required</td>
<td>text</td>
<td>String</td>
<td>The text whose language needs to be identified</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

(3) Example response

Here a sample of the response:

```json
{
    "sentences": [
        {
            "offsetStart": 0,
            "offsetEnd": 7
        },
        {
            "offsetStart": 6,
            "offsetEnd": 21
        }
    ]
}
```
4.6 Customisation API

The customisation is a way to specialize the entity recognition, disambiguation and resolution for a particular domain. This API allows to manage customisations for the entity-fishing instance which can then be used as a parameter by the entity-fishing services.

Customisation are identified by their name (or, also called profile in the API).

4.6.1 Customisation body

The JSON profile of a customisation to be sent to the server for creation and extension has the following structure:

```json
{
    "wikipedia": [
        4764461,
        51499,
        1014346
    ],
    "language": {"lang": "en"},
    "text": [
        "World War I (WWI or WW1 or World War One), also known as Germany and Austria-Hungary."
    ],
    "description": "Customisation for World War 1 domain"
}
```

The context will be build based on Wikipedia articles and raw texts, which are all optional. Wikipedia articles are expressed as an array of Wikipedia page IDs.

Texts are represented as an array of raw text segments.

4.6.2 Response status codes

In the following table are listed the status codes returned by this entry point.

<table>
<thead>
<tr>
<th>HTTP Status code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Successful operation.</td>
</tr>
<tr>
<td>400</td>
<td>Wrong request, missing parameters, missing header</td>
</tr>
<tr>
<td>404</td>
<td>Indicates property was not found</td>
</tr>
<tr>
<td>500</td>
<td>Indicate an internal service error</td>
</tr>
</tbody>
</table>

GET /customisations

Returns the list of existing customisations as a JSON array of customisation names.

(1) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

(2) Example response

Here a sample of the response:
GET /customisation/{name}

Retrieve the content of a specific customisation

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>name of the customisation to be retrieved</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

(3) Example response

Here a sample of the response

```json
{
    "wikipedia": [
        4764461,
        51499,
        1014346
    ],
    "language": {
        "lang": "en"
    },
    "texts": [
        "World War I (WWI or WW1 or World War One), also known as the First World War or the Great War, was a global war centred in Europe that began on 28 July 1914 and lasted until 11 November 1918."
    ],
    "description": "Customisation for World War 1 domain"
}
```

Or in case of issues:

```json
{
    "ok": "false",
    "message": "The customisation already exists."
}
```

POST /customisations

Creates a customisation as defined in the input JSON, named following the path parameter. The JSON profile specifies a context via the combination of a list of Wikipedia article IDs and text fragments. A text describing informally the customisation can be added optionally.

If the customisation already exists an error is returned.
(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>String</td>
<td>name of the customisation to be created</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>String</td>
<td>JSON representation of the customisation (see example)</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>value</td>
<td>Set the response type of the output</td>
</tr>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td></td>
</tr>
</tbody>
</table>

(3) Example response

Here a sample of the response

```json
{
  "ok": "true"
}
```

Or in case of issues:

```json
{
  "ok": "false",
  "message": "The customisation already exists."
}
```

PUT /customisation/(profile)

Update an existing customisation as defined in the input JSON, named following the path parameter. The JSON profile specifies a context via the combination of a list of Wikipedia article IDs, FreeBase entity mid and text fragments.

A text describing informally the customisation can be added optionally.

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>profile</td>
<td>String</td>
<td>name of the customisation to be updated</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>name</td>
<td>value</td>
<td>Set the response type of the output</td>
</tr>
<tr>
<td>optional</td>
<td>Accept</td>
<td>application/json</td>
<td></td>
</tr>
</tbody>
</table>

(3) Example response

Here a sample of the response

```json
{
  "ok": "true"
}
```

Or in case of issues:

```json
{
  "ok": "false",
  "message": "The customisation already exists."
}
```
DELETE /customisation/{profile}

(1) Parameters

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>content-type value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>profile</td>
<td>String</td>
<td>name of the customisation to be deleted</td>
</tr>
</tbody>
</table>

(2) Request header

<table>
<thead>
<tr>
<th>required</th>
<th>name</th>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accept</td>
<td>application/json</td>
<td>Set the response type of the output</td>
</tr>
</tbody>
</table>

(3) Example response

Here a sample of the response

```json
{
   "ok": "true"
}
```

Or in case of issues:

```json
{
   "ok": "false",
   "message": "The customisation already exists."
}
```
5.1 Datasets for long texts

It is possible to evaluate entity-fishing entity disambiguation models with several well-known available datasets. For convenience, the following datasets are present in the entity-fishing distribution:

