Modeling and optimization of an online publishing application

Milos Cuculovic

Université de Haute-Alsace

Mathématiques, Sciences de l'Information et de l'Ingénieur

Institut de Recherche en Informatique, Mathématique, Automatique et Signal

DOCTORAL THESIS

Modeling and optimization of an online publishing application

---

Author: Milos Cuculovic

Examiners:

Prof. Abderrafiaa Koukam

Prof. Ileana Ober

Dr. Cédric Dumoulin

Supervisors:

Prof. Michel Hassenforder

Dr. Frédéric Fondement

Dr. Maxime Devanne

A thesis submitted in fulfillment of the requirements

for the degree of Doctor of Philosophy in Computer Sciences

January 5, 2023

Declaration of Authorship

I, Milos Cuculovic, declare that this thesis titled, "Modeling and optimization of an online publishing application" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this university;
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
- Where I have consulted the published work of others, this is always clearly attributed;
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

SIGNATURE: Milos Cuculovic

DATE: January 5, 2023

Modeling and optimization of an online publishing application

by Milos Cuculovic

Abstract

In recent years, we have been observing a constant increase in the number of scientific articles. This phenomenon is mainly due to the rapid development of science and technology and the growth in higher education. Each of these articles has to go through a laborious process, from peer review through author revision rounds to the final decision made by the Editor-in-Chief. In addition to the pressure to publish scientific papers, senior scientists are also asked to conduct peer review and be members of journal editorial boards and part of conference committees where they are in charge of, in addition to many other activities, making final decisions regarding new articles' acceptance.

We propose within this PhD thesis a set of tools to be used in the academic publishing process by authors, peer-reviewers and Editors-in-Chief with the purpose of automating parts of their activities. Those tools are based on two research topics: document comparison and Named Entity Recognition (NER). Regarding the document comparison part, we propose a novel XML diff algorithm, called jats-diff, able to make a bijection between the author modifications and the changes observed between two article versions. As regards the NER part, a deep learning neural network model was trained, able to annotate review comments and extract meaningful information.

During the work on the document comparison, 12 existing XML diff algorithms were assessed with the purpose of evaluating their comparison capacities for JATS XML documents—JATS being the de facto standard for the XML representation of academic articles. Most of those algorithms only support basic edit pattern detection (insert, delete, update and move); however, their capacity to detect higher-level edit

patterns (structural and style changes) is very limited. Due to that limitation, none of the tested XML diff algorithms were suitable for JATS document comparison, and there is a need for a new XML diff algorithm able to make a bijection between author edits on one side and changes detected on their respective XML versions on the other side. For this, the jats-diff algorithm was developed allowing a human readable author change extraction during the revision rounds, which allows the automation of article comparison activity made by peer-reviewers and Editors-in-Chief. Using jats-diff, extracting actual changes made by the authors is made possible.

In the NER part, we first assessed different NER approaches and focused our research on deep learning models that are achieving state-of-the-art results in many natural language processing (NLP) tasks. After assessing different models that we further fine-tuned on the reviewer comments annotation task, we created the "review-annotation" model based on SciBERT, which is able to achieve an weighted average F1 score of 0.87. Use of this model will allow authors and Editors-in-Chief to better understand the review request by having the four important named entities extracted: *Location*, *Action*, *Modal* and *Trigger*. Using the "review-annotation" model, extracting the requested changes made by the reviewers is made possible.

Finally, the correlation of the actual changes provided by jats-diff and the requested changes provided by the "review-annotation" NER model was carried out. This combined information will allow the Editor-in-Chief to assess the requested changes asked by the reviewer and the actual changes made by the authors with the purpose of automating the final decision-making process during the evaluation of academic articles and conference proceedings. By using the different tools created, the author can have a better understanding of the requested changes made by the reviewers; the reviewers can extract actual changes made by the author; and the Editor-in-Chief can extract both the requested and the actual change information and then correlate them together in order to make the final decision.

Acknowledgement

Although this thesis is based on my individual work, there were many direct and indirect contributors that helped me to reach this achievement. I sincerely acknowledge and thank all of the following actors that contributed to this work.

First of all, I would like to thank my thesis director, Prof. Dr. Michel Hassenforder, and my thesis supervisors, Dr. Frederic Fondement, Dr. Maxime Devanne and Dr. Jonathan Weber, for their knowledge and work ethics they have transferred to me during this exciting journey. A great thank you also for their patience and all the fruitful exchanges we had.

I would also like to thank MDPI for funding my PhD thesis. A great thank you goes to Dr. Shu-Kun Lin, president and owner of MDPI, and to the entire management team for their financial and technical support.

I also want to acknowledge the work of my thesis examiners, Prof. Dr. Abderrafiaa Koukam, Prof. Dr. Ileana Ober and Dr. Cédric Dumoulin. Thank you for your valuable feedback and advice regarding the thesis writing and improvement.

A special thank you also goes to my colleague Dr. Bastien Latard, who kindly agreed to take over some of my responsibilities at MDPI that allowed me to fully concentrate on the thesis.

Another thank you goes to my colleagues at MDPI, namely Sasa Simic and Chams Azouz, who helped me with some practical implementation of different applications that we developed during the thesis; also thank you to the entire Sysadmin team, namely, Xianhua Zhou, Dr. Yannis Soupionis, Marc Bigler and others for their support. A great thank you to Sara Faes and Noel Smaragdakis for their support regarding the English editing of this thesis and other scientific papers I wrote.

Last but not least, I would like to thank my family and all my friends for their constant support and motivation. A special acknowledgment goes to my lovely wife Aleksandra and my adorable son Milan for being with me, encouraging and motivating me on a daily basis. I would not have been able to go through this journey without you two.

Contents

1 Introduction **5**

1.1 Motivation 5

1.2 Context 6

1.2.1 Thesis sponsor—MDPI 6

1.2.2 Academic publishing process 7

1.3 Publications 12

1.4 Outline 13

2 State of the Art **15**

2.1 Document Comparison 15

2.1.1 XMLDiff 22

2.1.2 DeltaXML 23

2.1.3 XyDiff 23

2.1.4 XDiff 23

2.1.5 DiffXML 24

2.1.6 XOp 24

2.1.7 FC-XmlDiff 25

2.1.8 DiffMK 25

2.1.9 JXyDiff 26

2.1.10 XCC 26

2.1.11 JNDiff 27

2.1.12 Node-delta 27

2.1.13 Existing XML diff algorithm characteristics 28

2.2 Named Entity Recognition 31
2.2.1 Evaluation criteria 40
2.2.2 Datasets 44

3 Document comparison **47**
3.1 XML Diff Comparison Study 47
3.1.1 Evaluation metrics 48
3.1.2 JATS Edit Actions 51
3.1.3 XMLDiffAnalyzer Script 53
3.1.4 Coarse-grained evaluation 54
3.1.5 Fine-Grained Analysis 57
3.2 New jats-diff algorithm 62
3.2.1 Impact of author edits on XML: a JATS example 64
3.2.2 Paragraph merge and split 66
3.2.3 Text move 67
3.2.4 Subsection upgrade/Section downgrade 67
3.2.5 Style edit 68
3.2.6 Citable object reference edit 69
3.2.7 XML diff base—JNDiff 70
3.2.8 XML tree annotation 70
3.2.9 Text similarity vs. text equality 72
3.2.10 Structural upgrade / downgrade 72
3.2.11 Structural merge/split 74
3.2.12 Inline style edit 76
3.2.13 Text move 77
3.2.14 Order of the edit detection 79
3.2.15 Citable node 80
3.3 Similarity index 81
3.3.1 Text similarity index propagation 81
3.3.2 Element lists and special objects similarity 83

3.4 Algorithm . . . 85
3.5 Performance analyses . . . 87
3.5.1 Information extraction capacity . . . 88
3.5.2 Execution performance . . . 89
3.6 Discussion . . . 91
3.7 Conclusion . . . 93

4 Named Entity Recognition **95**
4.1 Review comments . . . 96
4.2 Training, valid and testing datasets . . . 99
4.2.1 Dataset annotation . . . 99
4.2.2 Dataset conversion to CONLL format . . . 100
4.2.3 Dataset cleanup . . . 101
4.2.4 Imbalanced dataset . . . 101
4.3 Coarse-grained evaluation . . . 102
4.3.1 Models evaluation script . . . 103
4.3.2 Results . . . 103
4.4 Fine-grained evaluation . . . 110
4.5 Final long-epoch evaluation . . . 113
4.6 Manual human evaluation—*Location* class . . . 116
4.7 Discussion . . . 117
4.8 Conclusion . . . 117

5 Information matching **119**
5.1 Location . . . 121
5.1.1 Out-of-the-shelf matching possibilities . . . 122
5.1.2 Custom matching . . . 124
5.1.3 Location correlation result . . . 126
5.2 *Modal* . . . 126
5.3 *Action* . . . 127
5.4 *Trigger* . . . 130

5.5 Conclusion 131

6 Conclusion / Discussion **133**

6.1 Contributions 133

6.1.1 Document Comparison 133

6.1.2 Named Entity Recognition 134

6.1.3 Information Matching 135

6.2 Perspective 135

6.2.1 Academic 135

6.2.2 Industrial 136

7 Résumé **139**

7.1 Introduction et Motivation 139

7.2 Contributions 141

7.2.1 Comparaison de documents XML - jats-diff 141

7.2.2 Reconnaissance d'entités nommées - NER 143

7.2.3 Corrélation des changements requis et effectifs 144

7.3 Conclusion 145

7.4 Publications 146

A jats-diff delta output examples **1**

B jats-diff semantics output examples **3**

List of Figures

1-1 Article peer review and decision-making process. 8
1-2 Bird's eye view on the new straightened publication process. 10

2-1 line-based text diff (top-right) VS XML diff (bottom-right) XML compare. 17
2-2 Data-centric (top-right) vs. text-centric (bottom-right) XML compare. 19
2-3 Tai vs. Zhang and Shasha tree traversal unique node numbers assignment. 31
2-4 Pre-training NSP phase principles for BERT. 35
2-5 Fine-tuning phase principles for BERT on NER task. 36
2-6 Main difference between XLNet and BERT on MLM task. 39
2-7 Cross-entropy loss function use. 41
2-8 Precision, recall and F1 score. 42

3-1 Author modifications' impact on JATS article version. 50
3-2 Memory and Time evaluation—minimal vs. real-life author changes. . 55
3-3 Delta size—minimal vs. real-life author changes. 56
3-4 Example of bijection used while detecting a paragraph merge. 63
3-5 Two article versions side by side written in a text processor. 65
3-6 Two JATS XML versions side by side. 66
3-7 JNDiff base workflow. 71
3-8 Document A XML tree. 72
3-9 Paragraph merge. 73
3-10 Impact of structural upgrade on an XML tree. 74

3-11 Impact of structural merge on XML tree 76
3-12 Impact of inline style change on XML tree. 78
3-13 Impact of text move on XML. 79
3-14 Edit pattern detection order. 79
3-15 Impact of citable node insert on XML tree. 81
3-16 Text node modifications between two XML tree. 83
3-17 Removing an author from the authors list. 84
3-18 jats-diff algorithm workflow. 86
3-19 Execution time and memory usage. 90

4-1 NER on review comments. 98
4-2 Docanno json vs. CONLL txt formats. 100
4-3 Single location with a list of actions. 101
4-4 Coarse-grained grid-search for all classes. 106
4-5 Coarse-grained grid-search without *Content* class. 109
4-6 Fine-grained grid-search without *Content* class. 112
4-7 Confusion matrix: *Action*, *Location*, *Modal*, *Trigger* and *O* class. . . . 115

5-1 Analogies between change information. 120
5-2 Example of PDF line numbers usage 125
5-3 Analogies between NER *Location* namedentity and jats-diff. 127
5-4 Analogies between NER *Action* named entities and jats-diff. 129

7-1 Vue d'ensemble sur le processus de publication scientifique amélioré. . 140
7-2 Flux de travail jats-diff. 142
7-3 Exemple d'application NER sur un commentaire de relecteur. 143
7-4 Analogies entre la classe *Location* et la sortie jats-diff. 145

List of Tables

2.1 XML diff algorithms. 21
2.2 State-of-the-art XML diff algorithm characteristics. 28

3.1 Delta output analysis. 56
3.2 XML diff algorithms analysis recap. 61
3.3 Delta output edit action attributes. 87
3.4 Level 1/2 edit detection and similarity index calculation capacities for jats-diff, JNDiff, XYDiff and XCC. 89

4.1 Optimal hyperparameter combination per model—coarse. 105
4.2 Top coarse-grained grid-search scores for BERT-based and XLNet models. 105
4.3 Optimal hyperparameter combination per model—coarse w.o. *Content*. 108
4.4 Top coarse-grained grid-search scores without *Content* class. 110
4.5 Optimal hyperparameter combination per model—fine. 111
4.6 Top fine-grained grid-search scores without *Content* class. 113
4.7 Final model training scores without *Content* class. 114
4.8 Final evaluation results. 114

5.1 Common *Location* named entity observed within the training dataset. 122
5.2 Common *Modal* named entities observed within the training dataset. 127
5.3 Common *Action* named entities observed within the training dataset. 128
5.4 *Action* named entities correspondence table example. 128
5.5 Common *Trigger* named entities observed within the training dataset. 130

Glossary

accuracy Number of correctly predicted data points out of all the data points. 27, 37, 43, 46

deep learning Machine learning method based on artificial neural networks. 13, 33, 35, 38, 39, 40, 46, 95, 105, 111, 114, 117, 134, 136, 137

docx Microsoft Word text document file type. 7

F1 score Combines the precision and recall of a classifier into a single metric by taking their harmonic mean. 13, 41, 42, 43, 44, 46, 95, 102, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 116, 118, 134

grid-search Technique that searches exhaustively through a manually specified subset of the hyperparameter space of the targeted algorithm. 13, 95, 96, 103, 105, 106, 109, 110, 112, 113, 117, 134

HTML HyperText Markup Language - standard markup language for documents designed to be displayed in a web browser. 16, 18, 59, 91

hyperparameter Parameter whose value is used to control the learning process. 13, 95, 99, 102, 103, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 117, 118, 134

JATS Journal Article Tag Suite. v, vi, xiii, 7, 13, 17, 18, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 60, 61, 65, 67, 68, 70, 80, 82, 84, 85, 86, 87, 89, 90, 91, 92, 93, 119, 123, 124, 125, 126, 131, 133, 135, 137

MS Word Microsoft Word - a word processor developed by Microsoft. 12, 18

named entity Predefined categories used to categorise raw text and classify text spans. xiv, xv, 11, 14, 16, 27, 31, 32, 33, 41, 92, 96, 97, 98, 99, 100, 101, 107, 118, 119, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 134, 135, 137

NER Named Entity Recognition. v, vi, xiii, xiv, 10, 11, 13, 14, 31, 32, 33, 34, 36, 39, 41, 46, 92, 95, 96, 98, 99, 100, 101, 107, 113, 114, 116, 117, 119, 126, 129, 130, 134, 135, 136, 137

NLP Natural Language Processing - branch of artificial intelligence giving computers the ability to understand text and spoken words in much the same way human beings can. 33, 34, 36

node Everything in an XML document is a node. xiii, 15, 16, 18, 20, 21, 22, 23, 24, 27, 29, 30, 31, 49, 51, 52, 57, 58, 59, 60, 61, 62, 63, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 93, 94, 134

Open Office Open-source office software suite for word processing developed by Apache. 18

peer reviewer Expert who volunteers his time to help improve the manuscripts he reviews. 5, 7, 8, 9, 11, 14, 92, 96, 124, 127, 132

peer review Evaluation process of scientific / academic work by other experts working in the same field. v, xiii, 5, 6, 7, 8, 9, 12, 46, 95, 125, 137

pre-trained model Saved neural network that was previously trained on a large dataset. 34, 35, 36, 39

precision Fraction of relevant instances among the retrieved instances. 32, 41, 42, 43, 44, 46, 105, 108, 110, 111, 113, 122

recall Fraction of relevant instances that were retrieved. 32, 41, 42, 43, 44, 46, 105, 108, 110, 111, 113, 114

SQL Structured Query Language. 24

supervised learning Subcategory of machine learning and artificial intelligence defined by its use of labeled datasets to train algorithms that classify data or predict outcomes accurately. 13, 32, 44, 99, 134

tex LaTeX source document file type. 7

training Calculate the best possible weights that will allow the model to make good predictions. 13, 32, 34, 37, 38, 39, 40, 41, 42, 46, 96, 99, 100, 101, 102, 103, 104, 105, 106, 107, 109, 111, 112, 113, 114, 115, 116, 134

tree XML documents have a hierarchical structure and can conceptually be interpreted as a tree structure. xiii, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 29, 30, 31, 47, 49, 51, 52, 55, 56, 57, 58, 59, 60, 61, 62, 68, 70, 71, 72, 75, 76, 78, 79, 81, 82, 83, 85, 86, 87, 88, 91, 92, 94, 134

XML Extensible Markup Language. xiii, xv, 7, 11, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 47, 48, 51, 52, 53, 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 76, 78, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 119, 123, 124, 125, 126, 131, 133, 134, 135

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Motivation

As stated in the brief description of the thesis, the number of scientific, peer reviewed articles has increased exponentially for more than a decade. Every article has to go through a specific publication process, including peer review, author revision rounds and the final decision made by the Editor-in-Chief. Peer review is carried out by the peer reviewers reading the initial version of the article and making recommendations and improvement requests to the authors. The revision is carried out by the authors implementing the recommended review changes and complementing with a response letter describing the changes made. The final decision is made by the Editors-in-Chief, analysing and comparing review comments to the revisions made by the authors. All those are manual processes that require a lot of time, attention and rigour. It is thus of interest to facilitate the tasks of all involved parties (authors, peer reviewers and final decision-makers), and new tools to help to strengthen this process are needed.

1.2 Context

1.2.1 Thesis sponsor—MDPI

MDPI (www.mdpi.com), established in 1996 in Basel, Switzerland, is a pioneer in scholarly open access publishing. With over 400 diverse, peer reviewed open access journals across all disciplines, MDPI is supported by more than 130,000 academic editors and has published over 1,000,000 articles, with 240,000 publications in 2021 alone. With additional offices in Beijing, Wuhan, Tianjian, Nanjing and Dalian (China), Barcelona (Spain), Belgrade and Novi Sad (Serbia), Manchester (UK), Tokyo (Japan), Cluj and Bucharest (Romania), Toronto (Canada), Kraków (Poland), Singapore (Singapore) and Bangkok (Thailand), MDPI has over 5500 employees worldwide. Publishing the research of more than 400,000 individual authors, MDPI journals are attracting over 25 million monthly web-page views. Most of the published content is covered by the main academic indexing databases such as Science Citation Index Expanded (SCIE[1]), maintained by Web of Science, and Scopus[2], maintained by Elsevier. The biomed-related journals are full-text archived in PubMed Central (PMC[3]) and with article abstracts to be found in PubMed/MEDLINE[4]. All MDPI journals are archived long term with the Swiss National Library[5] and CLOCKSS[6].

For over a decade, MDPI has been developing a large variety of platforms and initiatives around open science. The following can be distinguished:

- sciforum.net—a scientific event planning and management platform;
- scilit.net—a database of scholarly works indexing over 145 million articles;
- preprints.org—a platform for early versions of research outputs;
- sciprofiles.com—a social network for researchers and scholars.

[1]https://clarivate.com/webofsciencegroup/solutions/webofscience-scie/
[2]https://www.scopus.com/home.uri
[3]https://www.ncbi.nlm.nih.gov/pmc/
[4]https://pubmed.ncbi.nlm.nih.gov/
[5]https://www.nb.admin.ch/snl/en/home.html
[6]https://clockss.org/

MDPI has also developed its own in-house submission system, used by authors, internal assistant editors, peer reviewers and external academic editors to process papers from submission to publication. The docx and tex conversion to XML is also made in-house by a team of production editors, using internal conversion tools. Among different tools and data available at MDPI, we are mostly interested in the following as those can be used and integrated with our research:

- An XML conversion tool that allows the docx / tex conversion to JATS XML;
- The different versions of an academic article during the peer review process;
- The review comments with requested changes made during the peer review;
- The database of editors, authors and peer reviewers that could test our tools.

1.2.2 Academic publishing process

Following recent growth in higher education and the rapid development of science and technology, the number of scientific, peer reviewed articles has been growing exponentially for over a decade [89]. The main reason for this is the growth of science and technology on one side, and higher education on the other side. In addition to the pressure to publish scientific papers, senior scientists are also asked to conduct peer review and be members of journal editorial boards and part of conference committees where they are in charge of, in addition to many other activities, making final decisions regarding new articles' acceptance. Some research groups started exploring the assisted peer review possibilities [103, 96] in order to help senior scientists with their daily tasks. The final decision making and the overall peer review process is, however, not receiving enough research attention. With the Editor-in-Chief or Conference Chair together with the author and peer reviewer playing the key roles in academic publishing, it is of interest to facilitate their tasks as much as possible. Currently, this process is very manual and requires a lot of time, attention and rigour. With all the previously mentioned duties, and with senior scientists lacking the time needed, the different tasks in the publishing process are prone to human errors, and new tools

to help to strengthen the entire process are needed.

After the article revision round—see Figure 1-1, in order to make the final decision the senior scientist must assess whether the author made the requested changes. This is achieved by reading the review comments, comparing different versions of the article and reading the author response letter. The article comparison together with the review comments and the author response letter reading tasks are time-consuming. Moreover, change description within the author response letter is written by the author and may not always reflect all, nor real changes made during the revision. It is then of great importance to automate this task as much as possible in order to obtain a higher-quality and faster final decision making.

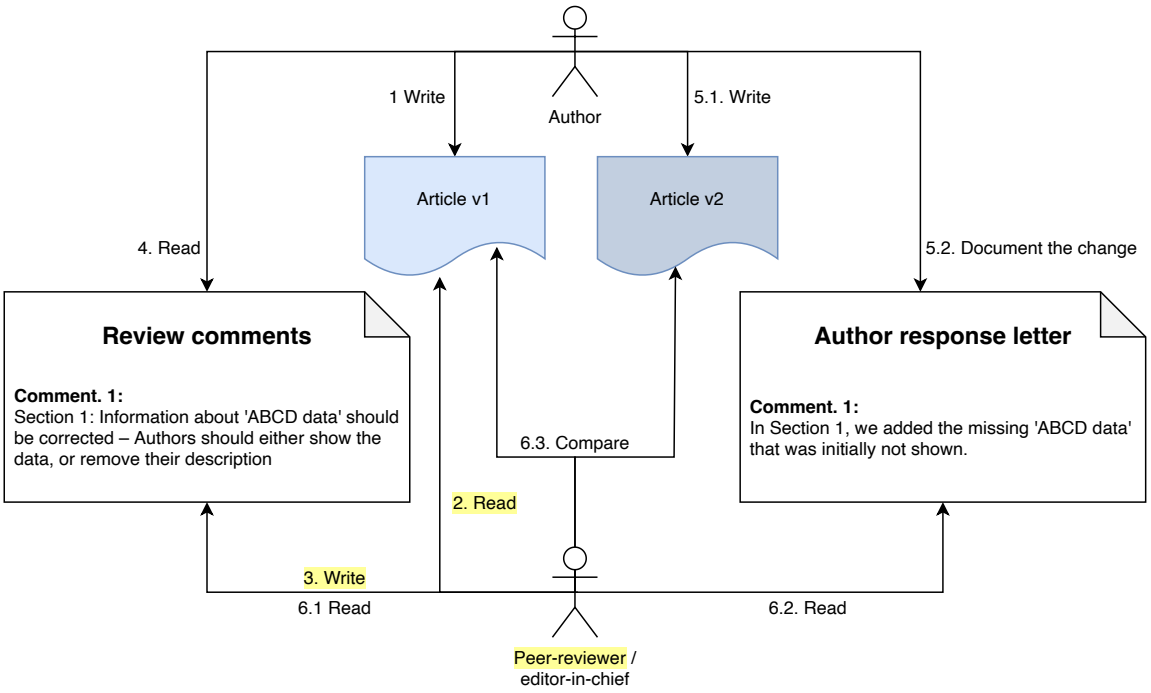


Figure 1-1: Article peer review and decision-making process.

Simplified version of the academic publishing process, limited to one revision round. The author writes (1) the first version of the article. It then goes to the peer reviewer that reads (2) the article and makes comments (3). The comments are then read (4) by the author that writes (5.1) the second version of the article by applying corrections requested by the peer reviewer, and also the author response letter (5.2). The decision-maker has to read the review comments (6.1) and author response letter (6.2) and compare different versions of the article (6.3) in order to evaluate whether the author made all the changes requested by the peer reviewer

On another note, the peer review process is an indirect exchange between the

author, the peer reviewer/decision-maker. The first version of the article goes into the peer review where it gets evaluated and commented on by the peer reviewer. The author then has to read the review comments and apply the requested changes in order to write the second (improved) version of the article together with the response letter containing the information on what and how was the article changed. The key elements here in the process, enabling the communication between the peer reviewer and the author, are the review comments and the author response letter documents. The peer reviewer explains which part of the article has to be revised and in which way. In the example of a review comment where the peer reviewer asks the author for a specific change—see Figure 1-1: "Section 1: Information about 'ABCD data' should be corrected – Authors should either show the data, or remove their description"—we can observe a specific location within the article where the change should take part ("Section 1") and the proposed corrections on how to improve the article—"either show the data or remove their description". The author will read that comment, identify the location where the change should be made and apply the proposed correction. Once the change has been made, the author has to document the change in the response letter so that the peer reviewer (or the Editor-in-Chief) can match that specific change description with the associated review comment. We can observe here how manual the entire process is for all involved parties, including the author, the peer reviewer and the Editor-in-Chief.

Figure 1-2 shows the bird's-eye view of how different key-players can be assisted in order to strengthen the entire publication process. For this, instead of having to do the manual work, i.e., read the author response letter and the review comments and compare different versions of the article, the new tools will automate that process. The author will be able to directly access the requested changes asked by peer reviewer; the peer reviewer will be able to directly access the actual changes made by the author; and the Editor-in-Chief will be able to access the correlated information between the requested and the actual changes. This way, all needed information is interconnected and can be found in one place.

In order to provide such tools, our work was separated into three different parts:

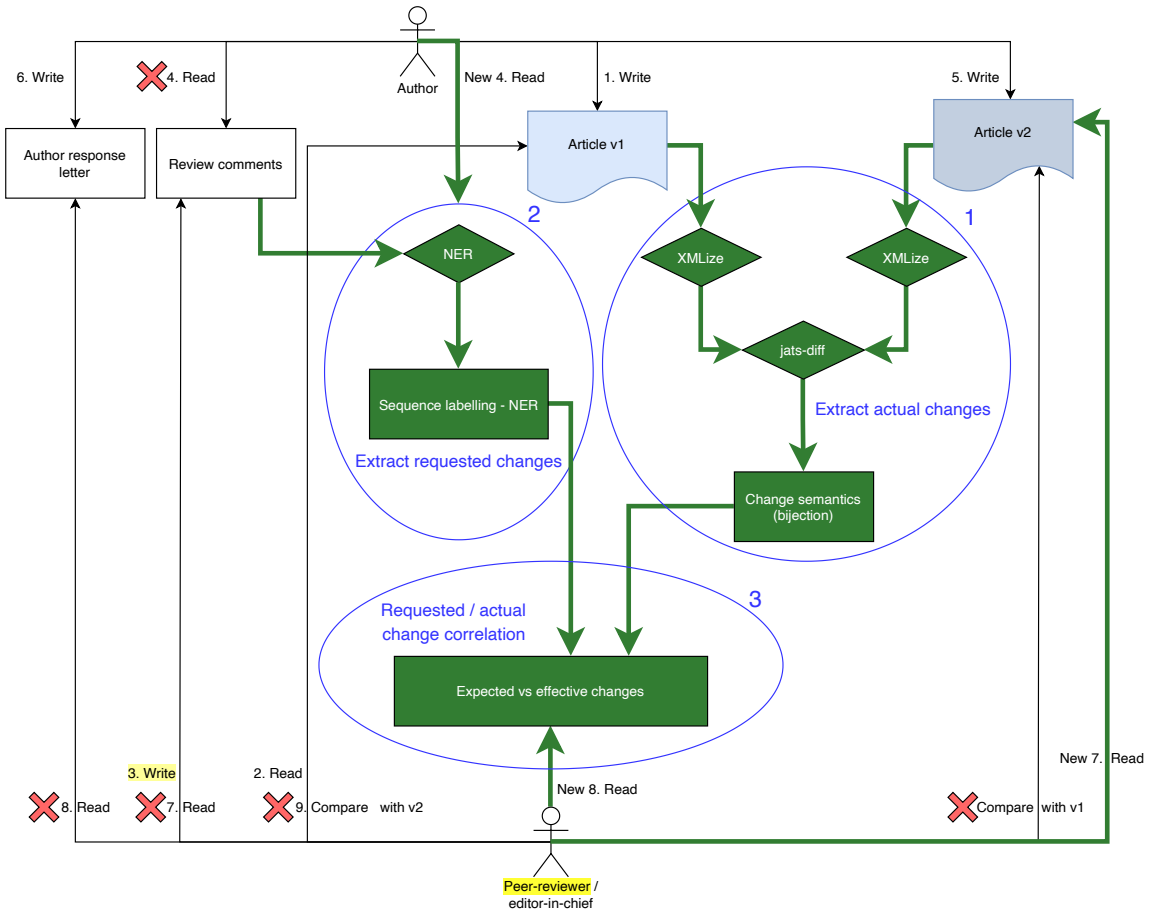


Figure 1-2: Bird's eye view on the new straightened publication process.

Instead of reading the author response letter and the review comments and comparing different versions of the article, the Editor-in-Chief will only have to obtain the correlated information between the requested and actual changes.

one regarding the extraction of actual changes by comparing different versions of the article (1); the second regarding the extraction of requested changes by analysing review comments (2); and the third regarding the correlation between the requested and actual changes (3). For this to happen, our research is based on the following topics: document comparison for extracting actual changes and Named Entity Recognition (NER) for identifying requested changes. The document comparison part is used to not only extract lexical changes made by the authors, such as text inserts, deletes and updates, but also syntactic changes such as paragraph split and merge, section upgrade and downgrade, style edits, citable object edits, text moves, etc. The main goal of the document comparison is to have a bijection between the modifications

made by the author and the changes detected between the two XML versions of the article. The NER part is used to label review comments with specific named entities in order to match the requested change type and location within the article. Finally, the two pieces of change information are correlated together using the modification location from the document comparison and the NER sides. This will result in five different output categories:

- **Requested but not detected changes**: we were able to match a change request written by the peer reviewer for a given location within the article, but there is no modification made by the author on that location;
- **Detected but not requested changes**: we observed a modification made by the author in a given location within the article, but there is no associated change request made by peer reviewer;
- **Requested and detected changes**: we were able to match a change request written by the peer reviewer for a given location within the article and specific modifications made by the author on that location;
- **Major requested change but minor change detected**: we were able to match a major/substantial change request written by the peer reviewer for a given location within the article, but only a minor modification made by the author was detected;
- **Minor requested change but major change detected**: we were able to match a minor change request written by the peer reviewer for a given location within the article, but a major/substantial modification made by the author was detected.

As seen in Figures 1-2, the current decision-making process within academic publishing is rather manual. The authors directly read review comments written in free-form-text where peer reviewers can use their own wording to describe a specific change request. Once article correction has taken place, the author writes the response letter, again in free-form-text, where each requested change is commented on regarding if and how the request was fulfilled. Finally, the Editor-in-Chief or the peer

reviewer has to assess if the requested changes were fulfilled by the author. For this, the two article versions are compared, usually using the document compare function in MS Word, and the actual changes are assessed by correlating them with the review comments. Within the current state of the art, there are no specific tools developed for the academic community that could be directly used during the publication process. On one side, there are document comparison tools available, but those are usually proprietary and depend on the typesetter tools. On the other side, we are not aware of any tool that can further process or directly annotate review comments in order to extract meaningful information. Finally, there is no research carried out on correlating requested and actual changes within the peer review process nor in any other similar domain.

1.3 Publications

The work that has been carried out during the thesis produced one international journal and three international conference articles:

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "Change Detection on JATS Academic Articles: An XML Diff Comparison Study." In Proceedings of the ACM Symposium on Document Engineering 2020, pp. 1-10. (2020)

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "Semantics to the rescue of document-based XML diff: A JATS case study." Software: Practice and Experience (2022)

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "A JATS XML Comparison Algorithm for Scientific Literature." In: Journal Article Tag Suite Conference (JATS-Con) Proceedings (2022)

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "Named Entity Recognition for peer review disambiguation in academic publishing." Accepted in: ICICT 2023 Conference - IEEE

1.4 Outline

Chapter 2—State of the art

After Chapter 1, Introduction, Chapter 2 is divided into the state of the art on XML document comparison and the state of the art on NER. Within the document comparison part (Section 2.1), we describe the current XML diff algorithms, their comparison capacities and their shortcomings in comparing JATS XML versions of academic articles. Following is the state of the art on NER (Section 2.2) and the usage of deep learning models in NER tasks.

Chapter 3—Document Comparison

The third chapter is about our contribution to the XML document comparison research topic. In Section 3.1, we present an evaluation study of the existing state-of-the-art XML diff algorithms and their capacities in comparing JATS XML documents. After finding that none of the existing XML diff algorithms are suitable for JATS document comparison, we describe a novel XML diff algorithm we have developed, called jats-diff, presented in Section 3.2. Jats-diff is able to make a bijection between the actual modifications made by the author on typesetter tools on one side, and differences observed between two JATS XML documents on the other.

Chapter 4—Named Entity Recognition

The fourth chapter is about our work on training deep learning models on the review comments annotation task, using the supervised learning approach. We start with a coarse-grained evaluation of different models such as BERT, SciBERT, DistilBERT, RoBERTa and XLNet using the grid-search technique. In order to select the best-scoring model on our NER task, we go further through the hyperparameters' fine-tuning process by finally achieving an weighted average F1 score of 0.87.

Chapter 5—Information matching

Finally, we present a correlation method between the jats-diff output, representing the actual changes, and the NER output, representing the requested changes. The goal here is to give to the Editor-in-Chief the possibility to evaluate if the requested changes written by the peer reviewer are made in a satisfactory manner by the author. For this, we start with the correlation between the review comments *Location* and the location in the article where actual changes are detected with jats-diff. The *Location* named entity can be precise, for example, a given line or paragraph; semi-precise, for example, a given section or multiple paragraphs; or fuzzy, for example, the entire manuscript, or several sections. Next, we continue with the *Action* named entity in order to link it with an actual change detected by jats-diff. We then continue with the modality of the requested change by using the *Modal* named entity. The modality will give us the information about the obligatory and optional change. If a change is requested using the "must" or "should" modal, the change is obligatory. By contrast, if requested using the "might" modal, we can estimate the change as being optional. Finally, we describe the *Trigger* named entity usage in order to further refine the correlation of the *Action* named entity identifying questions or multiple choice change types.

Chapter 2

State of the Art

2.1 Document Comparison

Since the beginning of the digital age in the 1970s, researchers have expressed their interest in comparing textual documents with the purpose of extracting, analysing and understanding differences. Text diff algorithms [88, 58] have been studied and described. Some of them are still in use, such as Hunt–McIlroy's [32] algorithm (currently used in the GNU Diff utility) and Myers' [60] algorithm. Most of these algorithms are line-based and rely on two edit operations: Insert and Delete. The difference is calculated by comparing each line of the original with the corresponding line of the modified text file. The range of applications for text diff algorithms is wide; they are present in version control systems (Git, Apache Subversion) and many other tools such as IDEs (Eclipse Compare) or text editors (Notepad++ Compare), where tracking textual differences is important. Starting from the early 2000s, with the explosion of the World Wide Web and semi-structured text documents, several research groups had their focus on a specific type of text document—XML [91]. The main reason for this was the promising future of XML, adopted widely in a short period by many key players in the domain of high technology. Compared with plain text documents, the particularity of XML resides in its hierarchical structure, also called tree structure, and nodes can also have attributes to carry specific information. Moreover, for text-centric XML document types (see the paragraph describing the

differences between text-centric and data-centric XML documents), the order of the child nodes within a given parent provides an important information on how the text is structured within the document.

Regarding the comparison of XML documents, the tree-to-tree editing problem defined by Selkow [80] makes existing text diff algorithms unsuitable for comparing XML documents. This was further demonstrated by several research groups [8, 7, 15]. Actually, XML diff algorithms compare attribute–value pairs of objects, each of those objects corresponding to XML nodes. The comparison is carried out on each node according to their position in the XML tree. Figure 2-1 shows the main differences between line-based text diff algorithms (top-right) and proper XML diff algorithms (bottom-right). We observe that while comparing XML documents, there is a need for semantics and a higher level of abstraction in order to take into consideration the XML tree structure properties. We no longer only compare text elements between different named entities of documents A and B, but we also compare node objects that are organised within a defined tree structure. In addition to the basic text insert, delete and update edit actions, XML diff algorithms use those edit actions on the XML tree structure and also additional edit actions such as move, attribute edit, etc. As an extreme example, there are one-line valid XML documents where the line breaks between different nodes are removed. Comparing such a document with a line-based text diff tool does not make any sense as any change would be shown as one delete–insert sequence.

Several projects [92] also developed HTML-based diff algorithm implementations. Both XML and HTML are markup languages; however, the difference between them is that XML documents are meant to be used for data storage and transportation, while HTML files are meant to be used for displaying the data. Thus, HTML diff compared to XML diff algorithms are more complex as they have to detect and represent content, structure and styling/layout differences between two HTML files. Within our environment, we are only interested in content changes and not in layout changes made by authors, so this approach of using HTML diff algorithms is not considered.

Figure 2-1: line-based text diff (top-right) VS XML diff (bottom-right) XML compare. The line-based diff tools use three edit actions: insert (>), delete (<) and update (a combination of <, — and >). They compare the two documents line by line and show the differences. The XML diff approach compares XML tree objects (nodes) and uses additional edit actions such as move, attribute-update, etc. We can observe here that moving the <keywords> sub-tree is shown as a sub-tree move using the XML diff tool and as a full insert-delete combination using a line-based text-diff tool.

In the scientific literature, there are several main document types largely used by authors while writing their articles. Among them are well-known and established formats such as Tex[1], docx[2] and odt[3]. In order to facilitate archiving and interchange, academic publishers convert articles from their initial document types to JATS[4] (Journal Article Tag Suite) XML developed by NISO (National Information Standards Organization). JATS is the de facto standard for the XML representation

[1]https://foldoc.org/TeX
[2]https://loc.gov/preservation/digital/formats/fdd/fdd000397.shtml
[3]https://loc.gov/preservation/digital/formats/fdd/fdd000428.shtml
[4]https://jats.nlm.nih.gov

of academic articles and is used by major indexing companies, including PubMed Central[5] and SciELO[6]. It has the advantage of being machine-readable and independent of text processors, has no layout information and carries only the article data and structure. The main text is contained in paragraphs (<p>), which is similar to what is done within HTML documents. In addition to text, paragraphs are composed of <xref> elements used for citing objects contained in the back part of JATS (references, figures, tables) and styling elements such as <b>, <i>, <sub>, <sup>, etc.

