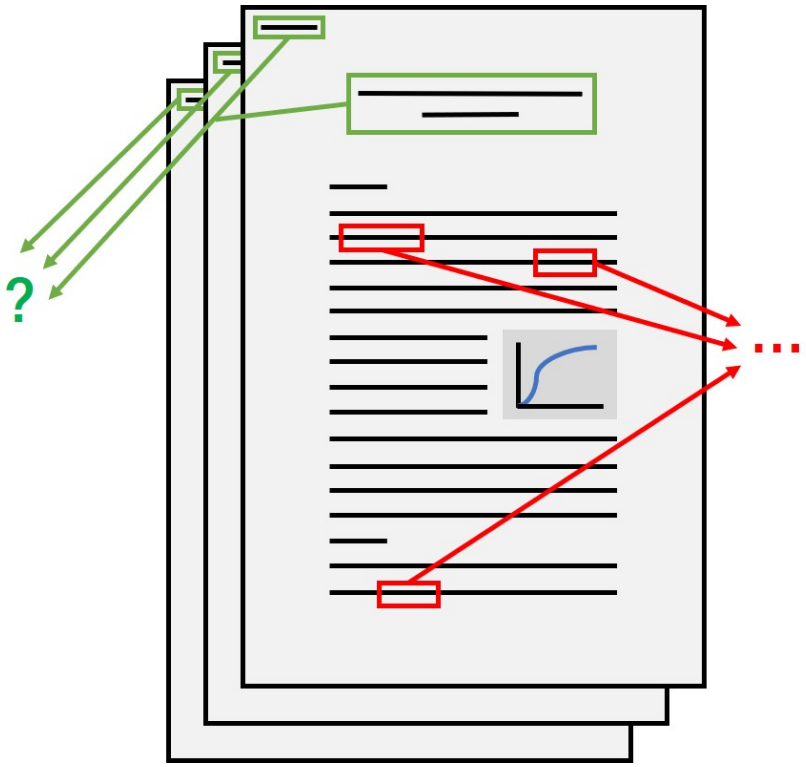
UNIVERSITY OF COPENHAGEN
FACULTY OF HEALTH AND MEDICAL SCIENCES



Project Outside Course for credit transfer students from SCIENCE

Should we treat articles equally?

Introducing article weights in text mining of the scientific literature

Zelin Li

Advisors: Lars Juhl Jensen and Katerina Nastou

February 3, 2022

This project has been submitted to the Faculty of Health and Medical Sciences, University of Copenhagen

Contents

Introduction

Text mining is the process that combines information from multiple papers to find connections from unstructured text data. Text mining tools have been extensively used in many disciplines such as drug discovery, proteomics, ecology, healthcare and medicine. The JensenLab tagger is a highly efficient tagging algorithm, used in biomedical text mining, implemented in C++, which can match a document against a dictionary of terms of interest, such as genes/proteins and diseases[1]. Associations between different terms are identified based on their co-occurrence in text. More specifically, different weights are attributed to the associations based on the distance of the entities, with higher scores for entities appearing in the same sentence, followed by entities in the same paragraph and finally entities in the same document; the co-occurrence score's computation first calculates a weighted count ($C_{(i,j)}$) for each pair of entities i and j as equation (1) shows:

$$C_{(i,j)} = \sum_{k=1}^{N} \delta_{d(i,j)k} w_d + \delta_{p(i,j)k} w_p + \delta_{s(i,j)k} w_s \tag{1}$$

$$S_{(i,j)} = C_{(i,j)}^{\alpha} \left(\frac{C_{(i,j)} C_{(\cdot,\cdot)}}{C_{(i,\cdot)} C_{(\cdot,j)}} \right)^{(1-\alpha)} \tag{2}$$

where w_d=1.0 (weight for co-occurrence within the same document), w_p=2.0 (same paragraph) and w_s=0.2 (same sentence) under default settings; if the entities i and j are co-mentioned in the document k, a paragraph of k or a sentence of k, then $\delta_{d(i,j)k}$, $\delta_{p(i,j)k}$ and $\delta_{s(i,j)k}$ are 1, and otherwise are 0 [1, 2]. Afterwards the weighted counts are used to calculate the final co-occurrence score ($S_{(i,j)}$) as shown in equation (2). The details of the calculations are explained in [2].

Notice that the information from different articles is considered equally "good" in the co-occurrence score calculation, and thus all articles carry equal weights in the final scores. Therefore, a trivial way to introduce

weights on different articles (documents) is to adjust the weighted count calculation as follows:

$$C_{(i,j)} = \sum_{k=1}^{N} \Big(\delta_{d(i,j)k} w_d + \delta_{p(i,j)k} w_p + \delta_{s(i,j)k} w_s \Big) w_k \tag{3}$$

where w_k is the 'corpus weight' of a document k (in the case where the corpus is the scientific literature the document is an article). This weight could be assigned by using a TSV file containing two columns: the article's PMID and its corpus weight.

The text-mined associations produced by tagger are used to populate various biological databases, such as the STRING database[3] of protein interactions. Thus, to test the implications of document weighing in the scores of text-mined associations, a comparison between the results produced with and without corpus weights was performed in this project. This allowed us to evaluate whether the latter can produce better quality scores. The weighting scheme that was tested in this project was based on the influence of the journal in which the paper is published in.

In this project, a software infrastructure to allow articles (documents) weighing was built successfully. As a first test case, the SJR weighting scheme was used and we evaluated its applicability. We opted for the SJR evaluation metrics instead of the most commonly known and used Clarivate Impact Factors (IF), because bulk access to the latter requires payment, and the calculations of the two metrics are roughly similar[1].

All in all, in the first part of this project, the aim is to implement the functionality to allow the addition of the optional input of corpus weights file; in the second part of this project, the aim is to acquire a weighting scheme that related to journal's influence, and then test whether we should treat the articles equally regardless of the journal in which they are published in or not.

