University of Zurich[UZH]

The Argument Annotator Pipeline - Generate Visually Annotated Documents

BSc Thesis October 12, 2021

Joel Watter
of Bad Zurzach AG,
Switzerland

Student-ID: 17-719-303
joel.watter@uzh.ch

Advisor: Florian Ruosch

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to express my sincerest gratitude to my supervisor Florian Ruosch for his immense support and feedback during the process of this thesis. I would also like to thank Professor Abraham Bernstein, Ph.D. for the opportunity to conduct my Bachelor thesis at the Dynamic and Distributed Information Systems Group of the University of Zurich. Last, I would like to thank my family, my friends and my colleagues for the support they gave me during the time of this thesis.

Zusammenfassung

Die Forschung zu argumentativen Strukturen in natürlichen Texten entwickelt sich kontinuierlich weiter. Eine perfekte Methode zur Modellierung, Annotation und Identifikation argumentativer Strukturen muss aber noch gefunden werden. Qualitativ hochwertige Annotationskorpora werden in komplexer und zeitaufwändiger Handarbeit erstellt, um als Daten Grundlage für das Trainieren, Testen und Verbessern von automatisierten Argument Mining Tools zu dienen. Der Wert, den solche Korpora für eine Maschine haben, steht ausser Frage. Aber die Tatsache, dass die referenzierten, argumentativen Strukturen in der Annotationsdatei des Korpus vollständig von ihrem eigentlichen Kontext im Originaltext getrennt sind, erschwert es einem menschlichen Leser, auf einer ähnlichen Ebene von den darin enthaltenen Informationen zu profitieren. In dieser Arbeit widmen wir uns diesem Problem und implementieren ein Werkzeug, um visuell annotierte PDF-Dokumente aus Korpusdaten zu erzeugen. Die resultierenden Dokumente unterstützen den menschlichen Leser dabei, die sichtbaren Annotationen und die dargestellten Beziehungen zu anderen Annotationen im Text, verstehen und nachvollziehen zu können. Durch das Anhängen und Einbetten der ursprünglichen Text- und Annotationsdateien und der, während des Generierungsprozesses erstellten, Annotationsstruktur in unsere Dokumente, werden diese PDF-Dokumente zu einer Komplettlösung in einer Datei. Zur Validierung unseres Konzepts haben wir einen Beispielkorpus mit unserem Tool prozessiert.

Abstract

The research on argumentation in natural text is evolving, but a perfect way to model, annotate and mine argumentative structures is yet to be found. High-quality annotation corpora are created in complex and time consuming manual work, to represent annotations for the training, testing and improvement of automated Argument Mining tools. The value such corpora have for a machine is out of question. But the fact, that the referenced argumentative structures in the annotation file of the corpus are completely separated from their actual context, within their original text, makes it difficult for a human reader to benefit on a similar level from the data they incorporate. In this thesis, we address that problem and implement a tool to generate visually annotated PDF documents from corpus data. The produced documents support human readers to understand and comprehend the visible annotations and the presented relationships they have to other annotations within the text. Attaching and embedding the original text and annotation files as well as the annotation structure, created during the creation process to our documents, makes these PDF documents to an all in one file solution. As proof of our concept, we processed an example corpus with our tool.

Contents

1 **Introduction** 1

2 **Related Work** 5
2.1 The Argument Web . . . 5
2.2 Ontology and Models . . . 7
2.3 Argument Mining . . . 8

3 **Design** 11
3.1 Motivation and Idea . . . 11
3.2 Ontology Engineering . . . 12
3.3 Software Development . . . 13
3.3.1 Module Phase . . . 13
3.3.2 Connection Phase . . . 14
3.3.3 Test Phase . . . 14

4 **Implementation** 17
4.1 Pipeline Overview . . . 17
4.2 Run the Pipeline . . . 18
4.3 Configurations . . . 18
4.4 Generate the Meta Data Structure . . . 20
4.5 Create the LaTeX-File . . . 23
4.5.1 Colour Definition . . . 23
4.5.2 Custom Commands . . . 24
4.5.3 Write the LaTeX File . . . 25
4.5.4 Command Embedding . . . 26
4.5.5 Unicode Conversion . . . 27
4.6 Compile the LaTeX File . . . 28
4.7 PDF Attachments . . . 29
4.8 The Extraction . . . 31
4.8.1 The Extraction Configuration . . . 31
4.8.2 Read the Extraction Configuration . . . 32
4.8.3 Attachment Retrieval . . . 32
4.8.4 Generate a Custom Annotation File . . . 33