XML documents, being simple and general in nature, are suitable for both text and data, and so there are two main categories of XML documents: text-centric (or document-centric) and data-centric, as described in [13] and [54] (see chapter *Text-centric vs. data-centric XML retrieval*). The size of individual text nodes is usually larger in text-centric XML documents, while data-centric nodes are smaller in size but higher in number. Most of the early XML diff algorithms were developed for data-centric XML documents with the main focus on execution time, memory usage and delta size efficiency, and with less attention on the way changes are modelled and displayed. In order to maintain that high computational performance, most of them reduced their computational complexity to $O(n * \log(n))$, with n being the number of nodes. The authors of those algorithms were working with a large number of XML files and had to discard operations of higher complexity in order to improve execution performance. Moreover, higher complexity operations have lower potential gain when applied on data-centric XML documents [47]. Several research groups [74, 70, 13] have demonstrated that XML diff algorithms for data-centric documents are not suitable for text-centric documents, and there is a need for specific algorithms adapted to their needs. Those algorithms strive to achieve similar results to those of MS Word and Open Office track change tools, where differences between the two documents are represented as close as possible to the changes originally produced by their authors. Figure 2-2 shows a comparison of two XML documents carried out by

[5]https://www.ncbi.nlm.nih.gov/pmc/
[6]https://scielo.org/en/

a data-centric XML diff algorithm on the top-right (XyDiff), and a text-centric XML diff algorithm on the bottom-right (jats-diff). We can observe that data-centric XML diff algorithms are able to only detect basic edit operations compared to text-centric XML diff algorithms that are able to detect a greater number of higher-level edit operations. A paragraph split and a text-style edit detected by the text-centric XML diff algorithm is shown as a basic delete–insert sequence when using data-centric XML diff algorithms.

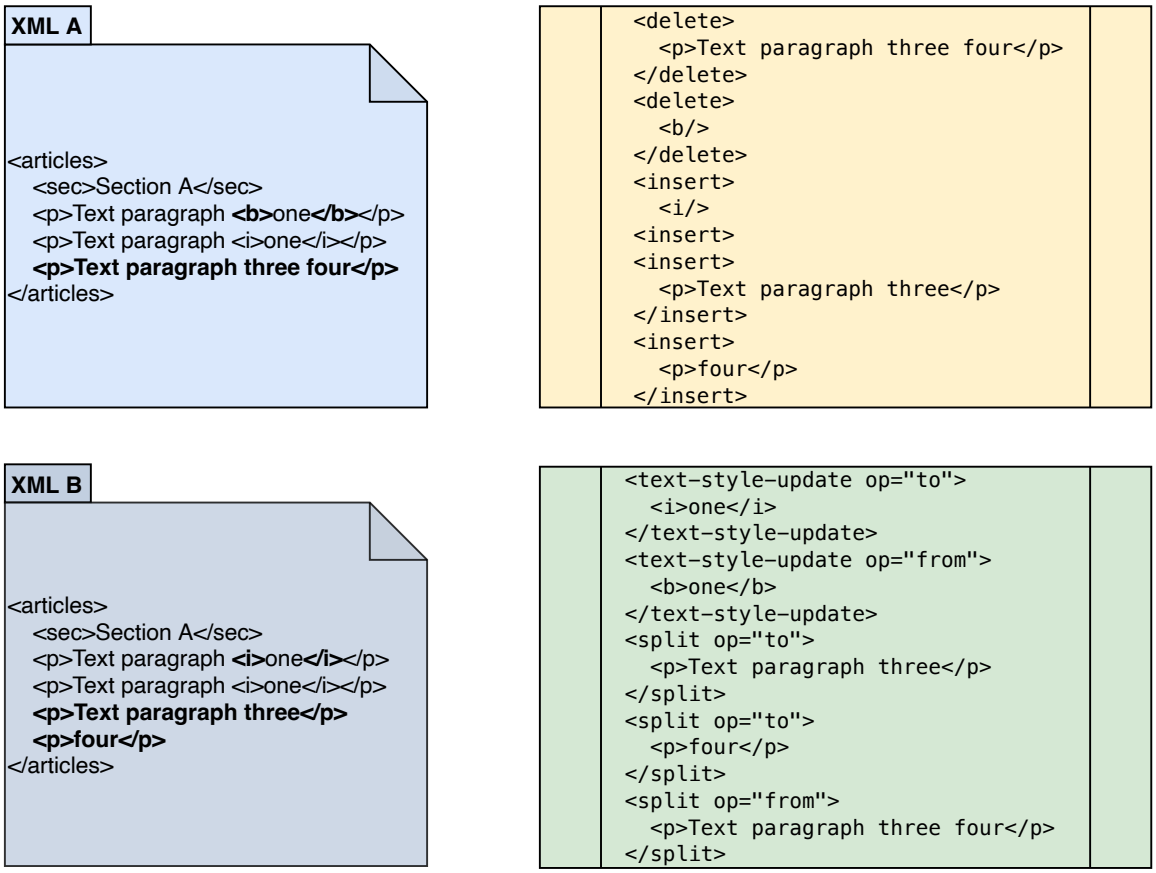


Figure 2-2: Data-centric (top-right) vs. text-centric (bottom-right) XML compare.

The data-centric XML diff tools (XyDiff in our example) have a limited set of edit pattern detection capacities. By contrast, text-centric XML diff tools (jats-diff in our example) can detect additional edit patterns. Within this example, a paragraph split and a style edit is represented as delete–insert sequences.

Historically, data-centric XML diff algorithms inherited the existing text comparison capacities and additionally added so-called "Level 1" or "basic" edit operations [36] on tree elements: insert, delete and attribute change. Those basic edit oper-

ations are very important for the XML tree structure, as without attribute change detection, modifying the highest parent attribute would result in deleting the entire XML tree and inserting the same tree with its attribute change. Data-centric XML diff output is rather large using only those basic edit actions, which is not an issue as the change information consumers were usually machinamed entities looking for precise data modifications. The number of individual text nodes being high but small in size, modifying a text node was naturally represented as a simple delete and insert combination. Another reason for the restrictive list of changes is their detection performance. Higher-level changes require more computation power and time to execute; thus, keeping the list of changes simple would allow better performance.

Recent XML diff algorithms are mostly text-centric and propose a specific, higher level or "Level 2" changes that are not found in data-centric XML diff algorithms. "Level 2" changes are composed of a combination of "Level 1" basic edit operations. Each time a specific insert–delete sequence is observed, it then gets converted to a unique "Level 2" change. The individual text nodes are usually larger in size but smaller in number; thus, it makes sense to think about higher-level changes that replace a combination of basic insert and delete operations. Moreover, as humans are usually the main producers of those text-centric XML documents, there is a need to use a strong relationship between the modifications made and the changes detected. Among the "Level 2" tree edits, existing algorithms are able to detect and represent tree move and wrap/unwrap (observed in [13]). Wrap is used to detect edit patterns where a specific portion of text within a given node was wrapped by another node, unwrap being the opposite of wrap. Adding/deleting styling nodes around a text portion is represented as wrap/unwrap. In the literature, there is a discussion about an additional number of "Level 2" changes. Some research groups propose element merge and split [36] and also text move [13]. Unfortunately, none of the existing algorithms propose a solution regarding how to detect and represent such changes.

In order to cope with performance issues while comparing large textual nodes, some XML diff algorithms [15, 71, 13] use the two-pass logic where the first pass is used to assign fingerprint values to the XML tree structure on both versions of the

document. In the second pass, differences are detected by comparing node fingerprints, instead of comparing their content.

Among the existing XML diff algorithms, we were able to find documentations (or research literature) and execute 12 of them that are shown on Table 2.1: 11 are state-of-the-art algorithms from the scientific literature, and 1 is a commercial implementation. In these algorithms, edit operations specific to the tree structure were introduced: Insert, Delete, Update and Move.

Algorithm	Language	Link	XML type	Published date
XMLDiff [8]	Python	Pypi.org	data	1996
DeltaXML	Java	Commercial	data	2001
XyDiff [15]	C++	Github	data	2002
Xdiff [97]	C++	Github	data	2003
DiffXML [10]	Java	Github	data	2004
XOp [42]	Java	Living-pages.de	data	2004
FC-XmlDiff [47]	Java	Github	data	2006
DiffMK [63]	Java	Sourceforge	data	2007
JXyDiff [73]	Java	Github	data	2009
XCC [71]	Java	Launchpad	text	2009
JNDiff [13]	Java	Sourceforge	text	2009
Node-delta [50]	JavaScript	Github	data	2012

Table 2.1: XML diff algorithms.

Deleting a node implies the same action on all of its child nodes, which means it is no longer a line-based approach but becomes a tree-structure-based approach. While moving a tree has no impact on the text content, it does move the content by changing its position within the tree. In the real-world scenario of an academic article, an operation as simple as inserting an additional author composed of their name, email and affiliation number would result in a node Insert action where the entire author tree substructure is impacted. The Update operation is also important in order to minimise the overuse of Insert/Delete operations. Ronnau et al. [72] explain that in the case of an attribute change on the root of the document tree, without Update, it is necessary to represent the change with a full Delete operation followed by a full Insert operation. The Move operation consists of changing the

position of a given child node among the other child nodes of a specific parent and can be used for author reordering. Without the Move action, changing the node order would result in removing and re-inserting them in the correct order. Attribute editing is another specificity of XML documents which the text diff algorithms are not able to deal with. By using all previously mentioned edit operations, XML diff algorithms should be able to detect both text and tree changes.

In the following, we will describe each of these algorithms and the research work carried out around them in chronological order.

2.1.1 XMLDiff

XMLDiff is the oldest among the algorithms we analysed. It was published in 1996 [8]. However, the XMLDiff implementation we tested dated from 2004. The paper was published two years before the first XML 1.0 Specification and did not directly mention the XML format. It was related to hierarchically structured data, which are closely related to data-centric XML documents. The algorithm was justified by explaining the importance of detecting and representing changes in databases and version and configuration management. Unlike most of the previous work carried out in the field of change management, where the differences were computed from flat-files and relational data, they focused on hierarchically structured data which are very close to XML documents. By converting the data into hierarchically structured documents, evaluating the changes between the old and new versions of the data will consist of computing hierarchical change detection and finding a “minimum-cost edit script” that transforms one data tree to another. XMLDiff uses so-called "key domain" characteristics in order to improve its performance compared to previous generic comparison algorithms. Four edit actions, all related to tree structure changes, were defined: Insert, Delete, Update and Move. The algorithm is still maintained and is at version 2.4.

2.1.2 DeltaXML

DeltaXML is a commercial suite of products that started in 2001. One of their solutions is XML Compare, which can be used as a command line or GUI to compare XML files. The delta results are passed through a pipeline so the output can be adapted to different needs. The default delta output uses their own namespace and is a summary of the original XML document enriched with embedded annotation attributes describing the change in each node in the XML tree. Three edit actions, all related to tree structure changes, were defined: Delete, Insert and Update. The move operation is not supported. For unchanged nodes, those are annotated with deltaxml:unchanged attribute. DeltaXML is still maintained, and their R&D teams have published several white papers [93] regarding new approaches in XML diffing.

2.1.3 XyDiff

XyDiff was developed by Gregory Cobena within a PhD project called Xyleme. The publication describing the algorithm was published in 2002 [15]. XyDiff was developed in C++ with the purpose of managing data changes on the Web by indexing and analysing parts of French websites stored as XML documents. In the first part of their work, changes at the "microscopic" scale were considered which resulted in the creation of the XyDiff algorithm. The algorithm parses each XML document twice: the first time for assigning so-called "XID persistent identifiers", acting as fingerprint values, to each node and then to compute the difference. Differences, called XyDelta, are all related to the XIDs. An additional .xidmap file is created grouping XIDs depending on the node differences between the two documents. XyDiff supports four edit actions, all related to tree structure changes: Insert, Delete, Update and Move. Its implementation has not been maintained since 2015.

2.1.4 XDiff

XDiff was published in 2003 [97]. The paper describes a new XML diff algorithm used to detect changes in internet query systems, search engines and continuous

query systems. Compared to previous XML diff algorithms that used an ordered tree model where the left-to-right order among siblings is important, the XDiff authors explain that for data-centric XML documents, the order of the elements is not important. They defended the theory that an unordered model, where only ancestor relationships are important, is more adapted for date-centric XML documents. This approach involved the vanishing of the Move edit action, and XDiff supports three edit actions, all related to tree structure changes: Delete, Insert and Update. On the other side, the complexity of comparing XML documents within the unordered tree model was described as higher than the ordered model comparison. The algorithm implementation was written in C++ and was maintained until 2015.

2.1.5 DiffXML

DiffXML was published in 2004 [10]. The algorithm introduces a new approach that consists of mapping the XML DOM tree structure to a relational database. Each node has two key pieces of information: its value and a unique path. Those pieces of information are stored in relational tables, and SQL operations are then used to detect changes between two XML documents stored in the database. In order to detect changes, for example, finding a node Move, SQL queries are executed to detect nodes whose path changed and value stayed the same. Each SQL query is adapted to one of the four specific edit actions supported by DiffXML: Insert, Delete, Update and Move. Its implementation was written in JAVA and and was maintained until 2018.

2.1.6 XOp

XOp stands for XOperator and was developed in 2004 by Living Pages Research GmbH as part of the Ercato project. The project was based on "thing-oriented programming" with the so-called ercatons representing "things" (i.e., XML documents). There is no research paper directly describing XOp; however, the Ercato project concept [42] was published in 2004 as a press article and one year later analysed in a

XMLDiff comparison study [30] that evaluated, among others, the XOp algorithm. In order to represent object-oriented inheritance, the XOp was developed within the project to compute the difference between two XML documents by using algebraic operations on XML trees. The default operation used while comparing two XML files is subtraction (-) where the initial document (a.xml) is compared to the modified document (b.xml) by subtraction. There is also a possibility to use different operations as addition (+), equality (==) and inequality (!=) where both return a Boolean, etc. Due to the algebraic comparison approach, XOp does not directly support any of the XML edit actions, and those have to be detected with a post treatment of the XOp delta. The algorithm implementation was written in JAVA and was maintained until 2009.

2.1.7 FC-XmlDiff

FC-XmlDiff, also called faxma, was published in 2006 [47]. The first implementation of the algorithm is from 2008. The algorithm uses a greedy heuristic approach by transforming the XML ordered trees to the domain of sequence alignment, computing the difference and transforming it back to the ordered trees domain. This approach reduces the complexity of the XML tree comparison and makes it simpler compared to the direct comparison of XML trees. FC-XmlDiff was compared to other existing algorithms and the results showed that, although simple and with lower complexity, their approach had similar output size and execution time performance compared to other XML diff algorithms with a more complex design. FC-XmlDiff supports four edit actions: Insert, Delete, Update and Move. The algorithm implementation has not been maintained since 2009.

2.1.8 DiffMK

DiffMK was initially developed by Sun Microsystems in 2001 as a generic file comparison tool and later adapted by Norman Walsh as an XML diff tool. There was no research literature describing DiffMK, but there was one paper [47] evaluating Nor-

man Walsh's, and one PhD thesis [14] evaluating the Sun Microsystems algorithm. The DiffMK algorithm we evaluate here is the one adapted by Norman Walsh and we use its version 3.0 from 2007. It annotates the changes on the original XML file, similar to DeltaXML. DiffMK uses the Unix diff algorithm and works in the sequence domain, which makes the tree move detection impossible. Thus, there are only three possible edit actions: Insert, Delete and Update. The algorithm implementation has not been maintained since 2015.

2.1.9 JXyDiff

JXyDiff is a Java implementation of the XyDiff algorithm developed by Adriano Bonat in 2009. The author claimed this implementation had some bug fixes compared to the original C++ implementation developed by Gregory Cobena in 2002. JXyDiff has not been maintained since its publication in 2009.

2.1.10 XCC

XCC was published in 2012 [71]; however, the implementation we evaluate here dates from 2009. XCC is the first algorithm fully dedicated to text-centric XML document comparison and has the purpose of comparing office documents where the content is saved in XML format—more precisely, the OpenDocument format. The algorithm has two main goals, diff and patch. Diff is used to identify and represent differences between two versions of a document, and patch is used for merging purposes of document versions resulting from different editing processes. Those two goals are already achieved in different typesetter tools by using the change tracking function during the editing phase. Those are, however, dependent on those typesetter tools and not applicable outside of the corresponding applications. In order to be able to achieve those goals outside of the typesetter tools, a different approach that is state-based is required. XCC also introduces the context fingerprints in order to identify the edit operation in a highly reliable and faster way. It supports four edit actions: Insert, Delete, Update and Move. The algorithm implementation has not

been maintained since 2009.

2.1.11 JNDiff

JNDiff was published in 2016 [13]; however, its implementation was carried out earlier, in 2009. Similar to the XCC algorithm approach and opposite to most existing XML diff algorithms, the authors of JNDiff made a clear distinction between text- and data-centric XML documents. They explained clearly that data-centric XML diff algorithms process both text-centric and data-centric XML documents the same way, which is not optimal. JNDiff has the focus on delta output quality (human readability, accuracy and clear named entities) rather than high execution performance. The goal of the algorithm was to compare textual documents in XML format and represent the differences in a similar way to the existing change-tracking functions that we can find in different typesetter tools. Compared to those functions which do record edit actions while they are performed, reconstructing the edit sequence by comparing two versions of an XML document is way more complex when using diff algorithms. JNDiff supports five edit actions: Insert, Delete, Update, Move and Wrap/Unwrap. This new Wrap/Unwrap edit action is specific to text-centric XML documents and represents a specific text wrapped inside a new node. An example would be the styling of some words using <bold>, <italic> etc. The research paper also mentioned the need for additional so-called "Level 2" edit operations to detect paragraph merge and split. The algorithm implementation has not been maintained since 2014.

2.1.12 Node-delta

Node-delta was part of the Delta.js JavaScript project developed by Lorenz Schori in 2012. There was no research literature describing the algorithm. However, its development was part of a BSc thesis [50]. The main purpose of the Delta.js project was the implementation of a version control system for structured documents. The author claimed that the current version control systems are well adapted for plain text files but do not work properly with structured documents. This is one of the

main reasons those systems are not used by people who mainly work with content. Node-delta was inspired by the XCC algorithm we mentioned earlier and supports four edit actions: Insert, Delete, Update and Move. In order to improve its execution performance, the algorithm also uses fingerprints. Its implementation was maintained until 2020.

2.1.13 Existing XML diff algorithm characteristics

In the above, we described 12 existing XML diff algorithms. There are probably many more that are less well known, proprietary or with no scientific literature. Each of these algorithms has its own way of describing differences with no universal delta model. This makes it challenging to compare them and measure the quality of their delta outputs, as described in [3]. Table 2.2 shows the main characteristics for every of the 12 XML diff algorithms we will further evaluate in Section 3.1. We can find there their supported edit actions and the fingerprint use that we represent with a binary success operator (✓for success and X for fail), together with the type of XML documents they were designed for, the published and the last updated date.

Algorithm	Supported edit actions					Fingerprint	XML type	Language	Published date	Last update
	Del	Ins	Mv	Upd	Other					
XMLDiff	✓	✓	✓	✓	X	X	data	Python	1996	present
DeltaXML	✓	✓	X	✓	X	?	data	Java	2001	present
XyDiff	✓	✓	✓	✓	X	✓	data	C++	2002	2015
Xdiff	✓	✓	X	✓	X	X	data	C++	2003	2015
DiffXML	✓	✓	✓	✓	X	X	data	Java	2004	2018
XOP	X	X	X	X	X	X	data	Java	2004	2009
FC-XmlDiff	✓	✓	✓	✓	X	X	data	Java	2006	2009
DiffMK	✓	✓	X	✓	X	X	data	Java	2007	2015
jXyDiff	✓	✓	✓	✓	X	✓	data	Java	2009	2009
XCC	✓	✓	✓	✓	X	✓	text	Java	2009	2009
JNDiff	✓	✓	✓	✓	✓	✓	text	Java	2009	2014
Node-delta	✓	✓	✓	✓	X	✓	data	JavaScript	2012	2020

Table 2.2: State-of-the-art XML diff algorithm characteristics.

Regarding the basic edit operations (insert, delete and update) the existing XML diff algorithms are able to detect, those were first discussed in the literature by Selkow [80] describing the problem of identifying the differences between two ordered labelled

trees, known as tree-to-tree correction problem. Selkow extended Sankoff's algorithm [79] used to solve the string-to-string editing problem and applied it to find the tree-to-tree edit distance between ordered labelled trees. Three basic edit operations were defined: relabel (change), delete and insert.

The Selkow algorithm being recursive, given two trees A and B and with their respective nodes A_i and B_j, it calculates the edit distance between different node pairs starting from the root nodes and propagating towards the child nodes. At each node pair, the algorithm sets a cost matrix that is computed by taking the minimum cost from the three previously mentioned edit operations that are detected as follows:

- **Insert**: Cost to edit $A_1, A_2, ..., A_i \rightarrow B_1, B_2, ..., B_{j-1}$ + cost to insert B_j
- **Delete**: Cost to edit $A_1, A_2, ..., A_{i-1} \rightarrow B_1, B_2, ..., B_j$ + cost to delete A_i
- **Change**: Cost to edit $A_1, A_2, ..., A_{i-1} \rightarrow B_1, B_2, ..., B_{j-1}$ + cost to edit $A_i \rightarrow B_j$

Although able to adapt the Sankoff's string-to-string editing algorithm to solve the tree-to-tree editing problem, Selkow's algorithm has a major limitation because of the fact that delete and insert edits are restricted to the leaf nodes, meaning that deleting nodes containing other child nodes or inserting nodes on parents containing other child nodes is not possible. This results in having to perform 2n+1 edits for a simple insert/delete, n being the number of child nodes belonging to the parent where the edit action is made.

In order to solve this issue, Tai's algorithm [87] uses a dynamic programming approach without recursion where deletes and inserts are possible within parent nodes containing other child nodes. While deleting, all remaining child nodes are attached to the parent of the deleted node. While inserting, all remaining child nodes are attached to the newly inserted node. In order to adapt the string-to-string edit algorithm to trees, Tai's algorithm uses a so-called preorder tree traversal with the purpose of assigning unique numbers to each node, starting with the root node. Once all nodes on trees A and B are annotated, the algorithm compares the two trees by using the idea of traces (called mappings by Tai) from the Wagner and Fischer's string-to-string edit algorithm [94]. With the same edit operations as Selkow's algorithm

(insert, delete and change), Tai's algorithm detects insertions where for a given node in the B tree, there is no pair node on the A tree. The deletion is detected, while for a given node in the A tree, there is no node pair on the B tree. Finally, the change is detected while there are differences between two paired nodes. Although having improvements compared to the Selkow's algorithm, Tai's algorith has another limitation that forces the structure of the trees A and B to be preserved. While the two tree structures are different, Tai's algorithm is not able to operate correctly.

Another algorithm that is an improved version of Tai's algorithm was proposed by Zhang and Shasha [81, 104]. The main difference between the two algorithms is the tree traversal for assigning unique node numbers—see Figure 2-3. Zhang and Shasha's algorithm performs a postorder tree traversal instead of the preorder tree traversal carried out by Tai. Contrarily to the preorder tree traversal that starts the numbering at the root node annotating the remaining child nodes left to right, Zhang and Shasha's algorithm performs the postorder tree traversal and starts with the leftmost leaf descendant of the root, proceeding to the leftmost descendant of the right sibling of that leaf, the right sibling, then the parent of the leaf and so on up the tree to the root node. Compared to Tai's algorithm where the minimum cost computation is calculated once the tree traversal has been completed, Zhang and Shasha's algorithm is able to calculate the minimum cost distance right away, before the node is encountered. This is made possible by keeping track of the tree keyroots—defined as the root of the tree plus all nodes which have a left sibling. In addition to the tree distance defined as the distance between two nodes considered separately from their siblings and ancestors, the keyroots allow the calculation of the forest distance, defined as the distance between two nodes considering their left siblings in the trees A and B.

The previously mentioned algorithms were further reused in XML document comparison where XML trees are seen as ordered labelled trees. Besides the insert, delete and change edit actions, the move edit action was further added, seen as a combination of insert–delete edit actions of the same node.

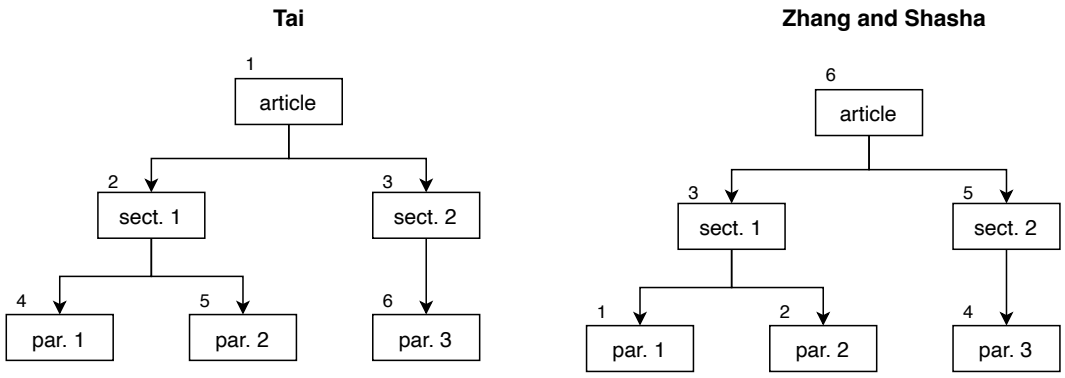


Figure 2-3: Tai vs. Zhang and Shasha tree traversal unique node numbers assignment. Tai's algorithm performs the preorder tree traversal and starts with the root; Zhang and Shasha's algorithm performs the postorder tree traversal and ends with the root node.

2.2 Named Entity Recognition

The goal of the NER task we plan to achieve is to annotate review comments with specific named entity categories in order to extract the changes requested by the reviewers. This additional information will be used by human readers to better understand the change requests and to correlate those requests with the actual changes extracted by XML document comparison. NER was first introduced in 1996 at the sixth Message Understanding Conference (MUC-6) [27]. It seeks to identify named entities in raw text and classifies text spans into predefined categories. The usual named entities we observe in the literature are person name, location, date, organisation, monetary value, etc. By analysing a specific raw text, for example, "John Doe was in Paris, France on February 21–24, 2022", important text spans can be identified and classified in specific categories. Within our example, "John Doe" can be categorised as a person, "Paris, France" as a location, and "February 21–24, 2022" as a date. In Natural Language Processing (NLP), NER plays an important role for tasks such as question answering [59], machine translation [2], text understanding [106, 11], relation extraction [102], entity linking [100], text summarisation [43], etc.

Although conceptually simple, NER is a complex task that usually requires large training datasets. The right choice of a named entity depends on multiple factors involving text semantics and the neighbourhood of the specific text span. After three

decades, researchers are still working on NER [98, 62, 20], and the NER task is not yet solved [55]. There are two main reasons for this: the limited set of named entities, especially for domain-specific NER tasks, and the different languages the NER needs to operate in.

There are two main named entity types: generic (or coarse-grained) and domain-specific (or fine-grained). Initially, coarse-grained NER [27, 77], representing named entities such as person name, location, date, organisation, etc. were developed. Later, fine-grained NER tasks [48, 69, 1, 38] appeared with a broader number of named entities where a text span can also be assigned to multiple named entities. Domain-specific named entities are proper to the domain they are used in. Researchers are working on NER for domains such as bio-medicine [86] and history [37], but also for some newly emerging topics such as analysing the COVID-19 literature [17]. The language being another important factor for NER, some research groups are working on the use of NER in languages other than English [83, 90, 52].

In the literature, there are different ways to perform NER [46]:

- **The handcrafted rules approach** uses grammar-, syntactic- and orthographic-based linguistic techniques [6]. This approach does not need annotated data, has a high precision but also a lower recall, is domain/language-specific and requires a large amount of work by experienced computational linguists. Linguists have to write specific, so-called handcrafted rules that will be able to analyse the raw text and identify specific named entities;
- **The feature-based supervised machine learning approach** [82] uses statistical models and turns named entity identification into a classification task. It is faster to put in place but requires a large amount of manually or semi-automated labelled data for training purposes. The model learns from the labelled data in a supervised manNER in order to correctly carry out named entity distinctions;
- **The hybrid approach** [56, 84, 85] is a combination of both handcrafted rules and feature-based supervised machine learning approaches. This approach can

achieve better results than each of the two individual approaches but also inherits the handcrafted rules technique weakness, where experienced linguists have to write specific rules by hand;

- **The unsupervised machine learning approach** uses clustering [61] with the idea of extracting named entities based on context similarity. The use of lexical patterns and statistics on a large corpus of raw text gives the possibility to match specific named entities and build their representations from raw data. This approach is not very popular in NER today;
- **The deep learning approach** [44] is the most used NER technique today [46], with many research groups [18, 31, 39, 12, 66] working on the topic and achieving state-of-the-art performance compared to the other approaches. deep learning is composed of multiple artificial neural network layers—see Figure2-4—created to learn representations of data with multiple levels of abstraction. This makes it possible to understand the context of a sentence using vector representation and neural processing. Recent deep learning models we can find in the literature are ELMo [67], GPT [68], BERT [23], XLM [40], XLNet [101], etc. BERT has also some derivative models as RoBERTa[49], SciBERT[4], DistilBERT [78] and ALBERT [41].

As deep learning is the most used NER technique today, achieving state-of-the-art performance, we will focus our research on this NER approach. As seen, there are already several pre-trained deep learning language models that can be further trained on the NER task, one of them being BERT. BERT was first introduced by researchers at Google AI Language [23]) in 2018, achieving state-of-the-art results in a large variety of NLP tasks. As seen in Figure 2-4, BERT is based on stacked layers of encoders and makes use of transformers, an attention mechanism that learns contextual relations between words and sub-words in a text. BERT is pre-trained on two NLP tasks: Masked Language Modelling (MLM) and Next Sentence Prediction (NSP). MLM consists of using randomly masked tokens where random words in a sentence are masked, and then the model is trained to predict those masked tokens. NSP is a binary classification task based on understanding the relationship between

two sentences. The model is trained to distinguish, for a randomly chosen sentence A from the training corpus, if sentence B is the sentence subsequent to sentence A. This is shown as beneficial to several NLP tasks, such as Question Answering (QA) and Natural Language Inference (NLI).

During the pre-training phase—see Figure 2-4 showing the NSP training—BERT learns contextual embedding for words which is computationally expensive and requires large training sets. Afterwards, BERT models can be fine-tuned for specific tasks such as NER—see Figure 2-5—text classification, question answering and semantic textual similarity, using fewer resources and smaller datasets. BERT has achieved state-of-the-art performance in those NLP tasks: General Language Understanding Evaluation (GLUE), Stanford Question Answering Dataset (SQuAD), Situations With Adversarial Generations (SWAG) etc. It uses the WordPiece [99] subword-based tokenisation algorithm for constructing its BERT base and BERT large vocabularies. The difference between the BERT base and BERT large lies on the number of encoder layers. The BERT base has 12 and BERT large 24 transformer encoder layers stacked on top of each other. Besides the number of encoder layers, BERT also uses cased and uncased text. The difference between BERT cased and BERT uncased is during the WordPiece tokenisation step. BERT cased uses cased text, accents and diacritical marks. By contrast, BERT uncased converts everything to lowercase and also deletes the accents and diacritical marks. For example, the following text, "Université Haute Alsace", while being transformed from cased to uncased, will result in "universite haute alsace". Note the first letter of each word that became lowercase and the accent removal from "é". In total, there are four combinations of BERT models: BERT-base-cased, BERT-large-cased, BERT-base-uncased and BERT-large-uncased.

During fine-tuning—see Figure 2-5—the same pre-trained model parameters are used and further fine-tuned on the NER task. Each input example starts with a [CLS] token, and each sentence is delimited with a [SEP] token. The input example first goes through the WordPiece tokenisation, where it gets converted into tokens. Once tokenisation has taken place, the input tokens go to the embedding phase where each

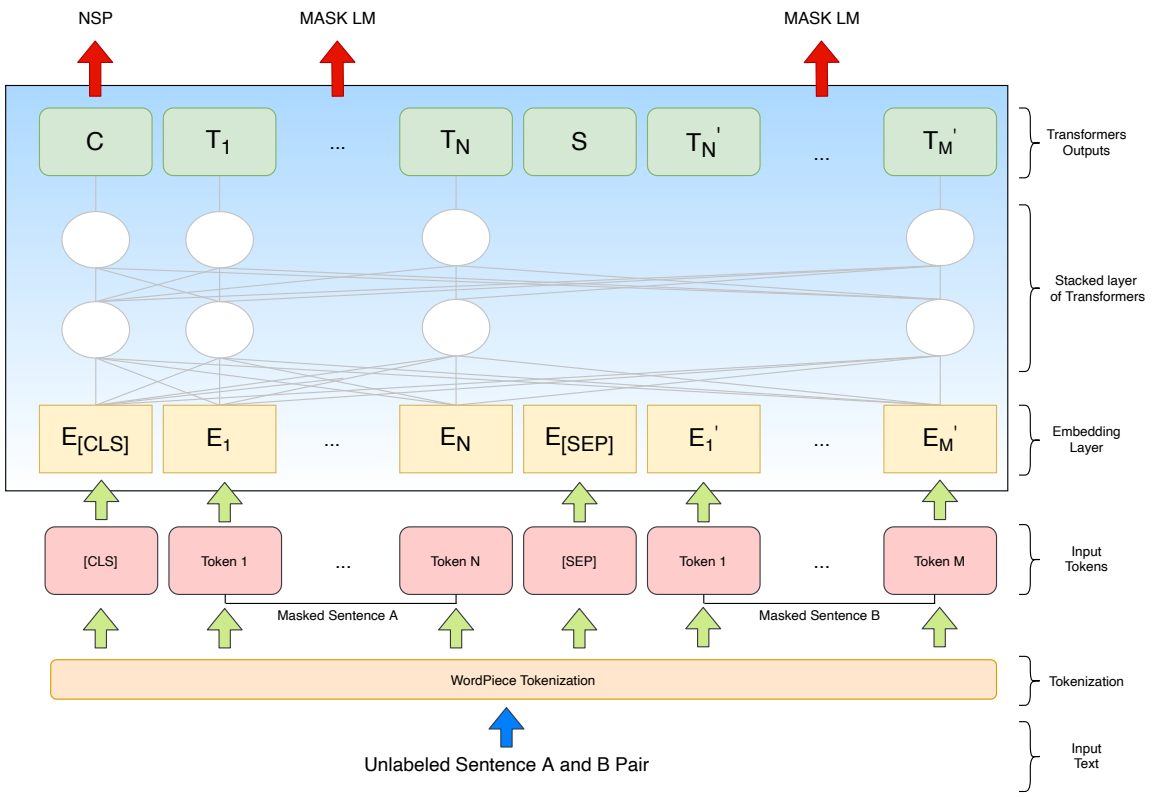


Figure 2-4: Pre-training NSP phase principles for BERT.

NSP is a binary classification task in which, given a pair of sentences, it is predicted if the second is the actual next sentence of the first sentence. The [CLS] symbol is added in front of every input example, and [SEP] is a separator token separating different sentences.

input token will have its embedding representation (Ex). Input embedding is the sum of the token, segmentation and position embedding. Each embedding representation then goes through different layers of transformers where every layer performs multi-headed attention computation on the word representation of the previous layer. In a n-layer BERT model, a token will have n intermediate representations.

We will describe some of the BERT-derived deep learning models in order to understand their specificity compared to the original BERT:

- **SciBERT** is a BERT-based pre-trained model on scientific data. Compared to BERT vocabulary usage, SciBERT can use both BASEVOCAB (BERT base) and a new SCIVOCAB vocabulary constructed using the SentencePiece [35] subword-based tokenisation algorithm. There are four variants of SciBERT, using different combinations of BASEVOCAB and SCIVOCAB as vocabularies,

and cased and uncased text: scibert_scivocab_cased, scibert_scivocab_uncased, scibert_basevocab_cased and scibert_basevocab_uncased. The models using BASEVOCAB are fine-tuned from the BERT base models, while the SCIVOCAB models are trained from scratch. SciBERT can perform the following NLP tasks: NER, Dependency Parsing (DEP), PICO Extraction, Relation Classification (REL) and Text Classification (CLS).

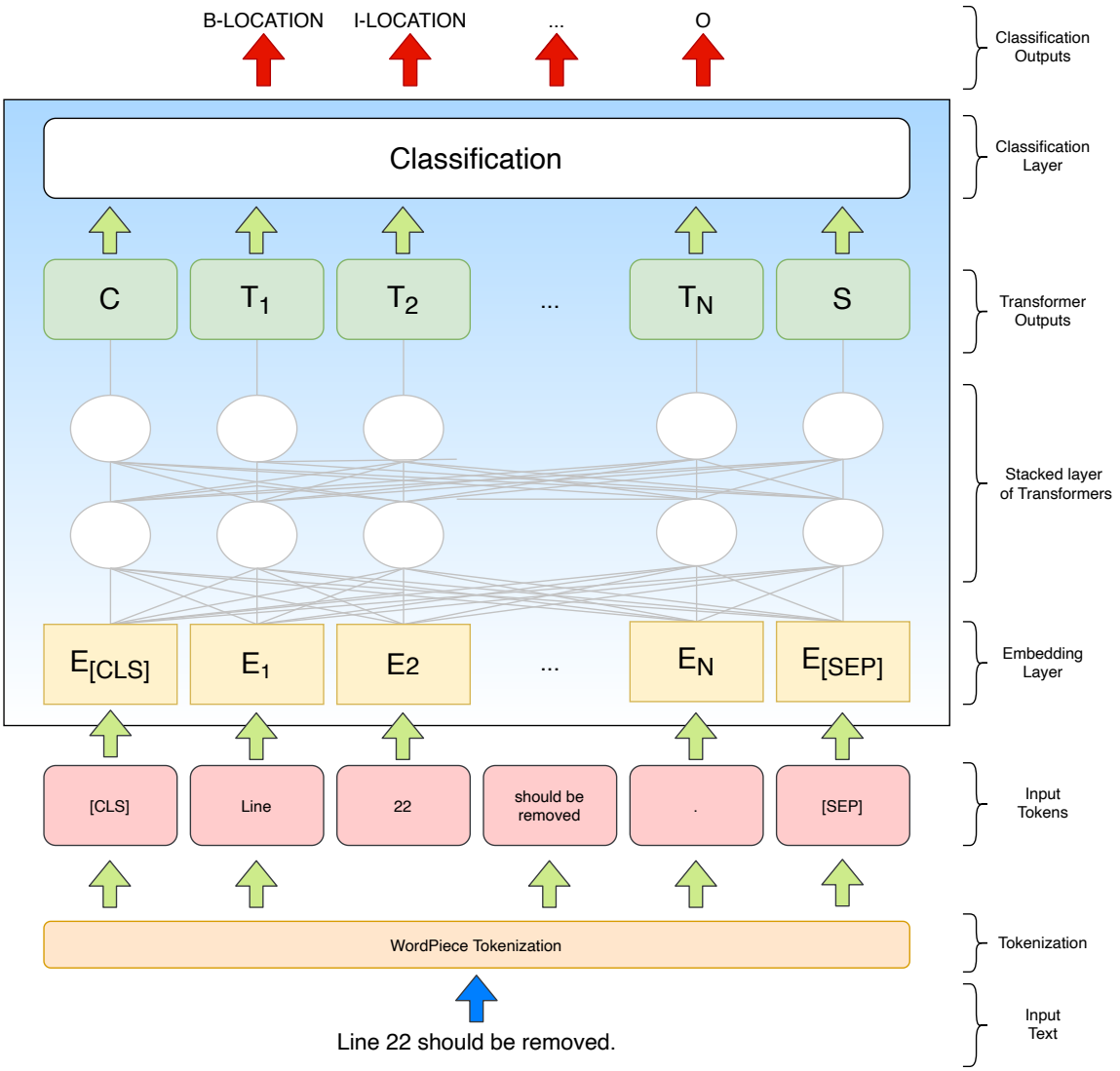


Figure 2-5: Fine-tuning phase principles for BERT on NER task.