[1] IF calculation: The Clarivate Analytics Impact Factor; SJR calculation: Description of Scimago Journal Rank Indicator

Methods

The working environments of this project are Linux operating systems, which include the Danish Life Sciences Supercomputer Computerome 2.0 (Linux Distribution: CentOS 7) and my personal laptop WSL (Windows Subsystem of Linux: Ubuntu 20.04.3 LTS). I used my laptop for upgrading/developing the JensenLab tagger software and making small local tests on it. As for generating the SJR weighting scheme and running the benchmarking experiments, I used the infrastructure provided by Computerome 2.0.

Tagger upgrade

The general idea to implement this new optional input is to change the code as follows: instead of first implementing the methods for reading and parsing the document and then changing the score calculation method, first, I change the score calculation method and manually assign weights to all the documents, so that I can always perform tests to verify the results. I went through the code to find the method that gets the final $C_{(i,j)}$ of each document. The class `ScoreDocumentHandler` will handle the process in the order of: `on_document_begin` → `on_paragraph_begin` → `on_sentence_begin` → ····· (multiple paragraphs and sentences) → `on_document_end`, therefore, `on_document_end` is the method I was looking for:

```
1 void ScoreDocumentHandler::on_document_end(Document& document) {
2     ScoreBatchHandler* score_batch_handler = (ScoreBatchHandler*)
      ↪  this->batch_handler;
3     this->commit_pairs(this->document_pair_set,
      ↪  score_batch_handler->document_weight);
4     this->commit_pairs(this->paragraph_pair_set,
      ↪  score_batch_handler->paragraph_weight);
5     this->commit_pairs(this->sentence_pair_set,
      ↪  score_batch_handler->sentence_weight);
6     /* omit code for update the overall pair score map (sum) */
7     ......
```

The above code uses the `commit_pairs` method to update this document's pair score map (`this->pair_score_map`) with three pair sets respectively, according to the method: if a pair in a `pair_set` matches the pair in this document's `pair_score_map`, then add the relevant weight (w_d, w_p or w_s) on this document's pair score map. I multiplied the corpus weight to each of the relevant weight before entering the `commit_pairs` method, then each time, the updated value will be the multiplication result rather

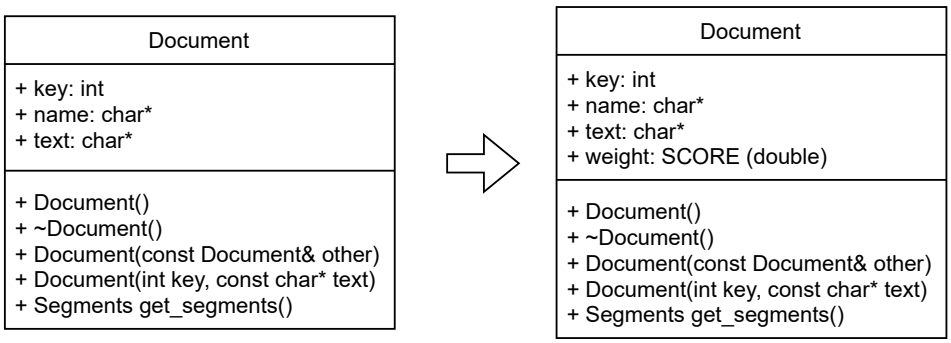


Figure 1: Class `Document` and its change

than the relevant weight, just like the code below[2]:

```
this->commit_pairs(this->xxx_pair_set,
↪  score_batch_handler->document_weight*(corpus_weight));
```

where `corpus_weight` can be introduced from the `document` object, and it is the input of `on_document_end`. By doing so, `corpus_weight` in the above code has to change to `document.weight`. The new attribute `weight` needs to add to the class `Document`, as Figure 1 shows.

This change makes each time the variation of the document's pair score map equal to $\delta_{d(i,j)k} w_d w_k + \delta_{p(i,j)k} w_p w_k + \delta_{s(i,j)k} w_s w_k$, which is equivalent to the right side of the equation (3) (without summing).

Now every document has a corpus weight. When the document is instantiated with `new Document()`, the weight assigned to it by default is 1.0.

The next step is to read the TSV-formatted corpus' weights file and assign a corpus weight to every document according to their identifier (an integer: `key`, in the case of scientific publication in PubMed, they are PMID). From `tagcorpus.cxx`, I notice that the documents are read by `TsvDocumentReader`:

```
    TsvDocumentReader *document_reader;
    if (validate_opt(documents)) document_reader = new
    ↪  TsvDocumentReader(documents);
    else document_reader = new TsvDocumentReader(stdin);
```

where `validate_opt()` is a function to check whether the optional input filename exists or not.

[2]For a complete modified version of the `on_document_end(Document& document)`, please see the Appendix on_document_end

