EFFICIENT AUTOMATED PROCESSING OF BIOMEDICAL LITERATURE

NICO COLIC

1. Introduction

The rate at which biomedical research papers are published is ever increasing. Because of this, professionals rely on manual curation of dedicated databases that list information relevant to their research. However, curators struggle to cope with the amount of papers published, and thus need to rely on automated processing of such articles.

The OntoGene systems aims at automatically identifying various relations, such as in between proteins, genes, diseases and medicaments, that can then be fed into databases such as the one mentioned before.

2. Existing Pipeline

Currently, the OntoGene system is a pipeline that is patched together from different modules written in different programming languages, which communicate with each other via files. Each module takes as input a file, and produces a file, typically in a predefined XML format (called OntoGene XML). The subsequent module will then read the files produced from the antecedent modules. These modules are coordinated by bash scrips [4] [6] [5]. This is inefficient for two reasons:

Firstly, because every module needs to parse the precedent module's output, and because of that there is demand for a streamlined pipeline. The repeated accessing of the disk, reading and writing considerably slows down the processing.

Secondly, usage of the pipeline is not easy for new users, since the different modules are written in different languages; and since there's no centralised documentation.

3. python-ontogene

3.1. **Project Overview.** The goal of this project is to rewrite the existing OntoGene pipeline in *python3*, and in particular to reduce communication between modules via files. Through this, processing should be accelerated. Furthermore, the pipeline should be consistent in documentation, and hence easier to understand for the user.

The acceleration of processing is particularly relevant as the pipeline should be able to process the entire PubMed, which is a database of 25 million biomedical articles that can be downloaded freely.

The pipeline is currently developed up to the point of entity recognition, and can be accessed at https://pub.cl.uzh.ch:11443/colic/python-ontogene.

3.2. **Architecture of the System.** The system is composed of several independent modules, which are coordinated by a control script. The main mode of communication between the modules is via objects of a custom *Article class*, which mimics an XML structure. All modules read and return objects of this class, which ensures independence of the modules.

The modules are coordinated via a control script written in python, which passes the various Article objects produces by the modules to the subsequent modules.

3.3. **Configuration.** All variables relevant for the pipeline (such as input files, output directories and bookkeeping parameters) are stored in a single file, which is read by the control script. The control script will then supply the relevant arguments read from the configuration file to the individual modules. This ensures that the user only has to edit a single file, while at the same time keeping the modules independent.

3.4. **Backwards Compatibility.** In order to preserve compatibility to the existing pipeline (2), the article objects can be exported to OntoGene XML format at various stages of processing.

3.5. **Timing.** Because processing can occur on very large scales, some modules provide timing options.

4. Usage

The exact usage of the individual modules is described in the subsequent chapters. Furthermore, the in-file documentation in *example.py* file can serve to provide a more concrete idea of how to use the pipeline.

5. Module: Article

The article module is a collection of various classes, such as token, sentence and section. The classes are hierarchically organised (eg. article has sections), but kept flexible to allow to future variations in the structure. Each class offers methods particularly suited to dealing with its contents, such as writing to file or performing further processing.

However, in order to keep the pipeline flexible and modular, the article class relies on other modules to perform tasks such as tokenization or entity recognition. While this leads to coupling between the modules, it also allows for easy replacement of modules. For example, if the tokenizer that is currently used needs replacing, it is easy to just supply a new tokenization module to the article module to perform tokenization.

5.1. **Implementation.** Currently, there are the following classes, which all implement an abstract *Unit* class: Article, Section, Sentence, Token and Term. Each of these classes have a *subelements* list, which contains objects of other classes. In this fashion, a tree like structure is build, in which an Article object has a subelements list of Sections, which each have a subelements list of Sentences and so on.

The abstract Unit class implements, amongst other, the *get_subelement(argument)* function, which will traverse data structure recursively until the elements of type *argument*

have been found. In this fashion, the data structure is kept flexible for future changes. For example, Articles may be gathered in Collections, or Sections might contain Paragraphs.

As for tokenization, the Article class expects the *tokenize(argument)* function to be called with an tokenizer object as argument. This tokenizer object needs to implement the following two function: *tokenize_sentences(string)*, and *tokenize_words(string)*. The first function is expected to return a list of strings; the second one to return a list of tuples, which store token text as well as start and end position in text.

Finally, the Article class should implement functions such as *add_section()* and *add_term()*, and internally create the respective objects. This is done so that other modules only need to import the Article class, which in turn will take care of accessing and creating other classes.