During fine-tuning, the same pre-trained model parameters are used and further fine-tuned on the NER task. The [CLS] symbol is added in front of every input example, and [SEP] is a separator token separating different sentences.

- **RoBERTa**, i.e., the Robustly Optimised BERT Pre-Training Approach, was developed by researchers at Facebook and Washington University with the purpose of optimising the training of BERT in order to reduce the pre-training time. This is made possible by using specific BERT design choices and training strategies for better performance. For this, the following was carried out:
  - Disable the NSP which improved the downstream task performance;
  - Train with bigger batch sizes (number of training examples utilised in one iteration) and longer sequences: compared to BERT that was trained for 1M steps with a batch size of 256 sequences, RoBERTa was trained for 125 steps with a batch size of 2K sequences and 31K steps with a batch size of 8k sequences. Large batches are easier to parallelise via distributed parallel training, and their use improves the perplexity on masked language modelling objective and end-task accuracy;
  - Dynamically change the masking pattern applied to the training data: in the BERT architecture, masking is carried out once during the data preprocessing step. With RoBERTa, the training data are duplicated and masked 10 times over 40 epochs, each time with a different mask strategy, resulting in 10 x 4 epochs with the same mask.

  RoBERTa achieved state-of-the-art performance on the SQuAD 1.1, SQuAD 2.0 and RACE benchmark datasets. The same was found in four GLUE tasks: Multi Natural Language Inference (MNLI), QuestionNLI, Semantic Textual Similarity Benchmark (STS-B), and Recognising Textual Entailments (RTE).

- **DistilBERT** is presented as a distilled version of BERT that is 40% smaller in size while being 60% faster and able to retain 95% of BERT's language understanding capabilities while tested on a GLUE language understanding benchmark. In order to leverage the inductive biases learned by larger models during pre-training, the authors introduce the triple loss technique combining language modelling, distillation and cosine-distance losses. The knowledge distillation technique [5, 29] consists of training a smaller student model to reproduce the

behavior of a larger teacher model, which is what RoBERTa does with BERT.

- **ALBERT** is another BERT-derived model with reduced memory usage compared to its predecessor. For this to happen, ALBERT uses the following techniques:
  - Cross-layer parameter sharing: same as BERT base, ALBERT architecture also has 12 transformer encoder layers stacked on top of each other. However, compared to BERT, where each encoder layer is initiated with a specific set of weights, the encoder weight initiation for ALBERT is carried out by repeating the first encoder weights to the remaining encoders. This directly reduces the number of unique parameters, which reduces memory usage;
  - Embedding factorisation: In BERT, the vocabulary is of the same size as the hidden layer. ALBERT adds a smaller layer between the vocabulary and the hidden layers, which decomposes the embedding matrix into two smaller matrices. This approach reduces the total number of parameters between vocabulary and the first hidden layer;
  - Sentence-order prediction (SOP): Compared to BERT, which uses next sentence prediction (NSP), ALBERT uses SOP that focuses on inter-sentence coherence and self-supervised loss, which improve loss compared to BERT.

XLNet is another deep learning model that achieved significant performance improvements using the benefits of transfer learning approach. The authors of XLNet claim that this new model outperforms BERT on 20 tasks, including question answering, natural language inference, sentiment analysis and document ranking. XLnet is an extension of the Transformer-XL model [21]. Compared to BERT that is an auto-encoding based model, XLNet is pre-trained using an auto-regressive method with the goal of learning bidirectional contexts using Permutation Language Modelling (PLM). PLM is the concept of training a bi-directional auto-regressive model on all permutation of words in a sentence. This difference mainly impacts the MLM task,

which is illustrated within the example in Figure 2-6. While selecting the two tokens [Clermont, Ferrand] as the prediction targets and streaming to maximise the log (Clermont, Ferrand | is, a, city), XLNet is able to capture the dependency between the pair (Clermont, Ferrand), which is omitted by BERT. The XLNet pre-trained model can be used and further fine-tuned in two different versions: the XLNet-large-cased and the XLNet-base-cased. The XLNet-large-cased model variant has 24 and the XLNet-base-cased model variant 12 transformer encoder layers stacked on top of each other.

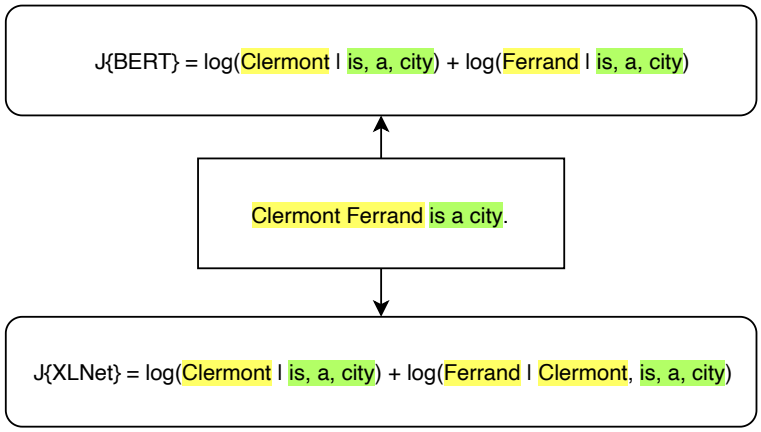


Figure 2-6: Main difference between XLNet and BERT on MLM task.

Compared to BERT that is an auto-encoding based model, XLNet is an auto-regressive-based model. Within the example, this approach allows XLNet to capture the dependency between the pair (Clermont, Ferrand), which is omitted by BERT.

All previously described models have the advantage of tokenising the words before going into the embedding layer. This avoids having to pre-process the input text with stemming or lemmatisation. If we take the example of the word run that can also be mentioned as running, BERT will for example decompose that word into two tokens "run" and "##ing" before going to the embedding layer.

Once the different deep learning models we will fine-tune on our NER task have been described, we will continue with the state of the art regarding the evaluation criteria during the model training and testing phases.

2.2.1 Evaluation criteria

Training phase

In deep learning, the goal of the training (or the fine-tuning) phase is to calculate the best possible weights that will allow the model to make good predictions. Due to the complexity of the problem, calculating the perfect weights is not possible as there are too many unknowns. In order to improve the model prediction, the learning process is projected as an optimisation problem with an initial arbitrary weight-set that is further optimised through different training epochs. For this, the training dataset is used in order to train the neural networks using the stochastic gradient descent optimisation algorithm [45] where the gradient is the error gradient. Starting with the initial arbitrary weight-set, the model predictions are evaluated on the training dataset, and the prediction error is calculated using a loss function [95]. With the prediction error, the weights are updated using backpropagation [76] so that the next evaluation reduces the error. The gradient descent algorithm seeks to change the weights with the goal to reduce the error on the next evaluation, meaning the optimisation algorithm navigates down the error gradient. The parameter that determines the step size at each iteration while navigating down the error gradient is called learning rate or gain. This parameter represents the speed at which the model learns.

One of the major issues faced during the training phase is the overfitting problem [28], where the model learns the training data to the extent that it negatively impacts the performance of the model on new data. In order to avoid overfitting, a technique called weight decay is used for regularisation. It consists of adding a small penalty to the loss function with the purpose of shrinking the weights during backpropagation.

Compared to weight decay and the learning rate that have a direct impact on the model prediction success, the batch size defines the number of samples that will be propagated through the network and also impacts the training speed and memory usage. The bigger the batch size is, the higher the memory usage and the slower the model is to train. On the other side, the smaller the batch, the less accurate the

estimate of the error gradient will be. Another standard parameter while training neural network models is the warm-up learning rate. The warm-up phase represents the beginning of the training where the warm-up learning rate is a lower percentage of the learning rate initially defined. Over the initial epochs, the warm-up learning rate is doubled per epoch until it reaches the initially defined learning rate. This parameter is very useful, while the training dataset is highly differentiated and the number of training epochs is low. Without its usage, the model could face the early overfitting problem as it can quickly learn with a high learning rate on a specific batch of examples that are strongly related.

Our NER task being a multi-class classification problem, assigning the right class to a specific word consists in classifying that word as belonging to one of the classes. This problem is projected as the prediction of the likelihood of that specific word belonging to each of the possible classes. During training in a multi-class NER task and under the maximum likelihood framework [75], the commonly used loss function is cross-entropy [22]. This loss function is used to calculate the error between two probability distributions. Figure 2-7 shows the use of the cross-entropy function that enables loss calculation.

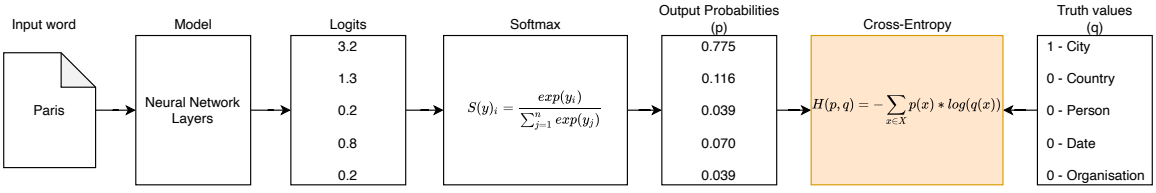


Figure 2-7: Cross-entropy loss function use.

On the left, the input word goes through the neural network layers in order to calculate its logits per different named entity classes. The Softmax function then converts logits into probabilities. The cross-entropy of the distribution of truth values q relative to the distribution of output probabilities p is then calculated in order to measure the distance from the output probabilities to the truth values.

Testing phase

In order to evaluate a neural network model on a given task, the standard approach consists of evaluating its precision, recall and the F1 score [26]. This is carried out

during the testing phase and consists in executing the model on a dataset containing examples the model never saw during the training. Figure 2-8 shows how the precision, recall and F1 scores are obtained in a binary classification task. The model is executed on the testing dataset examples in order to evaluate the differences between the predicted and the correct values. Four different result groups are obtained during the evaluation:

- False Positives: Negative labels that were wrongly predicted as being positive;
- True Positives: Positive labels that were correctly predicted as being positive;
- False Negatives: Positive labels that were wrongly predicted as being negative;
- True Negatives: Negative labels that were correctly predicted as being negative.

Figure 2-8: Precision, recall and F1 score.

Once the results have been grouped, the calculation of the precision and the recall is made possible. The precision is defined as the ratio of successful predictions to all attempted predictions. The recall is defined as the ratio of successful predictions to

the total number of the positive cases in the dataset. The higher the precision and the recall are, the better the prediction is. The two following formulas are used in order to calculate precision and recall:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

Once precision and recall have been calculated, the calculation of the F1 score is made possible, that is, by definition, the combination of both. The F1 score is the harmonic mean representing the average of precision and recall. In order to have a high F1 score, both precision and recall have to be high. The mathematical formula to calculate the F1 score is as follows:

$$F1\ score = 2 * \frac{Precision * recall}{Precision + recall}$$

Another basic metric for evaluation of neural network models on classification tasks is accuracy. This performance indicator represents the proportion of correct predictions over all predictions. The mathematical formula to calculate accuracy is as follows:

$$Accuracy = \frac{TruePositives + TrueNegatives}{TruePositives + FalsePositives + TrueNegatives + FalseNegatives}$$

However, the accuracy performance indicator is ineffective in assessing model performance on imbalanced datasets—see Section 2.2.2. Thus, we will not calculate, nor use, this performance indicator.

It is important to note that compared to the binary classification where each performance indicator represents a single overall score, in multi-class classification, the different performance indicators are calculated for each class separately in a one-vs.-rest approach as if there are distinct classifiers for each class. Compared to binary classification performance indicators, for multi-class classification, there are additional

useful metrics, such as the micro, macro and weighted averages that are calculated per precision, recall and F1 score within the one-vs.-rest approach.

The macro average is calculated by taking the arithmetic (unweighted) mean of all the per-class performance indicators. The mathematical formula to calculate the macro average (of the F1 score in this example), n being the number of total classes, is as follows:

$$MacroAvg = \frac{\sum_1^n F1_n}{n}$$

The micro average is calculated as a global average by counting the sums of the True Positives, False Positives and False Negatives. The mathematical formula to calculate the micro average (of the F1 score in this example), n being the number of total classes, is as follows:

$$MicroAvg = \frac{\sum_1^n TruePositives_n}{\sum_1^n TruePositives_n + \sum_1^n FalseNegatives_n}$$

The weighted average is calculated by taking the mean of all per-class performance indicators while being aware of the dataset imbalance (per-class representation in the dataset represented by the support proportion indicator). The mathematical formula to calculate the weighted average (of the F1 score in this example), n being the number of total classes and sup being the support proportion indicator, is as follows:

$$WeightedAvg = \sum_1^n F1_n * sup_n$$

2.2.2 Datasets

The first thing to check while annotating the datasets to be used in supervised learning is the representation ratio for each class. Having a so-called imbalanced dataset where certain classes are more represented compared to others could be confusing for the model that will learn to assign higher probability weights to the over-represented classes and lower probability rates to the under-represented classes. In order to cope with imbalanced datasets, there are different techniques that can be applied on the

dataset directly, by changing its composition in order to improve the class distribution, or during the model evaluation phase by measuring the right performance indicators. The following techniques can be used in order to improve the dataset balance:

- **Collect more data**: add examples of the under-represented class to the dataset;
- **Oversampling**: copy dataset instances of the under-represented class;
- **Undersampling**: delete dataset instances of the over-represented class;
- **Combination of over– and undersampling**: copy dataset instances of the under-represented and delete dataset instances of the over-represented class.

The first approach that consists in collecting more data is the ideal but less commonly used technique as it requires additional data which are not always possible to obtain. Regarding the data-sampling techniques, there are many variants that can be divided into three categories: oversampling, undersampling and a combination of both. The simplest oversampling technique is random oversampling which, as its name indicates, consists in randomly adding more support examples of the under-represented classes. This technique is, however, prone to the overfitting problem, where the model models the training data too well due to the random duplicates. The most popular oversampling technique that helps to overcome the overfitting problem is the Synthetic Minority Oversampling Technique (SMOTE) [9]. This technique consists in selecting support examples that are close in the feature space and, with the help of interpolation between the positive instances that lie together, copying those support examples in order to reduce the dataset imbalance. Regarding the undersampling techniques, the simplest to implement is random undersampling, which is the opposite of the random oversampling technique and comes with the under-fitting problem where the model cannot model the training data nor generalise to new data. Alternative approaches exist for the undersampling techniques, allowing to solve the under-fitting problem, such as the Condensed Nearest Neighbor Rule (CNN), Near Miss undersampling, etc. Those techniques are, however, less used as they consist in decreasing the size of the training dataset, which is a costly approach. It is also common to have a combination of over- and undersampling techniques in order to

cope with the dataset imbalance. For example, combining the oversampling SMOTE technique with an undersampling method applied to the dataset after SMOTE allows improving the dataset imbalance while keeping a similar dataset size. Unfortunately, it is not possible to use any of those techniques for our dataset due to the fact that all examples are imbalanced in a similar way and there are no examples in which the under-represented classes are solely available. Moreover, the classes are linked together, dependent on each other, and each have similar word sizes.

Another approach to coping with less severe imbalanced datasets is accepting it and being aware of it during the model testing phase. This can be achieved by measuring the right performance indicators. The first basic performance indicator while evaluating models on classification problems is its accuracy—see Section 2.2.1. Accuracy represents the ratio between the number of correct predictions over all predictions and does not taking into consideration the representation of a given class within the dataset at all. If we take as an extreme example a dataset having 1000 representations of class A and only 10 representations of class B, having 5 wrong predictions in both classes A and B will result in having an accuracy of 0.95 for class A and 0.5 of class B. With imbalanced datasets, exclusively measuring accuracy is not optimal, and the use of additional performance indicators such as precision, recall and F1 score—see Section 2.2.1—better addresses imbalanced datasets. In addition to those performance indicators, there is the weighted average indicator that takes into consideration the support examples ratio per class while making the averages—see Section 2.2.1. Using those performance indicators and having a non severe imbalanced dataset can give us accurate performance indicators while evaluating our models.

In the above, we described some of the existing deep learning models that can be fine-tuned on our NER task. In addition, we covered how to evaluate a given model during the training and testing phase. Finally, we saw what the different impacts and solution techniques are to deal with imbalanced datasets. In Chapter 4, we will apply some of those models in the domain of academic peer review comments with the goal to improve the final decision-making process by review comments annotation.

Chapter 3

Document comparison

In order to evaluate the current XML diff algorithms' capacity to have a bijection between the author edits and the XML diff while comparing JATS academic articles, we have identified 12 implementations of XML diff algorithms as seen in Table 2.1: 11 are state-of-the-art algorithms from the scientific literature and 1 is a commercial implementation. Our goal is to asses each of those algorithms by analysing their capacity to detect and represent common author edit actions we have extracted during the article revision process. Each of those common edit action was then applied on our dataset by creating one JATS XML file pair per edit action.

3.1 XML Diff Comparison Study

In order to perform the analyses of the 12 XML diff algorithms, we start with describing different common edit actions an author applies during the article revision and the impact those edits have on JATS XML . Following is the XMLDiffAnalyzer script we developed in order to automate one part of the analysis. We continue with carrying out an initial high-level performance and suitability analysis of the algorithms for JATS document comparison. The performance is evaluated in terms of execution time, average and maximal memory used and resulting delta file size. The suitability depends on the results obtained by analysing the delta files for specific author edit operations which impact the XML document in both the text and tree structure.

Finally, we focus on the three algorithms we identified as most suitable and three algorithms identified as low-performing in JATS comparison. A deeper analysis of the delta outputs they generated is performed in order to identify their strengths and weaknesses.

3.1.1 Evaluation metrics

In previous XML diff comparison studies [16, 30], the main criteria for an appropriate algorithm were based on execution time, delta size, CPU and memory usage. In our case, we are mainly focused on the delta output—that is, all the differences between two versions of an academic article should be detected, correctly interpreted and represented. In order to assess the delta output, we need to understand the modifications an author is making during the revision process and correlate those with the changes observed on JATS. During the revision rounds, the following author modification actions are seen:

- **Paragraph correction** is the most common author modification we observed. This is because paragraphs represent the largest text content part of the article (over 95%). Authors mostly modify paragraph content but also move, merge or split paragraphs. Content changes include text additions, removals, moves and style changes. Paragraphs are also composed of other objects such as mathematical formulas and citations that are subject to editing. Smaller corrections can be interpreted as updates, while larger corrections can be seen as rewrites;
- **Section correction** is mostly about article structural change. A section is composed of other sub-elements such as its title, subsections, paragraphs, figures, etc. Authors modify section sub-elements, merge and split sections or upgrade subsections into sections or downgrade sections into subsections;
- **Author correction** is observed while modifying author information, adding or deleting an author or changing an author's position within the authors list;
- **Title correction** is observed in the article title, which is a relatively short portion of text;

- **Citable object correction** is observed on objects that can be cited within the article (i.e., references, figures, tables, sections, algorithms, etc.). The particularity of those objects is that their order of appearance within the article is used to generate their incremental citation number; moving a table, reference or figure impacts the number with which this object is cited within the article;
- **Embedded object correction** is observed on objects that are externally produced and inserted in the article. The most common examples are figures.

By further analysing the previously mentioned modifications observed on a corpus of 50 academic article revision pairs, we identified nine general edit actions that an author can apply: Add, Remove, Update, Move, Merge, Split, Upgrade, Downgrade and Styling. Add and Remove are the two main modifications, and the others can be reinterpreted with their sequence. Figure 3-1 shows the correlation between the typesetter version of an academic article and its JATS representation. As mentioned in the introduction, JATS has structured data, and important information resides in text nodes, tree element structures and attributes. The following list shows a short description of each edit, divided into two groups, content and structural edits:

Content edits:

- **Add** is the first main modification observed on all levels of the article. Authors can add characters, sentences, paragraphs, sections, references, etc.;
- **Remove** is the opposite of Add;
- **Update** is observed where changes are minor. Instead of representing a typo correction or a rephrasing in a paragraph as a full Remove/Add sequence, Update is used as a more fine-grained approach;
- **Styling** is observed on portions of text, mostly within paragraphs. Authors can add or remove styling elements such as italic, bold, subscript, superscript, etc. Those styling elements can also have their range changed while extending or shrinking the styled portion.

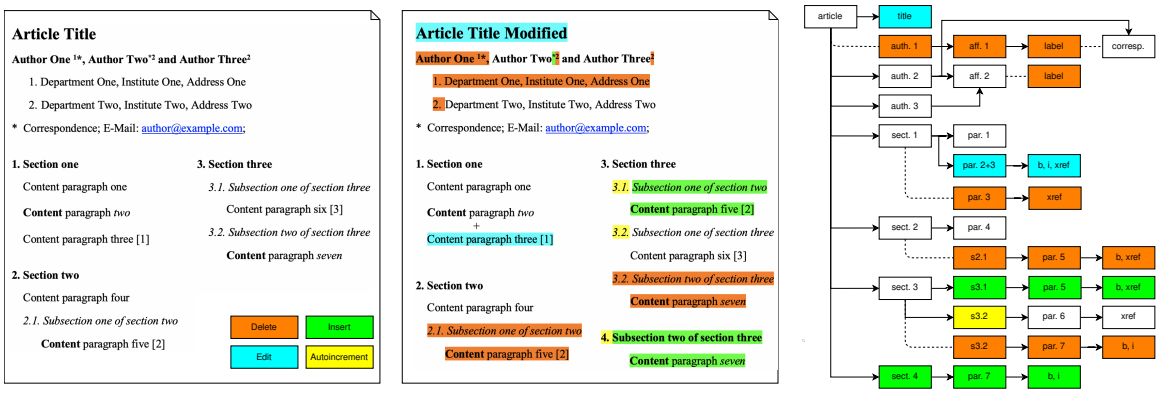


Figure 3-1: Author modifications' impact on JATS article version.
Left, the original typesetter article; middle, the modified version; right, the impact on JATS.

Structural edits:

- **Move** is observed where the order of items within the article is changed. It has no impact on the content itself, only on its position within the article;
- **Merge** is observed on paragraphs, subsections and sections. For paragraphs, it usually consists of removing the line breaks between two paragraphs in order to form one larger paragraph. For subsections and sections, the line breaks are removed and, additionally, their titles are merged;
- **Split** is the opposite of Merge;
- **Upgrade** is observed on subsections and consists of changing a subsection to a section;
- **Downgrade** is observed on sections and consists of changing a section to a subsection.

Each of the previously described modifications that an author applies to an article is a sequence of the edit actions we identified. Paragraph correction, for example, can be composed of text Insertions, Deletions, Updates, Moves, Merges with other paragraphs, Splits or Styling changes.

3.1.2 JATS Edit Actions

In order to understand how these common author modifications impact the JATS versions of the article, we analysed each of them by observing the changes from an XML perspective. This gave us the opportunity to define 16 edit actions produced by the author and reflected on the JATS. Those actions are divided into two edit groups: six text and ten tree edits. Text edits are actions observed on text nodes, mostly paragraphs, but also article title, author name, etc. Tree edit actions are observed on the tree structure of the JATS while adding or deleting elements, moving paragraphs or sections, etc.

Text edits:

- **Delete**: basic edit operation that can be executed on any text node within the article. Can be found in text deletion in paragraphs, sections, authors, title and other nodes;
- **Insert**: basic edit operation that can be executed on any text node within the article;
- **Move**: composite operation that is often seen in paragraphs. Entire sentences or parts of sentences are moved in order to correct the grammar or improve the writing clarity;
- **Update**: composite edit type that can be seen in any of the XML nodes. Makes sense while doing relatively small modifications to a given text. Having a text edit action with a high volume of inserted/deleted text should not be considered as an update, but rather as a full rewrite using Delete–Insert actions;
- **Style**: specific edit type where styling elements range change while extending or shrinking the styled portion. One example is the extension of a portion of text that is in italics;
- **Complex environment edits**: any text edit performed on text nodes that have the specificity of embedding other child nodes such as styling, reference, mathematical formulas, etc. Being very common in academic article XML rep-

resentations, text edits applied on such nodes should be detected in the same way as edits on non-complex nodes.

Tree edits:

- **Delete**: produced by deleting specific parts of the article, for example an author, section, paragraph, reference or figure. Specific enough that when a parent node is removed, all child nodes are also removed;
- **Insert**: the opposite of Delete;
- **Update attribute**: specific to XML documents. Observed when changing the correspondence of an author, the license information of the article or the reference type within the reference;
- **Move**: observed in the reordering of authors, references, figures and other elements. Important edit action for text-centric XML documents and is reflected by changing the order of child nodes within their parent node;
- **Merge**: observed when paragraphs or sections are merged. Implies the merging of multiple parent nodes with all their child nodes into a single parent node containing all child elements from each merged parent node;
- **Split**: the opposite of Merge;
- **Upgrade**: observed when subsections are changed to sections;
- **Downgrade**: opposite of Upgrade and is observed when sections are changed to subsections;
- **Style** consists of adding/removing styling such as bold or italics to portions of text;
- **Complex environment edits**: any tree edit performed on nodes that contain text nodes embedding other markups.

After observing a sample batch of 50 real-life author JATS articles[1], we found that the frequency of complex paragraphs was much higher compared to that of plain text paragraphs. The average ratio is 80% vs. 20%. The largest edited part of a JATS document is the article body, composed mostly of paragraphs. We conclude that edits on complex text and trees are important.

[1] github.com/milos-cuculovic/XMLDiffAnalyzer/tree/master/Supplement/XML

3.1.3 XMLDiffAnalyzer Script

In order to automate the execution and the results collection for the 12 algorithms we are comparing, we developed the XMLDiffAnalyser script available online[2]. The results collected and JATS testing files prepared from the original research paper [53] are also available[3]. Algorithm 1 presents the pseudo code of the script written in Python. The script embeds the executable of 12 algorithm implementations, the compared articles and the resulting delta files per algorithm per article. It requires the user to input the number of times each algorithm will be executed, and which articles the algorithms will run on. It measures and returns the time, memory and delta size performance data, writing them to a CSV file and generating SVG vector figures. In case of further tests, the algorithm can be easily reused for other XML documents.

Algorithm 1 XMLDiffAnalyzer

```
procedure XMLDiffAnalyzer()
    tools ← XyDiff, JNDiff, JXyDiff, ..., DiffXml
    rounds ← 5
    articles ← { orig_1  new_1
               { orig_2  new_2
               { ...     ...
               { orig_n  new_n
    delta_dir ← Full path of the delta save directory
    start(rounds, articles, delta_dir)
    function start(rounds, articles, delta_dir)
        build the CSV file
        for article in articles do
            for round in rounds do
                for tool in tools do
                    result ← proc(tool, rounds, article)
                    for result_item in result do
                        generate result_item pyplot graph
                        save result_item into SVG vector figures
                    end for
                    results[] ←result
                end for
            end for
        end for
        write results[] in the CSV file
    end function
    function proc(tool,rounds,article)
        build and execute the command
        results.execution_time ← Execution time
        results.memory ← Max and average memory used
        results.delta_file_size ← Size of the delta output
        return result
    end function
end procedure
```

[2]github.com/milos-cuculovic/XMLDiffAnalyzer

[3]github.com/milos-cuculovic/XMLDiffAnalyzer/tree/master/TestingCorpus

3.1.4 Coarse-grained evaluation

The initial evaluation phase consists of a high-level performance and suitability analysis of the 12 XML diff algorithms. Here, we carry out a coarse-grained evaluation with the purpose of identifying the potential suitable algorithms for JATS article comparisons. These are further analysed in Section 3.1.5 with a more fine-grained approach. The coarse-grained evaluation is divided into two parts: first the performance evaluation, and then the delta output analysis.

Performance Evaluation

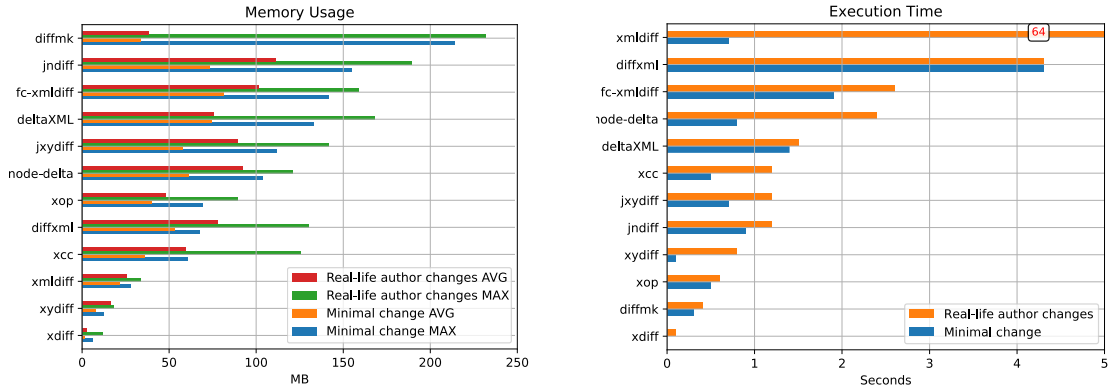
JATS articles are large text-centric XML files that may vary from 100 KB to 400 KB. In order to calculate the minimum performance requirements for suitable algorithms, we measure the time and memory needed per algorithm for a comparison. The tests[4] were run on two different edit scenarios: the first on a JATS representing one minimal change on the title, and the second on a JATS representing real-life author changes to the article title, authors, affiliations, paragraphs, figures, tables, references, etc.

Figure 3-2a represents the average and maximum RAM usage per algorithm. The results range from 5 MB for XDiff to 120 MB for DiffMK in terms of average RAM used, which is more than acceptable within our environment. We noticed that DiffMK has an imbalance between the average and maximum RAM used, where the maximum peaks are up to six times larger than the average.

Figure 3-2b shows the execution time each algorithm takes on average to perform a comparison of the XML file pairs. The average was calculated on five comparison round executions per algorithm. Except for XMLDiff, where the execution time scales exponentially while the number of changes increases, the rest of the algorithms are able to complete the comparison in under three seconds (under five seconds for DiffXML), which is acceptable within our environment.

The purpose of the delta file size measure is to obtain initial insights on the number of potential operations that each algorithm will produce, and how those are

[4]The evaluation was carried out on an Apple MacBook Pro (15-inch, 2016); Processor: 2.7 GHz Quad-Core Intel Core i7; Memory: 16 GB 2133 MHz LPDDR3; SSD Hard Drive



(a) Memory (b) Time

Figure 3-2: Memory and Time evaluation—minimal vs. real-life author changes.

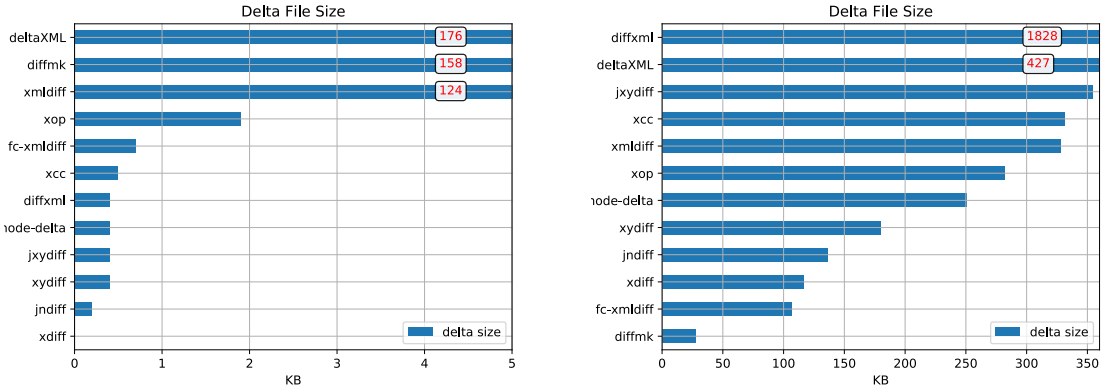
Average and maximum memory and time measured per algorithm in two edit scenarios: one with minimal text edit and the other with real-life author changes. In both scenarios, the algorithms were run to compare the same original file with its modified version depending on the two edit scenarios.

stored. For one minimal change comparison, Figure 3-3a shows that three algorithms, DeltaXML, DiffMK and XMLDiff, produce large delta files compared to others due to the fact these algorithms represent differences by annotating one of the two compared XML files. While evaluating the real-life author changes comparison, Figure 3-3b shows that DiffXML produces the largest delta file with over 10k edit actions, which seems to be oversized. JXyDiff and XCC are both heavily affected by the number of changes. Overall, with the exception of DiffXML, the algorithms produce delta files with acceptable sizes within our environment.

Delta Output

The evaluation of delta outputs is carried out within the two modification groups we have presented—text and tree edits—with a total of the 16 edit actions. Using the XMLDiffAnalyzer, we applied the algorithms to 16 different JATS file pairs, each pair reflecting one of the edit actions. The full comparison results are shown in the supplementary supporting information[5] organised into individual tables, each table

[5]github.com/milos-cuculovic/XMLDiffAnalyzer/tree/master/Supplement/Supplementary.pdf



(a) Minimal change

(b) Real-life author changes

Figure 3-3: Delta size—minimal vs. real-life author changes.

Delta size per algorithm applied to the same original JATS—once on the modified JATS with minimal text edit and once with real-life author changes.

describing our observation per edit action. The scoring system gives between zero and two points per edit action by comparing the expected and obtained delta results. The number of points is weighted by a factor two for complex edit actions, both on text and trees. This decision is supported by the importance of the author edits on complex content, as mentioned in 3.1.2.

We present the delta output results in Table 3.1. The total score is calculated by summing the successful matching of expected versus obtained delta outputs per edit action. Based on this, we can rate the JNDiff algorithm as first; XyDiff sharing the second place with XMLDiff; and XCC as third.

	Text edits						Tree edits										
Algorithm	Del	Ins	Mv	Upd	Style	Complex	Del	Ins	Attr	Mv	Merge	Split	Upgr	Dwngr	Style	Complex	Res/36
XOP																	0
DiffXML	1	1															2
FC-XmlDiff								2	1								3
DiffMK		1		1				2									4
Xdiff							2		2							2	6
Node-delta	1	1					2		1		1	1	1			2	10
DeltaXML	2	2					2	2				1	1	1			11
jXyDiff	2	2		1	1		1	2	2	2							13
XCC	2	2		1	1			2	2		1	1				2	14
XMLDiff	2	2		2	2	4		1					1			1	15
XyDiff	2			1	1		2	2	2	2	1	1	1				15
JNDiff	2	2		1	1	4	2	1	2	2	1	1	1	1		4	25

Table 3.1: Delta output analysis.

Coarse-grained analysis of delta output per edit action for each of the 12 XML diff algorithms with results divided into two groups: text and tree edits.

By further analysing the obtained results, we first observe that none of the algorithms are able to deal with text move nor style addition nor removal detection. In the best-case scenario, those are considered full text node updates, and in other cases treated as a suite of Delete–Insert actions. Operations on complex text nodes are common in text-centric XML documents, and only one of the 12 algorithms, JNDiff, is able to successfully treat complex text edits. Regarding complex tree operations, the best-scoring algorithm is XDiff because it only detects tree edits and is not affected by analysing text edit operations. Tree move is only supported by jXyDiff, XyDiff and JNDiff. Most of the remaining algorithms present this change as a sequence of tree Delete–Insert operations. Tree merge, split, upgrade and downgrade are not fully supported by any of the tested algorithms. Node-Delta, DeltaXML, XyDiff, XCC and JNDiff are partly able to represent some of those operations as a short sequence of tree Delete–Insert operations.

3.1.5 Fine-Grained Analysis

As seen in Table 3.1, none of the tested XML diff algorithms are able to fully fulfil our expectations for comparing JATS documents. Moreover, XMLDiff is too slow within our environment (see Figure 3-2b). However, three of them (JNDiff, XyDiff and XCC) are interesting, as their delta outputs could eventually be improved or post-processed in order to obtain the desired results; we decided to further analyse these three algorithms in order to understand their functional principles and identify their strengths and weaknesses. Moreover, we also perform a quick analysis of the three algorithms that scored the lowest in order to identify the principal reasons for their low performance in our testing scenario. The analysis is more technical and allows us to extract important aspects for a suitable JATS XML diff algorithm.

JNDiff

JNDiff is able to fulfil 69% of our expectations and is the only algorithm able to deal with text changes in complex environments—one of the reasons it rates as top-

performing. Complex environments are common in JATS documents, as the majority of the text content contains bold, italics, xref and other styling or reference nodes. The algorithm is also able to make distinctions between text updates and replacements (Delete–Insert), depending on the size of the modification compared to the original text. While editing short text sequences such as the author name, JNDiff considers this action a full replace. The article title being longer, the same text insertion is considered a text update. All tree edits are well or partly detected, more than any of the 11 other algorithms. As a concrete example, JNDiff was able to detect all modifications to article authors: insertions and removals, changes in the correspondences via attribute updates and position/ordering changes. Although scoring higher than others regarding the ability to detect changes in JATS documents, there are several aspects we would like to mention where the algorithm was not able to fulfil our expectations. One of them is the missing ability to represent text Move operations. When an author moves large portions of text, it is considered a complete rewrite using the Delete–Insert edit sequence. Tree moves are presented for the parent but also for all its child nodes, which represents an author Move operation containing three child nodes (firstname, lastname and email) as four Move operations. Although the distinction between text Update and text Replace has its strengths, it also presents a weakness where small changes have to be detected. This can be the case for text corrections where changes have been made on the character level. JNDiff will, in this case, consider those edits complete word rewrites. Tree Merge, Split, Upgrade and Downgrade operations are represented as Delete–Insert sequences. In this way, merging or splitting two paragraphs and upgrading or downgrading sections within a document are not interpreted as we would expect. When on Style edit operations, styling a text is represented as a complete deletion of the existing text followed by a complete insertion of the new text containing the styling node. One last note about JNDiff is the fact that the documentation is written in Italian, which could present further difficulties for future improvements.