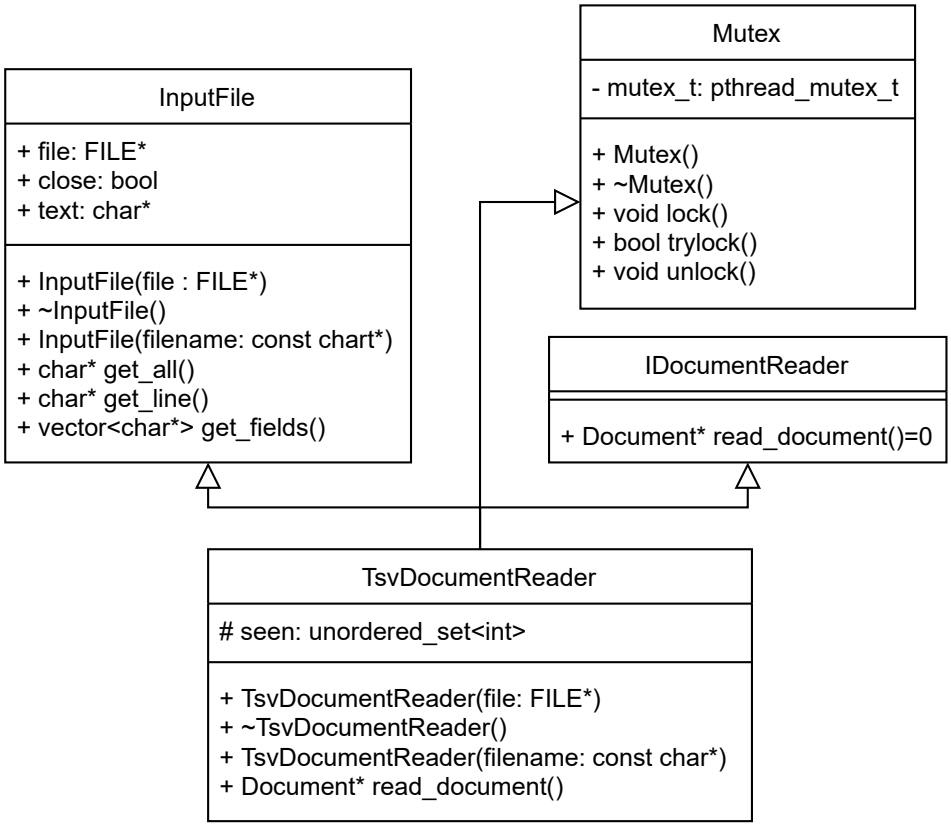


Figure 2: The diagram of class `TsvDocumentReader` and its parent classes `IDocumentReader`, `InputFile` and `Mutex` (multiple inheritance)

The documents file is a TSV-formatted file, and the method `read_document()` processes it; this method will return a `TsvDocument`[3] object, as Figure 2 shows. Therefore, the `weight` of the `TsvDocument` object should be assigned during the process of `read_document()`.

`read_document()` can assign the `weight` of `TsvDocument` object by mapping a `key` (in this case the PMID) to the `weight` using the following code[4]:

```
1 unordered_map<int, SCORE>::iterator it=weights.find(document->key);
2 if (it != weights.end()) document->weight = it->second;
```

where `weights` is the map to store the relation. The task is to read the TSV corpus weights file and record relations into the map (`weights`). Since

[3] `TsvDocument` inherits from `Document`, it has a protected `Segments` type attribute `segments` and a public `char*` type attribute `line`.

[4] For the complete modified version of `read_document()`, please see the Appendix read_document

method `get_fields()` in class `InputFile` can be used for reading and parsing the TSV file, method `load_weights` uses it:

```
void TsvDocumentReader::load_weights(InputFile file) {
    while (true) {
        vector<char*> fields = file.get_fields();
        int size = fields.size();
        if (size == 0) break;
        if (size >= 2 && *fields[1] != '\0' && *fields[1] != '\t')
        ↪  weights[atoi(fields[0])] = atof(fields[1]);
        for (vector<char*>::iterator it = fields.begin(); it !=
        ↪  fields.end(); it++) delete *it;
    }
}
```

For each line, a TAB separates the columns; `get_fields()` can extract the contents of each column and put them into one vector named `fields`; `load_weights` will stop reading the corpus weights file when it encounters the first empty row. `load_weights` can continue reading the file even though the row is incomplete (only one column has content) or exceeds two columns. If there are more than two columns, then `load_weights` only processes the first two columns. Notice that the first column is the `key` (PMID), and the second column is the `weight`.

Thus, the class `TsvDocumentReader` has changed to Figure 3 shows.

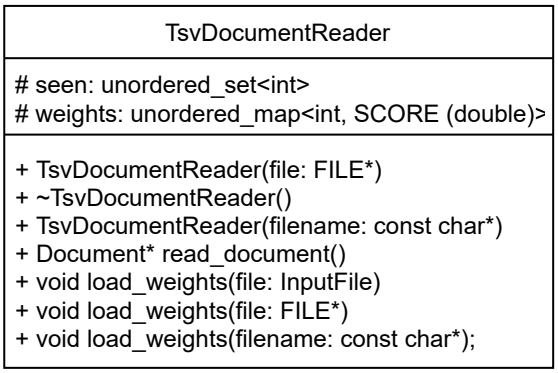


Figure 3: Changed class `TsvDocumentReader`

The final step is to alter the `tagcorpus.cxx` to enable the interface to show and take in the new optional input and let the −−h or −−help argument print the guidance[5]. The detailed changes are shown in the GitHub pull request new optional input (corpus weights) #1.

[5]see the Appendix tagcorpus guidance

Weighting scheme

Scimago Journal Rank (SJR) was used to obtain the weighting scheme in this project. I used `wget` to download the SJR directly. The downloaded file `journalrank.xls` is badly formatted; to use the semicolon as delimiters, I used the following command (convert `journalrank.xls` to `journalrank.txt`):

```
awk -F '"' -v OFS='' '{ for (i=2; i<=NF; i+=2) gsub(";", ",", \$i) } 1' journalrank.xls | sed -e 's/"-"//g' | sed -e 's/;-;/;;/g' | sed -e 's/"//g'  > journalrank.txt
```

After that, the ISSNs and SJR were selected with `awk` for further analysis. The processed `journalrank.txt` looks like the example below:

```
15424863, 00079235      62.937
10575987, 15458601      40.949
14710072, 14710080      37.461
00335533, 15314650      34.573
20588437        32,011
15518922, 15518930      28,083
```

The first column is two ISSNs (but it can have one ISSN only or even without any ISSN), and the second column is the SJR score (could be empty too).

The SJR scores statistics (in total 32953 journals, but only 32604 journals have scores available) of all the journals are illustrated in Table 1.

max	median	min	mean	std*
62.937	0.259	0.100	0.572	1.175

Table 1: SJR scores statistics; *std: standard deviation

Notice that for each journal, there could have more than one ISSN. For example, a journal could have one ISSN that is the ISSN of the printed version, while having another is the online/electronic ISSN, but a single ISSN-L (linking ISSN) is designated for all media of a serial publication, irrespective of how many there are[4].

In the JensenLab database, there are multiple XML files to store the information about the articles, including the article's first time published ISSN (in the database, the type of ISSN could only either be 'Print' or 'Electronic') and its ISSN-L. Each article in the XML file is a <PubmedArticle>. The structure of a <PubmedArticle> is like the example shown in Figure 4.