- `ace`: this is a subset of the documents used in the ACE 2004 Coreference documents with 36 articles and 256 mentions, annotated through crowdsourcing, see [1].
- `aida`: AIDA-CONLL is a manually annotated dataset based on the CoNLL 2003 dataset, with 13881 Reuters news articles and 27817 mentions, see [2]. Note that the raw texts of this dataset are not included in entity-fishing, they have to be obtained from NIST (free for research purposes). AIDA-CONLL dataset can be considered as the most significant gold data for entity disambiguation both in terms of size, ambiguity rate and annotation quality. In addition to the complete AIDA-CONLL dataset, this corpus is divided into three subsets that can be used for evaluation separately:
  - `aida-train`: corresponds to the training subset of the CoNLL 2003 dataset
  - `aida-testa`: corresponds to the validation subset of the CoNLL 2003 dataset
  - `aida-testb`: corresponds to the test subset of the CoNLL 2003 dataset
- `aquaint`: this dataset has been created by Milne and Witten [3], with 50 documents and 727 mentions from a news corpus from the Xinhua News Service, the New York Times, and the Associated Press.
- `iitb`: manually created dataset by [4] with 50 documents collected from online news sources.
- `msnbc`: this dataset is based on 20 news articles from 10 different topics (two articles per topic) and contains a total of 656 mentions, see [5].
- `clueweb`: W-NED-Clueweb 12 dataset is a large dataset created by [6] from the Clueweb corpura automatically - it is this far less reliable than the previous ones.
- `wikipedia`: similarly as the Clueweb dataset, this set has been created automatically by [6] from Wikipedia, thus also clearly less reliable.
- `hirmeos`: manually created dataset using open accessible books (licence CC-BY), financed from the European project H2020 Hirmeos [7].
All these reference datasets are located under data/corpus/corpus-long.

## 5.2 Evaluation commands

Use the following maven command with the above dataset identifier for running an evaluation:

```bash
$ ./gradlew evaluation -Pcorpus=[dataset]
```

For instance for evaluating against the testb subset of the AIDA-CONLL, use:

```bash
$ ./gradlew evaluation -Pcorpus=aida-testb
```

The evaluation process will provide standard metrics (accuracy, precision, recall, f1) for micro- and macro-averages for the entity disambiguation algorithm selected as ranker and for priors (as baseline).

The recall of the candidate selection with respect to the gold annotations is also provided (e.g. the proportion of candidate sets containing the expected answer before the ranking).

## 5.3 Generation of pre-annotated training/evaluation data

In case a new corpus needs to be created, entity-fishing includes the possibility to automatically generate an XML file of entity annotations from text or pdf files in the same format as the other existing corpus. These generated files can then be corrected manually and used as gold training or evaluation data, or they can be used for semi-supervised training.

For a given new corpus to be created, for instance the corpus toto, the following directory must be created: data/corpus/corpus-long/toto/ The documents part of this corpus must be placed under the subdirectories RawText and/or pdf.

If there is a directory called pdf or PDF, the process will extract information (title, abstract, body) from each pdf and save it as pdfFileName.lang.txt inside the RawText directory. The tool will then look into the subdirectory RawText and process the files *.txt found inside. If the files name is in the form filename.lang.txt then the lang will be used as reference, otherwise en will be the default choice.

Use the following maven command with the above dataset identifier for generating the annotation xml file:

```bash
$ ./gradlew annotatedDataGeneration -Pcorpus=[corpusname]
```

For instance, for a new corpus toto, with text or pdf documents prepared as indicated above:

```bash
$ ./gradlew annotatedDataGeneration -Pcorpus=toto
```

## 5.4 References


5.4. References


Train and evaluate

Trained models for entity recognition and disambiguation are provided in the project repository. The following section explains how to retrain the models.

6.1 Training with Wikipedia

Currently a random sample of Wikipedia articles is used for training. The full article content is therefore necessary and a dedicated database will be created the first time the training is launched. This additional database is used and is required only for training. You will need the Wikipedia XML dump corresponding to the target languages available in a directory indicated in the yaml config files by the parameter dataDirectory. A warning here, as this additional database contains the whole textual content of all Wikipedia articles (with wiki markups), it is quite big, around 7.6G GB for the English Wikipedia (dump from May 2020). This database (stored under the dbDirectory indicated in the language config file and called markupFull) will be built automatically if not present, so typically at first launch of the training for a given language, and the process will take a bit more than one hour for building the English version for example.

The following command will build the two models used in entity-fishing, the ranker and the selector model (Gradient Tree Boosting for the first one, Random Forest for the second one) and preliminary build the full article content database the first time for the English Wikipedia:

```bash
$ ./gradlew train_wikipedia -Plang=en
```

For other languages, replace the ending language code (en) by the desired one (fr, de, it and es are supported), e.g.:

```bash
$ ./gradlew train_annotate -Plang=fr
$ ./gradlew train_annotate -Plang=de
```

Models will be saved under data/models. ARFF training data files used to build the model are saved under data/wikipedia/training/.
6.2 Evaluation with Wikipedia

An evaluation is produced at the end of training based on a random sample of Wikipedia articles, providing macro- and micro-average precision, recall and f1-score.