5 Limitations and Future Work **37**
5.1 Limitations 37
5.2 Future Work 38

6 Lessons Learned **41**
6.1 Knowledge Acquisition 41
6.2 Technologies 42
6.2.1 LaTeX 42
6.2.2 PDF 42

7 Conclusions **45**

A Appendix **49**
A.1 User Guide 49
A.2 Run the Pipeline 51
A.3 Installation 52

1

Introduction

Research is done every day in every kind of field. Exciting findings are made and prior assumptions or hypotheses are being accepted or rejected. Today the distribution and gathering of knowledge and information is as easy as it gets, with the World Wide Web being accessible almost everywhere at any time and so are scientific documents. While this might seem as good as it gets, we currently miss out on a huge opportunity to inter-connect knowledge and generate a tremendous increase of its overall value because as of now, the documents and web pages on the Internet have no (machine comprehensible) semantic coherence. In every paper, news article, or discussion on a forum, people use argumentation to express their opinion, to prove their point, or to underpin why something is true or false. This is done on every topic and across many different platforms. If someone wants to write a thesis or paper on a specific topic, a lot of research with a time-consuming manual search is necessary to gather as much information as possible about the topic of interest. While the progression of the World Wide Web certainly simplified and expedited this process, finding all relevant and related information is still a tedious task. The main approach is to gather the most relevant papers first and then manually explore the references for their content and their respective references and so on. The World Wide Argument Web, as proposed by Rahwan et al. [2007], could potentially be helpful in the development of tools that simplify and support the acquisition relevant research documents. Showing the possibility to inter-connect arguments that exist in a structured manner to build a large-scale knowledge repository, the World Wide Argument Web could potentially also embody references to documents with specified and structured arguments. Once established, this knowledge repository can be used to search for Argumentative structures on a semantic and relation-based level. This vision requires large quantities of well-structured argument annotation data [Lawrence and Reed, 2019].

The work of Toulmin [1958] is still relevant concerning the process and impact of Argumentation and gives the baseline of how the representation of Arguments is done. Nowadays, various approaches to model and represent Arguments are applied [Al Khatib et al., 2021]. Nevertheless, researchers have not yet developed one perfect way to mine or create the Argument data of scientific publications widely and across specific applications and domains [Al Khatib et al., 2021].

Al Khatib et al. [2021] mention in their work the importance of the context-dependence of argumentative structures. Documents are written for a human audience and as a hu-

man, we have the ability to conclude certain things from the context of what we read. We know how to communicate and argue, can grasp the meaning from context and put semantics to what we read. For a machine, it is the exact opposite. Files and documents are to a machine just what they effectively are, bits and bytes. A machine needs to extract and annotate Argumentative structures or have this information available in the processing step to make sense out of a file. One way of introducing such structures to a text file is to create annotations while writing a text. Groza et al. [2007] have developed the tool Semantically Annotated LaTeX (SALT), which allows to create rich annotations directly in LaTeX. They provide a framework to simplify and encourage argument annotation in the writing process among other functionalities possible with their tool. If the annotation of Arguments becomes a standard task in scientific writing, all newly published documents will carry their respective data. But we also want to benefit from documents that already have been published. Annotating existing documents is either done manually by humans or automatically with Argument Mining.

As seen in the work of Lawrence and Reed [2019], the field of Automated Argument Mining is evolving. The precision of such systems is increasing, but there persists a gap in accuracy and performance from the automated extraction of Argumentative Structures to manual, human-made, annotations as can be seen from the work of Stab and Gurevych [2014]. Especially the recognition of relations among the Argumentative Components is complex, also for humans [Lippi and Torroni, 2015], and due to the context dependence and semantic [Al Khatib et al., 2021]. With further improvements in the area of Automated Argument Mining in the future, a big amount of data can be processed and made machine understandable in a short amount of time. As for now, the most exact and qualitative Argument Annotations are manually created Argument Corpora like the one from Lauscher et al. [2018b]. Besides the fact that a manual annotation is very time consuming and complex [Lippi and Torroni, 2015], the created corpus at the end, consisting of plain text files and structured annotation files, is in no way directly connected to its original documents and has to be provided additionally.