It is clear that the PMID is in <MedlineCitation> and the ISSN-L is in <MedlinejournalInfo>, while the first published ISSN (in this example

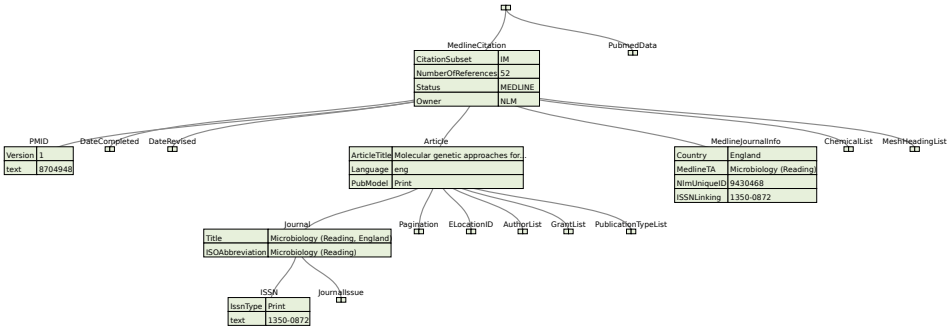


Figure 4: Example of the structure of a <PubmedArticle> in XML format

ISSN of the printed version) is in <Article>. Based on these observations, a Python script for reading and parsing the PMID and ISSNs of the XML files were created and shown in the Appendix Get the PMID and its ISSNs.

After extraction, the ISSNs have "-" between the first four digits and the last four digits. Using `awk` to remove these "-" and then the results are like the example below:

```
6149890 00702137    Current topics in cellular regulation   00702137
↪  Curr Top Cell Regul
6149891 00702153    Current topics in developmental biology 00702153
↪  Curr Top Dev Biol
6149892 00702153    Current topics in developmental biology 00702153
↪  Curr Top Dev Biol
6149893 00114162    Cutis   00114162    Cutis
6149894 00094722    Der Chirurg; Zeitschrift fur alle Gebiete der
↪  operativen Medizen  00094722    Chirurg
6149895 00014001    Chirurgie; memoires de l'Academie de chirurgie
↪  00014001    Chirurgie
```

Columns are separated by TAB. The first column is the PMID; the second column is the first published ISSN (print ISSN or electronic ISSN); the third column is the title of the journal retrieved from its first published ISSN; the fourth column is ISSN-L, and the fifth column is the title of the journal retrieved from ISSN-L, which is commonly the abbreviation.

Finally, all articles in the JensenLab abstracts database obtain corpus weights through scripts similar to the example in Appendix example script. In total, 2794 scripts were automatically generated and distributed through the batch system (TORQUE resource manager[5]).

In the example script, `pubmed21n0001.tsv` is the file extracted from the original XML file `pubmed21n0001.xml.gz`, and `pubmed21n0001.tsv` has five columns as the above example shows. `journalrank.txt` is the SJR that I have processed previously, which has two columns (ISSNs and SJR score). The final output in this example is `Wc_pubmed21n0001.tsv`, which has two columns; the first column is the PMID, the second column is the SJR

score. Thus, the last step for generating the weighting scheme is combining all the `Wc_xxx.tsv` files using `cat` and then `sort` and `uniq`. The results `all_uniq_Wc.tsv` is the weighting scheme (corpus weights file) I want, and the following content is a fraction of it:

```
25974306        7.985
25974307        1.240
25974308        0.701
25974309        1.240
25974310        0.833
25974311        1.441
25974312        1.028
2597431     2.287
25974313        1.053
25974314        1.088
```

Benchmarking

Having the SJR weighting scheme and the upgraded tagger, the final step of this project is benchmarking. First, I had to read the specified abstracts and full texts as input, then one script runs with the old version of tagger without the weighting scheme, another script runs with the new version tagger, which has the weighting scheme[6]. Notice that the new version tagger can also running without the corpus weights file, and it will output the same results as the old version tagger since it will automatically set all documents' corpus weight to 1.0.

After running the tagger two times, I got two output files, they are: `old_all_pairs.tsv` (without any weighting scheme); `new_all_pairs.tsv` (with the SJR weighting scheme). Then run the Perl script in the Appendix (`altered_create_pairs.pl`) to replace the entities' serial number with their type and identifier by using the file `all_entities.tsv`. The script also keep the raw co-occurrence score instead of calculating the normalized Z-score or star score[6]. Since I only want to benchmark against one species (human), it is not necessary to eliminate the effects of inter-species differences, and the normalized score is not required for the benchmark.

I ran the script two times with different input and output names (replace 'xxx' with 'new' and 'old'). The outputs are `old_database_pairs.tsv` and `new_database_pairs.tsv`; they have five columns, the first two columns are the first entity's type and its identifier; the third and fourth columns are the second entity's type and identifier and the last column is the raw co-occurrence score of this pair.

[6]For the complete related command in the script, please see the Appendix with the SJR weighting scheme

As previously mentioned, the benchmarking considers only pairs between human proteins. So the next step of benchmarking is to run the script[7] to select human protein pairs and prepare data for the downstream analysis.

In the script, the cutoff were is set to those pairs within the top 10^4, 10^5, 10^6 and 10^7 highest co-occurrence score (if the cutoff range is greater or equal to `max`, then it is automatically set to `max`), because from the overview (Appendix Figure 6) I got, I noticed that the performance differences (cumulative true positive counts) between the two are too small, by setting the cutoff range exponentially, I can quickly find out the scale of the differences. Finally, all the cutoff files and the original file (all pairs) are taken as the input of a Python script[8] to compare with the KEGG[7] pathway gold standard dataset and output the true positive and false positive cumulative counts.

After running the script, the cumulative counts files were generated (the row counts of which are shown in Appendix Table 4). Each pair of 'old' and 'new' files were concatenated together to form the overall cumulative counts files, and they served as the input of an R script[9] for plotting the results.