XyDiff

XyDiff is one of the fastest XML diff algorithms, able to fulfil 42% of our expectations and to deal with most of the tree edits (Delete, Insert, Attribute Update and Move) with the exception of Split, Merge and Downgrade, which are only partly carried out. Tree downgrade is, for example, represented as four move actions, which makes post-processing difficult. On the other hand, the text edits score is much lower for XyDiff. Text Update precision is fine-grained, which is good for small text changes such as corrections; however, in our test environment, this is more often considered a weakness for larger text changes, as the algorithm calculates the longest common substring (LCS) and tries to minimise the edit distance at the cost of increasing the difficulty of post-processing the delta representations. Comparisons of complex text nodes are not optimal. Minor text changes are represented as very large due to the fact that the change is shown on the entire non-complex part of the text node; changing one letter in such an environment results in a multi-sentence change. Note that XyDiff also considers complex those text nodes containing HTML character representations. In the same way as other algorithms, text moves are represented as complete Delete–Insert rewrites. The same behaviour occurs in tree merges, where paragraph/section merges are not represented as we would expect.

XCC

XCC is able to fulfil 39% of our expectations. It always presents old and new values, which could be very useful for eventual post-treatment of the delta results. XCC is also able to detect text Updates. Tree Insert is represented as one edit action, which is easier to interpret compared to JNDiff. Furthermore, attribute updates are also well represented. There are, however, several important edit actions that are not represented as we would expect. The most important is the detection of all edit operations in complex environments. Changing one character in a large paragraph containing one bold node will result in a large update of the entire text content outside of the bold node. Text Move operations are not detected and are represented

as an Update. XCC is also not able to deal with Style changes; they are represented as two actions—an Insert of the new style element with the following text node, and an Update of the original edited element in order to remove the duplicated text. Tree Move detection is another weakness of the XCC algorithm. This edit action is represented as two Inserts, two Deletes and four Updates. Tree Delete impacts the parent, but also all its child elements, similar to how JNDiff interprets tree Move, making the further processing of this edit action difficult. Tree Upgrade and Downgrade are represented by complete Delete–Insert actions, followed by some Updates.

Low-Performing Algorithms

XOP, DiffXML and FC-XmlDiff (faxma) are the lowest-scoring among the tested algorithms. XOP is not able to represent text-node differences but only tree differences. DiffXML is able to represent text Insert, Delete and Move operations. While performing tree edits, a tree Upgrade operation results in close to 1000 edit operations, mostly composed of Move actions. FC-XmlDiff, also called faxma, presents the difference with a diff-copy XML tag, and there is no information about the edit type.

Analysis conclusion

As seen during the XML diff comparison study, we have identified 12 existing XML diff algorithms (Table 2.1) that we assessed for their suitability in comparing text-centric JATS XML documents that represent academic articles. We started with a performance evaluation using the XMLDiffAnalyzer script we developed, followed by a manual evaluation of the resulting deltas. Table 3.2 shows the key detection capacity differences between the XML diff algorithms we have evaluated. We present those with a success operator (✓) within different edit groups: text edits, tree edits, style edits, level 2 edits (paragraph split, merge, upgrade and downgrade), the execution time below five seconds, a fair number of operations that could represent changes in a human readable way and the last updated date.

The overall execution-time performance was rather acceptable among the algo-

Algorithm	XML Type	Text edits	tree edits	Style edits	Level 2	Time <10sec	Fair nbr. of operations
XOp	data					✓	✓
DiffXML	data					✓	
FC-XmlDiff	data					✓	✓
DiffMK	data					✓	✓
Xdiff	data		✓			✓	✓
Node-delta	data		✓			✓	✓
DeltaXML	data	✓	✓			✓	✓
JXyDiff	data	✓	✓			✓	✓
XCC	text	✓	✓			✓	✓
XMLDiff	data	✓	✓				✓
XyDiff	data	✓	✓			✓	✓
JNDiff	text	✓	✓	✓		✓	✓

Table 3.2: XML diff algorithms analysis recap.

rithms (<10sec) except for XMLDiff implementation that presents large execution times for real-life JATS document comparison and is not suitable for production usage. Regarding the second part of the evaluation where we analysed the delta output produced by each algorithm, the results show that none of the algorithms fully meet our expectations. The main weakness shown by most of the algorithms, except JNDiff, is the inability to represent edit operations in complex environments where styling nodes are embedded into text nodes, which is very common in a text-centric XML document environment. Moreover, the existing edit operations' detection capacity (Insert, Delete, Update and Move) is not sufficient to efficiently represent the differences between two JATS XML versions of an academic article. In order to implement a suitable JATS diff algorithm, it would be necessary to detect, in addition to the existing edits, style changes, citable object reference changes, text moves, tree splits and merges, tree upgrades and downgrades and semantic differences. We can divide those new edit actions into three groups: styling, structural and inducted changes.

- **Styling changes** are observed inside different text nodes representing styles changes applied to text elements, as italic, bold, etc. They have no narrative nor textual structure impact but are heavily impacting the XML structure;
- **Structural changes** are another example where the XML structure is heavily impacted while the text structure stays the same. We can observe node split, merge, upgrade and downgrade there;

- **Inducted changes** are observed when citable object reference nodes are automatically modified when citable object lists such as bibliographies, figures and tables are changed. Those citable object reference nodes are observed inside different text nodes and use the citable object label and ID as a reference. Those auto-incremental values are dependent on the order of the citable object appearance within a list. Each time this order changes, this creates a so-called inducted change on every citable object reference.

All those changes are currently represented as simple delete–insert sequences that have to be further refined by using proper change semantics and convert specific delete–insert sequences into higher level changes that better reflect real author modifications. Once those new change detection mechanisms are added, it would be possible to properly detect, represent and store most of the edit operations presented in Section 3.1.1. Having this in place would be useful for an eventual future versioning system for academic articles.

3.2 New jats-diff algorithm

As mentioned above, XML documents are divided into two types, text- and data-centric. Historically, data-centric XML diff algorithms inherited the existing text comparison capacities and also added so-called "Level 1" or "basic" lexical edit operations [36] on tree elements: insert, delete and attribute change. Recent XML diff algorithms are mostly text-centric and propose specific higher level or "Level 2" syntactic changes that were not found in data-centric XML diff algorithms. The size of individual text nodes is usually larger, but they are smaller in number; thus, it makes sense to think about higher level changes that replace a combination of basic insert and delete operations. Moreover, as humans are usually the main producers of those text-centric XML documents, there is a need to use bijection—see Figure 3-4—in order to detect and represent their modifications in the same way as how they are produced. Among "Level 2" changes, existing algorithms are able to detect and represent tree move and wrap/unwrap (observed in [13]). Wrap and unwrap is used

to detect edit patterns where a specific portion of text within a given node was wrapped/unwrapped by/from another node. Adding/deleting styling nodes around a text portion is represented as wrap/unwrap. In the literature, there is a discussion around an additional number of "Level 2" changes. Some research groups propose element merge and split [36] and also text move among the nodes [13]. Unfortunately, none of the existing algorithms propose a solution on how to detect such changes.

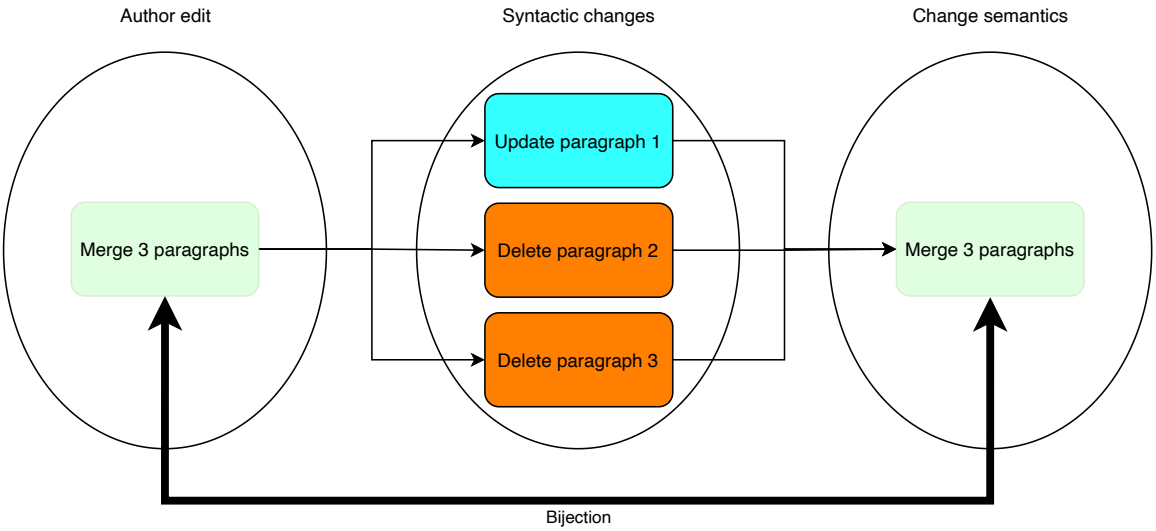


Figure 3-4: Example of bijection used while detecting a paragraph merge.

Three paragraphs were merged together by the author and detected as two delete and one update. Using change semantics, the merge edit is detected in the syntactic change sequence and presented to the change information consumer.

In addition to detecting existing "Level 1" and "Level 2" changes, there is also a need for the detection of additional "Level 2" changes such as paragraph split and merge, section upgrade and downgrade, style edit, text move and citable object edits. For this to happen, we need to apply similar bijection methods as seen in Figure 3-4, where change semantics are used to recognise author edits as a specific sequence of lexical and syntactic changes. Several research groups have started working on XML diff algorithms for semantic change extraction [105, 24, 64, 65]. The main idea shared by those research groups is to track the evolution of an XML document in time by extracting change semantics between elements that do not necessarily share

the same structure and the same identifiers. There are algorithm proposals [105] able to support an XML versioning system where the XML structural changes are ignored and change detection is achieved by first detecting identifiers for elements that are common across the versions and then using these identifiers to associate elements among the versions. The XKeyMatch algorithm proposed in [24] follows the same goal and uses XML keys to match elements that refer to the same entity among the versions. They use this extension to pre-process the structural analysis phase that ordinary XML diff algorithms carry out in order to match similar elements, but with structural changes between the versions. Another algorithm was proposed [64] that identifies syntactic change patterns in order to deduce semantic changes. In their example, a combination of employees with a salary increase and employees with title change could be used to extract employee promotion semantics. Alessandreia Oliveira et al. [65] describe the Phoenix algorithm used for the same goal to match the same elements among different document versions that have their identifiers changed. Phoenix uses similarity metrics for this purpose.

3.2.1 Impact of author edits on XML: a JATS example

The existing text processor track change tool is very efficient for generating a human-readable description of the edits applied by the author on a given digital textual document. Unfortunately, this tool lies in the author's hands and depends on the text processors. On the other hand, the current XML diff algorithms generate delta outputs with a limited number of edit patterns where one author edit action is interpreted as a sequence of different lower-level edit actions (insert/delete), which makes the delta hard to read and understand for humans. In this section, we analyse some common edit actions observed on digital textual documents (academic articles being used as an example) that authors regularly produce and correlate those with their impact on XML .

Academic articles usually follow a standard structure in terms of how they are written regarding the sections they embed and in which order those appear. An article usually starts on the first page with the journal, title, authors, affiliation,

abstract and keywords information. What follows is the largest part with several sections, each of which can contain subsections, paragraphs, citations, figures, tables, maths formulas, etc. At the end, we usually find the acknowledgments and reference list. As the largest part of an article is text blocks known as paragraphs, most of the changes made by authors are mainly observed there. Figure 3-5 shows some common author edit actions and their impact on JATS XML —see Figure 3-6. The current "Level 2" edit actions are unfortunately not able to represent all those edits in a human-readable way. Instead, those are mostly presented with a combination of "Level 1" (insert and delete) edits. In the following text, we use abbreviations I, D, A, U and M for Insert, Delete, Attribute edit, Update and Move representations of XML edits, respectively.

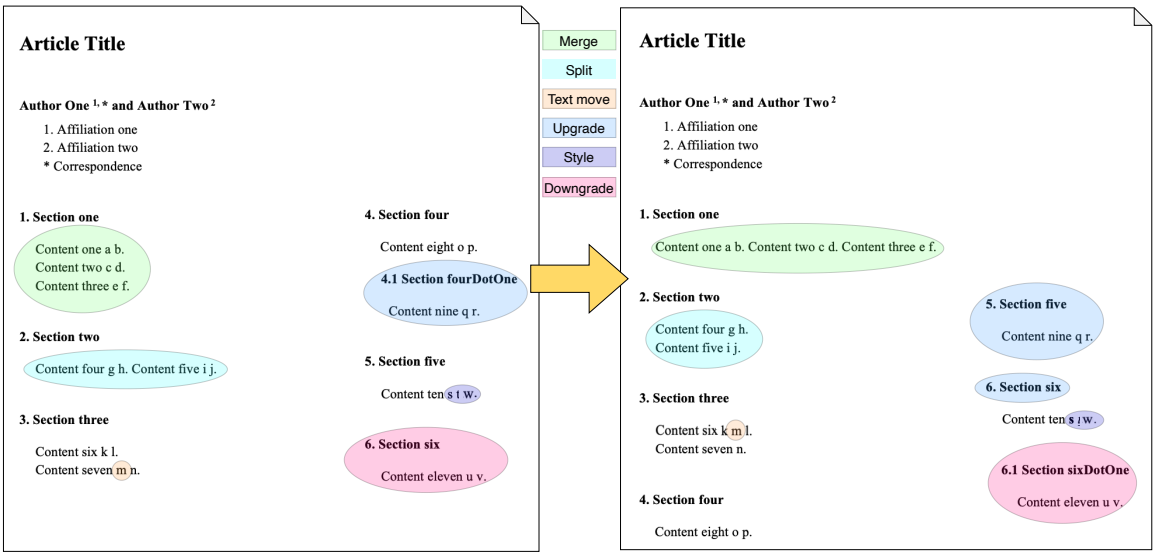


Figure 3-5: Two article versions side by side written in a text processor.

The following author changes are highlighted: Three paragraphs from Section 1 merged into one; one paragraph from Section 2 split into two; text portion moved from the second into the first paragraph in Section 3; Subsection 4.1 upgraded as Section 5, implying the auto-increment of the previous Section 5 to Section 6; styling edits on the paragraph in Section 5; initial Section 6 downgraded as a new Subsection 6.1.

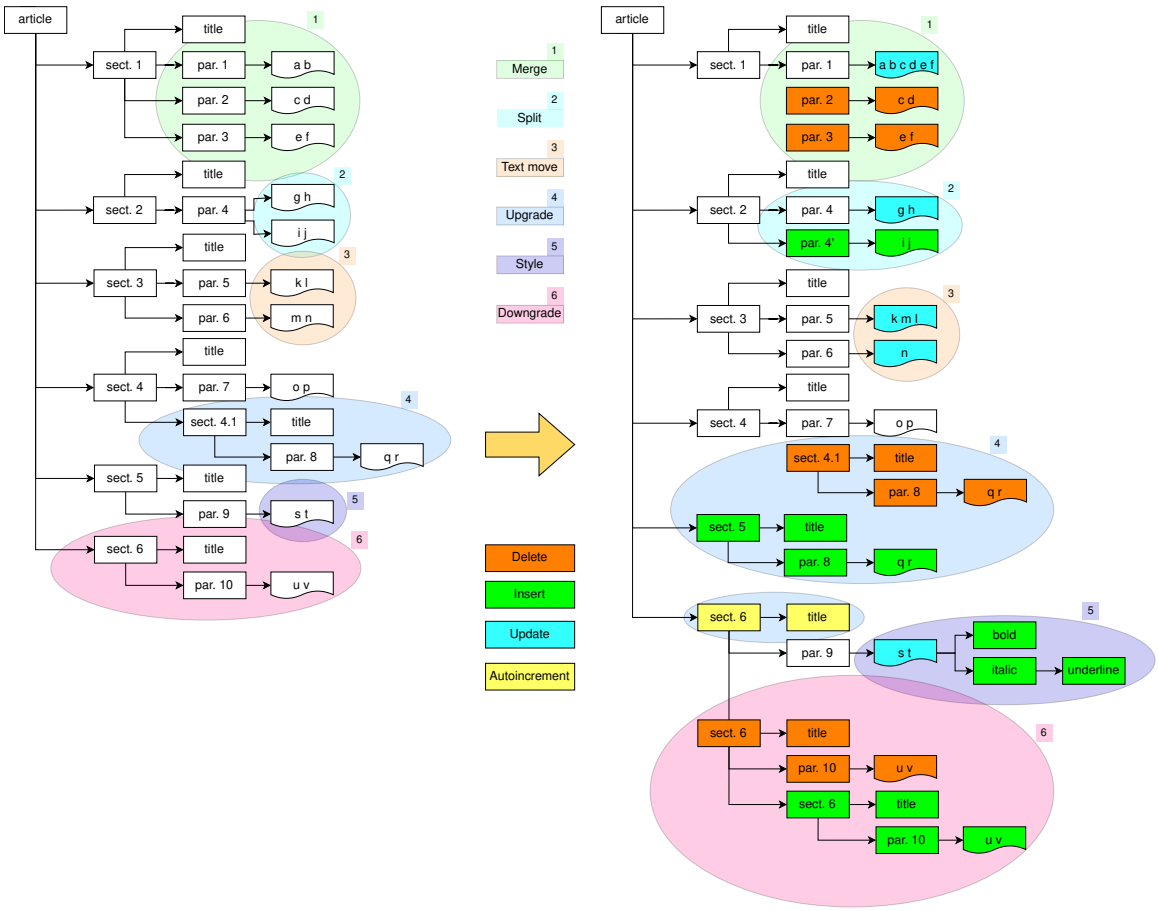


Figure 3-6: Two JATS XML versions side by side.

The following author changes are highlighted: Three paragraphs from sect. 1 merged into one; one paragraph from sect. 2 split into two; text portion moved from the second into the first paragraph in sect. 3; sect. 4.1 upgraded as sect. 5, implying the auto-increment of the previous sect. 5 to sect. 6; styling edits on the paragraph in sect. 5; initial sect. 6 downgraded as a new sect. 6.1.

3.2.2 Paragraph merge and split

Two of the first paragraph edit actions we observed are merge and split. Authors usually split large paragraphs into smaller ones or merge several small paragraphs into a bigger one. This action is relatively easy in text processors and consists of adding or removing line breaks between text blocks.

Figure 3-5 (Section 1) shows how a paragraph merge action is carried out by the author on the text processor and its impact on XML (see Figure 3-6, sect. 1), with a higher complexity than simply removing line breaks. We can observe a combination

of U+(n-1)D, n being the initial number of paragraphs to merge. Looking at the delta output of existing XML diff algorithms, the merge modification in our example will appear as three or more edit actions: U+2D in the best-case scenario or I+3D, while U is seen as a combination of D+I. The split edit action is the opposite of merge as can be seen in Figure 3-5 (Section 2), where content "i j" lands in a new paragraph annotated 4' in Figure 3-6 (sect. 2), representing the split edit impact on JATS. The delta representation for this change is symmetrical and of similar complexity to that of merge.

From the human reader perspective, reading a delta output representing a structural split or merge edit action as a combination of three or four basic edit actions is not convenient and requires some higher-level interpretation.

3.2.3 Text move

Another common edit action we observed is the text move where authors move sentences or part of text from one paragraph to another within the document. Figure 3-5 (sect. 3) shows how part of a text labelled "m" was moved from the second to the first paragraph. The impact of this edit on XML will appear as 2U in the best-case scenario—see Figure 3-6, sect. 3—or as 2(D+I).

Again, from the human reader perspective, representing a text move as two node updates is not self-intuitive and cannot be easily interpreted as a text move.

3.2.4 Subsection upgrade/Section downgrade

Subsection upgrade is yet another edit action we observed and happens when authors upgrade a subsection to a section. Each section/subsection being composed of one label, one title and the section body, upgrading a subsection in the text processor is usually seen as a combination of font increase, indent decrease and label change—see Figure 3-5 (Section 4). As JATS XML does not hold any layout information, the changes observed in the text processor cannot be directly detected, which increases the detection complexity. We can observe in Figure 3-6 (sect. 4.1 -> sect. 5) that

multiple nodes are affected by the change. The entire Section 4.1 is removed and inserted as Section 5. Moreover, an important inducted change is observed in sections following the upgraded subsection as their label, and the ID will automatically be changed according to the numbering plan. In our case, the initial Section 5 will be renumbered as Section 6. Depending on the XML diff algorithm, comparing the two JATS files will result in a delta showing a full D of Section 4.1 and a full I of Section 5, followed by an attribute and label change on the initial Section 5 becoming Section 6. Eventually, the full delete–insert combination is also seen on some XML diff algorithms as a simpler Section 4.1 parent and title node D, followed with the new Section 5 parent and title node I and a M of Section 4.1 child elements in the newly created Section 5, resulting from a combination of 2(D+I)+M.

The downgrade edit action is the opposite of upgrade. Downgrading a section in the text processor is usually seen as a combination of font decrease, indent increase and label change—see Figure 3-5 (Section 6) for author edits in a text processor. Its impact on XML —see Figure 3-6 (sect. 6.1)—is exactly the same as for an upgrade, but in the opposite direction.

Both of the mentioned author edit patterns are not recognised in the existing XML diff algorithms and make the delta difficult to interpret from a human reader perspective.

3.2.5 Style edit

Although JATS XML documents do not hold any layout information, textual styling is still part of it and represents important information in the document. Most of the paragraphs within the document hold text styling information, where we can observe bold, italic, underline, subscript, superscript and others. Styling is purely a visual change in the text processor—see Figure 3-5 (Section 5 -> Section 6) where one part of the text was styled bold and another one italic and underlined. On the other hand, styling information is represented as node elements on XML , and their insertion/removal directly impacts the XML tree—see Figure 3-6 (sect. 5 -> sect. 6). This adds another layer of complexity for the XML diff algorithms as inserting a

bold style around a text portion consists in inserting a new node that will wrap this text content. Most XML diff algorithms will represent this change as a new node I containing the edited text portion and an U of the edited paragraph text node. This kind of delta output is, again. not easy for human readers and is hard to interpret as a style edit action made by the author.

3.2.6 Citable object reference edit

Another common element we can observe in paragraphs are citable object references. Authors usually cite bibliographies, figures, tables and other sections within the article. Those references appear as <xref> nodes in XML and are visible in most of the paragraphs. In order to cite an object, its label and ID can be used. Those auto-incremental values are assigned to each citable object and are dependent on its appearance in the citable objects list. A reference citation is seen as <xref ref-type="bibr" rid="B2-molecules-25-00430">2</xref>, where "B2-molecules-25-00430" represents the ID, and the number 2 represents the label. If the paragraph is changed from citing the bibliography B2-molecules-25-00430 to B3-molecules-25-00430, both the ID and the label will be changed within the <xref> node. Those two properties make it difficult to track author edits on the citable objects list. Let us assume a scenario where there are five references that the author is using as the bibliography. If a new reference were to be added at the position 2, the IDs and labels of all the following references would be incremented, i.e., the "B2-molecules-25-00430" would switch to "B3-molecules-25-00430", and so on. The same applies for their labels. This change would then also impact all the references those objects have within the article, where the <xref> from our previous example would change to <xref ref-type="bibr" rid="B3-molecules-25-00430">3</xref>, and the same for the following references. Having those inducted changes while adding or removing citable objects would result in them being represented in most of the XML diff tools as, instead of a simple citable object I, I+n(A+U) in the best-case scenario, or I+n(D+I) if the A and U are seen as full <xref> D+I, n being the number of auto-incremented bibliographies. This type of simple author edits creates a lot of noise in the delta output that needs to be detected

and eliminated by the XML diff algorithm. Without any filtering, one reference I or D can reproduce hundreds of other edits due to the inducted changes it generates.

3.2.7 XML diff base—JNDiff

The JNDiff algorithm has achieved state-of-the-art performance [19] in comparing text-centric JATS XML documents. Its two-pass logic on detecting "Level 1" edits and the existing logic of analysing and converting sequences of level "Level 1" to "Level 2" edits makes JNDiff a suitable candidate to build our new jats-diff algorithm on top of the existing code. We decided to ground our new edit pattern recognition using the core functionalities of JNDiff—see Figure3-7—in order to avoid rewriting the existing "Level 1" edit actions detection logic that is composed of over 12k lines of code. Besides having a good performance, JNDiff is also highly reliable in respect of detecting differences between two XML documents. The logic of the algorithm is as follows: It first builds one virtual tree object per document; then, it detect inserts and deletes as basic/"Level 1" edit actions; finally, it tries to refine the detected "Level 1" differences and convert them to "Level 2". Each time a specific insert–delete sequence is turned to "Level 2" change, it is removed and replaced in the change list by its "Level 2" representation. Our new edit pattern detection will be added to the JNDiff refinement logic in order to recognise and represent them in the delta.

3.2.8 XML tree annotation

We use the conventional labelled ordered tree model to represent the XML trees in the following edit pattern detection on JATS XML . As we always compare two XML documents A and B—two XML trees—each node belonging to the document A is labelled α and the document B β. We assign to each node an identifier m for document A and p for document B. In addition to the identifier, the node depth regarding the tree represented by n for document A and q for document B is also added, resulting in the annotation $\alpha[m, n]$ for document A and $\beta[p, q]$ for document B.

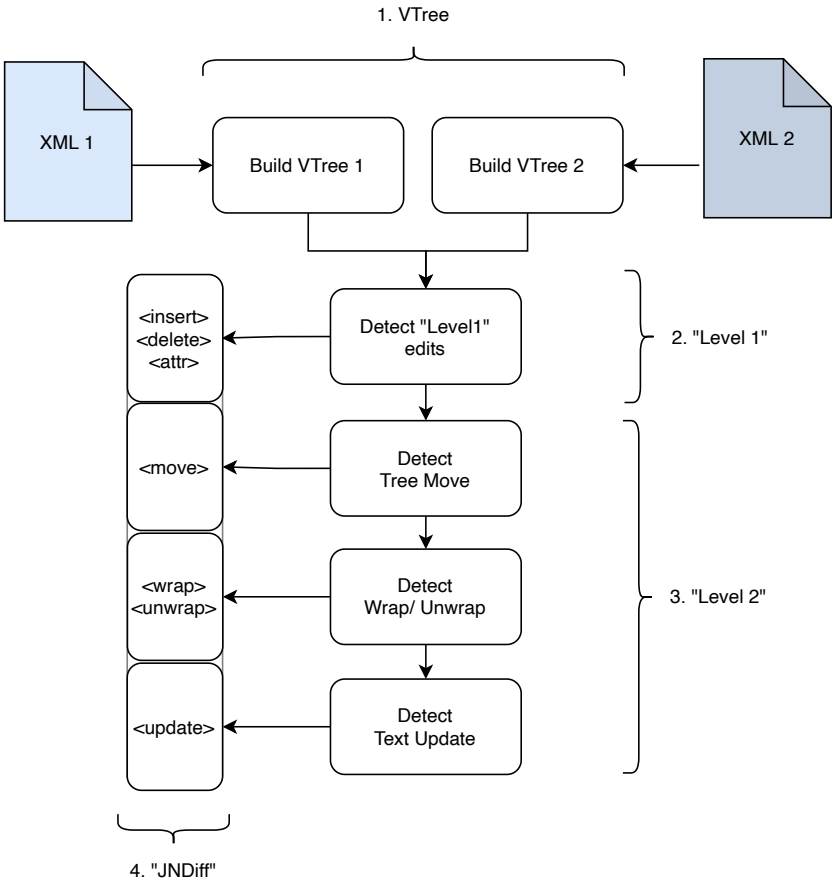


Figure 3-7: JNDiff base workflow.

The JNDiff base workflow is as follows: 1. Build one virtual tree object per XML document; 2. Detect "Level 1" edit actions (insert, deletes and attribute edits); 3. Refine the detected "Level 1" edits and convert them to "Level 2". Each time a specific insert–delete sequence is turned to "Level 2" change, it is removed and replaced in the change list by its "Level 2" representation; 4. The final XML diff output is built.

As we can see in Figure 3-8 representing the XML tree of the document A, each node has a specific annotation. $\alpha[0, 0]$ represents the article node, being the root node. $\alpha[0.0, 1]$ and $\alpha[0.1, 1]$ represent Section 1 and Section 2, respectively, Section 1 being the node number 0.0 at depth 1 and Section 2 the node number 0.1 at depth 1. Section titles and paragraphs are within depth 2 and have node numbers 0.0.0 to 0.1.1. Finally, text nodes are at depth 3 and have node numbers 0.0.1.0 and 0.1.1.0. On the left, we can observe shallow nodes, and on the right, deeper nodes.

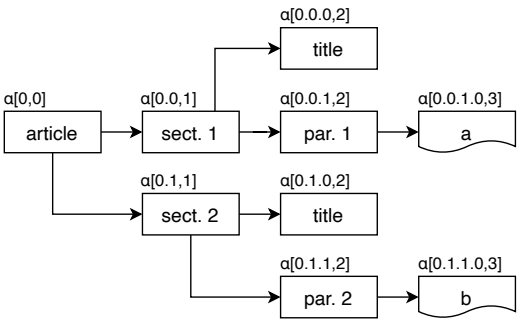


Figure 3-8: Document A XML tree.

Example of XML tree on document A using the conventional labelled ordered tree model $\alpha[m, n]$ and $\beta[p, q]$, where m and p represent the node identifier and n and q the depth.

3.2.9 Text similarity vs. text equality

All existing XML diff algorithms we tested use text equality while trying to match "Level 2" among the "Level 1" changes. Taking the example of structural merge, see Figure 3-9 where the P1 and P2 nodes, representing paragraph 1 and 2, are merged into P3. The merge is detected only if the text in P3 is exactly the same as the sum of the texts in P1 and P2. This behaviour is suitable for data-centric XML documents where high precision is required. On the other hand, text-centric XML documents are prone to small textual edits where grammar and sentence rephrasing are common. It is then important to replace text equality checks with text similarity while evaluating "Level 2" changes. This way, the algorithm is more flexible and can detect "Level 2" changes even with small textual change interference represented in Figure 3-9 by the addition of "but the". All the following "Level 2" change patterns we present use text similarity with a threshold that we experimentally defined at 95%; however, this value can be changed for further fine tuning. Once the "Level 2" change pattern has been detected, and if the text node has a similarity different from 100%, the text updates have to be detected in the usual way regardless of the "Level 2" change.

3.2.10 Structural upgrade / downgrade

As seen in Figure 3-6 (sect. 4 -> sect. 5), upgrading one child node to the same depth as its parent represented by upgrading one subsection with x nodes into a section

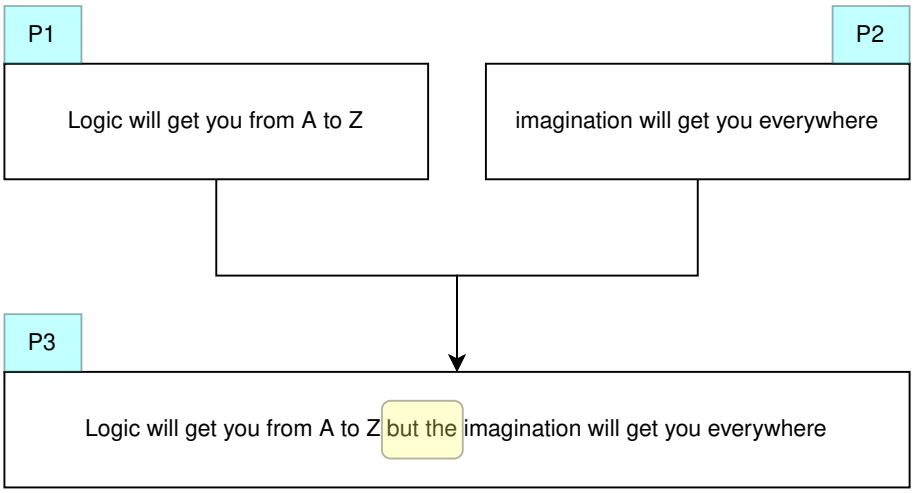


Figure 3-9: Paragraph merge.

Nodes P1 and P2 merged into P3 while a text change common to text-centric XML document was introduced, showing the benefits of text similarity over text equality usage for "Level 2" change detection.

results in x(D+I) "Level 1" edits. Moreover, as the remaining section numbering will also change due to their auto-incremental nature, we observe additional y(A+D+I)) "Level 1" edits, y being the number of remaining sections following the upgrade: A for id; D+I for the title. In total, a simple section upgrade will result in x(D+I) + y(A+D+I) "Level 1" edits. Within our example—see Figure 3-10—the structural changes that consist in upgrading (sec 1.1) node into (sect. 2) result in depth lowering on each of the upgraded nodes followed by attribute id and title change.

By applying the following mathematical formula on the lists of node changes detected between documents A and B, respectively annotated cα and cβ, we can evaluate if the specific change pair fulfils the structural upgrade condition:

$$(\forall c\beta)(\forall c\alpha)\beta[p,q].content \simeq \alpha[m,n].content \wedge q < n \tag{3.1}$$

The formula verifies if the contents of the (sect. 1.1) and (sect. 2) nodes are similar and, in addition, if the depth of (sect. 2) is lower than the depth of (sect. 1.1) node. Having this condition satisfied will result in a structural upgrade detection. By running the formula on our example, the following scenario occurs: for

each change detected in document B ($\beta[p, q]$), evaluate if the text content of $\beta[0.1, 1]$ "title"+"cd" is similar to the text content of $\alpha[0.0.2, 2]$ "title"+"cd". In addition, evaluate if the depth of the modified $\beta[0.1, 1]$ element is lover than the depth of the $\alpha[0.0.2, 2]$ element. As both conditions are satisfied, the structural upgrade pattern is recognised. The delta output of such pattern detection is represented as one "Level 2" change named "upgrade", having two information elements: "upgrade_from" and "upgrade_to".

Structural downgrade is the opposite of upgrade, where it is enough to invert the depth comparison in order to adapt the upgrade formula to detect downgrade patters. Once a structural downgrade pattern has been detected, the delta output will contain one "Level 2" change named "downgrade", having two information elements: "downgrade_from" and "downgrade_to".

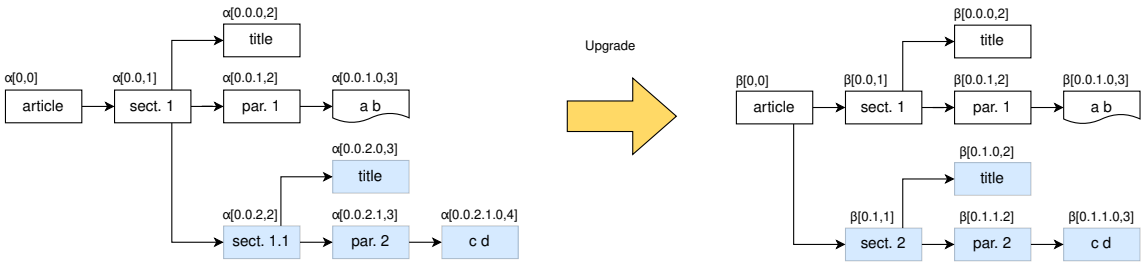


Figure 3-10: Impact of structural upgrade on an XML tree.

We observe that the depth of the upgraded (sect. 1.1) node is lowered by one, becoming, on the depth level, the same as their previous parent (sect. 1).

3.2.11 Structural merge/split

As seen in Figure 3-6 (sect. 1), merging x nodes, represented by paragraphs, into a unique one will be seen as x+1 "Level 1" edits, composed of xD+I. Using the update "Level 2" edit, the number of edits observed is lowered to x, with (x-1)D+U. We propose here a new way of detecting structural merge. Within our example—see Figure 3-11—nodes $\alpha[0.0.2, 2]$ and $\alpha[0.0.3, 2]$ are merged with node $\alpha[0.0.1, 2]$. Represented with "Level 1" changes, this edit pattern detection results in 3D ($\alpha[0.0.1, 2]$

with content "ab", $\alpha[0.0.2, 2]$ with content "cd" and $\alpha[0.0.3, 2]$ with content "ef"), followed by I of $\beta[0.0.1, 2]$ with content "abcdef".

By applying the following mathematical formula on the lists of node changes detected between documents A and B, respectively annotated cα and cβ, we can evaluate if specific node pair fulfils the structural merge condition, the merge being valid only if two or more nodes from cβ fulfil the condition:

$$(\forall c\beta)(\forall c\alpha)\alpha[m, n].content \subset \beta[p, q].content \wedge q = n \tag{3.2}$$

This mathematical formula verifies for every node content on β if there are nodes on α whose content is subset β node content and where both depths on β and α are identical. By running the formula on our example from Figure 3-11, the following scenario occurs: for each change detected in document B $\beta[p, q]$, test if the mathematical condition is verified for a given node in document A ($\alpha[m, n]$); if so, the examined node is a merge candidate. The algorithm will test for a given node in document B $\beta[0.0.1, 2]$ all nodes within the same depth in document A $\alpha[0.0.1, 2]$, $\alpha[0.0.2, 2]$ and $\alpha[0.0.3, 2]$, with their respective text content "ab", "cd" and "ef". As their text contents are all contained in $\beta[0.0.1, 2]$ "abcdef" and they all have the same depth 2, they will be added to the merge candidate pool. At the end, if there is more than one merge candidate in the pool, a "Level 2" tree merge edit is detected. The resulting delta using the structural merge pattern detection while merging n nodes into a unique node will be seen as one "Level 2" edit, containing n-1 merge_from and one merge_to information elements.

Structural split is the opposite of merge, where instead of evaluating if the text content of changed $\alpha[m, n]$ nodes are contained by $\beta[p, q]$, we evaluate if the text content $\beta[p, q]$ nodes are contained by $\alpha[m, n]$. While using structural split edit pattern detection, splitting one into n nodes will be represented as one "Level 2" edit containing one split_from and n split_to information.