We believe this circumstance could be improved to increase the value of such corpora for human readers as well as for visions like the World Wide Argument Web described by Bex et al. [2013] and more formally by Rahwan et al. [2007]. The goal of this thesis is to use high-quality corpus data to automatically generate documents with visual annotations to enable readers to see argumentative components within their context. Annotations, as well as the Argument Structure with its relations, are embedded directly into the documents to have all information within one file. As we see the PDF file format as the de facto standard for the exchange of immutable documents and inspired by the work of Groza et al. [2007], we implement a pipeline to generate a PDF file from a corpus. We use LaTeXto visualise the annotations within the text and embed the created Annotation data structure within the metadata of the PDF file structure. Documents with their Annotations embedded give the opportunity to reconstruct the original corpus data. Working with a simple and generic Ontology to model our Argument Data we aim to provide good re-usability in terms of the capability to adapt to different domains and use cases. We created our Ontology by adapting the basic principles of the Claim and Premise Model from Walton [2009].

As the development of the Semantic Web, the World Wide Argument Web, and the Argument Mining and Annotation progresses, we hope that in the future every published paper or thesis will carry its argument data. This will allow for huge inter-connections of the Arguments and their relations among and across documents and research fields. In that way, related work to a topic of interest can be found in minutes, with up-to-date research results and cutting-edge tools available, potentially supporting research in many fields. If this kind of information can be generated and collected in large numbers, new opportunities will emerge. Machines could be able to search for connections between areas of research that may not appear to be closely related at first glance, potentially providing an impulsion for new and previously overlooked approaches and connections in any type of research.

This work is structured as follows: First, the related work to our topic will be outlined in Chapter 2. From there, we first describe our work and planning prior to the implementation process in Chapter 3 followed by the presentation of the implementation of the pipeline itself in Chapter 4. We will further look at the limitations of our work and what future work can be done with or to improve our work in Chapter 5. Finally, the lessons learned will highlight the eye-openers that occurred and what has been learned during the research and the implementation of the pipeline in Chapter 6, followed by our conclusions in Chapter 7. How to use and install our pipeline is elaborated within the Appendix A.

2

Related Work

Argument Mining and Annotation and natural language processing in general is a fast-evolving field with many exciting applications. New approaches, ideas and publications are being created at a fast pace [Bex et al., 2013]. This section is first going to cover the promising possibilities of the Argument Web, which is a part of the Semantic Web context. The ontology and models used in existing applications and potential candidates to (re)use for a pipeline are highlighted in the second section. Finally, the related work in the area of argument mining is shown, its challenges and different approaches that already have been formulated for automated argument annotations.

2.1 The Argument Web

Rahwan et al. [2007] state in their work that one of the most important components for the Argument Web are the arguments themselves. According to them, arguments are found everywhere on the Web. In blogs, discussion forums or on news sites, opinions and viewpoints with embedded argumentative structure are presented and exchanged every day on the Web. The Argument Web is described as the tool that manages on a large scale the inter-connected arguments that are posted on the Web in a structured way. They support their description with a motivational example. When searching the Argument Web with a question you receive arguments related to your question, ordered according to their respective strength in the given context. These arguments are connected to the respective scientific paper or a blog where they were presented [Rahwan et al., 2007]. Bex et al. [2013] say that with the continuously rising digital communication, new publications, opinions and discussions on various topics are published and made by the minute. Their work implies that new technologies are needed to cope with the vast amount of data produced. To connect and increase the value of the data at hand the Argument Web was introduced as a vision for the not so far future. Bex et al. [2013] show the immense potential and also some requirements that need to be met to accomplish such a system. On the internet exist uncountable platforms where people state their opinions, discuss different viewpoints and share thoughts and most of the time only a hand full of people get to take part in such an exchange. A lack of interconnection between all the different media platforms where these arguments are stated makes it nearly impossible to capture an adequate picture of all the available information say