Results and Discussion

Before testing on the weighting scheme, I checked whether the software works or not by doing two local tests:

In the first test, set 1, 2, and 3.7 as all documents' corpus weights (run three experiments: **1**, **2**, and **3.7**), Figure 2 shows the results for a specific pair 1845149-1846565.

scheme	1_{st} entity i	2_{nd} entity j	$S_{(i,j)}$
1	1845149	1846565	2.520334
2	1845149	1846565	3.820112
3.7	1845149	1846565	5.525597
with	1845149	1846565	3.281015
without	1845149	1846565	2.520334

Table 2: The first test set all weights equal to **1**, **2**, and **3.7** respectively and then observe the co-occurrence score results on the same pair; the second test set two different weights (1.0 and 3.0) to two documents (**with**), and don't assign any weights to all documents (**without**), then observe the co-occurrence score results on the same pair

In the second test, one experiment set the corpus weights of two documents to 3.0 and 1.0, while the other does not assign corpus weights to

[7] see the Appendix `sel_hum.sh`

[8] `string_score_benchmark.1.1.py`, can be found within JensenLab

[9] `string_score_benchmark_plots.1.1.R`, can be found within JensenLab

any documents (run two experiments: **with** and **without**). Using the same specific pair as the first test used, the results are shown in Figure 2 too.

From these two tests, one can see that the new functionality of the JensenLab tagger was implemented successfully. Now the tagger can take a new optional input file containing document identifier as its first column and corpus weight as its second column, and it will give the correct calculation. Thus, the next step is to benchmark tagger with and without using the SJR weighting scheme.

First, an observation on the tagger outputs `xxx_database_pairs_human-only.tsv` were made ('xxx' is 'old' or 'new', 'old' is the old version tagger without any weighting scheme, and 'new' is the new version tagger run with SJR weighting scheme). Among the pairs that only included human proteins, I recorded 26,516,295 pairs in total.

In Table 3a, I recorded the highest, median, lowest, mean, and standard deviation of co-occurrence score among all human protein pairs in two different tagger runs ('no SJR' is ran on the old version tagger without any weighting scheme, while 'with SJR' is ran on the new version tagger that runs using SJR weighting scheme).

After running the script `string_score_benchmark.1.1.py` and obtaining the cumulative counts, I also calculate for Table 3b which shows the highest, median, lowest, mean and standard deviation of co-occurrence score among pairs that both proteins are included in the gold standard in two different tagger runs, this is because `string_score_benchmark.1.1.py` ignores the protein pairs, for which both proteins are not both included in the gold standard.

$S_{(i,j)}$	max	median	min	mean	std
no SJR	4811.084485	1.672147	0.021292	4.13588	16.1775
with SJR	7365.565793	2.366951	0.010575	7.45176	28.6106

(a) Statistics of $S_{(i,j)}$ of all human protein pairs

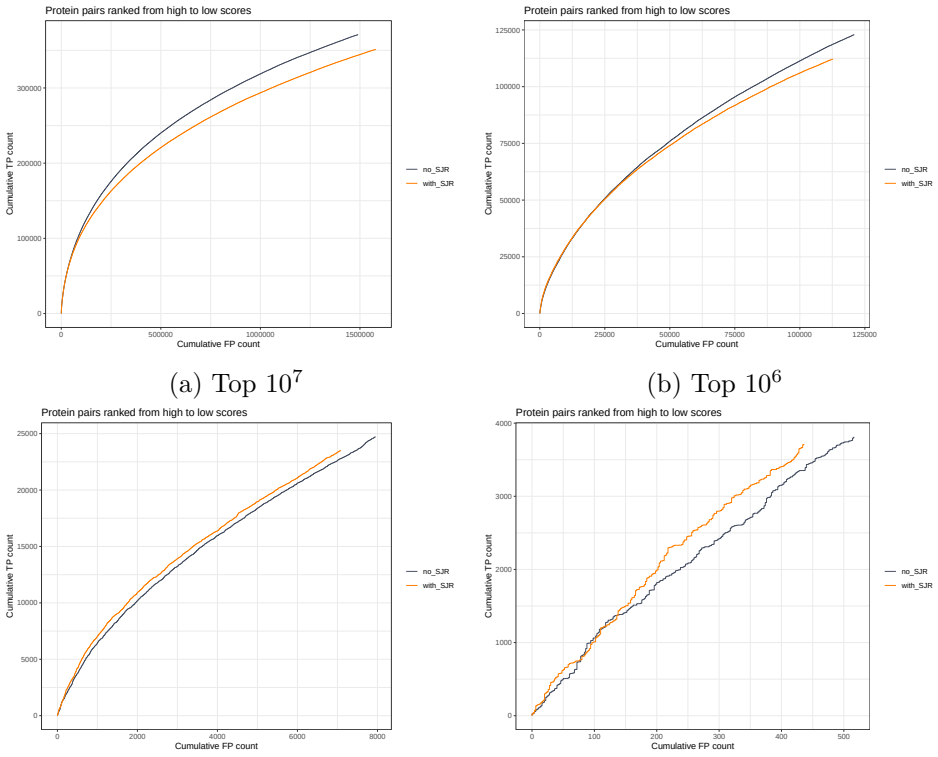
$S_{(i,j)}$	max	median	min	mean	std
no SJR	4811.084485	1.6059	0.081193	4.79059	23.1266
with SJR	7365.565793	2.375879	0.013286	8.57118	39.1322

(b) Statistics of $S_{(i,j)}$ of human protein pairs that both proteins included in the gold standard

Table 3: Statistics of co-occurrence score ($S_{(i,j)}$)

From Table 3 I can see that no matter if it is all human protein pairs or only human protein pairs of proteins present in the gold standard, the 'no SJR' weighting scheme tends to give lower scores (lower median and mean), and the scores have less undulation (lower standard deviation) compared to 'with SJR'.

Then, I observed the cumulative counts of true positive and false positive pairs. An overview is shown in Appendix Figure 6. Looking closer, Figure 5 provides different scaling; it presents the true positive and false positive cumulative counts of the top 10^7, 10^6, 10^5 and 10^4 highest scoring pairs.



(a) Top 10^7

(b) Top 10^6

(c) Top 10^5

(d) Top 10^4

Figure 5: The cumulative counts of false positive and true positive with the human protein pairs that their co-occurrence score cutoff at different ranks

From these results, I know that for the pairs at higher rank (higher co-occurrence score), the SJR weighting scheme will give better predictions on correlation, while for the lower co-occurrence scores, when using the SJR weighting scheme, the cumulative true positive count will be lower compared to not using the scheme.

One possible explanation for these results is that highly influential articles and journals are more likely to be review articles/journals. The reviews tend to describe definite results rather than papers with experimental results that may lack verification – some experiments may not be credible enough and could give biased results and conclusions, and they may be difficult to repeat by other researchers. Some less influential journals can still publish these results, while the review articles might not record them.