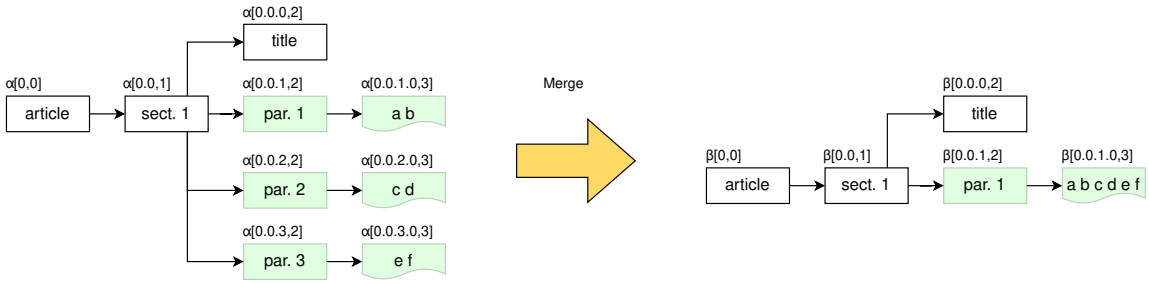


Figure 3-11: Impact of structural merge on XML tree

We observe that the text content "cd" and "ef" from nodes (par. 2) and (par. 3) is merged with the text content "ab" of the (par. 1) node

3.2.12 Inline style edit

As seen in Figure 3-6 (sect. 5) -> (sect. 6), styling information is seen as XML nodes. The most often observed styling elements are for bold, <i> for italic, <sub> for subscript, <sup> for superscript and <u> for underline. Style edits have no narrative impact, and the node textual structure remains the same. On the other hand, the XML structure is heavily impacted by those styling nodes, which makes their change detection complex. Most of the existing XML diff algorithms have difficulties representing text changes in nodes containing styling elements. Having no impact on the narrative structure, one of the possible solutions we propose to deal with inline style edits is to separate styling and textual change detection on XML . This is possible by converting style nodes to simple text using encryption (XML tags to text conversion). This way, the bold "hello" text is encrypted from initially hello to a pure text variant, for example, _|b|_hello_|/b|_. This significantly simplifies the detection of the inline style changes as there is no need to operate with complex tree changes—everything is seen and treated as simple text.

In Figure 3-12, we can see an example where some parts of the (par. 1) node content are styled. Text "a" is made bold, "b" is made italic and underlined and "c" remains unchanged. By analysing the impact of this modification on XML , we can observe that the node $\alpha[0.0.1, 2]$ changes from having one child text node "abc" $\alpha[0.0.1.0, 3]$ to six nodes: $\beta[0.0.1.0, 3]$, $\beta[0.0.1.0.0, 4]$, $\beta[0.0.1.1, 3]$, $\beta[0.0.1.1.0, 4]$, $\beta[0.0.1.1.0.0, 5]$ and $\beta[0.0.1.2, 3]$. By encrypting all those newly added styling elements

to simple text, we retrieve only one text node for (par. 1), which facilitates change detection.

Once we have simple text nodes on both sides, we split them into two lists, $listOfTextA$ and $listOfTextB$, by using the styling encrypted tags as separators. The two lists are then compared using the JAVA DiffUtils[6] library that returns the $diffList$ containing two parameters: difference content and type. With the type having one of three values (insert, delete or change), we are able to find inline style insertions, deletions and updates. In our example, DiffUtils will return three style inserts: bold, italic and underline. Deletions are observed when styling is removed and edits when styling type or styled text portion change. An example can be demonstrated in Albert Einstein's quote "Logic will get you from A to Z; imagination will get you everywhere" that is changed to "Logic will get you from A to Z; imagination will get you everywhere". Note the bold part change from A to Z to from A to Z. DiffUtils will return two differences in this example: the bold part content change with "from" added and the text part changed with "from" deleted. We interpret this change as an inline style edit where the styled portion of text changed.

By using the described approach, jats-diff is able to detect three different inline style changes: "text-style-insert", "text-style-delete" and "text-style-update". The "text-style-update" is used for both style type changes and style content changes.

Using the inline styling "Level 2" pattern recognition in our previous example allows us to change the delta output from D+6I to three text-style-insert. The change consumer can understand this way that there is only inline style and no content changes applied by the author.

3.2.13 Text move

As seen in Figure 3-6 (sect. 3), moving text portions from one node to another will result in four "Level 1" edit actions, 2(D+I). Making text moves within the document will be represented in a similar way to making real content changes, which does not

[6]https://github.com/java-diff-utils/java-diff-utils

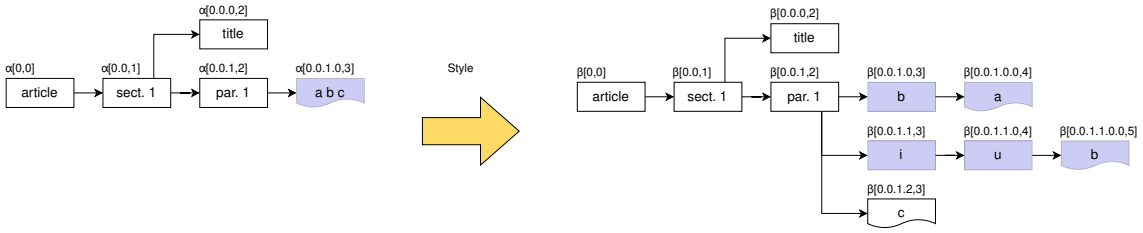


Figure 3-12: Impact of inline style change on XML tree.

We observe that adding bold to one part of the text and italic + underline to another changes the XML tree structure, as each inline style addition is seen as a new element added to the existing XML tree.

represent real modification made by the author. Within our example (see Figure 3-13), text "c" from node $\alpha[0.0.2.0, 3]$ has been moved to node $\alpha[0.0.1.0, 3]$. There, we can observe two change pairs: $\alpha[0.0.1.0, 3]$ - $\beta[0.0.1.0, 3]$ and $\alpha[0.0.2.0, 3]$ - $\beta[0.0.2.0, 3]$.

By applying the following mathematical formula for each of the detected change pairs between documents A and B, respectively annotated cα and cβ, we can evaluate if two specific change pairs fulfil the text move condition:

$$(\forall c\beta)(\forall c\alpha)c\beta[m,n].content - c\alpha[m,n].content \simeq c\alpha'[m,n].content - c\beta'[m,n].content \tag{3.3}$$

The formula evaluates if each of the change pairs differences have common text between them. If true, then the text move pattern is detected. By running the formula on our example, the following scenario occurs: for each change pair between document A and B, $\alpha[m,n]$ and $\beta[m,n]$, evaluate if there is another change pair, $\alpha'[m,n]$ and $\beta'[m,n]$, where the content difference between both change pairs is similar. This results in verifying whether $\beta[0.0.1.0, 3]$.content - $\alpha[0.0.1.0, 3]$.content is similar or equals to $\alpha'[m,n]$.content - $\beta'[m,n]$ content. As both content differences will return "c", the text move condition is satisfied.

The delta output of such pattern detection is represented as one "Level 2" change named "text-move", having two information elements: "text-move_from" and "text-move_to".

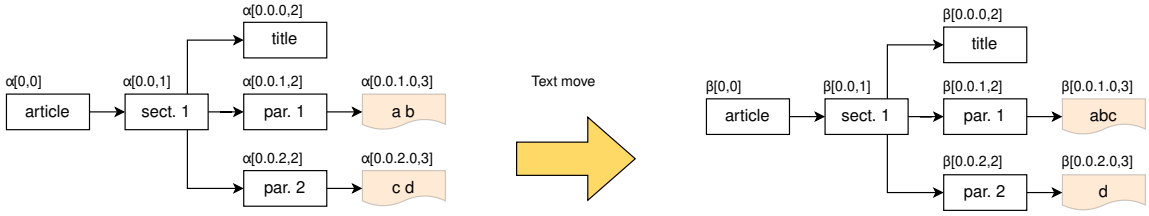


Figure 3-13: Impact of text move on XML.

We observe that moving some text content from the (par. 2) to the (par. 1) node has an impact on both nodes.

3.2.14 Order of the edit detection

The order in which we recognise the previously described edit patterns is important as some "Level 2" edits are composed of a combination of "Level 1" or even other "Level 2" edits. In the example of two structural merge, seen as one node delete ("Level 1") and one node update ("Level 2"), it is important to run the structural merge pattern detection before the node delete and text update detection. Regarding the structural upgrade, this edit pattern is composed of several tree moves and text updates, and it is thus important to run this pattern detection before tree move and text update. Following an empirical evaluation, we decided to choose the following "Level 2" edit pattern detection order:

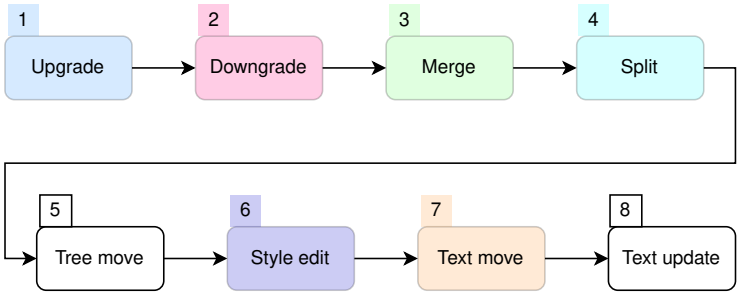


Figure 3-14: Edit pattern detection order.

Higher-level edits are usually composed of other "Level 1" or "Level 2" edits, which makes their detection order important.

3.2.15 Citable node

As seen in Section 3.2.6, in addition to styling nodes, text nodes are also composed of references used to cite other nodes within the document. The most common citable nodes on JATS XML are bibliographies, but we also observe figures, tables and sections. References are inserted as <xref> nodes containing the "ref-type" and "id" as attributes and the citing reference label as text. The "id" and the label are auto-incremental values dependent on the cited node appearance order; thus, inserting or removing a cited node automatically changes the remaining citable nodes' auto-incremental values. Those inducted changes are not interesting for the human reader and should be ignored as they are not directly made by the author.

Figure 3-15 shows the impact of adding one additional bibliography node $\beta[0.1.0, 2]$ at position one in the bibliography list. This change will move the initial position one bibliography node $\alpha[0.1.0, 2]$ to position two $\beta[0.1.1, 2]$, which implies that its label and attribute ID auto-incremental numbering values will change from 1 to 2. This then has a direct impact on the citing xref node $\alpha[0.0.1.1, 3]$ that will change to $\beta[0.0.1.1, 3]$ with a different attribute "ID" and a different label, citing the previous (ref. 1) that became (ref. 2) node. This kind of inducted changes is not interesting for the human reader, for whom the only relevant information is the insertion of the new bibliography. We propose here a solution on how to ignore those non-relevant changes and only keep the relevant changes made by the author. The main idea is to first scan the citable nodes list, detect insertions, deletions and the impact of those edits on their positions within the list. Citable node insertion will auto-increment and deletion will auto-decrement all following citable node IDs and labels, which will then impact all citing references within the text nodes. A precise list containing the original and new citable node numbering values is then used to scan all citing references within the paragraphs and ignore the changes detected where the original numbering value is changed to the new value as an inducted change. This way, only real citing reference changes are kept in the delta output, and the inducted ones are ignored.

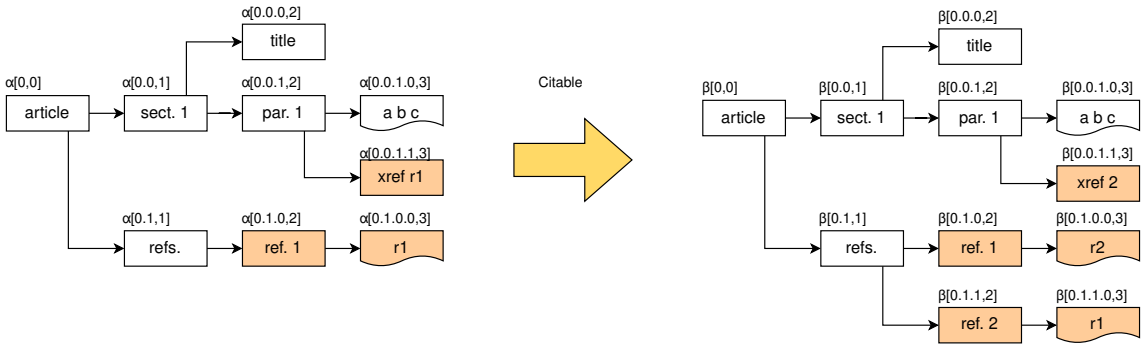


Figure 3-15: Impact of citable node insert on XML tree.

We observe that adding a new citable node (bibliography reference) in the references list will change the auto-incremental values (label and ID the following reference nodes, which directly impacts all citing (xref) nodes within the document.

3.3 Similarity index between the two documents

Text nodes are the most important part of text-centric XML documents. Having a similarity index between the two documents is beneficial for the final decision-maker that can evaluate the impact that the modifications had on the textual content of the article. Due to the XML tree structure, using ordinary text diff algorithms is not possible, which is why we developed a simple and efficient algorithm that can calculate text similarity between modified text nodes and propagate upwards in the XML tree.

3.3.1 Text similarity index propagation

After evaluating different text diff algorithms, we decided to embed the Jaccard index [33] and Term Frequency (TF) [51] that is calculated for every change node pair between document A and document B, regardless of whether the change is of "Level 1" or "Level 2". Once the similarity index is calculated for every change in the delta, those are propagated upwards in the XML tree by applying the following equation, using p as the parent node number, n as the child node number, S as their text similarity index, I as their text content ratio and N as the total number of child nodes for a given parent:

$$ParentSym = Sp * Ip + \sum_{n=0}^{n=N-1} Sn * In$$

A similar result can be achieved without using propagation but is more expensive in calculation power where, instead of completing one similarity calculation per change and propagating it upwards in the XML tree, one similarity calculation has to be carried out per node pair, increasing the number of similarity calculation operations.

Figure 3-16 presents two JATS XML versions where node $\alpha[0.0.0.0, 3]$ lost "b", representing half of its initial content, and node $\alpha[0.0.1.0, 3]$ had 50% textual content changes on "d" that represent 50% of the entire text node content. The delta output will show in this example one text update per modified node $\alpha[0.0.0.0, 3]$ and $\alpha[0.0.1.0, 3]$.

Using our similarity calculation algorithm, we could deduce that the text node $\alpha[0.0.0.0, 3]$ has a similarity of 50% compared to its document B version node $\beta[0.0.1.0, 3]$. Calculating the same for the $\alpha[0.0.1.0, 3]$ and $\beta[0.0.1.0, 3]$ nodes, we can deduce that the two text nodes have a similarity of 75% ("d" representing 50% of the entire text node, and modified by 50%). Once both similarities have been calculated, we can now propagate those upwards on the tree in order to measure the similarity between the two section trees $\alpha[0.0, 1]$ and $\beta[0.0, 1]$, both containing three paragraph nodes each. Here, you can find details on applying the previous formula to the Figure 3-16 example:

- N = 3 as Section 1 has three child nodes;
- n0 represents $\alpha[0.0.0, 2]$; n1 represents $\alpha[0.0.1, 2]$; n2 represents $\alpha[0.0.2, 2]$;
- $S_{n0} = 0.5$; $S_{n1} = 0.75$; $S_{n2} = 1$;
- $I_{n0} = 0.25$; $I_{n1} = 0.25$; $I_{n2} = 0.5$.

$$ParentSym(\alpha[0.0, 1]) = 0.5 * 0.25 + 0.75 * 0.25 + 1 * 0.5 = 0.8125 \sim 81\%$$

The example previously provided in Figure 3-16 is rather simple for comprehension purposes, but the same mathematical formula can be applied to more complex cases where, for example, we can observe text moves, node moves, structural up-

grades, downgrades, etc. as soon as we convert an XML sub-tree to simple text by concatenating their individual text nodes to a single text block.

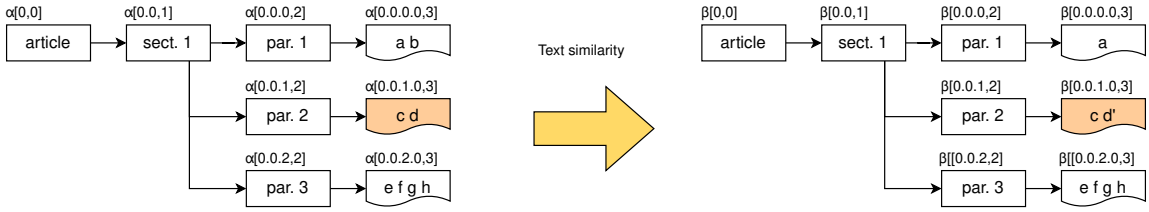


Figure 3-16: Text node modifications between two XML tree.

Paragraphs (par. 1) and (par. 2) represent 25% each of the (sect. 1) content, while (par. 3) represents the remaining 50%. We can observe that text element pairs "ab", "cd" and "efgh", respectively, on text nodes $\alpha[0.0.1.0, 3]$, $\alpha[0.0.2.0, 3]$ and $\alpha[0.0.3.0, 3]$ represent 100% each of the text node content and 12.5% each regarding the (sect. 1) node content.

3.3.2 Element lists and special objects similarity

We previously saw how to calculate text similarity and propagate this similarity upwards on the XML tree. In JATS, having the text similarity makes sense for paragraphs, subsections and sections; it is, however, rather useless for other types of sub-trees that are presented as lists (authors, references, tables and figures). For those, it makes more sense to express the similarity in number of changed/unchanged elements (4/5 authors, or 28/30 references).

Figure 3-17 shows a modification on the authors list where (author 2) was removed. If we use the similarity propagation to calculate the similarity between the "authors" parent nodes $\alpha[0.0, 1]$ and $\beta[0.0, 1]$, we would observe a similarity percentage that is highly influenced by the length of the removed author first and last names. To accentuate this problem, let us assume authors 1 and 3 have very short first and last names, and author 2 has long first and last names. The author two first and last names could, for example, be composed of 20 characters, while the other first and last names are composed of only five characters, meaning that regarding its size, the author 2 first and last names are double the size compared to the two remaining authors together.

We can conclude that text similarity calculation for those special types of JATS XML sub-trees can be inappropriate as this is purely based on text content. In such cases, it is much better to use child element counters and represent their parent element similarity that way. For this concrete example, we would say that authors have a similarity of 2/3, as two authors are exactly the same, and one was modified. In order to have even a higher precision, we propose to use the following semantic information for such lists:

- **Initial**: number of child elements on document A;
- **Final**: number of child elements on document B;
- **Modified**: number of modified child elements;
- **Deleted**: number of deleted child elements;
- **Inserted**: number of inserted child elements.

This way, the human reader can judge the changes applied by the author on such special sub-trees in an easy and convenient way. For additional information, consulting the delta output remains always available.

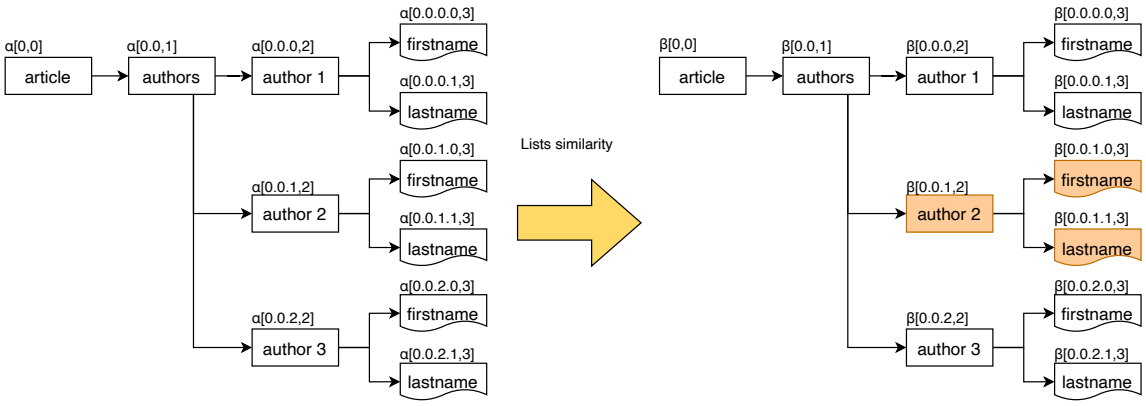


Figure 3-17: Removing an author from the authors list.

The (author 2) node $\alpha[0.0.1, 2]$ was removed during the revision. Measuring the impact of this change on the parent (authors) node $\alpha[0.0, 1]$ using similarity propagation does not reflect well the real changes made by the author. It is much better to use child element counters and represent their parent element similarity that way.

3.4 Algorithm workflow and outputs

While comparing two XML documents, jats-diff generates two distinct files: one XML file containing the delta output and a second text file containing the similarity index and refining the delta output using different change semantics proper to JATS documents: citable objects, special objects (math formulas and figures) and lists containing tables, references and authors. Figure 3-18 shows the algorithm workflow. In the first step, each of the JATS documents are converted into virtual XML tree object. The algorithm will then compare the two Vtree files and identify the differences represented as "Level 1" Insert–Delete and Attribute edit actions. Once those differences have been detected, jats-diff will try to convert them to higher "Level 2" edit actions such as paragraph split/merge, section upgrade/downgrade, paragraph move, etc. For this, the mathematical formulas we described in Section 3.2 are used. Different classes of the algorithm check the list of Insert–Delete sequences and try to identify patterns that satisfy specific mathematical conditions. Each time a condition is satisfied, that given Insert–Delete sequence is removed from the change list and converted to its higher level representation. Once the delta output is built, the algorithm will parse it and use JATS specific change semantics in order to build a human-readable tree structure file representing the change summary. Moreover, the similarity index is also calculated between different parts of the JATS document and propagated through the tree.

We will see in the next two subsections how the delta and the semantics output files are built and what information the end-user can retrieve there.

Delta output

The delta output generated by jats-diff is an XML document that represents both "Level 1" and "Level 2" edits. We observe there 11 possible edit actions: Delete, Insert, Update Attribute, Upgrade, Downgrade, Merge, Split, Move, Text Style edits, Text Move and Text Update. In order to understand the delta XML structure, we first describe within Table 3.3 the attributes used for additional information on the

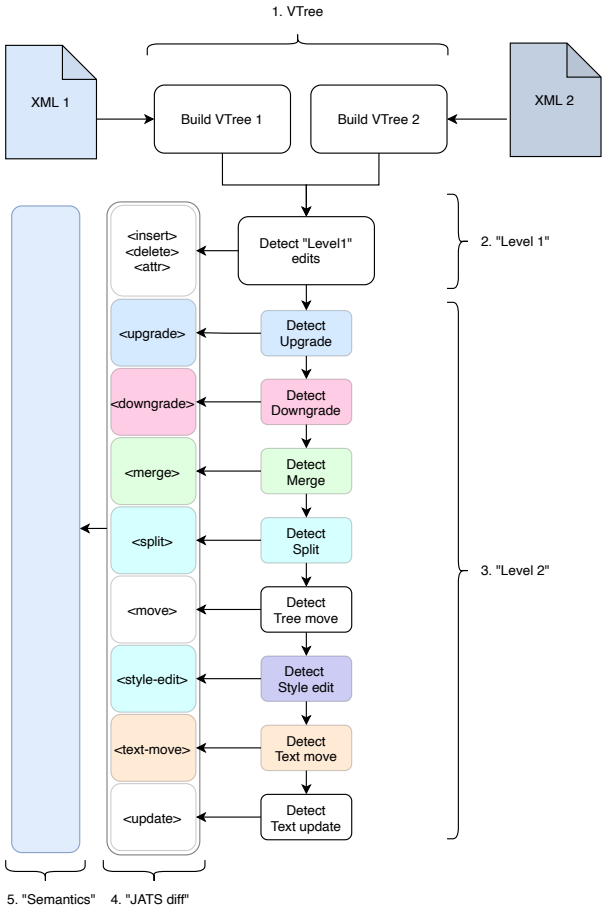


Figure 3-18: jats-diff algorithm workflow.

1. Build one virtual tree object per XML document; 2. Detect "Level 1" edit actions; 3. Refine and convert "Level 1" edits to "Level 2"; 4. Build the JATS diff output in XML format; 5. Analyse the JATS diff output and use JATS-specific change semantics in order to build a human readable tree structure and calculate the similarity index.

specific edit action. Those attributes allow the reader to identify the exact position of the change inside the original document A or the modified document B.

We also provide the following short version of the Document Type Definition (DTD) representing the possible edit actions represented in the jats-diff output:

```
<!DOCTYPE jats-diff [
    <!ELEMENT jats-diff (  delete|insert|update-attribute|upgrade|downgrade|merge|
                split|move|text-style-insert|text-style-delete|
                text-style-update|text-move|text-update)*>
```

The examples of the delta output per common edit action can be found within the Supplementary Materials A.

Attribute	Description
at	The parent node number where the child node was edited
nodenumberA	Node number in the document A
nodenumberB	Node number in the document B
direction	Parameters <from>:<to> used to represent the direction while moving, splitting and merging node
op	For edits composed of multiple XML operation tags, specifies the type of each operation
pos	For text edits, specifies the position within a given text node where the edit takes place
length	For text update, specifies the length of the modified text

Table 3.3: Delta output edit action attributes.

Semantics output XML

Once the initial "Level 1" and "Level 2" edits have been identified, jats-diff will refine those and use change semantics proper to JATS in order to improve the visual representation of the detected changes and avoid representing the so-called induced changes we saw in Section 3.2.1. Moreover, the similarity index will also be calculated and the results will be presented in a form of an XML tree structure. Some real-life examples where change semantics are used in order to refine a long list of "Level 1" and "Level 2" induced edits and summarise those in a human readable format and also the similarity index benefits can be found in the Supplementary Materials B.

jats-diff allows calculating different similarity indexes: similartext, similartext-word, Jaccard and Term Frequency–Inverse Document Frequency (TFIDF). By default, the similartext-word index is used that allows the reader to obtain insights about lexical changes.

As the new edit actions detection and the similarity index have been described, we compare in the next section the new jats-diff algorithm with the other text-centric state-of-the-art XML diff algorithms.

3.5 Performance analyses

The performance analyses of jats-diff[7] are divided into two subsections, one on the information extraction capacity and the other on execution performance. This being a state-of-the-art algorithm, our main effort was focused on the capacity to detect

[7] github.com/milos-cuculovic/jats-diff

new edit patterns and change semantics extraction, rather than its implementation performance.

3.5.1 Information extraction capacity

The initial evaluation phase consists in comparing the "Level 1" and "Level 2" information extraction capacities of the new jats-diff algorithm with JNDiff, XyDiff and XCC. During this performance analysis, we created different XML file pairs, one original and one modified version of the same document. The modified version is composed of one of the human edits that is described in the "Human edit description" column. The output of each of the compared algorithms is then verified for its ability to detect the given edit type.

As we can observe in Table 3.4, jats-diff is able to detect all of the "Level 1" and "Level 2" edits. In addition, there is a similarity index calculated and propagated upwards of the XML for each change detected. This is followed by JNDiff with a perfect score for "Level 1" and a low score for "Level 2" edits. JNDiff can also detect "wrap" and "unwrap" edit patterns that are similar to style edits. This is followed by XyDiff with similar results in addition to text insert detection, where XyDiff mostly uses text updates to represent text inserts. This is because XyDiff calculates the LCS in order to minimise the edit distance between two strings, which increases the complexity for a human reader to interpret the results. XCC follows XyDiff but with additional issues in detecting tree delete and tree move edits compared to JNDiff.

Concerning the delta output, Table 3.4 also shows that jats-diff uses the minimal number of edit actions for almost all edit pattern detection. For a few of them where JNDiff, XyDiff or XCC output a lower number of edit actions, they are usually represented as a simple delete–insert combination which does not reflect the real changes made by humans at all, which we observe in the next section where we evaluate the delta file size for each of the jats-diff. If we push the theory of minimising the number of edit actions, one could think of using the delete–insert combination on the complete document, which will minimise the number of edit actions but maximise the delta file size.

			Success				Nb. of actions			
Lev.	Edit Type	Human Edits	jats-diff	JNDiff	XyDiff	XCC	jats-diff	JNDiff	XyDiff	XCC
1	Text del.	Del. title part	✓	✓	✓	✓	1	1	1	1
1	Text ins.	Ins. title part	✓	✓		✓	1	1	2	1
1	tree del.	Del. author	✓	✓	✓		1	1	1	5
1	tree ins.	Ins. author	✓	✓	✓	✓	1	1	1	1
1	tree attr.	Corr. author	✓	✓	✓	✓	1	1	1	1
2	Text upd.	Update title	✓	✓	✓	✓	2	2	1	1
2	tree move	Move author	✓	✓	✓		2	2	2	8
2	Style ins.	Ins. bold	✓				1	3	4	2
2	Style del.	Del. bold	✓				1	3	4	2
2	Style type	Bold->italic	✓				2	2	4	2
2	Style cont.	Extend bold	✓				2	2	3	2
2	Upgrade	sec2.3->sec6	✓				2	6	4	2
2	Downgrade	sec5->sec2.4	✓				2	20	4	2
2	Split	Split one p	✓				3	2	3	2
2	Merge	Merge two p	✓				3	2	3	2
2	Text move	Move text	✓				2	4	4	2
2	Citable	Del. reference	✓				1	8	9	17
	Sim. index	Real-life edits	✓							

Table 3.4: Level 1/2 edit detection and similarity index calculation capacities for jats-diff, JNDiff, XYDiff and XCC.

3.5.2 Execution performance

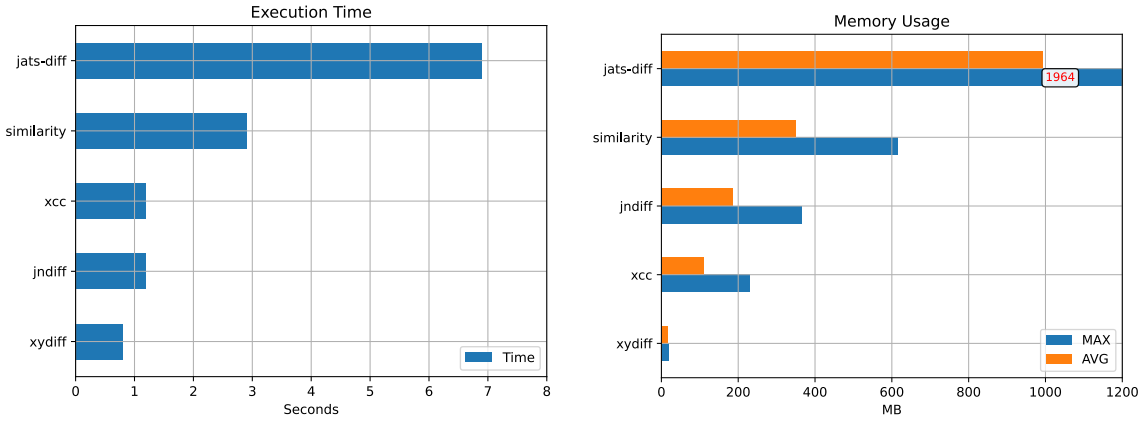
Although not critical in our working environment, algorithm performance for both execution time and memory usage stays important. Compared to other XML diff algorithms that are made with the purpose of comparing hundreds of thousands of XML documents (for example, XyDiff and webpage difference extraction), jats-diff will have to compare academic articles in JATS XML format during the publication process. The current large academic publishers publish from several hundreds to several thousands articles per day, which is far from the number of, say, the web-page changes tackled by XyDiff.

Using the JNDiff core functions for "Level 1" change detection, the execution time and memory usage of jats-diff is minimum at the level of JNDiff. Adding new "Level 2" pattern detection requires additional analysis of the detected "Level 1" differences; thus, it requires more time and memory for the algorithm to be executed. The similarity calculation and propagation are carried out separately by analysing the delta output and require additional execution time.

We divide jats-diff into two parts: first, "Level 1" and "Level 2" pattern detection,

and second, similarity calculation and propagation. JATS articles are large XML files that may vary from 100 KB to 400 KB. The tests[8] were run on a JATS document pair A and B representing real-life author changes during a revision round, affecting every aspect of the article: the title, authors, affiliations, paragraphs, figures, tables and references.

Figure 3-19 shows both the execution time (3-19a) and the maximum and average memory used (3-19b) during a comparison of two real-life author changes in JATS documents. The average was calculated on five comparison round executions per algorithm. As expected, both parts of the new tools take more time and memory to perform the diff and semantics extraction. If we compare jats-diff to JNDiff, the top-scoring algorithm from our previous comparison study, JNDiff is five times faster that jats-diff while using five times less average memory. However, those are acceptable within our environment as the information consumers are humans with the aim to compare the original and revised version of academic articles, which does not need to be carried out in real time while the authors submit their revised version.



(a) Execution time.

(b) Maximum and average memory used

Figure 3-19: Execution time and memory usage.
Comparison of two real-life author changes in JATS documents.

Once the execution time and memory usage have been assessed, we will also compare jats-diff to the other top-scoring algorithms regarding their delta file sizes.

[8]The evaluation was carried out on an Apple MacBook Pro (16-inch, 2019); Processor: 2.4 GHz 8-Core Intel Core i9; Memory: 32 GB 2667 MHz DDR4; SSD

3.6 Discussion

The previous section illustrates how new "Level 2" edit patterns can be detected and how important it is for the human reader to have a bijection of the changes made by the author and the modifications detected by analysing the two XML documents. In addition, the similarity index is also useful in order to obtain a broader picture of the impact that the changes made by the author have had.

The current algorithm still has possibilities for further improvements, one of which is when the document is heavily rewritten by the author in regard to sentence rephrasing and grammar changes. Taking JATS XML as an example, this can be requested by the reviewers in order to improve the article writing and organisation while maintaining the sentence meanings. In order to cope with this, additional text change similarity indexes could be included in the algorithm that will allow the reader to distinguish between simple sentence rephrasing and sentence meaning. Those indexes could be calculated using Topic model [25], word2vec [57, 34] and BERT [23]. This way, distinguishing between simple sentence rephrasing and sentence meaning changes would be possible.

On another note, the current output of jats-diff still needs to be improved; information is visualised as XML delta outputs for "Level 2" and tree structure text output for the similarity index. Having a better and more understandable representation of this information could help the human reader even more. An idea would be to convert the XML document pair to HTML in order to have a readable representation of the document, similarly to versioning control systems (Git/Subversion, etc.). On top of this, we could use our change pattern detection and similarity index in order to visually annotate the changes made by the author, and also the textual impact those changes had on specific key elements of the document—for JATS, as an example, on specific sections.

Regarding execution time and memory use efficiency, our algorithm is less efficient than the state-of-the-art algorithms available. The main reason for this is the fact that we mostly focused on new edit pattern detection being the main algorithm efficiency

indicator, rather than the execution time and memory use that we planned to improve during the industrialisation phase. "Level 2" changes are composed of a combination of "Level 1" basic edit operations. Each time a "Level 1" edit operation is observed, it is analysed as per "Level 2" changes until the right sequence is found, or after going through all "Level 2" types and not finding a match. For each of those "Level 2" edit patterns, there is a specific independent part of code that needs to be executed, which increases time and memory usage. By sharing information between the "Level 2" detection steps, a possibility could be to share different indexes, through which we could avoid constant analyse of the XML tree and reduce the execution time and improve the memory use efficiency.

As another practical use case of the presented jats-diff algorithm, we plan to further use it in assisting the final decision-maker by correlating the actual changes provided by jats-diff and the expected changes extracted from the review comments. In Figure 1-1, we can see that jats-diff helps with comparing different article versions; however, the final decision-maker should still read the review comments and match them with the changes made by the authors. Having a bijection of the modifications made by the authors and the changes detected by comparing the JATS versions and, in addition, calculating the similarity index could be used by the final decision-maker to understand the actual changes made by the author. In order to correlate those with the review comments, Named Entity Recognition (NER) could be used in order to extract information from the review comments on how, where and what should be changed in the article, representing expected changes. Once both pieces of information are available, we could correlate them and provide valuable insight to the final decision-maker that will assess if the author made the changes requested by the peer reviewer. Potential candidates for the named entities regarding review comments could be *Location*, *Action* and *Content*. *Location* could be matched with the change location within the article, *Action* with the modification pattern and *Content* with the change semantics. Additional author change information could also be used in order to further improve the software, one example being the author response letter that is usually uploaded together with the revised version of the article where the

authors claim their changes and modifications as an answer to the review comments. This information could be used to validate and enrich the modifications detected by the algorithm.

One last idea about a similar topic regarding document comparison semantics and their benefits would be its usage in version control systems (Git, Subversion, etc.) If those version control systems were to be aware of the programming language grammar they are detecting changes for, using change semantics to interpret some edit actions could be beneficial: for example, one class name change and its impact on other files where this initial class name is called could replace dozens or hundreds of detected modifications with one single edit which initiated all the other changes.

3.7 Conclusion

In this chapter, we assessed the current text-centric state-of-the-art XML diff algorithms and their capacity to detect higher-level changes made by authors on XML documents, using JATS as a text-centric XML document type for testing purposes. Those algorithms all support "Level 1" edit pattern detection; however, their capacity to detect higher "Level 2" edit patterns is very limited.

We proposed a new XML diff algorithm called jats-diff, based on the existing JNDiff core functions, able to detect "Level 2" edit patterns which are closely related to text document edits made by the authors regardless of their typesetter tools. This allows us to have a bijection of the author modifications and changes detected between two text-centric XML documents. In order to assess the need for the new "Level 2" edit pattern recognition, we started by evaluating different edit actions authors take during the document revision. We then assessed the impact that those edits have on XML and realised that there is a need for new edit pattern recognition: structural upgrade, downgrade, split and merge, inline style edit, text move and citable node edit. Afterwards, we proposed solutions on how to use change semantics on different combinations of existing "Level 1" and "Level 2" edit actions made by authors and how to recognise the new "Level 2" edit patterns. We also proposed a

way to calculate the XML node text similarity index and propagate it through the XML tree. Finally, we conducted a performance analysis comparing jats-diff with three other state-of-the-art XML diff algorithms: JNdiff, XyDiff and XCC. First, we evaluated the "Level 2" edit capacities where we could clearly observe that jats-diff is able to detect and represent all existing and new edit patterns described within this article. Afterwards, we evaluated the execution performance, where we measured the impact of the new "Level 2" edit detection and text similarity index computation on the time and memory used to compare two real-life author change documents.

Compared to existing XML diff algorithms that represent differences between two documents with a limited set of edit pattern recognition, jats-diff proposes an extended set of author modifications and changes detected by comparing the two XML versions. Among different use case scenarios, one of them is to help the Editor-in-Chief in the final decision-making process by automating the manual comparison of different article versions. This is achieved by offering enough details to the final decision-maker to assess whether author changes follow the reviewer requirements. The similarity index computed on different parts of the document also provides a clearer picture to the final decision-maker in order to understand which parts of the articles are the most impacted by the change.

As for the future of jats-diff, there is still work to be carried out on better visualisation, execution and memory use performance, and additional information that can be added in order to enrich change detection.

Chapter 4

Named Entity Recognition

As seen in Figure 1-2, in order to strengthen the academic publishing process, the automation of the three following parts is needed: the extraction of actual changes (1); the extraction of requested changes (2); and the correlation between the requested and actual changes (3). In the previous chapter, we were able to automate the first part of the process while comparing different article versions using jats-diff algorithm. This offers the possibility to understand the actual changes made by the author during the revision rounds. The second part of the thesis will consist in working on the requested changes made by the reviewers during the peer review process. In order to provide such a tool, we have taken the NER approach with the purpose of annotating the review comments and extracting meaningful information that will give us the possibility to further correlate the requested and the actual changes. For this, we have evaluated different deep learning models: BERT, SciBERT, DistilBERT, RoBERTa and XLNet. We have first started with a coarse-grained evaluation by applying the grid-search technique varying the learning rate, the weight decay and the train batch size. This approach gave us the possibility to identify different hyperparameter combination clusters each model performs the best. We have then continued with a fine-grained evaluation where, depending on the coarse-grained results, different fine-grained hyperparameters were used in order to further fine-tune the models. The top scoring model based on SciBERT was trained, achieving an weighted average F1 score

of 0.87. This new model, called "review-annotation", was published on Huggingface[1] and is able to annotate review comments extracting four meaningful classes.