5.2. **Usage.** This is an example which will create an Article, manually add a Section with some text, tokenize it and print it to console and file.

```
import article

my_article = Article("12345678") # constructor needs ID
my_article.addSection("S1","abstract","this is an example text")
my_article.tokenize()
print(my_article)
my_article.print_xml('path/to/output_file.xml')
```

5.3. **Export.** At the time of writing, the Article class implements a *print_xml()*, which allows exporting of the data structure to a file. This function in turn recursively calls an *xml()* function on the elements of the data structure. Like this, it lies in the responsibility of the respective class to implement the *xml()* function.

The goal of this function is to export the Article object in its current state of processing. For example, if no tokenization has yet taken place, it will not try to export tokens. This however, requires much processing work. Because of this, this function and the related functions need to be updated as the pipeline is updated.

5.3.1. *Pickling.* To store and load Article objects without exporting them to a specific format, the Article class implements the *pickle()* and *unpickle()* functions. These allow dumping the current Article object as a pickle file, and restoring a previously pickled Article object.

```
import article

my_article = None
# create Article object in some way

my_article.pickle('path/to/pickle')
new_article = article.Article.unpickle('path/to/pickle')
```

5.3.2. *Exporting Entities.* The Article class implements a *print_entities_xml()* function, which exports the found entities to an XML file. As with the general export function, the XML file is built recursively by calling an *entity_xml()* function on the Entity objects that are linked to the Article.

6. Module: File Import and Accessing PubMed

This module allows importing texts from files, or to download them from PubMed, and converts them into the Article format discussed above. From there, they can be handled to the other modules and exported to XML.

There are three ways how the PubMed can be accessed:

- **PubMed dump**. After applying for a free licence, the whole of PubMed can be downloaded as a collection of around 700.xml.gz files, each of which containing about 30000 PubMed articles. This dump is updated once a year (in November / December).
- **API**. This allows the individual downloading of PubMed articles given their ID. If *entrez* is used, PubMed returns XML, if *BioPython* is used, PubMed returns python objects.
- **BioC**. For the BioCreative V: Task 3 challenge, participants are supplied with data in BioC format.

6.1. **Updating the PubMed Dump.** Since the PubMed dump is only updated once per year, additional articles published throughout the year need to be downloaded manually using the API.

This takes substantial effort: Between the last publication of the PubMed dump in December 2014 and August 1st 2015, 800000 new articles were published. This takes about 3 days to download this via the API.

6.2. **Downloading via the API.** For downloading data from PubMed, the module keeps a copy of articles already downloaded from PubMed as a python pickle, in order to prevent repeated downloads from PubMed.

6.3. **Dealing with the large number of files.** Since the pipeline operates on the basis of single articles, the PubMed dump was converted into multiple files, each of which corresponding to an article. However, most file systems, such as FAT32, and ext2, cannot cope with 25 million files in one directory. Because of this, the following structure was chosen:

Every article has a PubMed ID with up to 8 digits. If lower, they are padded from left with zeros. All articles are then grouped by their first 4 digits into folders, resulting in to up to 10000 folders with each up to 10000 files. For example, the file with ID 12345678 would reside in the directory 1234.

However, different solutions for efficient dealing with the large number of files will be explored in the continuation of this project. Especially databases, inherently suited to large data sets, such as NoSQL, seem promising.

6.4. **Usage.** The following code snippet demonstrates how to import from file and from PubMed. The import_file module allows to specify a directory rather than a path. In that case, it will load all files in the directory and convert them to Article objects.

```
from text_import.pubmed_import import pubmed_import

article = pubmed_import("12345678","mail@example.com")
# email can be omitted if file has already been downloaded
# if file was downloaded before, the module will load it from local
    dump_directory
article.print_xml('path/to/file')

from text_import.file_import import import_file

articles = import_file('/path/to/directory/or/file.txt)
# always returns a list
for article in article:
   print(article)
```

7. Module: Text Processing

This module relies heavily on NLTK to perform sentence splitting, tokenisation and part-of-speech tagging. It is independent of the pipeline, but the object created can be used to perform these tasks on the Article class. This allows swapping out this module for a different one in the future, provided the functions *tokenize_sentences()*, *tokenize_words()* and *pos_tag()* are implemented.