On the other hand, the reviews will have higher SJR, which makes the

SJR weighting scheme perform better in high-ranking pairs. Another issue is that the SJR scores are neither distributed uniformly nor do they follow the normal distribution. According to the median in Table 1 and Table 3, we know that there are more journals/articles having low SJR scores than those having high SJR scores, which not only increases the discrimination (Assume articles have different qualities, and they should have different weights, then now its quality is defined by the SJR score of the journal they published on, rather than itself, which is a kind of discrimination), but also makes articles with lower SJR contribute far less than highly influential articles to the overall correlation. When using the SJR weighting scheme, the majority of pairs (low scoring pairs in lower ranks) will have a lower cumulative true positive count than when not using weighting scheme.

The solution for improving the performance could be normalizing the SJR scores, for example, using the logarithm transformation or min-max normalization. If it is still not ideal, we can try some different weighting schemes, for example, using the number of citations of each article. Besides those options, filtering the journals or articles type could also be a choice.

Appendix

Full version of updated `on_document_end` method.

```
void ScoreDocumentHandler::on_document_end(Document& document)
{
    ScoreBatchHandler* score_batch_handler = (ScoreBatchHandler*)
    ↪  this->batch_handler;
    this->commit_pairs(this->document_pair_set,
    ↪  score_batch_handler->document_weight*(document.weight));
    this->commit_pairs(this->paragraph_pair_set,
    ↪  score_batch_handler->paragraph_weight*(document.weight));
    this->commit_pairs(this->sentence_pair_set,
    ↪  score_batch_handler->sentence_weight*(document.weight));
    if (this->pair_score_map.size() >= 10000) {
        score_batch_handler->lock();
        score_batch_handler->pair_score_map +=
        ↪  this->pair_score_map;
        score_batch_handler->unlock();
        this->pair_score_map.clear();
    }
    else if (this->pair_score_map.size() >= 1000 and
    ↪  score_batch_handler->trylock()) {
        score_batch_handler->pair_score_map +=
        ↪  this->pair_score_map;
        score_batch_handler->unlock();
        this->pair_score_map.clear();
    }
}
```

Full version of updated `read_document` method.

```
Document* TsvDocumentReader::read_document()
{
    TsvDocument* document = new TsvDocument();
    bool valid = false;
    while (!valid) {
        this->lock();
        char* line = this->get_line();
        document->line = line;
        int index = 0;
        if (line) {
```

```
            // Find first key and skip additional keys.
            while (line[index] != '\0' && line[index] != '\t' &&
            ↪  line[index] != ':') {
                ++index;
            }
            if (line[index] == ':') {
                char* key = line+index+1;
                while (line[index] != '\0'  && line[index] != '\t'
                ↪  && line[index] != '|') {
                    ++index;
                }
                char replaced = line[index];
                line[index] = '\0';
                document->key = atoi(key);
                line[index] = replaced;
                // Check if key was previously seen.
                if (document->key && this->seen.find(document->key)
                ↪  == this->seen.end()) {
                    this->seen.insert(document->key);
                    valid = true;
                    if (this->seen.size() % 1000 == 0) {
                        cerr << "# Read documents (in thousands): "
                        ↪  << this->seen.size()/1000 << "\r";
                    }
                }
            }
        }
        this->unlock();
        if (valid) {
            while (line[index] != '\0' && line[index] != '\t') {
                ++index;
            }
            // Skip authors.
            if (line[index] != '\0') {
                do {
                    ++index;
                } while (line[index] != '\0' && line[index] !=
                ↪  '\t');
            }
            // Skip journal.
```

```
            if (line[index] != '\0') {
                do {
                    ++index;
                } while (line[index] != '\0' && line[index] !=
                ↪  '\t');
            }
            // Skip year.
            if (line[index] != '\0') {
                do {
                    ++index;
                } while (line[index] != '\0' && line[index] !=
                ↪  '\t');
            }
            // Find text.
            if (line[index] != '\0' && line[index+1] != '\0') {
                document->text = line+index+1;
            }
            else {
                valid = false;
            }
            // Map weight.
            unordered_map<int, SCORE>::iterator it =
            ↪  weights.find(document->key);
            if (it != weights.end()) {
                document->weight = it->second;
            }
        }
        else if (line) {
            free(line);
        }
        else {
            delete document;
            document = NULL;
            valid = true;
        }
    }
    return document;
}
```

Run tagger with the SJR weighting schema.

```
gzip -cd `ls -1
↪  /home/projects/ku_10024/data/databases/pmc/*.en.merged.filtered.tsv.gz`
↪  `ls -1r /home/projects/ku_10024/data/databases/pubmed/*.tsv.gz` | cat
↪  /home/projects/ku_10024/data/textmining/excluded_documents.txt - |
↪  /home/projects/ku_10024/people/zelili/tagger/tagcorpus --threads=40
↪  --autodetect
↪  --types=/home/projects/ku_10024/data/dictionary/curated_types.tsv
↪  --entities=/home/projects/ku_10024/data/dictionary/all_entities.tsv
↪  --names=/home/projects/ku_10024/data/dictionary/all_names_textmining.tsv
↪  --groups=/home/projects/ku_10024/data/dictionary/all_groups.tsv
↪  --stopwords=/home/projects/ku_10024/data/dictionary/all_global.tsv
↪  --local-
↪  stopwords=/home/projects/ku_10024/data/dictionary/all_local.tsv
↪  --corpus-weights=./all_uniq_Wc.tsv --type-
↪  pairs=/home/projects/ku_10024/data/dictionary/all_type_pairs.tsv
↪  --out-matches=./taggerout/new_all_matches.tsv
↪  --out-segments=./taggerout/new_all_segments.tsv
↪  --out-pairs=./taggerout/new_all_pairs.tsv
```