This chapter is divided in the following way: In Section 4.1, we explain the use of five classes that are common to each review comment and could offer us the possibility to make a correlation between the requested and the actual changes. In Section 4.2, we present the dataset preparation used for further NER neural network model training, validating and testing. In Section 4.3, we carry out a coarse-grained evaluation using the grid-search technique of the commonly used NER models, BERT, SciBERT, DistilBERT, RoBERTa and XLNet. In Section 4.4, we carry out a more fine-grained evaluation of the same models with a a final long epochs train selecting the highest-scoring model on our NER task. Finally, in Section 4.7, we discuss additional steps and tools needed in order to further facilitate the final decision-making process for the Editor-in-Chief.

4.1 Review comments

While reviewing an article, peer reviewers write comments about their overall impression of the quality but also give instructions on how to make improvements in order to be eventually accepted for publication. The authors then have to read the comments and assess them by completing the requested changes. For this to happen, the author needs to know the three following pieces of information in order to understand what should be modified and how: the location within the article where the modification should take place; the modification type that explains how the specific content at the given location should be modified; and the modification modality that gives information regarding whether the change is mandatory or optional.

Each peer reviewer has their own writing style and uses their own wording while writing their comments; however, we have identified five named entities that are common among all reviewers and could give us information that can help to understand

[1] https://huggingface.co/MilosCuculovic/review-annotation

the requested changes. Those named entities will also offer us the possibility to link them with the actual changes made by the authors and present this unique piece of information to the final decision-maker. Those five named entities are:

- **Location**: location/area in the article where the requested change has to happen. We distinguish here three sub-types of location:
  - Precise: usually represented by line numbers, figure and table names, quoted text, etc. Matching the review comment directly with a specific portion of text within the article is straightforward when using a precise *Location*;
  - Semi-Precise: usually represented by section names or a combination of section and paragraph numbers where the comment is not related to a specific portion of text but to a broader location;
  - Imprecise: we mainly observe this type of location in generic comments such as the level of English, some generic errors or other types of change requests that have to be applied to the entire article.
- **Action**: verbs or adjectives describing the type of change that should be realised. While using verbs, we usually observe "correct", "define", "specify", "add", "remove", etc. Besides verbs, we also observe adjectives that can specify the requested change type: "redundant", "wrong", "too long", "unclear", etc. A list of 37 actions was initially created with the goal to expand it during the analysis of additional new review comments;
- **Modal**: modals describing the modality of the action. In terms of use, we can divide the change requests into two categories: mandatory and optional. For mandatory changes, the modals that we usually observe are "must" and "should". For optional or nice-to-have changes, we usually observe "would", "might" and "could".
- **Trigger**: short words that are important for the overall comment understanding. Triggers allow us to identify if a comment is a question—e.g., "why", "what" and which"—but also to distinguish whether the reviewer is proposing more than one possible action in order to answer to their request by using

"either", "or", etc. We can also find other, different triggers such as "however", "rather" and "instead of" that give additional granularity to the action we described earlier;

- **Content**: information on what the change is about. The content is the largest part of the comment and is domain-specific. At the opposite of the action named entity that defines the type of change that is requested, the content defines what the change is about. Within the easiest examples, the content named entity represents nouns, but the NER for the content named entity is usually of higher complexity.

Figure 4-1 shows an example of the NER applied on a review comment where we identified the five classes *Location*, *Action*, *Modal*, *Trigger* and *Content*. In the first comment, we can interpret the review comment as: the author "should" - *Modal*, in "Section 1" - *Location*, "correct" - *Action* the "information" - *Content* about "ABCD data" - *Location* by "either" - *Trigger* "show" - *Action* "the data" - *Content* "or" - *Trigger* "remove" - *Action* "their description" - *Content*.

Figure 4-1: NER on review comments.

A review report where important named entities are highlighted using the five classes: *Location*, *Action*, *Modal*, *Trigger* and *Content*.

Matching the mentioned named entities on the review report is beneficial to the reviewer, the author and the final decision-maker. The reviewer can use the NER tool in order to validate their comments during or at the end of the writing. Comments that do not have specific entities matched, for example, *Location*, might not be clear enough to the author and could benefit from being further clarified. The author can use those highlighted named entities in order to better understand the review comment and further fulfil the reviewer requests. The decision-maker can on their end use the tool in order to assess if the requested changes are fulfilled by the author in an easier and a more automated way.

4.2 Training, valid and testing datasets

In order to evaluate different neural network models for our NER task, we will need train, valid and testing datasets composed of annotated review comments using the five named entities we defined in Section 4.1. Those models will be trained in a supervised learning. For this purpose, we exported a corpus of 7000 review comments, where one comment contains one or more change instructions.

4.2.1 Dataset annotation

We first pre-annotated the entire training set using the handcrafted rules approach, where we run different regular expressions[2] in order to identify the *Location*, *Action*, *Modal* and *Trigger* named entities. Once the training corpus was pre-annotated, we built a team of editors that made a manual pass with the purpose to verify and improve the training corpus annotation. The open-source NER annotation tool called Doccano[3] was used, where each of the editors had a specific number of review comments to annotate.

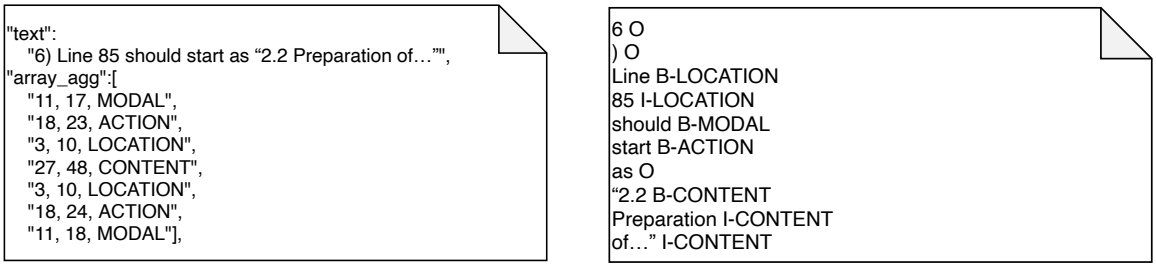
Once the manual labeling work has been carried out, we divided the dataset into three sets: one for training, one for validating the hyperparameters' fine-tuning and

[2]https://en.wikipedia.org/wiki/Regular_expression
[3]https://github.com/doccano/doccano

another for testing purposes. The training dataset is composed of 5000 and the validation and testing sets of 1000 review comments, representing a best practices dataset splitting ratio of 80–20 percent.

4.2.2 Dataset conversion to CONLL format

Once the three datasets become available, they have to be converted from the Docanno json format—See Figure 4-2a—to a format that can be understood by the PyTorch library we are using for the models training. We decided to opt for the CONLL format—See Figure 4-2b that is standard for NER datasets.



(a) Docanno json

(b) CONLL txt

Figure 4-2: Docanno json vs. CONLL txt formats.

As we can observe in Figure 4-2b, while converting the Docanno dataset format to CONLL, each word of a given sentence goes to a new line, and while multiple words are part of the same named entity, those are marked with B-<NE> and I<NE>. If we take the example of "Line 85" as being marked as the *Location* named entity on the json document, this is now converted into two separate words, but marked as B-LOCATION and I-LOCATION. The B-<NE> represents the beginning of a named entity right after another named entity, and the I<NE> represents then rest of the multi-word named entity. The remaining text that is not identified as belonging to a named entity will be marked as belonging to the O (other) named entity.

4.2.3 Dataset cleanup

While starting with the experiments, we noticed that part of the dataset was lowering the model performance due to missing important named entities. In most review comments, the first paragraph is about the global overview of the article mentioning its title, as well as (eventually) the authors and a short resume. This specific part is not about a requested change which we try to annotate. As over 90% of the remaining dataset will always have at least the *Action* and the *Location* named entities and in order to cope with this situation, we decided to exclude examples where the *Action* and the *Location* named entities are missing. In addition, some reviewers group several change requests within an ordered/unordered list by first mentioning the location, followed by a colon and then listing all requested changes for that specific location—see Figure 4-3.

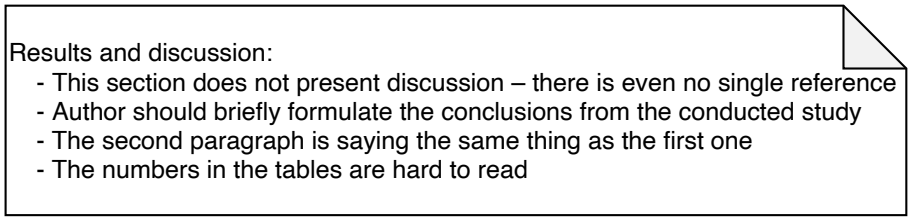


Figure 4-3: Single location with a list of actions.

Once the datasets are cleaned and available within the right format, we will move to the next step which consists of training, testing and evaluating the scores of different NER models.

4.2.4 Imbalanced dataset

The NER task we are striving to achieve can be categorised as a multi-class classification task where a neural network model has to classify each word of our dataset into one of the possible classes. For training and testing purposes, it is important to have a balanced dataset, where each of the classes is represented by a similar number of examples. Once our datasets were prepared and cleaned up, we measured the number of words representing each class within the test dataset (1000 examples):

- *Content*: 3600
- *Location*: 2112
- *Action*: 1491
- *Trigger*: 1935
- *Modal*: 813

As we can observe, our training dataset is rather imbalanced due to the fact that the classes are not represented equally within the dataset. If we compare the *Content* and the *Modal* classes—the two extremes—the number of *Content* support examples (the number of actual occurrences of the class in the training dataset) is four times larger compared to the number of *Modal* support examples. This is not extreme compared to datasets with severe imbalance such as 1:10, 1:100 or even 1:000, but it is worth noting and evaluating its impact on the training and the evaluation of our models.

As seen in Section 2.2.2, there are different techniques to cope with an imbalanced dataset. The first one is that additional data annotation was not possible in our case due to the fact that additional examples would come with a similar imbalance to the one already observed. The data-sampling techniques are also not applicable within our dataset as there are no examples where the under-represented classes are available in a solitary form. If we oversample the examples where the *Modal* class is represented, we will also increase the support for other classes, at least for the *Action* and the *Content* classes. If, however, we use the undersampling technique, we will do the opposite and by reducing the over-represented classes also reduce the under-represented ones. Finally, we decided to take the third approach and accept working with our imbalanced dataset by measuring the right performance indicators during the evaluation phase. The indicator we will use for comparing different model scores in the following is the weighted average of the F1 score. This indicator takes into consideration the support examples ratio per class while making the averages.

4.3 Coarse-grained evaluation

Before opting for a specific neural network model and concentrating on its hyper-parameters' tuning, we will start with a coarse-grained evaluation in order to select

the one that is best-performing within our dataset. Among the different models we described in Section 2.2, we will do our coarse-grained evaluation using BERT, SciBERT, RoBERTa, DistillBERT and XLNet.

4.3.1 Models evaluation script

With the purpose of evaluating different neural network models and before concentrating on the best-scoring model for fine-tuning purposes, we tested the five models using the nerModelsAnalyzer script. The script takes as input the list of models, the number of epochs each model will be trained through and the train, valid and test datasets. It also runs a grid search technique for hyperparameters' tuning, using the BUILD_HYPERPARAMS function where a total of 64 different hyperparameter combinations are evaluated per model. The number of epochs is set to 10 for coarse-grained evaluation. In order to accelerate the training process, the script is executed on an Nvidia Tesla V100 GPU with a total of 32.5G of memory. Once the grid-search per model is performed, the script runs the final evaluation that will select the best grid-search hyperparameter combination for the *Location* class and the weighted average. The choice of the *Location* class is due to the fact that we consider this class the most important for the future correlation between the requested and the actual changes using the location within the article where a change is requested and is eventually fulfilled.

The grid-search technique uses the 64 different hyperparameter combinations with different values for the weight decay, the learning rate and the train batch size. Within the experiment, we took the value of 0.1 representing 10% of the defined learning rate. As the total number of epoch is defined to 10, the final epoch will operate at the initially defined learning rate.

4.3.2 Results

During our experimentation phase, we started evaluating our models on all classes together, namely, *Location*, *Action*, *Content*, *Modal* and *Trigger*. A short experiment

Algorithm 2 nerModelsAnalyzer

```
models ← BERT, SciBERT, RoBERTa, DistillBERT, XLNet
n_epoch ← 10
train_data ← train.txt
valid_data ← valid.txt
test_data ← test.txt
START(models, n_epoch, train_data, valid_data, test_data)
function START(models, n_epoch, train_data, valid_data, test_data)
    best_loss = 1
    for hyperparam in BUILD_HYPERPARAMS( )do
        for model in models do
            for i = 1, i++, while i < n_epoch do
                for batch in train_data do
                    loss ← TRAIN(model, batch, hyperparam)
                    train_losses += loss
                end for
                train_loss = train_losses/len(train_data)
                if train_loss < best_loss then
                    best_loss ← train_loss
                    model_to_save ← model
                end if
            end for
            eval_results[model, hyperparam] ← EVAL(model_to_save, valid_data)
        end for
    end for
    eval_compare ← EVAL_COMPARE(eval_results, test_data)
end function
function BUILD_HYPERPARAMS
    weightdecay ← [0.1, 0.01, 0.001, 0.0001]
    learningrate ← [0.01, 0.001, 0.0001, 0.00001]
    warmupproportion ← [0.1]
    trainbatchsize ← [16,32,64,128]
    list_hyperparams ← build the hyperparameter combination list
    return list_hyperparams
end function
function EVAL_COMPARE(eval_results, test_data)
    for model_results in eval_results do
        best_model ← SORT_BEST_MODEL(model_results)
        test_results[model] ← TEST(best_model, test_data)
    end for
    return SORT(test_results)
end function
```

was also carried out on training each class separately by removing all the remaining classes each time. The obtained scores for individual class training were much lower compared to training with all classes. This confirms that the tested deep-learning models are well aware of the sentence structure and the relationship between classes which allows them to perform better, as requested, on multi-class labelling vs. labelling each class separately.

Evaluation results including all classes

After running the nerModelsAnalyzer evaluation script on the five previously mentioned models using the grid search technique, we were able to identify different high-

scoring hyperparameter combination clusters and also obtain the best coarse-grained hyperparameter combination per model.

Figure 4-4 shows the 4D graphs for each of the models representing the weighted average F1 score variation per unique combination of learning rate, weight decay and train batch size (the numerical values are reported in Table 4.2). For all the models, we can deduce that by using a high learning rate of 0.01, the F1 scores are very low. In addition, XLNet has low F1 scores for high and low learning rates and operates the best while the learning rates are 0.001 or 0.0001. Regarding the train batch sizes across all models, the highest scores were obtained while using lower train batch sizes of 16.

The following hyperparameter combinations shown in Table 4.1 were used per model while achieving their optimal scores:

Model	Learning rate	Weight decay	Train batch size
BERT	0.0001	0.1	16
SciBERT	0.0001	0.001	16
DistilBERT	0.0001	0.01	16
RoBERTa	0.0001	0.0001	16
XLNet	0.0001	0.001	16

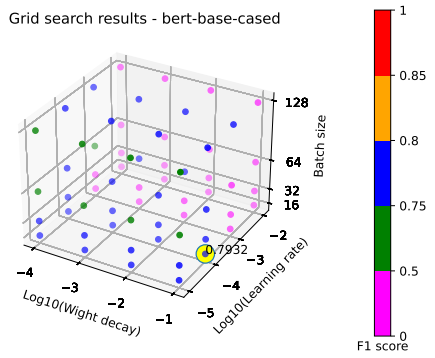
Table 4.1: Optimal hyperparameter combination per model—coarse.

Table 4.2 shows the different precision, recall and F1 scores per model and per class including the micro, macro and weighted averages. We have highlighted in , and the top scores per class and per different averages.

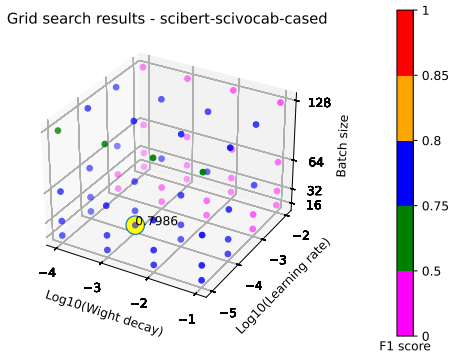
	BERT			SciBERT			DistilBERT			RoBERTa			XLNet		
	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1
ACTION	.7720	.8129	.7919	.7749	.8082	.7912	.6631	.7391	.6990	.7759	.8035	.7895	.7062	.7316	.7187
CONTENT	.6628	.6886	.6781	.6830	.6947	.6888	.4083	.5469	.4676	.6729	.6389	.6678	.5515	.5253	.5381
LOCATION	.8201	.8300	.8052	.7858	.8409	.8124	.6049	.7074	.6521	.7894	.8059	.8045	.7137	.7024	.7080
MODAL	.9342	.9434	.9388	.9251	.9422	.9336	.8947	.9410	.9173	.9205	.9262	.9234	.8733	.8916	.8824
TRIGGER	.9316	.9364	.9340	.9346	.9385	.9366	.9123	.9245	.9184	.9250	.9054	.9151	.8339	.8357	.8348
micro avg	.7797	.8063	.7927	.7869	.8104	.7985	.6048	.7154	.6555	.7824	.7860	.7842	.6927	.6836	.6881
macro avg	.8175	.8423	.8296	.8207	.8449	.8325	.6967	.7718	.7309	.8168	.8236	.8200	.7357	.7373	.7364
weighted avg	.7807	.8063	.7932	.7873	.8104	.7986	.6259	.7154	.6658	.7823	.7860	.7840	.6899	.6836	.6866

Table 4.2: Top coarse-grained grid-search scores for BERT-based and XLNet models.

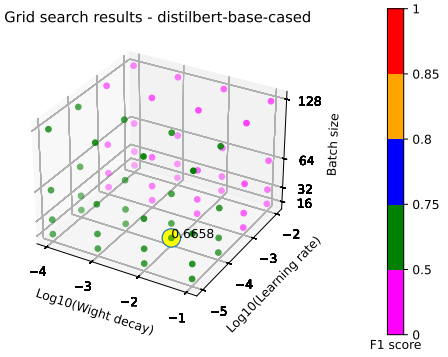
Coarse-grained scores for the top grid-search hyperparameter combinations training the BERT-based and XLNet deep learning models. Highlighted per class are in the top precision, in lime the top recall and in green the top F1 score.



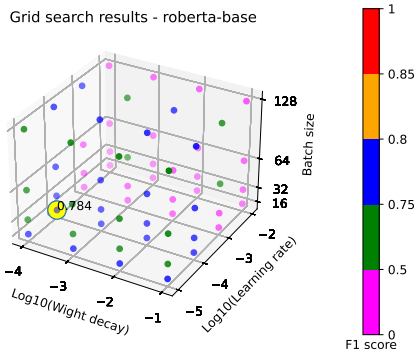
(a) BERT



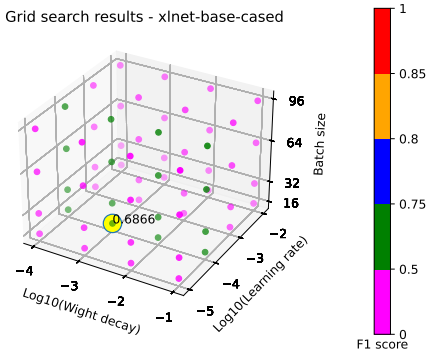
(b) SciBERT



(c) DistilBERT



(d) RoBERTa



(e) XLNet

Figure 4-4: Coarse-grained grid-search for all classes.
During 10 training epochs, the weighted F1 scores are shown in the function of different grid-search hyperparameter combinations with the top scores circled for each model.

As seen from the results table, the SciBERT model obtained the highest scores during our coarse-grained approach. This is most probably due to the fact that the SciBERT model was pre-trained on scientific data using a similar vocabulary to the review comments that are composed of different critiques made on scientific articles. SciBERT is closely followed by BERT and RoBERTa, while DistilBERT had the lowest scoring. As seen in Section 2.2, DistilBERT was made to be smaller in size and faster while retaining 95% of BERTs language understanding. Within our training dataset, this had a direct impact on the prediction result scores that are, compared to BERT, 16% lower. The XLNet model also achieved lower scores that are higher than those of DistilBERT but lower compared to the other BERT-based models. Regarding the top weighted average F1 scores of different models, BERT is able to achieve a weighted average F1 score of 0.7932, SciBERT of 0.786, DistilBERT of 0.6658, RoBERTa of 0.7840 and XLNet of 0.6866. In addition, we can see that the lowest-scoring class is the *Content* named entity with scores lower that 0.7 compared to other classes that are close to 0.8 for *Action* and *Location* and 0.9 for *Trigger* and *Modal*. This is mainly due to the two following reasons: on one hand, the complexity of the content part in the review comments which can be difficult to judge; and on the other, the number of poorly annotated examples within our dataset as the humans who made the manual annotation had different criteria regarding what the content really is. For these reasons, we decided to experiment on the training by ignoring the *Content* class in order to increase the final scores and check the impact this has on the scores of the labelling of the remaining classes.

Evaluation results excluding the *Content* class

During the last experiment, we excluded the *Content* class from the datasets. Without using the NER tool, in most cases we could assume the content being the remaining text that is not annotated within a given sentence. The same evaluation nerModels-Analyzer script was used to train the BERT, SciBERT, RoBERTa, DistillBERT and XLNet models using the same hyperparameter combinations within our new datasets, excluding the *Content* named entity.

Figure 4-5 shows the 4D graphs for each of the four BERT-based and XLNet models representing the F1 score variation per unique combination of learning rate, weight decay and train batch size (the numerical values are reported in Table 4.4). Compared to the previous experiment, on this occasion, we trained the models without the *Content* class as this class achieved the lowest scores and is the hardest to predict. The first requested impact of removing the lowest-scoring class is the increase in the average scores across all models. BERT increased its weighted average F1 score from 0.7932 to 0.8583, SciBERT from 0.7986 to 0.8609, DistilBERT from 0.6658 to 0.7658, RoBERTa from 0.784 to 0.8483 and XLNet from 0.6866 to 0.7762.

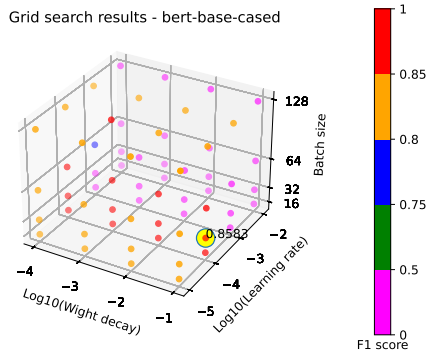
The following hyperparameter combinations shown in Table 4.3 were used per model while achieving their optimal scores:

Model	Learning rate	Weight decay	Train batch size
BERT	0.0001	0.1	32
SciBERT	0.0001	0.1	16
DistilBERT	0.0001	0.0001	16
RoBERTa	0.0001	0.01	16
XLNet	0.0001	0.1	16

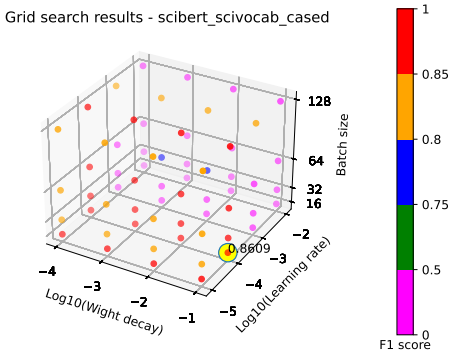
Table 4.3: Optimal hyperparameter combination per model—coarse w.o. *Content*.

Table 4.4 shows the different precision, recall and F1 scores per model and per class including the micro, macro and weighted averages. We have highlighted in , and the top scores per class and per different averages. In addition to the previously mentioned improvement of the overall score averages by ignoring the *Content* class, the other scores per class remained similar to their previous versions. We can notice, however, a slight F1 score increase for the *Location* and the *Action* classes regarding the BERT and SciBERT models on one side and a slight decrease for those scores for the DistilBERT, RoBERTa and XLNet models on the other side.

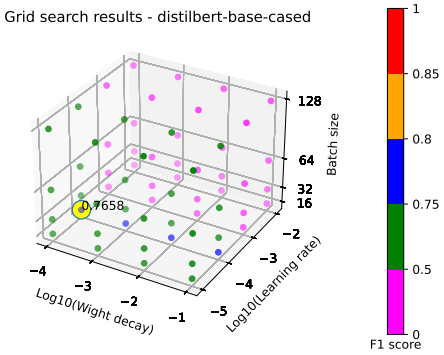
Similar to the previous experiment results, the BERT and the SciBert models achieved the best scores but now within all the measured indicators including the precision, recall and the F1 scores. We can also observe from the results that, within the newly revised datasets ignoring the *Content* class, there is almost a clear division between the precision and the recall capacities of the BERT and the SciBERT models.



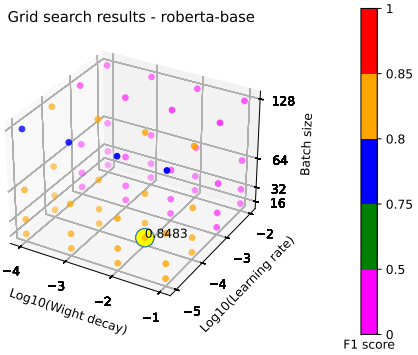
(a) BERT



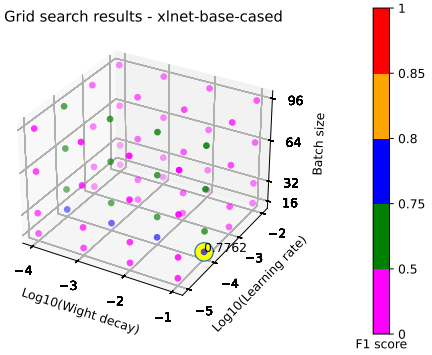
(b) SciBERT



(c) DistilBERT



(d) RoBERTa



(e) XLNet

Figure 4-5: Coarse-grained grid-search without *Content* class.
Excluding the *Content* class from the initial datasets, during 10 training epochs, the weighted F1 scores are shown in the function of different grid-search hyperparameter combinations with the top scores circled for each model.

	BERT			SciBERT			DistilBERT			RoBERTa			XLNet		
	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1
ACTION	.7722	.8048	.7882	.7790	.8015	.7901	.6732	.7143	.6931	.7697	.7981	.7837	.7250	.7351	.7300
LOCATION	.7919	.8376	.8141	.7997	.8281	.8137	.5934	.6662	.6277	.7857	.8125	.7989	.6971	.7004	.6988
MODAL	.9214	.9373	.9293	.9253	.9299	.9276	.8863	.9200	.9028	.9250	.9250	.9250	.8654	.9027	.8837
TRIGGER	.9326	.9292	.9309	.9408	.9370	.9389	.9113	.9189	.9151	.9270	.9127	.9198	.8597	.8399	.8497
micro avg	.8450	.8706	.8576	.8527	.8681	.8603	.7407	.7870	.7631	.8412	.8540	.8476	.7748	.7774	.7761
macro avg	.8545	.8772	.8656	.8612	.8741	.8676	.7661	.8048	.7847	.8519	.8621	.8568	.7868	.7945	.7905
weighted avg	.8467	.8706	.8583	.8539	.8681	.8609	.7465	.7870	.7658	.8428	.8540	.8483	.7751	.7774	.7762

Table 4.4: Top coarse-grained grid-search scores without *Content* class.

Coarse-grained scores for the top grid-search hyperparameter combinations without the *Content* class.

The BERT model has the highest recall scores for three out of the four classes, while the SciBERT model has the highest precision for all four classes. Overall, SciBERT has the highest F1 scores among all average indicators and half of the highest F1 scores, namely, for the *Action* and the *Trigger* classes. Regarding the top weighted average F1 scores of different models, BERT is able to achieve an weighted average F1 score of 0.8583, SciBERT of 0.8609, DistilBERT of 0.7658, RoBERTa of 0.8483 and XLNet of 0.7762.

With the coarse-grained evaluation carried out and the hyperparameter combination clusters where the specific model performs the best identified, we can move on to the fine-grained approach in order to further optimise the evaluated models.

4.4 Fine-grained evaluation

For each of the previously coarse-grained model evaluation results, we identified hyperparameter combination clusters where the specific model performs the best. We will now further fine-tune those hyperparameters by using additional fine-grained variations within the cluster.

The following hyperparameter combinations were built for each of the previously coarse-grained evaluated models, using Script 2:

- **BERT, SciBERT, DistilBERT and XLNet models**:
  - weight decay = [0.1, 0.01, 0.001, 0.0001];
  - learning rate = [1e-5, 2.5e-5, 5e-5, 7.5e-5, 1e-4, 2.5e-4, 5e-4, 7.5e-4, 1e-3];
  - train batch size = 128, 32, 24, 18, 12, 8]

- **RoBERTa model**:
  - weight decay = [0.1, 0.01, 0.001, 0.0001, 0.00001];
  - learning rate = [1e-5, 2.5e-5, 5e-5, 7.5e-5, 1e-4, 2.5e-5, 5e-4, 7.5e-4, 1e-3];
  - train batch size = [128, 32, 24, 18, 12, 8]

The main reason for adding an extra weight decay of 0.00001 to the RoBERTa model was that we observed during the coarse-grained evaluation with the *Content* class that the best weighted average F1 score was obtained with the 0.00001 weight decay, and we decided to explore this direction even further. Using the same evaluation nerModelsAnalyzer script, we trained the BERT, SciBERT, RoBERTa, DistillBERT and XLNet models using the previously mentioned fine-grained hyperparameters.

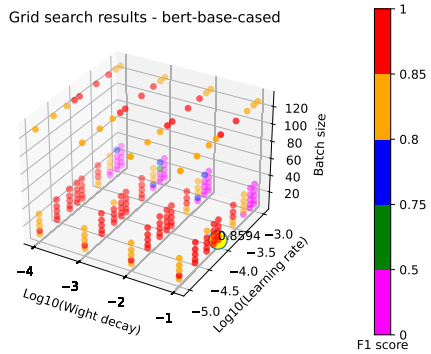
Figure 4-6 shows the 4D graphs for each of the four BERT-based and the XLNet models representing the F1 score variation per unique combination of learning rate, weight decay and train batch size. During the fine-grained training phase, BERT increased its weighted average F1 score from 0.8583 to 0.8594, SciBERT from 0.8609 to 0.8655, DistilBERT from 0.7658 to 0.7813, RoBERTa from 0.8483 to 0.8484 and XLNet from 0.7762 to 0.8068. The new scores rank the five deep learning neural network models as follows: 1st SciBERT, 2nd BERT, 3rd RoBERTa, 4th XLNet and 5th DistilBERT. We observe that the hyperparameter that influenced the weighted F1 scores the most is the learning rate, which we have varied the most.

The following hyperparameter combinations shown in Table 4.5 were used per model while achieving their optimal scores:

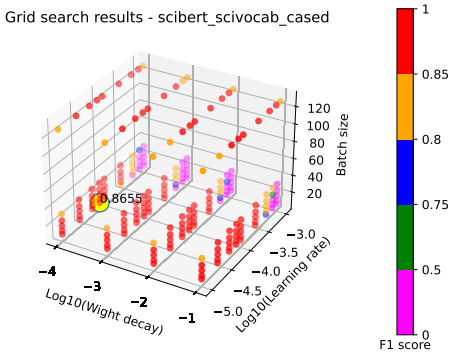
Model	Learning rate	Weight decay	Train batch size
BERT	0.0001	0.1	8
SciBERT	0.000075	0.0001	8
DistilBERT	0.0001	0.0001	16
RoBERTa	0.0001	0.001	18
XLNet	0.00025	0.1	12

Table 4.5: Optimal hyperparameter combination per model—fine.

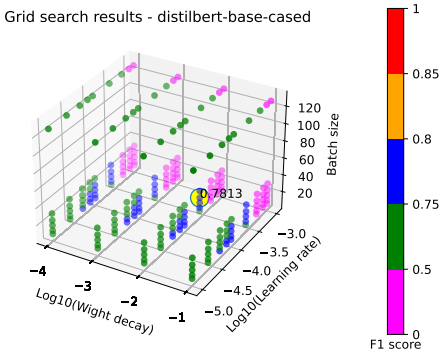
Table 4.6 shows the different precision, recall and F1 scores per model and per class including the micro, macro and weighted averages. We have highlighted in , and the top scores per class and per different averages. Besides the overall



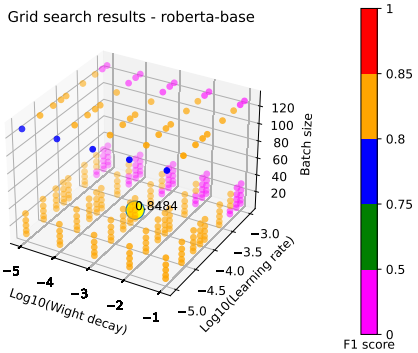
(a) BERT



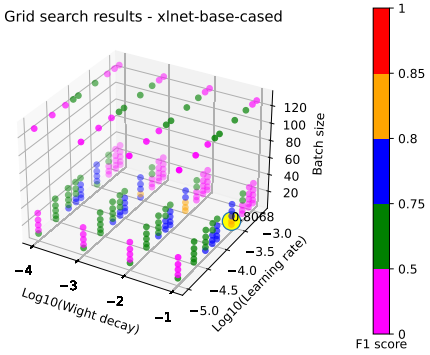
(b) SciBERT



(c) DistilBERT



(d) RoBERTa



(e) XLNet

Figure 4-6: Fine-grained grid-search without *Content* class.
Excluding the *Content* class from the initial datasets, during 10 training epochs, the weighted F1 scores are shown in the function of different fine-grained grid-search hyperparameter combinations with the top scores circled for each model.

increase in the weighted F1 score across all models, we can also observe that during the fine-grained evaluation, the SciBERT model had the best F1 scores across all classes. Compared to the coarse-grained evaluation where BERT, as the 2nd scoring model, had better recall scores, while SciBERT had better precision scores, SciBERT now has both the majority of best precision and recall scores. The largest weighted F1 score increase during the fine-grained evaluation was observed with the XLNet model that had its score increased from 0.7762 to 0.8068.

	BERT			SciBERT			DistilBERT			RoBERTa			XLNet		
	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1
ACTION	.7801	.7995	.7897	.8015	.7988	.8001	.7094	.7123	.7108	.7680	.7948	.7811	.7455	.7434	.7444
LOCATION	.7945	.8348	.8141	.7981	.8423	.8196	.6174	.6899	.6516	.7853	.8120	.7984	.7420	.7360	.7390
MODAL	.9219	.9287	.9252	.9164	.9434	.9297	.8951	.9237	.9092	.9186	.9299	.9242	.8965	.9064	.9014
TRIGGER	.9381	.9292	.9349	.9358	.9421	.9390	.9200	.9266	.9233	.9348	.9111	.9228	.9009	.8749	.8877
micro avg	.8497	.8681	.8588	.8554	.8755	.8653	.7619	.7972	.7792	.8420	.8533	.8476	.8110	.8023	.8067
macro avg	.8586	.8737	.8660	.8629	.8817	.8721	.7855	.8131	.7987	.8517	.8619	.8566	.8212	.8152	.8181
weighted avg	.8512	.8681	.8594	.8560	.8755	.8655	.7667	.7972	.7813	.8438	.8533	.8484	.8115	.8023	.8068

Table 4.6: Top fine-grained grid-search scores without *Content* class.

Fine-grained scores for the top grid-search hyperparameter combinations without the *Content* class.

With the fine-grained grid-search evaluation done, we can reuse the top-scoring hyperparameter combinations in order to do a final training for each model during 200 epochs (compared to the number of 10 epochs during the grid-search evaluation). At the end, we will compare their final scores with the purpose of selecting the best model for our NER task.

4.5 Final long-epoch evaluation

By using the top-performing hyperparameter combination per model obtained during the fine-grained grid-search evaluation, we used a similar algorithm to Script 2 for final training during 200 epochs by automatically selecting the best model variant with the lowest loss. Instead of getting the hyperparameter combination from the BUILD_HYPERPARAMS function described in the nerModelsAnalyzer evaluation script, those were now used depending on the weighted average F1 score results from Figure 4-6. Table 4.7 shows the different scores per model, with the top-scoring model being SciBERT with an weighted F1 average score of 0.8655. SciBERT is

closely followed by BERT with 0.8568 and RoBERTa with 0.8476. Those are then followed by XLNET with 0.8188 and DistilBERT as the lowest-scoring model with 0.7982.

	BERT			SciBERT			DistilBERT			RoBERTa			XLNet		
	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1	Precis.	recall	F1
ACTION	.7988	.7854	.7920	.7848	.7827	.7837	.7659	.7176	.7410	.7911	.7673	.7790	.7474	.7386	.7429
LOCATION	.7986	.8073	.8029	.8216	.8220	.8218	.6944	.6681	.6810	.7983	.8078	.8030	.7624	.7427	.7524
MODAL	.9177	.9323	.9250	.9191	.9360	.9275	.8983	.9237	.9109	.9206	.9127	.9166	.9059	.9126	.9092
TRIGGER	.9420	.9318	.9369	.9440	.9318	.9378	.9339	.9121	.9229	.9406	.9003	.9200	.9225	.8974	.9098
micro avg	.8574	.8561	.8568	.8626	.8608	.8617	.8119	.7868	.7991	.8547	.8533	.8472	.8266	.8111	.8188
macro avg	.8643	.8642	.8642	.8674	.8681	.8677	.8231	.8054	.8139	.8627	.8619	.8547	.8346	.8228	.8286
weighted avg	.8576	.8561	.8568	.8627	.8608	.8655	.8103	.7868	.7982	.8556	.8533	.8476	.8266	.8111	.8188

Table 4.7: Final model training scores without *Content* class.

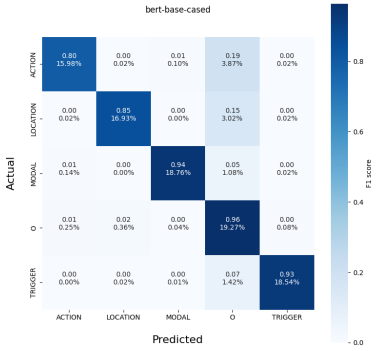
Figure 4-7 shows the confusion matrix per final trained model where we can see the different ratios between the predicted and the true values. We observe that the *Action* and the *Location* classes most often got confused with the *Modal* class. For higher-scoring models, the confusion of those classes is between 3% and 4% and for the lower-scoring models between 5% and 6%. As we have already fine-tuned our hyperparameters, we assume that with a larger and improved dataset, those class confusion errors could be reduced even further and brought closer to 0%.