Note that the ratio of disambiguated mentions in a Wikipedia article is low. As a consequence, the precision of our models will be very low because they are built for disambiguating a maximum of entities. Recall is probably a more meaningful measure when evaluating with Wikipedia.

For an evaluation of the NED aspect (ranker in our framework) with well-known datasets, which is much more standard and allows comparison with other similar works, see the evaluation section.

6.3 Training with an annotated corpus

It is possible to train the entity-fishing models with several well-known available datasets. For convenience, the datasets indicated here Evaluation are present in the entity-fishing distribution.

Use the following command with a dataset name and a language identifier for running a training with this dataset:

```
$ ./gradlew train_corpus -Pcorpus=aquaint -Plang=en
```

For instance for training with the train subset of the AIDA-CONLL, use:

```
$ ./gradlew train_corpus -Pcorpus=aida-train -Plang=en
```

entity-fishing also included the possibility to generate additional pre-annotated corpus, for instance to be further corrected manually. See Evaluation for the explanations.

The evaluation with annotated corpus is also described in the page Evaluation.

6.4 Creating entity embeddings

Entity embeddings are used to improve entity disambiguation. They are created from word embeddings and entity descriptions generated from Wikidata and Wikipedia. For creating these entity embeddings, the process is as follow:

0. Prepare packaging with maven:

```
$ mvn clean install
```

1. Download available pretrained word embeddings - this could be for instance word2vec, FastText, or lexvec.
   Word embeddings need initially to be in the standard .vec format (a text format). word2vec binary format can be transformed into .vec format with the simple utility convertvec

2. Quantize word embeddings

Quantize will simplify the vector given an acceptable quantization factor (by default the error rate for quantizing is 0.01, but it could be changed with the argument -error)

```
$ mvn exec:java -Dexec.mainClass=com.scienceminer.nerd.embeddings.Quantizer -Dexec.args="-i word.embeddings.vec -o word.embeddings.quantized -hashheader"
```

Here the FastText word embeddings wiki.en.vec given as input (-i) will be quantized and saved as wiki.en.quantized. -hashheader indicates that the first line (a header to be ignored) must be skipped.
3. Create Wikidata entity description to be used for producing entity embeddings. The command for creating description is the following one:

```shell
$./gradlew generate_entity_description -Plang=en
```

Replace the `en` argument by the language of interest.

As an alternative with maven:

```shell
$ mvn exec:java -Dexec.mainClass=com.scienceminer.nerd.embeddings.EntityDescription -Dexec.args="entity.description en"
```

The argument indicates then where to save the generated description (normally `data/wikipedia/embeddings/`) and the language of interest.

4. Create entity embeddings from the generated description.

This step might take a lot of time and exploiting multithreading is particularly helpful. The number of threads to be used is given by the argument `-thread`

```shell
$ mvn exec:java -Dexec.mainClass=com.scienceminer.nerd.embeddings.EntityEmbeddings -Dexec.args="-i entity.description -v word.embeddings.quantized -o entity.embeddings.vec -n 10"
```

The following parameters are available:

- `-h`: displays help
- `-in`: path to an entity description data file
- `-out`: path to the result entity embeddings file (not quantized, this is to be done afterwards)
- `-n`: number of threads to be used, default is 1 but it is advice to used as much as possible
- `-rho`: rho negative sampling parameters, if it’s < 0 use even sampling, default is -1 (must be an integer)
- `-max`: maximum words per entity, if < 0 use all the words, default is -1 (must be an integer)
- `-v`: the path to the word embedding file in .vec format (e.g. one originally of word2vec, faster, lexvec, etc.), optionally quantized

5. Quantize entity embeddings

Similarly as the steps 2.1 for the entity embeddings, the quantization:

```shell
$mvn exec:java -Dexec.mainClass=com.scienceminer.nerd.embeddings.Quantizer -Dexec.args="-i /mnt/data/wikipedia/embeddings/entity.embeddings.vec -o /mnt/data/wikipedia/embeddings/entity.embeddings.quantized -hashheader"
```

The entity embeddings are now ready to be loaded in the embedded database of `entity-fishing`.

6. Copy the embeddings files under the `entity-fishing` data repository (the one containing the csv files). `entity-fishing` expects compressed files with .gz extension: `word.embeddings.quantized.gz` and `entity.embeddings.quantized.gz`. Starting `entity-fishing` will load automatically the embeddings in the embedded database LMDB as binary data.

6.4. Creating entity embeddings
License and contact

Distributed under Apache 2.0 license. The dependencies used in the project are either themselves also distributed under Apache 2.0 license or distributed under a compatible license.

Main author and contact: Patrice Lopez (<patrice.lopez@science-miner.com>)