Bex et al. [2013] and they further point out the importance of an enhanced exchange and eased way to connect opinions of participants on the Web. Further, they criticise how today's online communication is too strongly focused on an entertainment aspect rather than on actually valuable information on a certain topic. They also mention that advanced communication types, like agree or disagree to a statement, are most often not available and the all too common "like" is not only ambiguous but also discourages oppositional engagement in a discussion. According to them, the overall goal of the Argument Web is to make the Internet intelligent in a way. Discussions and opinions shall be track-able over numerous otherwise not connected pages, complex questions could be answered with rated, structured and weighted arguments and related fields, topics and underlying questions and discussions can be suggested. To make the realisation of the Argument Web possible one day, Rahwan et al. [2007] worked out a detailed first look at the requirements that need to be fulfilled from a theoretical as well as a technical perspective. They point out the importance of the arguments themselves. To be able to classify and analyse arguments and their relations among them on a large scale, predefined corpora with highly structured argumentation schemes are required to support the future development of tools to search, navigate and visualise connected arguments. Following are the five key requirements introduced by Rahwan et al. [2007] that the World Wide Argument Web (WWAW) needs to address. First, it is necessary that the WWAW is able to handle the basic data operations for argumentative structures like storing, creating, updating and querying. As a next point comes the accessibility of the repositories from the WWAW over the Web. Further, an open standard must be used to support and encourage the development of new applications and tools. Hand in hand with the open standard go the last two points, first concerning the utilised ontology that it must be as unified and extensible as possible and secondly the capability of the WWAW to handle the creation, the annotation and the representation of arguments, no matter from which argumentation scheme they originated.

Rahwan et al. [2007] take the Argument Interchanged Format (AIF) as their core ontology and extended it with the Resource Description Framework (RDF) to the AIF-RDF ontology. They also state that the value of the WWAW scales with the number of participants that contribute arguments, interact and react to the existing arguments. Their pilot system, the ArgDF, provides the capability not only to create new arguments but also to query, manipulate and react in a supporting or attacking manner to existing arguments.

Another area of research that is of great importance to this thesis is the annotation of Arguments in text. Groza et al. [2006, 2007] developed the SALT framework to annotate arguments directly in LaTeX, visualising the annotations and storing the respective annotation data within the Extensible Metadata Platform (XMP) of the resulting PDF file. In addition, they implemented a publishing process tool, the SALT-WebPub, where annotated PDFs are taken as input to transform them into HTML files with their respective metadata. They present a framework with a big value to the field of semantic documents and with this to the Semantic Web and the Argument Web. Further, they state that they want to establish a standard with their application to enhance the number of human and machine understandable documents in PDF format. While their

framework is of great use to make annotations during the process of writing, it gives no capabilities to generate such rich annotated PDF documents from existing corpora.

2.2 Ontology and Models

To make raw argument data useful in a Semantic Web context and to be able to connect them in specific ways among each other, ontologies are needed to define how data is modelled and structured and what interactions and relations are possible among the existing entities [Zhou et al., 2019]. Al Khatib et al. [2021] state, that many studies concerned to create new models to represent argument data base their work on the argument model introduced by Toulmin [1958]. Al Khatib et al. [2021] state, that Toulmin [1958] models arguments in form of *claims* that can be supported by *data*. The *data* itself follows warrants and these can again be supported by *backing*. *Qualifiers* and *Rebuttals* form optional components of the model. Lauscher et al. [2018b] as well used the model of Toulmin [1958] as a starting point for their proposed annotation scheme and have built an argument-annotated corpus specifically for scientific writing in English. Lauscher et al. [2018b] and Rahwan and Reed [2009] say that there is already a big number of different models and also ontologies but the main issue is that they are very limited in their flexibility and only work for specific domains and often require their respective application to be of actual use. One of the core challenges that Al Khatib et al. [2021] mention in their work concerns the task of choosing the right model for the argumentation and further the right ontology to connect them. They say that there are numerous different models as seen in the work of Stede et al. [2018] and to choose the right one is not easily done. Further, they state that more often than not, the existing models are not a perfect fit for the specific task at hand. So the adaption of known models is an essential task to achieve satisfactory annotation results.

Lauscher et al. [2018b] adapt the model of Toulmin [1958] in their corpus to a simple data format with three different types. The *background_claim* represents a statement that originates from the background of the research area. An *own_claim* is closely related to the own work of the author and a *data* component consists of facts that support or attack a claim. The relations between them show how the respective text phrases are connected to each other. One of the reasons to take this model as a basis for their work was its simplicity and because it already proofed to be of good use with Artificial Intelligence and Argument Mining [Lauscher et al., 2018b]. They further simplified the model and optimised it to work with scientific writing by removing the not observed components *warrant*, *backing*, *qualifier*, and *rebuttal*. As mentioned before, to put the raw argument data into perspective and especially to be able to identify connections and relations across different documents, an ontology is needed.