`tagcorpus` guidance.

```
Usage: ./tagcorpus [OPTIONS]
Required Arguments
        --types=filename
        --entities=filename
        --names=filename
Optional Arguments
        --documents=filename    Read input from file instead of
        ↪  from STDIN
        --groups=filename
        --type-pairs=filename   Types of pairs that are allowed
        --stopwords=filename
        --local-stopwords=filename
        --autodetect Turn autodetect on
        --tokenize-characters Turn single-character tokenization on
        --corpus-weights=filename   If not specify then all weights
        ↪  default 1.0
        --document-weight=1.00
        --paragraph-weight=2.00
        --sentence-weight=0.20
        --normalization-factor=0.60
        --threads=1
```

```
        --out-matches=filename
        --out-pairs=filename
        --out-segments=filename
```

Get the PMID and its ISSNs.

```
import sys
import gzip
import xml.etree.ElementTree as ET
import csv
#first argv is input gz file
tree = ET.parse(gzip.open(sys.argv[1], 'r'))
root = tree.getroot()
#second argv is output tsv file
with open(sys.argv[2], 'w') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    for child in root:
        try:
            medcite = child.find("MedlineCitation")
            try: pmid = medcite.find("PMID").text
            except: pmid = " "
            try:
                journal = medcite.find("Article").find("Journal")
                try: issn = journal.find("ISSN").text
                except: issn = " "
                try: jname = journal.find("Title").text
                except: jname = " "
            except: pass
            try:
                medlinej = medcite.find("MedlineJournalInfo")
                try : issnlin = medlinej.find("ISSNLinking").text
                except: issnlin = " "
                try : sjname = medlinej.find("MedlineTA").text
                except: sjname = " "
            except: pass
        except: pass
        tsv_writer.writerow([pmid, issn, jname, issnlin, sjname])
```

Example script for getting the SJR scores of each XML file.

```
#!/bin/bash
sjr="/.../tagger_test/journalrank.txt"
input="pubmed21n0001.tsv"
Wc=Wc_$input
cd /.../tagger_test/xmlout
for PMIDandISSNs in $(awk -F '\t' '{print $1","$2","$4}' $input | sed s/[[:space:]]//g); do # tr -s [:space:], remove more than one space
    IFS=, read PMID ISSN1 ISSN2 <<< $PMIDandISSNs
    if [ -n "$PMID" ]; then
        if [ -n "$ISSN1" ] && [ ! -n "$ISSN2" ]; then
            SJR=`grep $ISSN1 $sjr | awk -F '\t' '{print $2}' | sed s/[[:space:]]//g`
            if [ -n "$SJR" ]; then
                echo "$PMID $ISSN1 NoISSN2 ISSN1_SJR_IsFound"
                printf "${PMID}\t${SJR}\n" >> $Wc
            else
                echo "$PMID $ISSN1 NoISSN2 ISSN1_SJR_NotFound"
            fi
        elif [ ! -n "$ISSN1" ] && [ -n "$ISSN2" ]; then
            SJR=`grep $ISSN2 $sjr | awk -F '\t' '{print $2}' | sed s/[[:space:]]//g`
            if [ -n "$SJR" ]; then
                echo "$PMID NoISSN1 $ISSN2 ISSN2_SJR_IsFound"
                printf "${PMID}\t${SJR}\n" >> $Wc
            else
                echo "$PMID NoISSN1 $ISSN2 ISSN2_SJR_NotFound"
            fi
        elif [ -n "$ISSN1" ] && [ -n "$ISSN2" ]; then
            SJR=`grep $ISSN1 $sjr | awk -F '\t' '{print $2}' | sed s/[[:space:]]//g`
            if [ -n "$SJR" ]; then
                echo "$PMID $ISSN1 $ISSN2 ISSN1_SJR_IsFound"
                printf "${PMID}\t${SJR}\n" >> $Wc
            else
                SJR=`grep $ISSN2 $sjr | awk -F '\t' '{print $2}' | sed s/[[:space:]]//g`
                if [ -n "$SJR" ]; then
                    echo "$PMID ISSN1_NoSJR $ISSN2
                    ISSN2_SJR_IsFound"
```

```
                    printf "${PMID}\t${SJR}\n" >> $Wc
                else
                    echo "$PMID $ISSN1 $ISSN2 Both_SJR_NotFound"
                fi
            fi
        else
            echo "$PMID Both_ISSN_NotFound"
        fi
    else
        echo "$ISSN1 $ISSN2 PMID_NotFound"
    fi
done
sed -i 's/,/./g' $Wc
```

The `altered_create_pairs.pl` for linking the entities' serial number to their type and identifier.

```
#!/usr/bin/perl -w
use strict;
use POSIX;
# factor for bin size. The larger, the smaller the bins
my $BINFACTOR = 200;
# read in entities file
my %serial_type = ();
my %serial_type_identifier = ();
open IN, "<
↪  /home/projects/ku_10024/data/dictionary/all_entities.tsv";
while (<IN>) {
        s/\r?\n//;
        my ($serial, $type, $identifier) = split /\t/;
        $serial_type{$serial} = $type;
        $serial_type_identifier{$serial} = $type."\t".$identifier;
}
close IN;
# raw textmining scores
open IN, "< xxx_all_pairs.tsv";
open OUT, "> xxx_database_pairs_orig.tsv";
while (<IN>) {
        s/\r?\n//;
        my ($serial1, $serial2, $raw_score, undef) = split /\t/;
        next unless exists $serial_type_identifier{$serial1} and
        ↪  exists $serial_type_identifier{$serial2};
```

```
        my $type1 = $serial_type{$serial1};
        my $type2 = $serial_type{$serial2};
        my $types;
        if ($type1 <= $type2) {
                $types = $type1."\t".$type2;
        }
        else {
                $types = $type2."\t".$type1;
        }
        print OUT $serial_type_identifier{$serial1}, "\t",
        ↪  $serial_type_identifier{$serial2}, "\t", $raw_score,
        ↪  "\n";
        print OUT $serial_type_identifier{$serial2}, "\t",
        ↪  $serial_type_identifier{$serial1}, "\t", $raw_score,
        ↪  "\n";
}
close IN;
close OUT;
close STDERR;
close STDOUT;
POSIX::_exit(0);
```

Select human protein pairs and generating files for getting the cumulative counts.