7.1. **Usage.** Since NLTK offers several tokenizers based on different modules, and allows you to train own models, this wrapper needs you to specify which model you want to use. The config module gives convenient ways to do this.

```
from config.config import Configuration
from text_processing.text_processing import Text_processing as tp

my_config = Configuration()
my_tp = tp(word_tokenizer=my_config.word_tokenizer_object,
          sentence_tokenizer=my_config.sentence_tokenizer_object)

for pmid, article in pubmed_articles.items():
   article.tokenize(tokenizer=my_tp)
```

8. Module: Entity Recognition

This module implements a dictionary-based entity recognition algorithm. In this approach, a list of known entities is used to find entities in a text. In this approach, the

quality of the entity recognition depends on the quality of the entities list. While the module is independent from what entity list is used, we used a list provided by *???* in our tests. This list was compiled using different existing resources.

8.1. **Usage.** The user first needs to instantiate an Entity Recognition object, which will hold the entity list in memory. This object is then passed to the *recognize_entities()* function of the Article object, which will then use the Entity Recognition to find entities. While this is slightly complicated, it ensures that different entity recognition approaches can be used in conjunction with the Article class.

When creating the Entity Recognition object, the user needs to supply a entity list as discussed above, and a Tokenizer object. The Tokenizer object is used to tokenise multi-word entries in the entity list. Obviously, this tokenisation approach needs to be the same as the one used to tokenise the articles.

```
from config.config import Configuration
from text_processing.text_processing import Text_processing as tp
from entity_recognition.entity_recognition import Entity_recognition as er

my_config = Configuration()
my_tp = tp(word_tokenizer=my_config.word_tokenizer_object,
          sentence_tokenizer=my_config.sentence_tokenizer_object)
my_er = er( my_config.termlist_file_absolute,
        my_config.termlist_format,
        word_tokenizer=my_tp )

# create tokenised Article object

my_article.recognize_entities(my_er)
my_article.print_entities_xml('output/file/path',pretty_print=True)
```

9. Evaluation

Two factors where evaluated: speed and accuracy.

9.1. **Speed.** Both pipelines were run on the same machine on the same data set and their running time measured using the Unix *time* command. The data set consisted of 9559 text files containing the abstract of a Pubmed article. All files together have a size of 38MB.

The Unix time command returns three values: real, user and system. Real time refers to the so-called wall clock time, that is, the time that has actually passed for the execution of the command. User and system refer to how much time the CPU was engaged in the respective mode. For example, system calls will add to the system time, but normal user mode programs to the user time. For the purposes below, we ignore the real time, and add system and user time.

On the old pipeline, the time command returned *real: 37m5.153s, user: 823m33.281s, sys:165m9.622s*, resulting in a total of 59 323 seconds (about 16.5 hours). This amounts to

about 6.206 seconds per article. The fact that the real time is so low is due to the program running on several cores.

On the new pipeline, the time command returned *real: 21m22.359s, user: 21m13.381s, sys: 0m6.918s*. This results in a total of 1280 seconds (about 0.36 hours), or 0.133 seconds per article. The new pipeline is not implicitly parallelised, and relies much less on system calls. However, it can be parallelised like the old pipeline, resulting in faster operation still.

9.2. **Accuracy.** For comparing the results of both pipelines, testing was done on the same data set of 9559 files as above. A testing script compares entities found by one pipeline and compares them against a gold standard. Here, we used the output of the old pipeline as gold standard. The test scripts requires the input to be in BioC format, an XML DTD focused on annotation [2]. Because of this, first the output of both pipelines was converted to BioC

The script calculates TP, FP, FN, as well as precision and recall values on a document basis, as well as average values for the entire data set evaluated. The evaluation script returned the following results:

```
OVERALL EVALUATION RESULTS

RESULTS FOR ENTITY TYPE Chemical
MACRO-AVG Precision: 0.8294495522
MACRO-AVG Recall: 0.853334601921

RESULTS FOR ENTITY TYPE ALL
MACRO-AVG Precision: 0.877999778493
MACRO-AVG Recall: 0.846837013278

RESULTS FOR ENTITY TYPE Disease
MACRO-AVG Precision: 0.930680723903
MACRO-AVG Recall: 0.83978660098

OVERALL SCORES
----
RESULTS FOR ENTITY TYPE Chemical
OVERALL PRECISION: 0.835085227273
OVERALL RECALL: 0.864601205941
OVERALL F-SCORE: 0.849586936102
MACRO-AVG F-Score: 0.882900453162
----
RESULTS FOR ENTITY TYPE Disease
OVERALL PRECISION: 0.946133662602
OVERALL RECALL: 0.826145552561
OVERALL F-SCORE: 0.882077847327
MACRO-AVG F-Score: 0.882900453162
```

Note though that precision and recall are measured against the output of the previous pipeline. This means that True Positives found by the new pipeline that the old pipeline did not find are treated as False Positives by the evaluation script. Below we list some examples of differences between what the pipelines produced.