In their research to lay the foundations for a WWAW, Rahwan et al. [2007] propose the AIF as a basis and extend it further with the RDF. They describe the core of the AIF as consisting of two types of nodes. The *infomation nodes* represent the information of an argument and the *scheme nodes* illustrate the presumed passage associated with an argument. While the AIF has great potential to represent a very detailed description of

the relations and structure of a text, the addition of the RDF scheme extends that even further. The AIF with or without its extensions will most certainly be of great use for inspiration to make or reuse an ontology and to keep it compatible. Highly important for the ontology of this thesis is Claim and Premise Model introduced by Walton et al. [2008]; Walton [2009] which proposes a very simple approach of how to model Argument data. The argumentative structures of a text are categorised into claims, which state something and premises, which in turn can support or attack a claim. These claims and premises can further be subdivided into major and minor claims or premises. The statement these components have, depending on what category they are assigned to, and how they stand in relation to each other produces a conclusion. Lauscher et al. [2018c] further define a claim as corresponding to an argument component where a point wants to be made. They give as examples a hypothesis, an assertion or more general a statement. The component premise, which is in their work named *data*, is defined by them as a fact. It can support or contradict a claim and the word premise or *data* can, according to them, be interchanged with evidence, prediction or generally a known and proven fact. For our work we will implement a simple ontology ourselves, leaning on the concepts of the just mentioned model of Walton et al. [2008]; Walton [2009]. We do not reuse one of the mentioned ontologies because we aim to implement an as simple model as possible, with the flexibility to adapt it to different domains and use cases.

2.3 Argument Mining

Argument Mining and Computational Argumentation in general is a growing field of research in recent years say Al Khatib et al. [2021]. Lawrence and Reed [2019] define Argument Mining as: "Argument mining is the identification and extraction of the structure of inference and reasoning expressed as arguments presented in natural language." [Lawrence and Reed, 2019, p.1] According to Lawrence and Reed [2019], the main goal for Argument Mining is to grasp argumentative elements from free text and turn it into structured argument data. Al Khatib et al. [2021] mention different tasks performed in Argument Mining. Despite the recognition of single argumentative units, the classification of different components of arguments and the identification of text structures are important as well. Depending on the used model, the components of arguments can be described as claims or premises. Premises can then either be supportive or attacking to a certain claim and come in the form of data or so-called warrants. Text structures describe different parts and sections of the document such as introductions, discussions and conclusions. A possible interpretation of claims, premises and the relation among them can differ according to their location within the text and their context as mentioned by Lawrence and Reed [2019]. Further, they mention the tasks of identifying the relations between the found arguments, like how a premise can be put into context to a given claim. They highlight six core challenges in Argument Mining.

The first challenge highlighted by Al Khatib et al. [2021] comes down to the modelling of the argumentation, which was already described in the preceding Section 2.2. The second big challenge concerns the various domains in which scientific texts are written.

In their work it is pointed out the fact, relying on the work of Weinstein [1990], that different research communities have different standards not only for the general structure of their documents but also how and what arguments they contain and that often a deeper understanding of the respective research domain is required, by humans as well as by the computational models, to properly handle such a document. These facts let them raise the question, if it is even possible to create a one-size-fits-all argument mining technique that can be adapted to every domain and its specific features or if there is the need to come up with a tailored solution for every individual community. Next, they look at the various different types of scientific documents, such as method papers, reviews or reports. They mention that each of these document types usually has their specific types of evidence they deliver and focus on the topics they cover in different ways. Al Khatib et al. [2021] again point out the open question of whether it is required to develop specific models for different document types or if there is the possibility to accommodate all of them with one representation approach. Another challenge, mainly for automated Argument Mining, are enthymemes. Enthymemes are described by Al Khatib et al. [2021] as implied meanings of an argument. The conclusion or premise they incorporate are unstated and can be constructed from context and common knowledge. While Al Khatib et al. [2021] say that they are mostly no problem for humans to understand, automated Argument Mining techniques easily flounder when it comes to required knowledge that can not be found within the text itself. Similar problems can occur for humans and automated Argument Mining techniques alike when it comes down to subjective interpretations. They also bring up the case of experimental papers, where it can be all too common to leave some connections open to interpret for the reader. For a machine as well as a human without domain expertise it can be very difficult up to not realistic to grasp the right or even some logical interpretation of the presented data. The last key point for Al Khatib et al. [2021] is the context-dependence. Based on the work of [Green, 2014; Stab et al., 2014; Vázquez Orta and Giner, 2009] they mention that the complexity of the text itself and the argumentation in scientific writings is by no means smaller than it is in other genres. Claims stated in one section could have their supporting premise in a completely different part of the text. This makes it difficult to choose adequate boundaries to look for argumentative units. These challenges are present regardless of the fact if Argument Mining is done manually or automatically.