Table 4.8 presents a summary of the five models' evaluation, with SciBERT being the top-performing (using the F1 weighted average score) on our NER task annotating review comments. SciBERT is closely followed by BERT, and the lowest-scoring model is DistilBERT, with a 7.8% lower weighted avg F1 score.

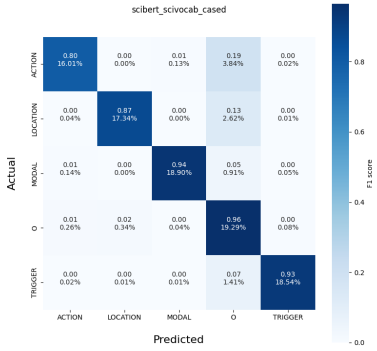
Model	↓ Weighted avg F1	Location F1	Action F1
SciBERT	0.8655	0.8218	0.7837
BERT	0.8568	0.8029	0.7920
RoBERTa	0.8476	0.8030	0.7790
XLNet	0.8188	0.7524	0.7429
DistilBERT	0.7982	0.6810	0.7410

Table 4.8: Final evaluation results.

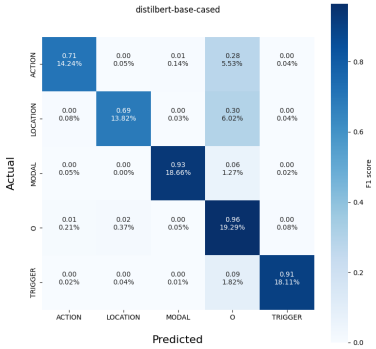
By performing the evaluation of different deep learning neural network models (BERT, SciBERT, DistilBERT, RoBERTa and XLNet), we were able to select the best-scoring model on our NER task, which is SciBERT. A final 200 epoch training



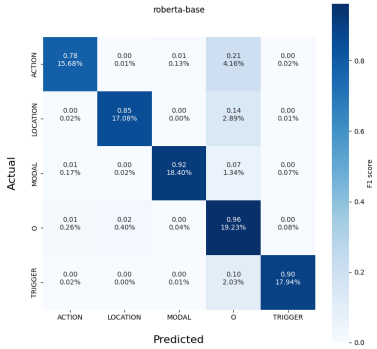
(a) BERT



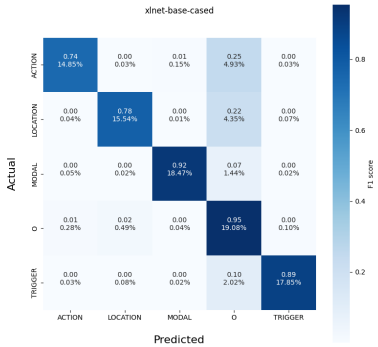
(b) SciBERT



(c) DistilBERT



(d) RoBERTa



(e) XLNet

Figure 4-7: Confusion matrix: *Action*, *Location*, *Modal*, *Trigger* and *O* class. During 200 training epochs, the predicted vs. true values are shown per class combination, showing the four classes *Action*, *Location*, *Modal*, *Trigger* and the class *O* representing the other words within the CONLL format.

resulted in a new model, based on SciBERT, which we called "review-annotation" and which was able to achieve a weighted average F1 score of 0.8655. In the next section, we will present the human evaluation of that model on its prediction capacities of the *Location* class.

4.6 Manual human evaluation—*Location* class

In order to validate the annotation of review comments for the *Location* class using our NER tool, we extracted 1000 random real-life review comment examples from the database, ran those through the annotation tool and asked internal MDPI editors to check them manually and document the examples where the model made a prediction error. Out of the 1000 examples, a success rate of 77.1% was achieved, with an error rate of 22.9%, from which 14.8% were missing and 10.1% were wrong predictions.

By analysing more closely different errors made by the model while predicting the *Location* class, we first observed that the model learned that everything within quotes is a location. This is not always true as there can be examples of text corrections where one quoted sentence (being the location) should be replaced by another quoted sentence (which is not the location). There is probably a need for additional training data in order for the model to better distinguish between quoted locations and other quoted text. Another issue is regarding some under-represented examples using the words "figure", "equation", "introduction", "conclusion", etc., which also result in missing predictions. We also observed that some generic (imprecise) location words are also not being predicted, such as "manuscript", "paper", "sentence" and "text".

Regarding the evaluation results that are lower than requested (lower than the score of 0.87 during the evaluation), we concluded that for better results, we first need to re-work and then also increase our training dataset in order to provide the model with better training examples. This way, we should be able to first increase the evaluation score, but also the real-life testing score.

4.7 Discussion

As seen within this chapter, we evaluated five pre-trained neural network models and assessed their capacities around achieving our NER tasks. Besides the five tested models, there are other models available that we did not test for various reasons, the first one being their lower state-of-the-art performance made available within the literature. There are also other models such as GPT-2 and GPT-3 achieving similar or even better state-of-the-art results than BERT or XLNet. Those are, however, not available in open source and are only available through the OpenAI API[4]. In order to further improve the prediction scores, we will continue to build a better annotated dataset, also higher in number of examples. In addition, evaluating other deep learning models is another possibility for increasing the prediction scores.

4.8 Conclusion

Within this chapter, we have described our work on extracting requested changes—those asked by the reviewer—by annotating the review comments and extracting the following meaningful change information: *Location*, *Action*, *Modal* and *Trigger*. For this, we evaluated different deep learning models that were fine-tuned on our NER task. The models we evaluated are BERT, SciBERT, DistilBERT, RoBERTa and XLNet. We first started with a coarse-grained evaluation by applying the grid-search technique having 64 different combinations of hyperparameters by varying the learning rate, weight decay and train batch size. The approach is coarse-grained as those variations are large, going from 0.1 to 0.00001 for the learning rate, 0.1 to 0.0001 for the weight decay and 16 to 128 for the train batch size. A script we developed, called nerModelsAnalyzer, was used to perform the grid-search and generate 4D graphs per evaluated model. Once the coarse-grained evaluation was carried out, we moved on to the fine-grained evaluation, where depending on the coarse-grained results, different fine-grained hyperparameters were used in order to further fine-tune the models and increase their scores. Once the fine-grained evaluation was completed, the models

[4]https://openai.com/api/

were further trained with the optimal hyperparameters for 200 epochs, with the top-scoring model selected, being SciBERT, achieving an weighted average F1 score of 0.87. Finally, a group of 10 editors tested the fine-tuned model on real-life review comments for its capacity to detect *Location* named entities. Those named entities are categorised as being the most critical within the next step that will consist of correlating the requested changes *Location* named entities with their corresponding locations within the article where the author made corrections. The obtained human evaluation result gave us a success rate of 77.1%.

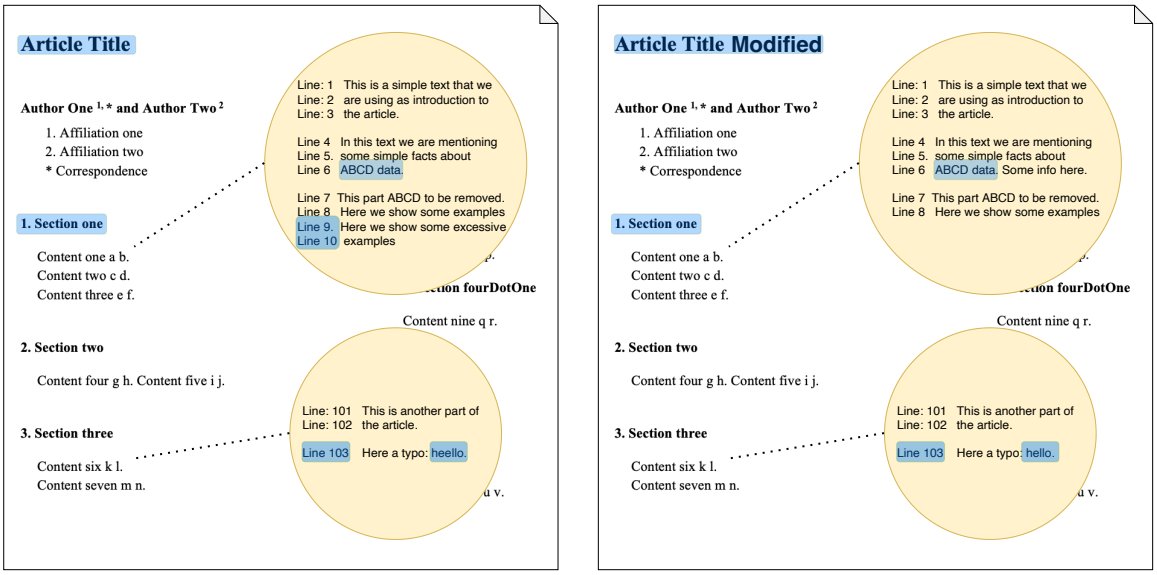
Chapter 5

Information matching

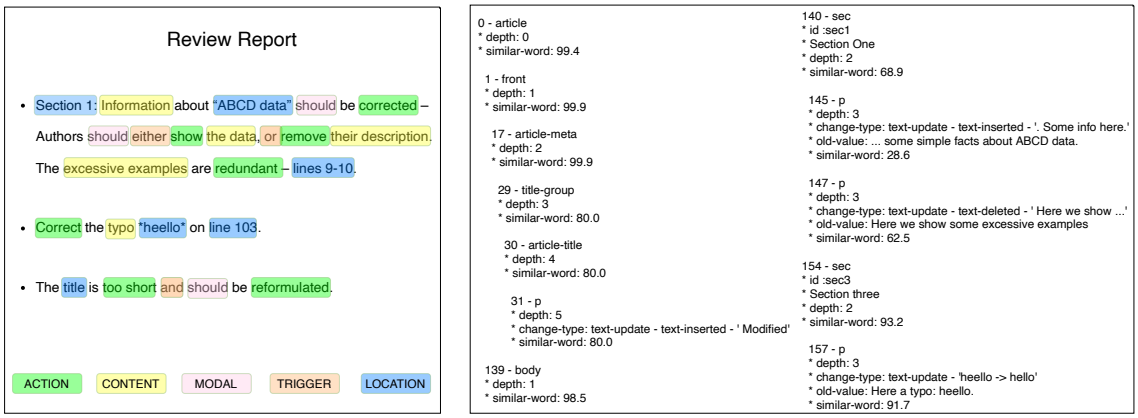
As seen in Figure 1-2, once the actual changes—those made by the authors—are extracted by comparing the two JATS XML versions of the article, and the requested changes—those asked by the reviewer—are extracted using our NER model, we can start correlating those two pieces of information in order to propose a unified view to the decision-maker.

The Figure 5-1 will be used in the following sections as a demonstration example. It is composed of: Figure 5-1a that shows two article versions, the original on the left and the modified on the right; and Figure 5-1b that shows the current change information we are able to extract, on the left the requested changes obtained by using the NER model while on the right are the actual changes obtained by using the jats-diff algorithm. The Figure 5-1b shows a practical use-case where correlating the *Location* entity extracted from the review comments and the location where the change should happen within the article gives to the final decision-maker a quick and precise overview of the requested and actual changes. Using the *Location* entity, we have the answer to "Where" the change should happen. By using the other entities (*Action*, *Modal* and *Trigger*), we can understand "How" the change should be realised.

The first and the most important step towards providing the decision-maker with information if the author fulfilled the expected changes during the revision round is taken by correlating the author modifications and the review comments. We will show in the following how different extracted named entities can be correlated within



(a) Two article versions



(b) Change information

Figure 5-1: Analogies between change information.

Figure (a) shows two article versions, on the left the original article and on the right its modified version. Figure (b) shows on the left the annotated review comments and on the right the differences extracted between two article versions using jats-diff algorithm.

the modifications from the author change list and what additional information each of those entities can provide.

5.1 Location

The *Location* named entity plays the most important role regarding the correlation between the requested and actual changes. This is made possible by using the location within the article where the actual change took place and the *Location* annotated in the review comment. Correlating those two pieces of information will result in three change categories:

- **Requested and detected changes**: the author made modifications on a specific location within the article following a review request;
- **Requested but not detected changes**: the author did not make the modifications requested by the reviewer on a specific location within the article;
- **Detected but not requested changes**: the author made modifications that were not requested by the reviewer on a specific location within the article.

In order to distinguish between those three categories, for each of the review comments *Location* named entities, a correlation trial is made to find a corresponding article location within the author change list. If the match is positive, the correlation worked and the given change is categorised as requested and detected. If the match is negative, either the change request is optional (using the modality information) or the author ignored that given change request. The remaining modifications in the author change list for which there is no correlation with the *Location* named entity within the review comments are categorised as detected but not requested.

As observed in Figure 5-1, there are four *Location* named entities: "Section 1", "title" and the two quoted examples "ABCD data" and "heello" that can be correlated with the delta information extracted by jats-diff. By analysing the training dataset,

we extracted the 20 most commonly used *Location* named entities—see Table 5.1—described them and assessed their precision within the text. The 3 most commonly used out of those 20 named entities (quoted text, line numbers and section names) cover over 90% of the total *Location* named entities observed within the dataset. We have decided to mainly concentrate on those in order to have a robust *Location* named entity correlation for the types covering the majority of the identified *Location* named entities.

named entity Type	Description	precision
quotes	quoted text that can be found within the article	precise
x section / section x	specific section identified by name or number	semi-pre.
line(s) x / x-y / x,y,...	line number available within the PDF	precise
table x	specific table identified by the number	precise
figure x	specific figure identified by the number	precise
title	title of the article, of the section, etc.	precise
keywords	keywords part of the article	precise
abstract	abstract part of the article	precise
references	references part of the article	semi-pre.
label/caption	the label / caption of a specific table / figure	precise
Introduction	common section cited without the section suffix	semi-pre.
Results and Discussion	common section cited without the section suffix	semi-pre.
Discussion	common section cited without the section suffix	semi-pre.
Conclusion	common section cited without the section suffix	semi-pre.
Supplementary x	common section cited without the section suffix	semi-pre.
paper/manuscript/body	generic comments applied to the whole article	generic
Reference(s) x / [x-y]	specific references identified by the number	precise

Table 5.1: Common *Location* named entity observed within the training dataset.

Once they are identified, we will assess in the following subsection how those *Location* named entities can be linked to a specific part of the article and the corresponding change detected by jats-diff.

5.1.1 Out-of-the-shelf matching possibilities

We will go through each of the previously identified *Location* named entity types—see Table 5.1—and assess the current out-of-the-shelf techniques to link those to their corresponding location within the article and the jats-diff output.

- **Quotes**: reviewers use single or double quotes to cite text parts of the article and request specific changes related to that text. Those are relatively easy to match by using search text and identifying those to a related location within the article and a given change detected using jats-diff;
- **Sections** (and synonyms such as chapter, etc.): there are different approaches when reviewers are identifying a section. The most straightforward one is by using section names or section numbers. However, we also face cases where reviewers describe a section as being the first, second, etc. or instead of using numerals, they use words as "Section two", "Section five" etc. This requires further processing in order to retrieve the right section name or number—for example, the reviewer can give a comment about the "last section" that we need to link to the last section of the article. In addition, subsections may also be mentioned by the reviewer with an example of requested change in the "Section two in the Methods section". Here, we have two levels of sections where we would need to identify the second subsection within the Methods section;
- **Line numbers**: reviewers also frequently use line number while identifying a specific text. Unfortunately, JATS XML representation of the article has no layout information, and thus, it does not provide any line number information, and those *Location* named entities are not possible to match out-of-the-shelf. For this, we will be using another custom technique described in Section 5.1.2;
- **Tables, Figures**: reviewers cite specific tables and figures by using their numbers and occasionally their labels. Both types of information are available within the article itself but also within the jats-diff output, as for each table and figure, jats-diff mentions its number and label;
- **Title, keywords, abstract, references**: those very specific parts of the article are always present and can be matched both on the JATS XML and also on the jats-diff output;
- **Citable references**: references available within the references list are numbered depending on their order of appearance. While mentioning a specific reference, for example, "reference 5", we use the reference number in order to

identify the right reference. For more complex examples where several references are mentioned with a from–to reference number, for example, "references [5-8]", further processing is needed as this annotation has to be converted to references 5, 6, 7 and 8. In addition, similarly to the section and line numbers, non-numerical references have to be converted to their numerical values.

We have observed examples where the exact match of the previously mentioned *Location* types was not successful due to different typos introduced by the peer reviewer. For such cases, instead of using exact match while searching the corresponding text, regular expressions or approximate string matching could be beneficial.

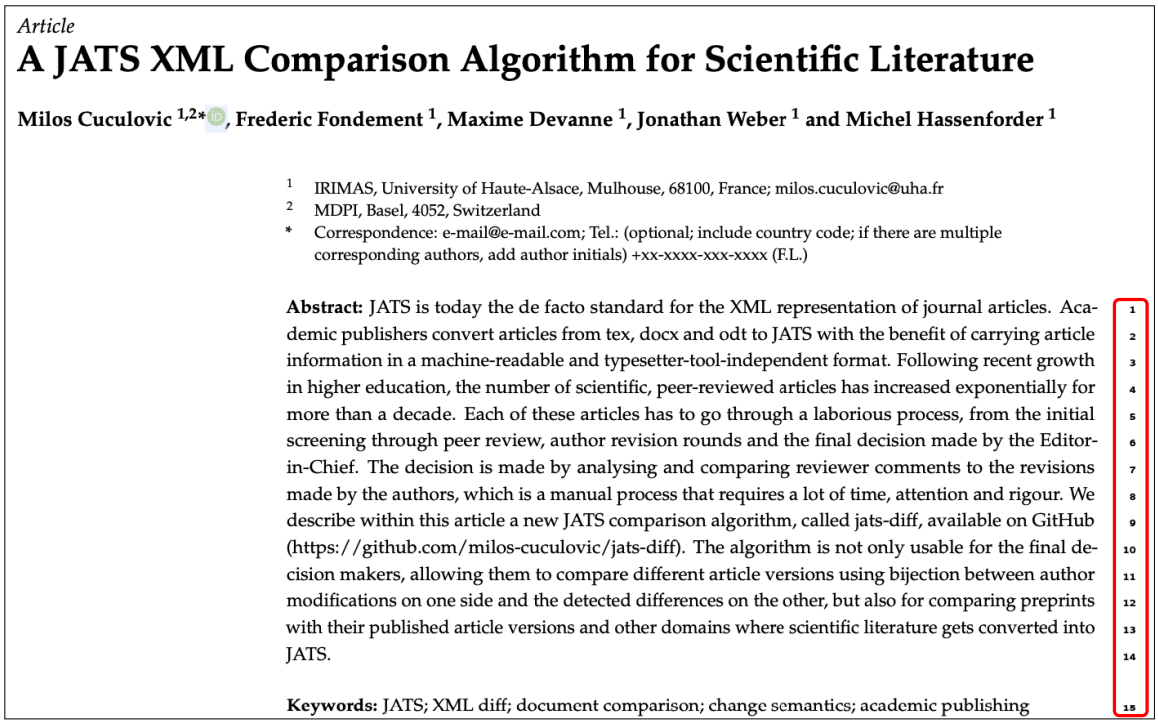
5.1.2 Custom matching

As seen previously, there are two use-cases where a simple and direct correlation of the *Location* named entity and the jats-diff output is impossible. We will discuss in the following which approach we took for those.

Line numbers

While reviewing an article, peer reviewers usually receive the PDF version of that article containing the line number annotated—see Figure 5-2. Very often, they make use of those line numbers to identify a specific location within the article the comments made are related to.

Carrying only the article data and structure and no layout information, assigning line numbers in JATS XML is not straightforward. We thus adopted an alternative approach that consists of extracting the line numbers from that PDF document and matching each line number with its corresponding text. The script 3 shows the pseudo-code on how the output map is generated, linking each line number to its corresponding text. The script first goes through the PDF document and extracts all character positions within the document. Those characters are afterwards evaluated and classified within sentence characters or line numbers. Once classification has been made, the output map is built, assigning each line number to its corresponding sequence of sentence characters (string).



Article

A JATS XML Comparison Algorithm for Scientific Literature

Milos Cuculovic [1,2]*, Frederic Fondement [1], Maxime Devanne [1], Jonathan Weber [1] and Michel Hassenforder [1]

[1] IRIMAS, University of Haute-Alsace, Mulhouse, 68100, France; milos.cuculovic@uha.fr
[2] MDPI, Basel, 4052, Switzerland
* Correspondence: e-mail@e-mail.com; Tel.: (optional; include country code; if there are multiple corresponding authors, add author initials) +xx-xxxx-xxx-xxxx (F.L.)

Abstract: JATS is today the de facto standard for the XML representation of journal articles. Academic publishers convert articles from tex, docx and odt to JATS with the benefit of carrying article information in a machine-readable and typesetter-tool-independent format. Following recent growth in higher education, the number of scientific, peer-reviewed articles has increased exponentially for more than a decade. Each of these articles has to go through a laborious process, from the initial screening through peer review, author revision rounds and the final decision made by the Editor-in-Chief. The decision is made by analysing and comparing reviewer comments to the revisions made by the authors, which is a manual process that requires a lot of time, attention and rigour. We describe within this article a new JATS comparison algorithm, called jats-diff, available on GitHub (https://github.com/milos-cuculovic/jats-diff). The algorithm is not only usable for the final decision makers, allowing them to compare different article versions using bijection between author modifications on one side and the detected differences on the other, but also for comparing preprints with their published article versions and other domains where scientific literature gets converted into JATS.

Keywords: JATS; XML diff; document comparison; change semantics; academic publishing

Figure 5-2: Example of PDF line numbers usage

A PDF version of the article sent to peer review containing line numbers on the right.

With the output map created, we will use it as a support database that will be accessed each time the line *Location* named entity containing line numbers is detected. This way, the initial correlation problem where line numbers were missing from the JATS XML is turned into a simple text quotation correlation problem which can be solved using text search.

Algorithm 3 PDFLineExtractor

```
input ← article.pdf
START(input)
function START(input)
    char_positions ← READCHARACTERPOSITIONS(input)
    sentence_characters ← EXTRACTSTRINGPOSITIONS(char_positions)
    line_numbers ← EXTRACTLINEPOSITIONS(char_positions)
    output_map ← BUILDOUTPUT(sentence_characters, line_numbers)
    return output_map
end function
```

Section numbers

Section numbers are easy to correlate between the *Location* named entity and the section numbers within the JATS XML and the jats-diff output. However, in cases where reviewers do not explicitly mention section numbers but use synonyms such as "First section" or "Section one" for the "1. Section", some additional pre-processing is needed in order to have a correspondence table between the real section numbers and different synonyms those can be referenced with.

Two main approaches are used here: first, the correspondence between the numeral and word representation of section numbers (1—one; 2—two, 3—three, ...); and second, between the ordinal and word representation of section numbers (1—first; 2—second, ..., n—last). This way, the capacity of correlating a *Location* named entity referring to a section is increased.

5.1.3 Location correlation result

Finally, once we know how to correlate most of the *Location* named entities extracted by our NER model to the location within the article and the location where specific changes took place, we show in Figure 5-3 how different named entities are correlated with different changes made by the authors during the revision round. Most of the named entities are directly correlated except the line numbers that go through the line numbers database map in order to retrieve the text behind each line.

5.2 *Modal*

The modality of a requested change is defined by the *Modal* NE. As seen in Chapter 4, the modality categorises a specific change request as being mandatory or optional. This information will help us to understand if a specific review request has to be linked or not to a change location in between. If we take the example seen in Figure 5-1, the modal "should" is categorising the requested change as mandatory, which includes a mandatory correlation between the requested change and the actual change linked to

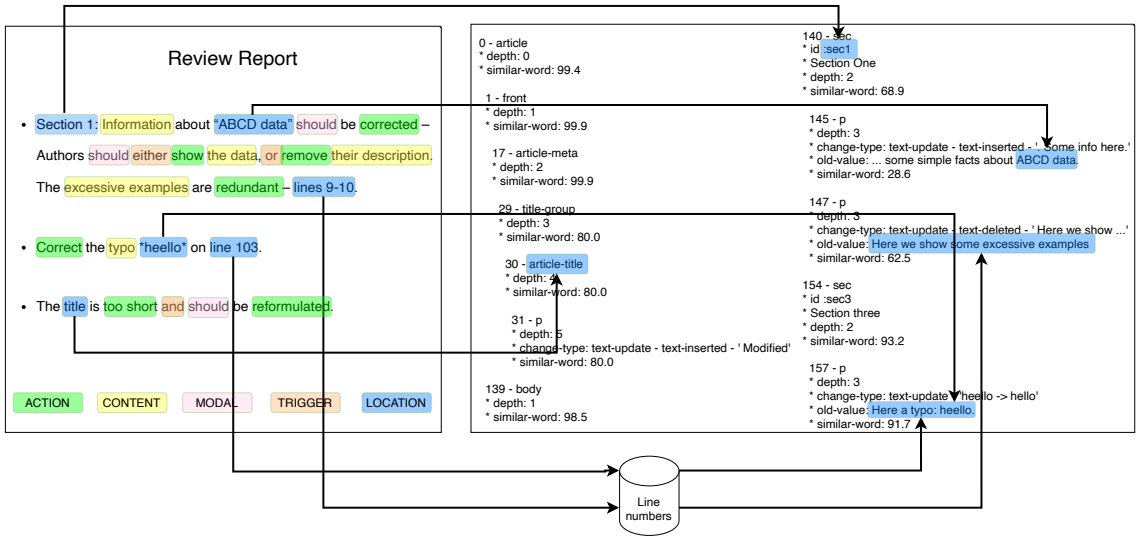


Figure 5-3: Analogies between NER *Location* namedentity and jats-diff.

On the left are the annotated review comments, and on the right the differences extracted between two article versions using jats-diff algorithm. The black arrows indicate the correlation between the *Location* named entities and the change location within the article. The line number map transforms the line number to its contained text.

a specific location within the article.

Table 5.2 shows the top six used modals within our training dataset. We can categorise those within the two initially defined categories for mandatory and optional change requests as follows: "should" and "must" for mandatory and "will", "can", "could" and "may" for optional changes. By analysing the total number of 4700 *Modal* named entity appearances, the top 6 modals represent over 90%, and those are divided as 86% being mandatory and 14% optional changes.

NE:	should	must	will	can	could	may
Count:	3150	639	349	110	66	59

Table 5.2: Common *Modal* named entities observed within the training dataset.

5.3 *Action*

The *Action* named entity identifies different type of actions the author should undertake in order to fulfil the requested changes described by the peer reviewer. Within

our training dataset, over 630 unique *Action* named entities were annotated with a total of 8300 appearances. Table 5.3 shows the top used named entities and their number of appearance. We see that the top 20 used named entities represent over 30% of the total number of appearances. Moreover, lemmatised versions of those words are also detected using similar root words by the models, meaning that the word mentioned being decomposed in "mention" + ("##ing" OR "ed" OR ...) covers additional examples within our dataset.

named entity	Count	named entity	Count	named entity	Count	named entity	Count
add	314	present	278	presented	260	revise	178
mentioned	150	provide	150	avoid	148	specified	134
check	130	mention	126	use	120	benefit	114
corrected	112	indicated	108	address	106	include	102
highlight	88	explained	84	removed	82	make	80

Table 5.3: Common *Action* named entities observed within the training dataset.

The next step is to correlate those *Action* named entities with specific edit actions observed in the jats-diff output. For this, we started creating a correspondence table that links different named entities to specific edit action(s). Table 5.4 shows some of those correspondences. As there are many more to be added, we also decided to use the synonyms of the top *Action* named entities in order to expand their correlation capacities.

Insert	Delete	Update	Split	Merge
add	remove	revise	split	merge
provide	avoid	correct	separate	join
include	excessive	update	break	unify
present	delete	explain		combine
insert		justify		stick
cite		specify		

Table 5.4: *Action* named entities correspondence table example.

The top used *Action* named entities are mostly verbs such as "add", "revise", "mention", "provide", "avoid", etc., but we can also observe, with a lower but still important number of occurrences, adjectives addressing a critique to a specific location within the article. Some examples of the adjectives used are "is missing", "too short",

"is not clear", "is not appropriate", etc. In a similar way as we categorised the verbs, there is also a need to categorise those adjectives where, for example, "is missing" and "too short" should be correlated to an Insert edit action, while "is not clear" and "is not appropriate" should be correlated an Update edit action.

Finally, having determined how to correlate some of the *Action* named entities extracted by our NER model to the edit action type on the jats-diff output, we show in Figure 5-4 how different named entities are correlated with different changes made by the authors during the revision round.

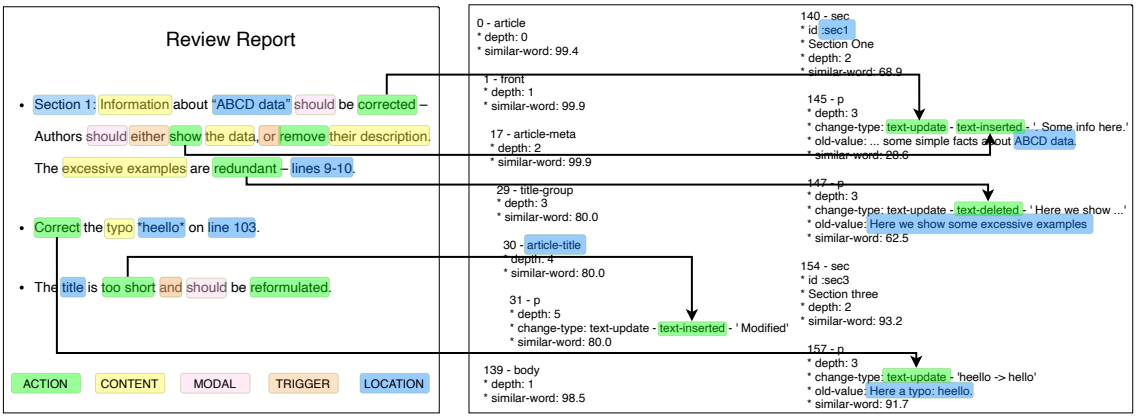


Figure 5-4: Analogies between NER *Action* named entities and jats-diff.

On the left are the annotated review comments and on the right the differences extracted between two article versions using the jats-diff algorithm. The black arrows indicate the correlation between the *Action* named entities and the change type extracted by jats-diff.

There are also other more ambiguous *Action* examples where the requested changes described can be achieved with different edits, or a combination of multiple edits that we are not yet able to solve. As an example, we have the *Action* "revise" that can be performed in multiple ways, using the Insert, Delete or Update edits, or a combination of all. Another example is the *Action* "check", where the author can apply different edits once that specific "check" is made.

5.4 *Trigger*

The last information provided by our NER model is the *Trigger* NE. The so-called triggers are used to further refine the correlation of the *Action* named entities. We do use Triggers in order to identify if a specific sentence is a question by using "what", "where", "why" and other triggers. In addition, within the examples, where there are multiple *Action* named entities, and triggers such as "or" "either", "instead of" and others specify that the author can choose one of the edit actions to fulfil the requested change. Moreover, within the same examples where there are multiple *Action* named entities, triggers such as "not only", "also", "both", "in particular" and others specify that the multiple *Action* named entities are not optional but have to all be addressed.

Within our training dataset, over 630 unique *Action* named entities were annotated with a total of 11,900 appearances. Table 5.5 shows the top used named entities and their number of appearance. We see that the top 20 used named entities represent over 85% of the total number of appearances.

named entity	Count	named entity	Count	named entity	Count	named entity	Count
and	4462	also	350	does	152	which	112
that	1902	but	341	why	147	however	110
what	699	how	196	while	143	like	94
or	617	etc	195	when	128	instead of	85
if	357	than	172	rather	124	without	80

Table 5.5: Common *Trigger* named entities observed within the training dataset.

If we take our example in Figure 5-4 with the two triggers "either" and "or" annotated in the comment "Authors should either show the data, or remove their description", we can link each of the two triggers to the edit actions that follow. This way, it can be deduced that the edit action to be executed could be either "show" or "remove". The author has chosen the 1st option and has added some text to the identified location within the article.

5.5 Conclusion

In this chapter, we have seen how the annotated named entities can be used in order to link specific change requests to their corresponding actual changes. We first described how to correlate the *Location* named entity with their corresponding locations where author modifications took place within the article. The most straightforward correlation is made using the quoted text, section names and specific parts of the article such as the title, keywords, abstract, etc. One of the most used *Location* named entity types is the line number, which is not directly available within the JATS XML , which was solved by extracting the line numbers from the article PDF. In the second part, we described how to use the *Modal* named entity in order to judge if a change request is mandatory or optional. This is valuable information during the correlation trials where changes are categorised within three groups: requested and detected; requested but not detected; and detected but not requested. If a change is requested but not detected, using the modality extracted with the *Modal* named entity, we can deduce if the change was optional, or if the author did not implement a requested change. Following was the correlation of the *Action* named entity with specific edit types observed in the jats-diff output. We categorised different *Action* named entity such as add, provide, avoid, etc. within their corresponding edit action groups: Insert, Delete, Update, etc. Using this type of information, we can assess if the requested edit from the review comments was fulfilled by the author with a corresponding edit detected by jats-diff. Finally, we explained how the *Trigger* named entity can be used in order to further refine the correlation of the *Action* named entity. Within the example provided where the change request mentions two different edit actions using the triggers "either" and "or", the author had to choose between "either" adding data "or" removing the data description. Using the correlation trials information where specific requested changes are correlated with their corresponding actual changes allows the final decision-maker to first assess the requested and detected changes within a unified view. For the remaining requested but not detected changes, we do provide the modality that is used to isolate mandatory changes that were requested but not

made by the author. And finally, the changes that were detected but never requested are shown separately so that the decision-maker can assess them and verify why the author performed changes that were never requested by the peer reviewer.

Chapter 6

Conclusion / Discussion

Over the last decade, the number of academic articles has been constantly growing. Besides their many other activities, senior scientists are also key players in the article publishing process, in charge of reviewing the articles and making the final acceptance decision. Within this PhD thesis, we assessed the article publishing process and proposed novel tools that can help to automate and further improve some of the manual tasks scientists perform during the academic publishing process.

6.1 Contributions

The contributions of this PhD thesis are divided into two research topics: first, document comparison; and second, Named Entity Recognition.

6.1.1 Document Comparison

The first contribution of this PhD thesis is in regard to assisting the reviewer or the final decision-maker in extracting the actual changes made by the author during the revision round. In order to extract the differences between the two article versions, we first analysed different XML diff algorithms and assessed their capacity in comparing JATS XML representations of academic articles. As their overall scoring was rather

low, we developed a novel XML diff algorithm, called jats-diff, which was able to make a bijection between the changes made by the author and the modifications detected by analysing the two XML documents. Compared to the existing XML diff algorithms, jats-diff is able to detect and represent the following author edit actions: structural upgrade, downgrade, split and merge, inline style edit, text move and citable node edit. In addition, jats-diff is also able to calculate and propagate towards the XML tree a so-called similarity index. This information is useful in order to obtain a broader picture of the impact that the changes made by the author have had. Finally, jats-diff uses some JATS-specific change semantics in order to properly detect and represent changes on citable objects, special objects (math formulas and figures) and lists containing tables, references and authors.

6.1.2 Named Entity Recognition

The second part of this PhD thesis was about assisting authors and final decision-makers in extracting the requested changes written by the reviewers. In order to achieve this, we took the sequence labelling approach by using deep learning models for the NER task. We started by comparing different deep learning models by training them in a supervised learning on a dataset of annotated review comments. We first started with a coarse-grained approach using the grid-search technique with different combinations of hyperparameters by varying the learning rate, weight decay and train batch size. Once the optimal hyperparameter cluster combinations were identified, we processed them with a more fine-grained approach until finding the optimal hyperparameter combination per model. The final step was a longer training (200 epoch) per model using the previously optimal hyperparameter combinations in order to select the best-scoring model, i.e., SciBERT, which achieved an weighted average F1 score of 0.87. During the human testing phase on real-life review comments carried out by a group of editors, our model obtained a success score on the *Location* named entity prediction of 77.1%.

6.1.3 Information Matching

With the actual and the requested changes retrieved using the document comparison jats-diff algorithm and the "review-annotation" NER model, we worked on the information matching part with the goal to correlate requested to actual changes. For each review comment, the *Location* named entity was correlated with its corresponding location within the article the jats-diff detected a change. Three types of *Location* named entities were identified: precise, semi-precise and geNERic. The *Action* named entity was correlated to the edit type regarding the correspondence table we made. The *Modal* named entity was used to define the modality of the requested edit action that we grouped into mandatory and optional. Finally, the *Trigger* named entity was used to add additional granularity to the *Action* named entity. Using the *Trigger*, we can define if there are multiple action options, or all proposed actions have to be fulfilled. In addition, triggers are also used to detect questions.

6.2 Perspective

6.2.1 Academic

The first output of this PhD thesis is the jats-diff algorithm that is meant not only to be used to compare JATS XML versions of academic articles but also any type of text-centric XML documents. Compared to other XML diff algorithms, jats-diff is able to detect additional high-level author edits and make a bijection between those edits on one side and the differences between the XML documents on the other side. Within future work, we do plan to make jats-diff modular, meaning the user can choose which edit action they are willing to detect and also create new edit actions by specifying the insert–delete sequence that specific edit actions are composed of. In addition, it could be very interesting to add a versioning option to jats-diff where the user could version different JATS articles. Yet another interesting idea would be to work on additional similarity indexes that jats-diff provides. The current similarity indexes measure the impact of textual and structural changes only. By adding additional

similarity indexes able to understand change semantics, we could distinguish between simple rephrase and sentence meaning changes. During document revision, the author could completely rewrite a specific paragraph, section or the whole paper by keeping a similar meaning. This additional similarity index could be very beneficial for the reviewer.

The second output of the thesis is the fine-tuned NER model trained on an annotated corpus of review comments. This is the first NER model available within this specific domain that can be further fine-tuned on similar datasets. The same sequence labelling approach could also be used on different domains where someone writes a review of a given text asking the author to revise it. Among the evaluated deep learning models, some additional models appeared recently, such as the GPT-3, which would be interesting to test on our dataset. Moreover, having extracted both the expected and the actual changes, additional experiments could be carried out in order to try to measure the fulfillment of the actual changes compared to the requested changes. An unsupervised machine learning approach could be taken, having a large corpus of requested and actual changes.