ID	Article	Description
15552511	These indices include 3 types of measures, which are derived from a health professional [joint counts, global]; a laboratory [erythrocyte sedimentation rate (ESR) and C-reactive protein (CRP)]; or a patient questionnaire [physical function, pain, global].	Here the new pipeline doesn't mark *C-reactive protein* as an entity, but the old one does (False Negative)
15552512	Patient-derived measures have been increasingly recognized as a valuable means for monitoirng patients with rheumatoid arthritis.	The new pipeline lists both *rheumatoid arthritis* as well as *arthritis* as entities in separate entries. This behaviour is quite common: The new pipeline will try to match as many entities as possible. Other examples include *tumor necrosis* and *necrosis* (in article 15552517)
15552518	It is now accepted that rheumatoid arthritis is not a benign disease.	Here, the old pipeline marks *not a* as an entity, and lists it with the preferred form of *1,4,7-triazacyclononane-N,N',N"-triacetic acid*. This is obviously a mistake, which the new pipeline does not make.

In conclusion, there needs some more evaluation to find out where exactly the differences in the entities missed by the new pipeline stem from. Also, the new pipeline lists many entities several times, due to them having several entries with different IDs in the term list. This is a behavior that should be made optional.

At the same time, however, the increase in speed is promising that the new pipeline and its architecture are a good base for further expansion.

10. Future Work

The next step is add more modules to this pipeline. In particular, in order to be able to detect relations between found entities, the articles need to be dependency parsed. While the old pipeline uses a parser that has not been updated in quite some time, the Stanford parser is the current state-of-the-art, and thus a good candidate to integrate into the python-ontogene pipeline.

The Stanford parser, however, is implemented in Java; but has been interfaced for python. Still, work needs to be done in order to integrate it in the new pipeline and evaluate its efficacy.

11. Appendix

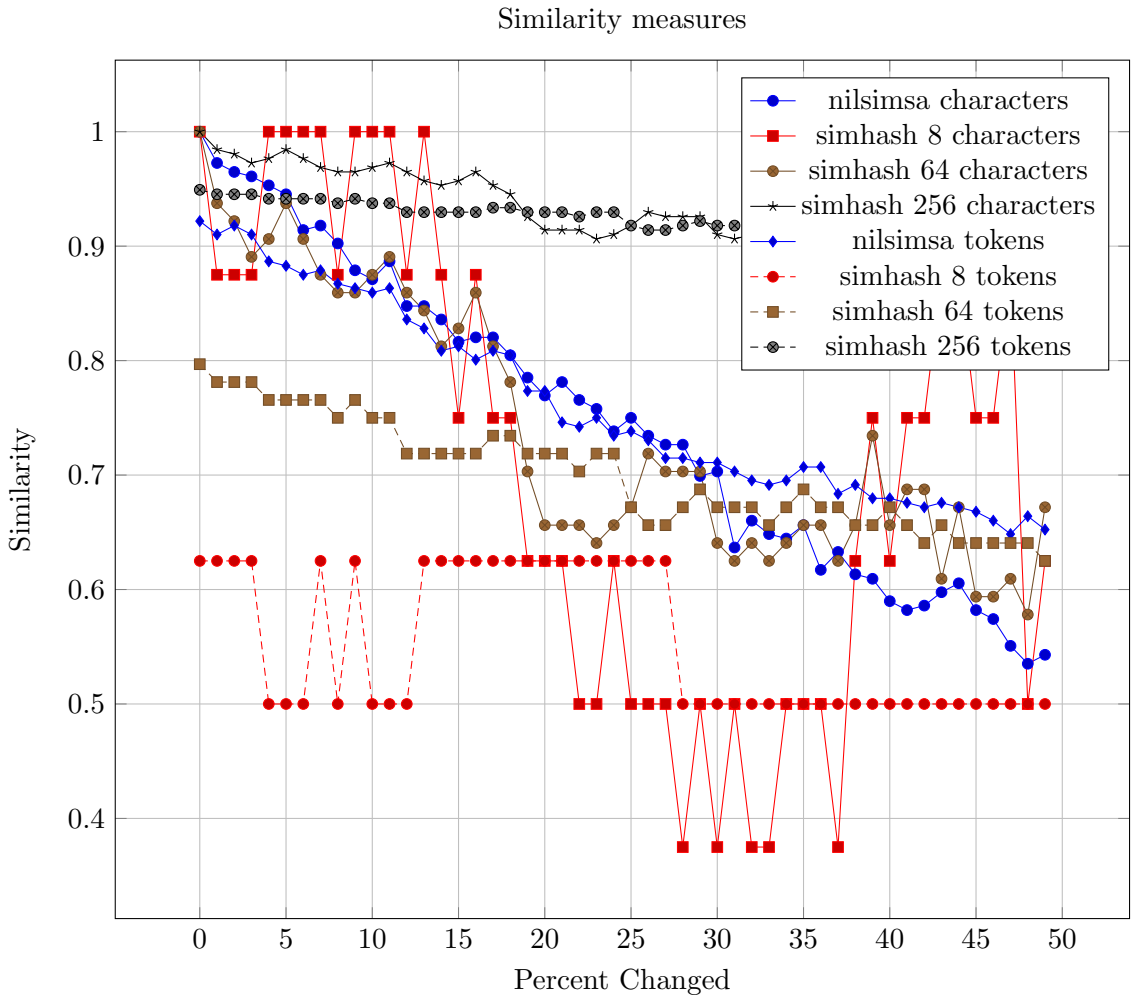
In the scope of this project we also explored briefly the following similarity algorithms. Since this module is not essential to the pipeline, and it was developed as a little side-project from the python-ontogene pipeline, it has not been modified to work with the Article class.