First, the script uses `awk` to select pairs that have both entities are human proteins (type>0 is protein, type=9606 is human protein), then the script sorts them by co-occurrence score in descending order and gets `4_columns_input.tsv` (first two columns are the protein's identifier, the third column is the raw co-occurrence score, the fourth column is the label 'no_SJR' or 'with_SJR'). After that, it will iterate on the predefined range of cutoff and create files that are originated from `4_columns_input.tsv` but having different cutoff ranges (smaller than the total number of pairs - `max`).

```
#!/bin/bash
human() {
    in=$1
    cutoff_downrange=$2
    cutoff_uprange=$3
    awk -v sjr="$4" '{if($1==9606 && $3==9606) {print
    ↪  $2"\t"$4"\t"$5"\t"sjr}}' ${in}.tsv > ${in}_human-only.tsv
```

```
    cat ${in}_human-only.tsv | sort -g -k3,3rn >
    ↪ ${in}_4_columns_input.tsv
    max=`wc -l ${in}_4_columns_input.tsv | awk '{print $1}'`
    for range in all `seq $cutoff_downrange $cutoff_uprange`; do
        if [ "$range" == "all" ]; then
            cutoff=$range
            cp ${in}_4_columns_input.tsv
            ↪ ${in}_4_columns_input_${cutoff}.tsv
        else
            let cutoff=10**$range
            if [ $cutoff -ge $max ]; then let cutoff=$max; fi
            head -$cutoff ${in}_4_columns_input.tsv >
            ↪ ${in}_4_columns_input_${cutoff}.tsv
        fi
        python3 string_score_benchmark.1.1.py -i
        ↪ ${in}_4_columns_input_${cutoff}.tsv -g
        ↪ benchmark_kegg_2col.tsv -o
        ↪ ${in}_cumulative_counts_${cutoff}.tsv
    done
}
human old_database_pairs 4 7 no_SJR
human new_database_pairs 4 7 with_SJR
```

file	all	10^7	10^6	10^5	10^4
no SJR*	9834128	1861774	243836	32676	4330
with SJR*	9834128	1930272	224816	30588	4150

Table 4: Row counts of different cumulative counts files; *: 'no SJR' is using the original tagger that without introducing any weighting schema, while 'with SJR' is using tagger with the SJR weighting schema

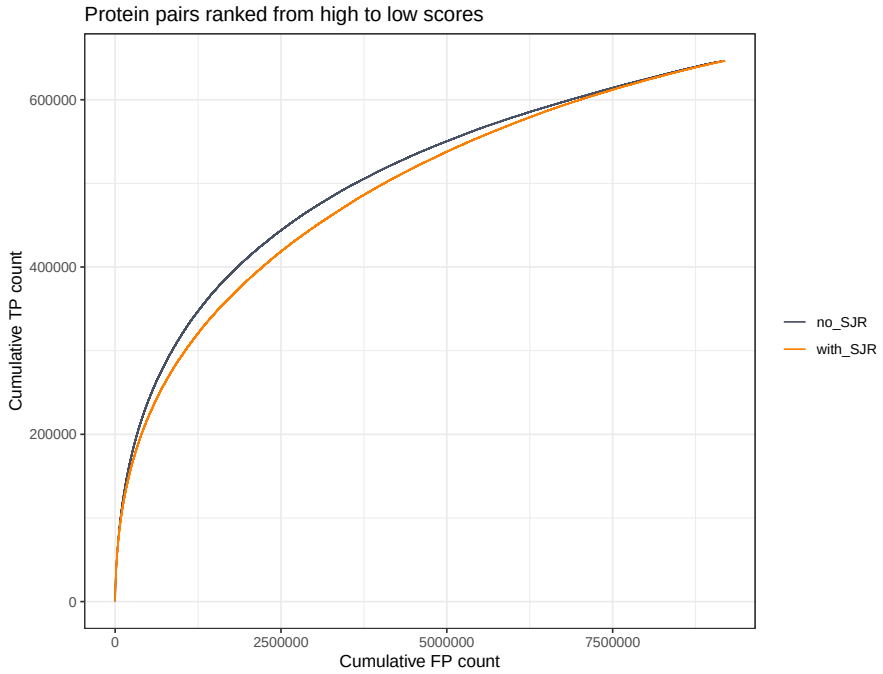


Figure 6: The cumulative counts of false positive and true positive with all human protein pairs

References

[1] Helen V. Cook and Lars Juhl Jensen. "A Guide to Dictionary-Based Text Mining". In: *Bioinformatics and Drug Discovery*. Ed. by Richard S. Larson and Tudor I. Oprea. New York, NY: Springer New York, 2019, pp. 73–89. ISBN: 978-1-4939-9089-4. DOI: 10.1007/978-1-4939-9089-4_5.

[2] Andrea Franceschini et al. "STRING v9.1: protein-protein interaction networks, with increased coverage and integration". In: *Nucleic Acids Research* 41.D1 (Nov. 2012), pp. D808–D815. ISSN: 0305-1048. DOI: 10.1093/nar/gks1094.

[3] Damian Szklarczyk et al. "The STRING database in 2021: customizable protein–protein networks, and functional characterization of user-uploaded gene/measurement sets". In: *Nucleic Acids Research* 49.D1 (Nov. 2020), pp. D605–D612. ISSN: 0305-1048. DOI: 10.1093/nar/gkaa1074.

[4] ISSN InterNational Centre. *The ISSN-L for publications on multiple media*. URL: https://www.issn.org/understanding-the-issn/assignment-rules/the-issn-l-for-publications-on-multiple-media/ (visited on 01/28/2022).

[5] Garrick Staples. "TORQUE Resource Manager". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 8–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188464.

[6] Sune Pletscher-Frankild et al. "DISEASES: Text mining and data integration of disease–gene associations". In: *Methods* 74 (2015). Text mining of biomedical literature, pp. 83–89. ISSN: 1046-2023. DOI: https://doi.org/10.1016/j.ymeth.2014.11.020.

[7] Minoru Kanehisa and Susumu Goto. "KEGG: Kyoto Encyclopedia of Genes and Genomes". In: *Nucleic Acids Research* 28.1 (Jan. 2000), pp. 27–30. ISSN: 0305-1048. DOI: 10.1093/nar/28.1.27.