6.2.2 Industrial

This PhD thesis was financed by MDPI and is meant to be scaled and used in an industrial/production environment. As already mentioned, the two research topics we covered, document comparison and NER, could help the key players in the publishing process to further automate the review and the decision-making tasks. By using jats-diff, the publisher, the reviewer and the decision-maker can compare different article versions and extract the changes an author made during the revision round. In addition to change extraction, jats-diff also provides information about the impact that the author changes had on the article. Using the NER model we have trained on annotated review comments, the author and the decision-maker can better understand the requested changes written by the reviewer. An important step within our model improvement process is to add a user interface where different users of the model can validate and correct the output results. This way, the model can

be constantly improved during its use in order to increase its prediction scores. We can also imagine that once we have a well-trained model, it could also be used by the reviewer while writing the review comments. The model will be used to automatically highlight different named entities and guide the reviewers to improve the writing of their comments. Finally, the correlation between the named entities labelled by our NER model and the jats-diff output could provide a unified source of information where the reviewer/decision-maker can view the requested and actual changes correlated and annotated on the article. A graphical interface while converting the JATS versions of the article to HTML and annotating the actual and requested changes on different locations of the article will have great potential and a great added value in the publishing industry. Providing such novel tools and a user-friendly graphical interface to the reviewers, authors and decision-makers can help MDPI to improve its overall user experience and facilitate the daily tasks of the mentioned parties within the publishing process. Moreover, the risk of human error will be reduced. As another benefit of extracting the requested and actual changes, future work could be carried out which explores the semi-automated peer review process. Having both types of information available, those could be used to train a new deep learning model in order to find context similarity between review requests and author changes. Using unsupervised machine learning, author corrections could be checked depending on the review comments in order to assess if the author fulfilled the requested changes.

Chapter 7

Résumé

7.1 Introduction et Motivation

Depuis plus d'une décennie, le nombre d'articles académiques publiés chaque année est en constante hausse. Chaque article soumis passe par un processus bien défini—voir Figure 7-1 (suivre les traits noir). Nous proposons dans cette thèse plusieurs outils d'automatisation destinés aux différents acteurs du processus de publication académique: les auteurs, les relecteurs et les éditeurs. Pour cela, un travail impliquant deux domaines de recherche a été réalisé: la comparaison de documents et la reconnaissance d'entités nommées (NER). Concernant la partie comparaison de documents, nous proposons un nouvel algorithme de comparaison de document XML appelé jats-diff. Cet algorithme permet d'extraire les changements apportés par l'auteur lors de la révision de l'article. Quant à la partie NER, un réseau de neurones profond a été entrainé, capable d'annoter les commentaires de relecteurs dans le but d'extraire les changements requis. Au final, les informations sur les changements requis et les changements effectifs sont corrélés afin de pouvoir évaluer si l'auteur a bien répondu aux remarques et corrections du relecteur durant la révision. En utilisant ces outils, l'éditeur peut extraire à la fois les changements demandés et les changements apportés par les auteurs et mesurer la corrélation entre les deux afin de prendre en toute conaissance de cause la décision finale concernant l'acceptation de l'article.

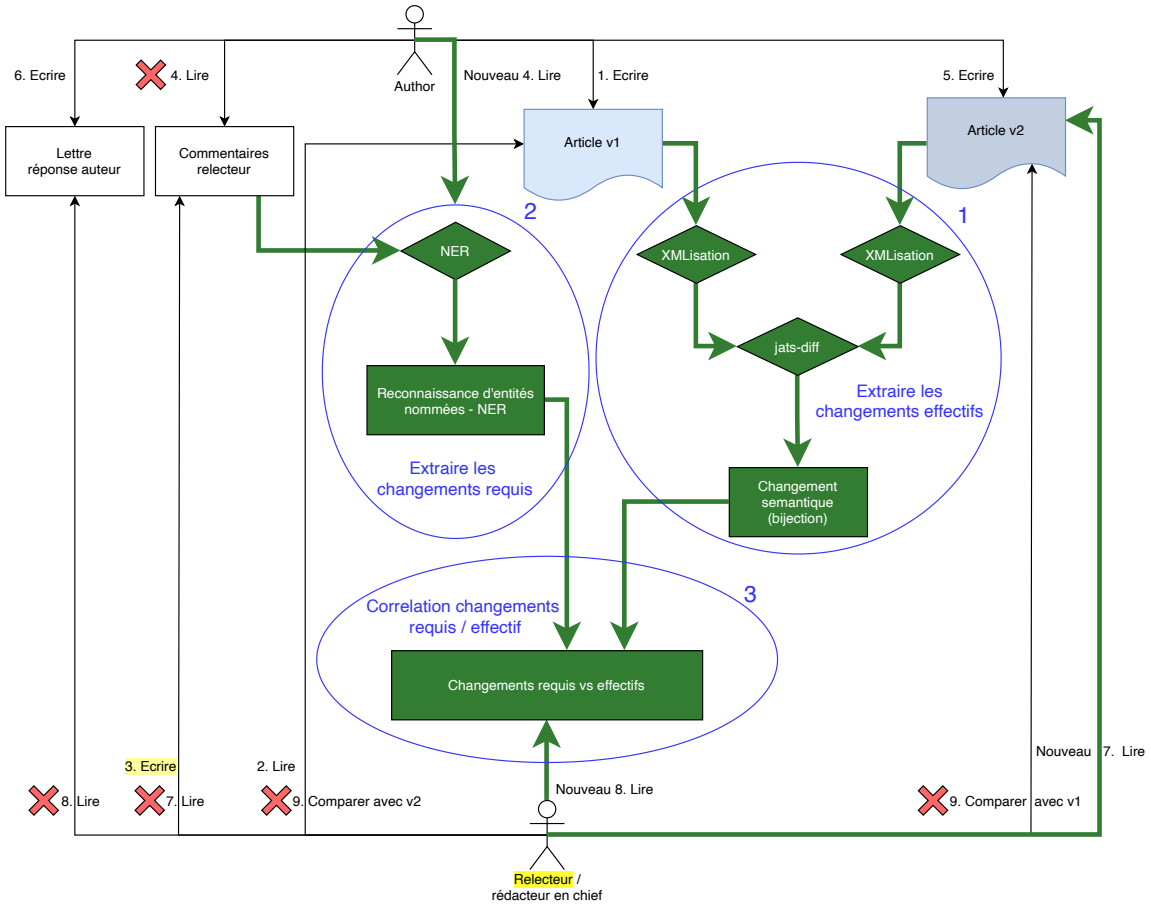


Figure 7-1: Vue d'ensemble sur le processus de publication scientifique amélioré.

Au lieu de consulter la lettre de réponse de l'auteur, les commentaires du relecteur et de comparer les versions de l'article, l'éditeur n'a plus qu'à consulter les informations corrélées sur les changements requis et effectifs afin de prendre la décision finale.

7.2 Contributions

Comme nous l'avons expliqué dans l'introduction, la contributions scientifique de cette thèse est concrétisé sous la forme de trois outils: (1) l'extraction de changements apportés par l'auteur; (2) l'extraction des changements requis; et (3) la corrélation entre les deux. La Figure 7-1 montre leur intégration dans le processus de publication actuel (la partie avec les traits verts).

7.2.1 Comparaison de documents XML - jats-diff

Jats-diff est l'algorithme de comparaison de documents XML que nous avons développé sur la base d'une réinterprétation des résultats de JNDiff. Contrairement aux algorithmes de comparaison existants uniquement capables de détecter des changements de bas niveau (insérer, supprimer, modifier et déplacer), jats-diff est capable de détecter des changements de plus haut niveau qui, de plus, représentent réellement les modifications de l'aauteur. Par exemple, un auteur fusionne trois paragraphes, ce changement est détecté par les algorithmes de comparaison existants comme une modification et deux suppressions de paragraphes. Jats-diff est capable de reconstruire cette fusion depuis la séquence de changements syntaxiques. Ceci donne la possibilité d'établir une bijection entre le changement réel fait par l'auteur et la différence détectée en comparant les deux documents XML.

La Figure 7-2 montre le principe de fonctionnement de jats-diff. L'algorithme produit deux sorties: un document XML avec les modifications: insérer, supprimer, changer l'attribut, promouvoir, rétrograder, fusionner, diviser, déplacer, modifier le style, déplacer et modifier le texte; et un document texte sous forme d'un arbre en intégrant un index de similarité entre les différentes parties de l'article. Ce dernier document, plus facile à lire par un être humain, est produit en analysant les résultats XML afin d'utiliser la sémantique de changement propre au XML JATS.

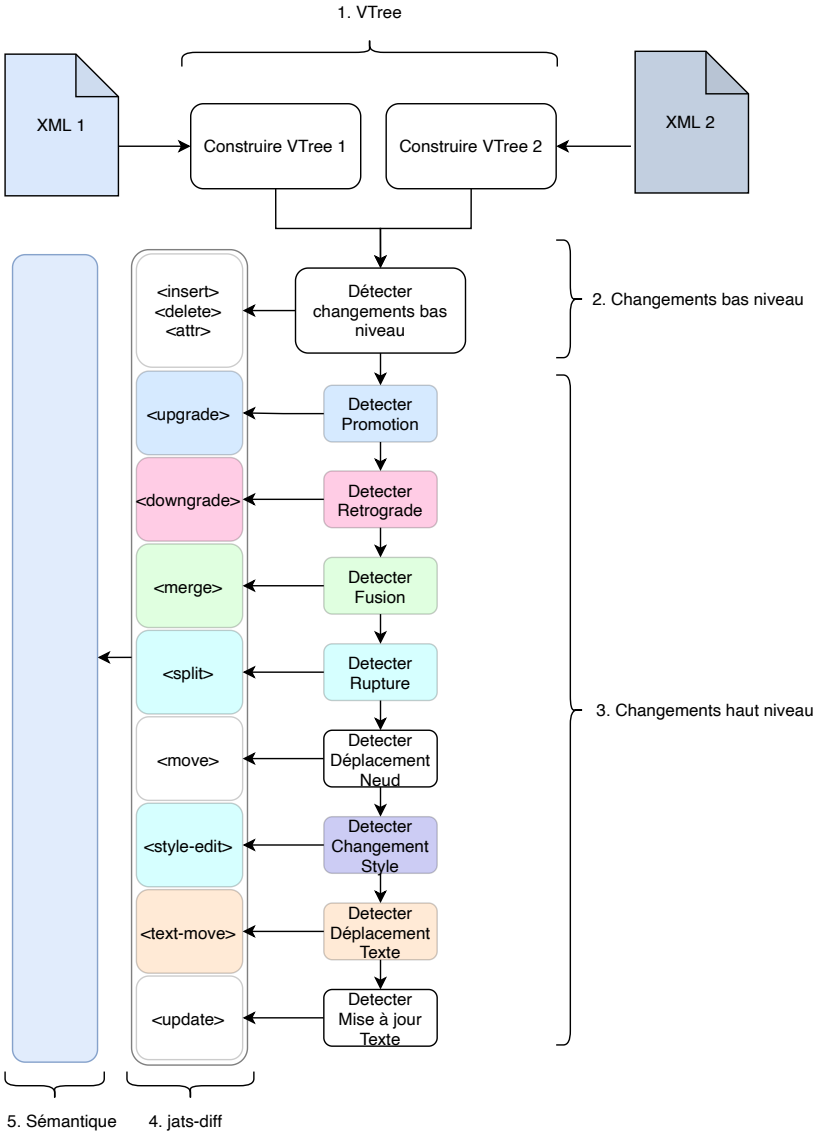


Figure 7-2: Flux de travail jats-diff.

1. Construire l'arbre XML; 2. Détecter les changements bas niveau; 3. Identifier les changements haut niveau parmi les séquences de changements bas niveau; 4. Construire la sortie jats-diff au format XML. 5. Analyser la sortie jats-diff en utilisant la sémantique propre au JATS pour construire une sortie sous la forme d'arbre et calculer l'index de similarité

7.2.2 Reconnaissance d'entités nommées - NER

Le but de cette partie est d'extraire les améliorations souhaitées par un relecteur. Pour cela, nous avons défini 5 classes d'entités nommées: *Location*, *Action*, *Modal*, *Trigger* et *Content*. *Location* est utilisé pour identifier la localisation de l'amélioration souhaitée dans l'article, et par extension où un changement devrait avoir lieu. *Action* est utiliser pour identifier le type de changement (ajouter, supprimer, préciser, reformuler, etc.). *Modal* définit la modalité de changement, c'est à dire si le changement est impératif ou non. *Trigger* représente des mots courts qui nous donnent la possibilité d'affiner le type de changement, de distinguer les questions ou de comprendre des solutions de corrections à choix multiples. Enfin, *Content* représente l'information sur le sujet de changement requis et contrairement à la classe *Action* qui elle définit le type de changement, la classe *Content* décrit en quoi consiste le changement. La Figure 7-3 montre un exemple d'application de NER sur les commentaires de relecteur où le fait d'annoter les différentes classes facilite la compréhension des changements requis.

Figure 7-3: Exemple d'application NER sur un commentaire de relecteur.

Un commentaire de relecteur avec la NER appliqué où nous retrouvons les 5 classes *Location*, *Action*, *Modal*, *Trigger* et *Content*.

Afin de compléter notre tâche NER, nous avons comparés plusieurs modèles de

réseau de neurones profonds: BERT, SciBERT, RoBERTa, DistilBERT et XLNet. Afin de faire un entrainement de ces modèles de manière supervisée, nous avons commencé par construire nos data sets d'entrainement et d'évaluation en extrayant 10 000 commentaires de relecteurs. Ces commentaires sont passés tout d'abord par une phase d'annotations automatiques en utilisant des expressions régulières afin de détecter les action, locations, triggers et modaux. La deuxième passe consistait à faire une annotation manuelle par des éditeurs de MDPI en utilisant un outil d'annotations appelé Doccano[1].

Une fois le data-set prêt, nous avons commencé par une évaluation grossière des 5 modèles en utilisant la technique de grid-search afin d'identifier les differents clusters d'hyper-paramètres où les modèles performent le mieux. Durant cette évaluation, nous avons constaté que les scores de prédiction de la classe *Content* sont plus bas comparé aux autres classes. Ceci est principalement dû à la qualité médiocre des commentaires des éditeurs et à leur très grande variété. Nous avons donc décidé de continuer les expérimentations en retirant la classe *Content*. Avec les différents clusters où les modèles performent le mieux, une évaluation plus fine a été réalisée. Elle nous a permis d'identifier le modèle le plus performant—SciBERT—pour effectuer la tâche NER sur les commentaires des relecteurs, capable d'avoir un F1-score de 0.8665. La performance de ce modèle a été également validée par un groupe d'éditeurs de MDPI qui ont manuellement vérifié la reconnaissance de la classe *Location* sur 1000 commentaires réels avec un succès de 77.1%. Parmi les 229 erreurs rapportées, 148 sont des prédictions manquantes, (le modèle n'a pas annoté la localisation) et 101 sont des mauvaises prédictions (le modèle a annoté une localisation erronée).

7.2.3 Corrélation des changements requis et effectifs

Une fois les informations sur les changements effectués par l'auteur et les changements requis part le relecteur identifiées, nous avons travaillé sur leur corrélation via la *Location* dans l'article. La Figure 7-4 montre un exemple où la classe *Location* des changements requis est corrélées avec la *Location* où les changements effectués ont eu

[1]https://github.com/doccano/doccano

lieu. Cette approche de corrélation est bien plus intuitive pour les éditeurs comparée à une analyse manuelle décrite dans la Figure 7-1.

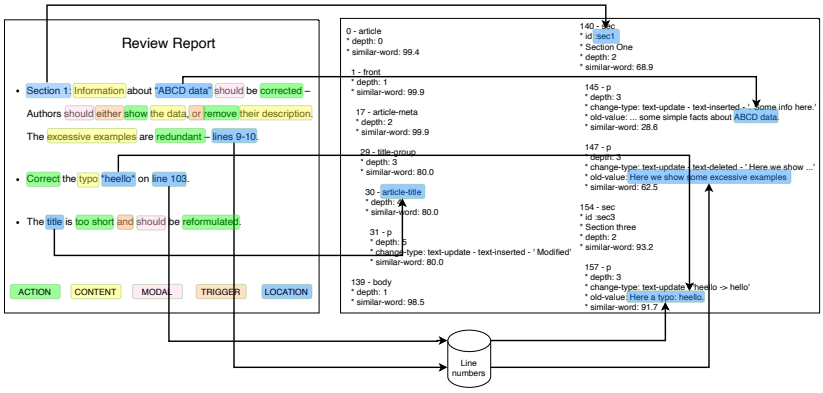


Figure 7-4: Analogies entre la classe *Location* et la sortie jats-diff.

À gauche les changements requis; à droite les différences extraites par jats-diff. Les flèches montrent la corrélation de la classe *Location* avec la location des changements

7.3 Conclusion

En plus de leur activité de recherche, les scientifiques ont un rôle très important dans le processus de publication d'articles où ils sont en charge de l'écriture, de la relecture et de la décision finale. Le but de cette thèse est d'améliorer ce processus en apportant des nouveaux outils quoi vont faciliter les tâches des différents intervenants. La contribution scientifique s'étends sur deux domaines de recherche: la comparaison de documents et la reconnaissance d'entités nommées. En ce qui concerne la comparaison de documents, nous avons travaillé sur la comparaison d'articles au format JATS XML afin d'extraire les changements effectifs apportées par l'auteur pendant une révision. Le nouvel algorithme développé, nommé jats-diff, est capable d'extraire des changements de haut niveau et faire une bijection entre les modifications apportées par l'auteur et les différences détectées entre deux documents XML. Le travail sur la NER a été appliqué sur les commentaires de relecteurs afin d'extraire les changements requis. Plusieurs modèles de réseau de neurones profonds ont été évalués avec un entrainement final du modèle SciBERT qui a obtenu le F1 score de 0.8665. L'utilisation

de différentes classes ont permis de mieux comprendre les requis des relecteurs afin de passer à l'étape finale de la thèse qui consiste à corréler d'un côté les changements effectifs et de l'autre côté les changements requis en utilisant la location où un changement spécifique devrait y avoir lieu comme élément de liaison. En mettant en place ces outils, le travail manuel de l'auteur, du relecteur et de l'éditeur est simplifié. L'auteur peut utiliser l'outil NER afin de mieux comprendre les changements requis. Le relecteur peut utiliser le même outil afin d'améliorer la rédaction de ses commentaires. Au final, l'éditeur peut utiliser l'ensemble des outils afin d'extraire les changements requis, les changements effectifs et avoir une information unique où ces deux types de changement sont corrélées.

7.4 Publications

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "Change Detection on JATS Academic Articles: An XML Diff Comparison Study." In Proceedings of the ACM Symposium on Document Engineering 2020, pp. 1-10. (2020)

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "Semantics to the rescue of document-based XML diff: A JATS case study." Software: Practice and Experience (2022)

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "A JATS XML Comparison Algorithm for Scientific Literature." In: Journal Article Tag Suite Conference (JATS-Con) Proceedings (2022)

- Cuculovic, Milos, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. "Named Entity Recognition to the rescue of academic publishing." Soumis à: ICTAI 2022 Conference - IEEE

Bibliography

[1] Abhishek Abhishek, Ashish Anand, and Amit Awekar. Fine-grained entity type classification by jointly learning representations and label embeddings. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, volume 1, pages 797–807, 2017.

[2] Bogdan Babych and Anthony Hartley. Improving machine translation quality with automatic named entity recognition. In *Proceedings of the 7th International EAMT workshop on MT and other language technology tools, Improving MT through other language technology tools, Resource and tools for building MT at EACL 2003*, 2003.

[3] Gioele Barabucci. Introduction to the universal delta model. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, page 47–56, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Iz Beltagy, Kyle Lo, and Arman Cohan. Scibert: A pretrained language model for scientific text. *arXiv preprint arXiv:1903.10676*, 2019.

[5] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541, 2006.

[6] Indra Budi and Stephane Bressan. Association rules mining for name entity recognition. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.*, pages 325–328. IEEE, 2003.

[7] Sudarshan S Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. *ACM SIGMOD Record*, 26(2):26–37, 1997.

[8] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *Acm Sigmod Record*, 25(2):493–504, 1996.

[9] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[10] Yan Chen, Sanjay Madria, and Sourav Bhowmick. Diffxml: Change detection in xml data. In YoonJoon Lee, Jianzhong Li, Kyu-Young Whang, and Doheon Lee, editors, *Database Systems for Advanced Applications*, pages 289–301, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Pengxiang Cheng and Katrin Erk. Attending to entities for better text understanding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7554–7561, 2020.

[12] Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *Transactions of the Association for Computational Linguistics*, 4:357–370, 2016.

[13] Paolo Ciancarini, Angelo Di Iorio, Carlo Marchetti, Michele Schirinzi, and Fabio Vitali. Bridging the gap between tracking and detecting changes in xml. *Software: Practice and Experience*, 46(2):227–250, 2016.

[14] Greégory Cobena. *Change management of semi-structured data on the Web*. PhD thesis, Institut Polytechnique de Paris, 2003.

[15] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting changes in xml documents. In *Proceedings 18th International Conference on Data Engineering*, pages 41–52, San Jose, CA, USA, 2002, 2002. IEEE.

[16] Grégory Cobéna, Talel Abdessalem, and Yassine Hinnach. A comparative study of xml diff tools, 2004.

[17] Nico Colic, Lenz Furrer, and Fabio Rinaldi. Annotating the pandemic: Named entity recognition and normalisation in covid-19 literature. In *ACL 2020 Workshop on Natural Language Processing for COVID-19 (NLP-COVID)*, 2020.

[18] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[19] Milos Cuculovic, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. Change detection on jats academic articles: An xml diff comparison study. In *Proceedings of the ACM Symposium on Document Engineering 2020*, pages 1–10, 2020.

[20] Leyang Cui, Yu Wu, Jian Liu, Sen Yang, and Yue Zhang. Template-based named entity recognition using bart. *arXiv preprint arXiv:2106.01760*, 2021.

[21] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[22] Pieter-Tjerk De Boer, Dirk P Kroese, Shie Mannor, and Reuven Y Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[24] Rodrigo Cordeiro Dos Santos and Carmem Hara. A semantical change detection algorithm for xml. *SEKE 2007*, page 438, 2007.

[25] Jianguang Du, Jing Jiang, Dandan Song, and Lejian Liao. Topic modeling with document relative similarities. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[26] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview. *arXiv preprint arXiv:2008.05756*, 2020.

[27] Ralph Grishman and Beth M Sundheim. Message understanding conference-6: A brief history. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*, 1996.

[28] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[29] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[30] Daniel Hottinger and Franziska Meyer. Xml-diff-algorithmen. Thesis, 2005.

[31] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[32] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, USA, 1976.

[33] Paul Jaccard. Distribution of the alpine flora in the dranse's basin and some neighbouring regions. *Bulletin de la Societe vaudoise des Sciences Naturelles*, 37(1):241–272, 1901.

[34] Derry Jatnika, Moch Arif Bijaksana, and Arie Ardiyanti Suryani. Word2vec model analysis for semantic similarities in english words. *Procedia Computer Science*, 157:160–167, 2019. The 4th International Conference on Computer Science and Computational Intelligence (ICCSCI 2019) : Enabling Collaboration to Escalate Impact of Research Results for Society.

[35] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[36] Robin La Fontaine. Standard change tracking for xml. In *Balisage: The Markup Conference*, pages 5–8, 2014.

[37] Kai Labusch and Clemens Neudecker. Named entity disambiguation and linking historic newspaper ocr with bert. In *CLEF (Working Notes)*, 2020.

[38] Anurag Lal et al. Sane 2.0: System for fine grained named entity typing on textual data. *Engineering Applications of Artificial Intelligence*, 84:11–17, 2019.

[39] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[40] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. *arXiv preprint arXiv:1901.07291*, 2019.

[41] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[42] Falk Langhammer. Bauen statt modellieren. *iX*, 2:100–103, 2004.

[43] Bjornar Larsen. A trainable summarizer with knowledge acquired from robust nlp techniques. *Advances in automatic text summarization*, 71, 1999.

[44] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[45] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[46] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[47] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Fast and simple xml tree differencing by sequence alignment. In *Proceedings of the 2006 ACM Symposium on Document Engineering*, DocEng '06, page 75–84, New York, NY, USA, 2006. Association for Computing Machinery.

[48] Xiao Ling and Daniel S Weld. Fine-grained entity recognition. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[49] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[50] Lorenz Schori. Delta.js - a javascript diff and patch engine for dom trees, 2020.

[51] Hans Peter Luhn. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4):309–317, 1957.

[52] Khai Mai, Thai-Hoang Pham, Minh Trung Nguyen, Tuan Duc Nguyen, Danushka Bollegala, Ryohei Sasano, and Satoshi Sekine. An empirical study on fine-grained named entity recognition. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 711–722, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics.

[53] Salvatore Manfreda, Matthew F. McCabe, Pauline E. Miller, Richard Lucas, Victor Pajuelo Madrigal, Giorgos Mallinis, Eyal Ben Dor, David Helman, Lyndon Estes, Giuseppe Ciraolo, Jana Müllerová, Flavia Tauro, M. Isabel De Lima, João L. M. P. De Lima, Antonino Maltese, Felix Frances, Kelly Caylor, Marko Kohv, Matthew Perks, Guiomar Ruiz-Pérez, Zhongbo Su, Giulia Vico, and Brigitta Toth. On the use of unmanned aerial systems for environmental monitoring. *Remote Sensing*, 10(641):1–28, 2018.

[54] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, USA, 2008.

[55] Mónica Marrero, Julián Urbano, Sonia Sánchez-Cuadrado, Jorge Morato, and Juan Miguel Gómez-Berbís. Named entity recognition: fallacies, challenges and opportunities. *Computer Standards & Interfaces*, 35(5):482–489, 2013.

[56] Andrei Mikheev, Claire Grover, and Marc Moens. Description of the ltg system used for muc-7. In *Seventh Message Understanding Conference (MUC-7): Proceedings of a Conference Held in Fairfax, Virginia, April 29-May 1, 1998*, 1998.

[57] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[58] Webb Miller and Eugene W Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.

[59] Diego Mollá, Menno Van Zaanen, Daniel Smith, et al. Named entity recognition for question answering. In *Australasian language technology workshop 2006*. Carlton, Vic: Australasian Language Technology Association, 2006.

[60] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[61] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[62] Zara Nasar, Syed Waqar Jaffry, and Muhammad Kamran Malik. Named entity recognition and relation extraction: State-of-the-art. *ACM Computing Surveys (CSUR)*, 54(1):1–39, 2021.

[63] Norman Walsh. Diffmk, 2015.

[64] Alessandreia Oliveira, Leonardo Murta, and Vanessa Braganholo. Towards semantic diff of xml documents. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 833–838, 2014.

[65] Alessandreia Oliveira, Gabriel Tessarolli, Gleiph Ghiotto, Bruno Pinto, Fernando Campello, Matheus Marques, Carlos Oliveira, Igor Rodrigues, Marcos Kalinowski, Uéverton Souza, et al. An efficient similarity-based approach for comparing xml documents. *Information Systems*, 78:40–57, 2018.

[66] Matthew E Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. *arXiv preprint arXiv:1705.00108*, 2017.

[67] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations, 2018.

[68] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[69] Xiang Ren, Wenqi He, Meng Qu, Lifu Huang, Heng Ji, and Jiawei Han. Afet: Automatic fine-grained entity typing by hierarchical partial-label embedding. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 1369–1378, 2016.

[70] Sebastian Rönnau and Uwe M Borghoff. Versioning xml-based office documents. *Multimedia Tools and Applications*, 43(3):253–274, 2009.

[71] Sebastian Rönnau and Uwe M Borghoff. Xcc: change control of xml documents. *Computer Science-Research and Development*, 27(2):95–111, 2012.

[72] Sebastian Rönnau, Christian Pauli, and Uwe M. Borghoff. Merging changes in xml documents using reliable context fingerprints. In *Proceedings of the Eighth ACM Symposium on Document Engineering*, DocEng '08, page 52–61, New York, NY, USA, 2008. Association for Computing Machinery.

[73] Sebastian Rönnau, Geraint Philipp, and Uwe M. Borghoff. Efficient change control of xml documents. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, page 3–12, New York, NY, USA, 2009. Association for Computing Machinery.

[74] Sebastian Rönnau, Jan Scheffczyk, and Uwe M. Borghoff. Towards xml version control of office documents. In *Proceedings of the 2005 ACM Symposium on Document Engineering*, DocEng '05, page 10–19, New York, NY, USA, 2005. Association for Computing Machinery.

[75] Richard J Rossi. *Mathematical statistics: an introduction to likelihood based inference*. John Wiley & Sons, 2018.

[76] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[77] Erik F Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

[78] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019.

[79] David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

[80] Stanley M Selkow. The tree-to-tree editing problem. *Information processing letters*, 6(6):184–186, 1977.

[81] Dennis Shasha and Kaizhong Zhang. Fast parallel algorithms for the unit cost editing distance between trees. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 117–126, 1989.

[82] Yu Song, Eunji Yi, Eunju Kim, Gary Geunbae Lee, and Soo-Jun Park. Posbiotm-ner: a machine learning approach for bio-named entity recognition. *Korea*, 305:350, 2004.

[83] Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. Portuguese named entity recognition using bert-crf. *arXiv preprint arXiv:1909.10649*, 2019.

[84] Rohini K Srihari. A hybrid approach for named entity and sub-type tagging. In *Sixth Applied Natural Language Processing Conference*, pages 247–254, 2000.

[85] Shilpi Srivastava, Mukund Sanglikar, and DC Kothari. Named entity recognition system for hindi language: a hybrid approach. *International Journal of Computational Linguistics (IJCL)*, 2(1):10–23, 2011.

[86] Cong Sun and Zhihao Yang. Transfer learning in biomedical named entity recognition: An evaluation of bert in the pharmaconer task. In *Proceedings of The 5th Workshop on BioNLP Open Shared Tasks*, pages 100–104, 2019.

[87] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

[88] Walter F Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems (TOCS)*, 2(4):309–321, 1984.

[89] WM To and BTW Yu. Rise in higher education researchers and academic publications [version 1; peer review: 2 approved]. *Emerald Open Research*, 2(3), 2020.

[90] Antti Virtanen, Jenna Kanerva, Rami Ilo, Jouni Luoma, Juhani Luotolahti, Tapio Salakoski, Filip Ginter, and Sampo Pyysalo. Multilingual is not enough: Bert for finnish. *arXiv preprint arXiv:1912.07076*, 2019.

[91] W3C. Extensible markup language (xml), 2016.

[92] W3C. Htmldiff, 2018.

[93] W3C. Htmldiff, 2019.

[94] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[95] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, 2022.

[96] Qingyun Wang, Qi Zeng, Lifu Huang, Kevin Knight, Heng Ji, and Nazneen Fatema Rajani. Reviewrobot: Explainable paper review generation based on knowledge synthesis. *arXiv preprint arXiv:2010.06119*, 2020.

[97] Yuan Wang, David J DeWitt, and J-Y Cai. X-diff: An effective change detection algorithm for xml documents. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 519–530, Bangalore, India, 2003. IEEE.

[98] Leon Weber, Mario Sänger, Jannes Münchmeyer, Maryam Habibi, Ulf Leser, and Alan Akbik. Hunflair: an easy-to-use tool for state-of-the-art biomedical named entity recognition. *Bioinformatics*, 37(17):2792–2794, 2021.

[99] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[100] Mengge Xue, Weiming Cai, Jinsong Su, Linfeng Song, Yubin Ge, Yubao Liu, and Bin Wang. Neural collective entity linking based on recurrent random walk network learning. *arXiv preprint arXiv:1906.09320*, 2019.

[101] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[102] Bowen Yu, Zhenyu Zhang, Tingwen Liu, Bin Wang, Sujian Li, and Quangang Li. Beyond word attention: Using segment attention in neural relation extraction. In *IJCAI*, pages 5401–5407, 2019.

[103] Weizhe Yuan, Pengfei Liu, and Graham Neubig. Can we automate scientific reviewing? *arXiv preprint arXiv:2102.00176*, 2021.

[104] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[105] Shuohao Zhang, Curtis Dyreson, and Richard T Snodgrass. Schema-less, semantics-based change detection for xml documents. In *International Conference on Web Information Systems Engineering*, pages 279–290. Springer, 2004.

[106] Zhengyan Zhang, Xu Han, Zhiyuan Liu, Xin Jiang, Maosong Sun, and Qun Liu. Ernie: Enhanced language representation with informative entities. *arXiv preprint arXiv:1905.07129*, 2019.

Appendix A

jats-diff delta output examples

Insert: addition of a new keyword "test keyword"

```
<insert at="373" nodecount="1" nodenumberB="388" pos="7">
    <kwd>test keyword</kwd>
</insert>
```

Delete: removal of an existing keyword "river monitoring"

```
<delete nodecount="2" nodenumberA="386">
    <kwd>river monitoring</kwd>
</deletes
```

Attribute Update: section 6 id change from "sec6" to "sec6dot1"

```
<update-attribute name="id" newvalue="sec6dot1" nodenumberA="38" nodenumberB="36"
     oldvalue="sec6" op="change-attr"/>
```

Section Upgrade: Section 2.3 that is upgraded as Section 6

```
<upgrade at="388" nodecount="5" nodenumberB="601" op="upgradedTo" pos="5">
    <sec id="sec6-remotesensing-10-00641"/>
</upgrade>
<upgrade nodecount="4" nodenumberA="447" op="upgradedFrom">
    <sec id="seczdot3-remotesensing-10-00641"/>
</upgrade>
```

Section Downgrade: Section 5 that is downgraded to Section 2.4

```
<downgrade at="410" nodecount="89" nodenumberB="452" op="downgradedTo" pos="5">
    <sec id="sec2dot4-remotesensing-10-00641"/>
</downgrade>
<downgrade nodecount="88" nodenumberA="517" op="downgradedFrom">
    <sec id="sec5-remotesensing-10-00641"/>
</downgrade>
```

Paragraph Merge: merge two paragraphs into one

```
<merge at="0" direction="9:9" nodenumberB="9" op="mergedTo" pos="4">
        <p>Text paragraph four with some additional text of paragraph five</p>
</merge>
<merge direction="9:9" nodenumberA="9" op="mergedFrom">
        <p>Text paragraph four</p>
</merge>
<merge direction="11:9" nodenumberA="11" op="mergedFrom">
        <p>with some additional text of paragraph five</p>
</merge>
```

Paragraph Split: split one paragraph into two different paragraphs

```
<split at="0" direction="9:9" nodenumberB="9" op="splitedTo" pos="4">
        <p>Text paragraph four</p>
</split>
<split at="0" direction="9:11" nodenumberB="11" op="splitedTo" pos="5">
        <p>with some additional text of paragraph five</p>
</split>
<split direction="9:9" nodenumberA="9" op="splitedFrom">
        <p>Text paragraph four with some additional text of paragraph five</p>
</split>
```

Move: move one keyword from its initial position 2 to position 4

```
<move move="376::378" nodecount="2">
    <kwd>remote sensing</kwd>
</move>
```

Style insert: insert bold style around the word "disasters"

```
<text-style-insert nodenumberB="117" op="insert-style" pos="319">
    <b>disasters</b>
</text-stvle-insert>
```

Style delete: remove bold style around the word "monitoring"

```
<text-style-delete nodenumberA="117" op="delete-style" DOS="14">
    <b>monitoring</b>
</text-style-delete>
```

Style edit: Moving one keyword from his initial position 2 to position 4

```
<text-style-update nodenumberB="117" op="update-style-to" pos="14">
    <i>monitoring</i>
</text-style-update>
<text-style-update nodenumberA="117" op="update-style-from" pos="14">
    <b>monitoring</b>
</text-style-update>
```

Text Move: move portion of thext from one paragraph to another

```
<text-move nodecount="1" nodenumberB="6" op="movedTo" text-position-to="19">text of
     paragraph five</text-move>
<text-move nodecount="1" nodenumberA="12" op="movedFrom" text-position-from="21">text of
     paragraph five</text-move>
```

Text Update: change one word from "central" to "centralised"

```
<text-update length="7" nodenumberA="368" nodenumberB="368" op="text-deleted"
     pos="33">central</text-update>
<text-update length="11" nodenumberA="368" nodenumberB="368" op="text-inserted"
     pos="33">centralised</text-update>
```

Appendix B

jats-diff semantics output examples

Citable objects: insertion of a new bibliography in the references list <ref-list>

```
0 - article
    * depth: 0
    * similar-word: 99.9

      606 - back
      * depth: 1
      * similar-word: 99.8

        1932 - ref-list
        * Initial: 156
        * Final: 157
        * depth: 2
        * similar-word: 99.7

          7993 - ref
          * id :B153-remotesensing-10-00641
          * depth: 3
          * change-type: insert
```

Being inserted at position 153 (out of 156), the reference insert edit action produces additional induced changes. While the delta XML shows for this simple bibliography insert over a dozen of different edit actions: text updates, inserts, deletes and attribute updates, the semantics output file is ignoring those induced edits and represents the change in a human readable way.

jats-diff considers as special objects: tables, bibliographies, figures and mathematical formulas. Those objects are special due to the fact that their edits are not human readable and have to be represented in a different way: for tables (as seen in

the previous example) the similarity index is calculated per table content and caption.

Element lists and special objects: modification of a table.

```
0 - article
* depth: 0
* similar-word: 100.0

  3977 - back
  * depth: 1
  * similar-word: 99.8

    6312 - sec
    * id :sec-type="display-objects"
    * depth: 2
    * similar-word: 99.5

    6313 - table
    * Initial: 6
    * Modified: 1
    * Final: 6
    * depth: 2
    * similar-word: 99.6

      6895 - table-wrap
      * id :remotesensing-12-02506-t006
      * depth: 3
      * change-type: table-edit
        * table
            * 66.7
        * caption
            * 98.0
      * similar-word: 89.9
```

Most of the special objects (tables, figures) but also bibliographies are within so-called element lists. jats-diff is also using change semantics in order to represent those in a more readable way. In the previous example where a table edit is shown, we can observe that the entire table list is shown with its initial, modified and final values. The initial value represents the total number of tables in document A, the modified value represents the number of edited tables and the final value represents the total number of tables in document B.

Similarity index: text edits in the Introduction and the Section 2.2

```
0 - article
* depth: 0
* similar-word: 99.5

  155 - body
  * depth: 1
  * similar-word: 99.3
```

```
156 - sec
* id :sec1-remotesensing-12-02506
* 1. Introduction
* depth: 2
* similar-word: 99.9

  169 - p
  * depth: 3
  * change-type: text-update - text-inserted
  * similar-word: 99.3

175 - sec
* id :sec2-remotesensing-12-02506
* 2. Materials
* depth: 2
* similar-word: 96.2

  199 - sec
  * id :sec2dot2-remotesensing-12-02506
  * 2.2. Imagery Surveyed from UAV and Satellites
  * depth: 3
  * similar-word: 77.8

    202 - p
    * depth: 4
    * change-type: delete
```

As seen in the previous examples, the similarity index is calculated and propagated through the JATS XML tree. jats-diff allows to calculate different similarity indexes: similartext, similartext-word, Jaccard, and TFIDF. By default, the similartext-word index is used that allows the reader to get insights about lexical changes.