In order to be able to find similar articles given a specific one, for example in order to build special training or test data sets, we evaluated and compared two similarity algorithms.

The idea of such algorithms is to process an article and return a digest value, which can then be used to measure similarity of different articles efficiently. In order to do this, small changes in the original file should only lead to small changes in the resulting digest values. These algorithms are also called *locality-sensitive hashes*. In particular, we evaluated Nilsimsa [3] and Simhash [1].

From 100 arbitrarily selected abstracts from PubMed, variations were generated. Firstly, characters were randomly replaced in varying degrees of intensity (0 to 50%); and secondly, tokens were randomly replaced with other tokens from a list of named entities (again, 0 to 50% of tokens were replaced).

The Nilsimsa and Simhash values (with bitlengths of 8, 64 and 256) were computed, as well as the distance of the permutated files to the original article. Note that when comparing Simhash values, the algorithm will return the percentage of same bits in the hashes; whereas Nilsimsa comparison will return a value between -128 and 128. 128 means that the Nilsimsa values are equal, and any value over 24 indicates that the original messages are likely to be not independently generated. To compare, these values where normalised to map to [0, 1].


Similarity measures
nilsimsa characters
simhash 8 characters
simhash 64 characters
simhash 256 characters
nilsimsa tokens
simhash 8 tokens
simhash 64 tokens
simhash 256 tokens
Similarity
1
0.9
0.8
0.7
0.6
0.5
0.4
0
5
10
15
20
25
30
35
40
45
50
Percent Changed

References

[1] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[2] Donald C Comeau, Rezarta Islamaj Doğan, Paolo Ciccarese, Kevin Bretonnel Cohen, Martin Krallinger, Florian Leitner, Zhiyong Lu, Yifan Peng, Fabio Rinaldi, Manabu Torii, et al. Bioc: a minimalist approach to interoperability for biomedical text processing. *Database*, 2013:bat064, 2013.

[3] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. An open digest-based technique for spam detection. *ISCA PDCS*, 2004:559–564, 2004.

[4] Fabio Rinaldi, Thomas Kappeler, Kaarel Kaljurand, Gerold Schneider, Manfred Klenner, Simon Clematide, Michael Hess, Jean-Marc Von Allmen, Pierre Parisot, Martin Romacker, et al. Ontogene in biocreative ii. *Genome Biology*, 9(Suppl 2):S13, 2008.

[5] Fabio Rinaldi, Gerold Schneider, and Simon Clematide. Relation mining experiments in the pharmacogenomics domain. *Journal of Biomedical Informatics*, 45(5):851–861, 2012.

[6] Fabio Rinaldi, Gerold Schneider, Kaarel Kaljurand, Simon Clematide, Therese Vachon, and Martin Romacker. Ontogene in biocreative ii. 5. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 7(3):472–480, 2010.