As of today, automated Argument Mining can not match the quality of a manual annotation as can be seen in the work of Stab and Gurevych [2014]. They achieve 90.5% of the performance of a human annotator in identifying argument relations and 88.1.5% in identifying argument components. Although these are impressive numbers, a gap remains. In contrast, while a manual annotation may be better, it is also very time consuming and complex as stated by Lippi and Torroni [2015]. Further Lawrence and Reed [2019] say that automated Argument Mining is needed and important to cope with the enormous amount of data today and in the future. Nevertheless, the manual annotation of Arguments and the resulting corpora, as seen from the work of Lauscher et al. [2018b], are important and by all means valuable to also train neural networks with high-quality data. The importance of these corpora is backed by the statement of Lippi and Torroni [2015] , saying that Argument Mining processes based on Machine

Learning and Artificial Intelligence, require such data to train and test their models.

3

Design

This chapter looks at the considerations and preparations leading up to the implementation. First, the idea and motivation to implement the system and how it stands out compared to other related tools is discussed. It continues with the ontology and the requirements engineering, and finally, the software development aspect is covered.

3.1 Motivation and Idea

After assessing the theoretical foundations and existing tools and applications, we started elaborating on which aspect or use case is not yet covered in a practical way and provide a benefit to the research community in this field.

Tools such as ArguminSci [Lauscher et al., 2018a] demonstrate and provide the possibility of state-of-the-art automated Argument Mining. As implied by Lawrence and Reed [2019] and Lippi and Torroni [2015], the progression of automated Argument Mining also depends on the availability of high-quality annotation data to train and test such systems. Based on this implication, the importance of work like that of Lauscher et al. [2018b] becomes clear. Referring to this information, such corpora are of great use to a machine. In contrast, in the format, these corpora currently come and due to the lack of context within an annotation file of a corpus, there is in our opinion little practical way to benefit from the value they incorporate from a human perspective. From our point of view, looking up annotations of a corpus manually by their index in a text file is not practical.

Other approaches with the objective to make the mined or annotated arguments accessible to humans exist in the form of tools such as SALT [Groza et al., 2006]. SALT makes it possible to manually annotate argumentative structures and their relations within LaTeX files according to the provided ontologies. The PDF documents resulting from these files have visual annotations and the metadata of the annotations packed within the XMP field of the PDF standard. Groza et al. [2006] focus mainly on the use case of making semantic annotations during the writing process of the document. To the best of our knowledge, SALT does not provide any functionality to process existing annotation data, such as a corpus, on a given LaTeX file in order to include it automatically. Because the website referenced on their paper is not working, we were unable to obtain the SALT tool.

This is where our application is meant to close the gap. Our goal is to increase the value and usability of high-quality annotation corpora, such as the one that emerged from the work of Lauscher et al. [2018b], by automating the conversion from corpus data into a file with visual annotations to supply a human reader with the information from the annotation file within the context of the text. We agree with the statement of Groza et al. [2006] that PDF is the standard file format for scientific publications and were inspired by their approach to enrich documents with visual annotations and the associated metadata.

Going one step further, our system generates PDF documents that carry the metadata and the visual annotated argumentative structures and relations from the corpus data. Each annotation is assigned with a globally unique target address. This target address is added to every annotation with the purpose to enable the opportunity for an unambiguous relation referencing among argument structures across documents. Further, our application is customisable regarding the used ontology for the creation and extraction process. In addition, the created documents have attached the corpus files they originate from to ensure complete data transparency and availability. The application we implement has a pipeline character and automatically generates enriched PDF documents from corpus data and can also reverse this process by extracting all metadata and attached files from such documents. Both the creation and extraction process work according to their respective customisable configuration.

3.2 Ontology Engineering

With the study of various scientific publications about used ontologies in the field of Argument Mining and also according to Al Khatib et al. [2021] we came to the conclusion that there exists not one perfect ontology. Further, Rahwan and Reed [2009] say more often than not the ontologies are tailored specifically for one application or use case. Our goal was to set as few boundaries as possible to the ontology required. The ontology we supply by default serves as a basis. The main requirement for our ontology and its utilisation was deliberately to keep it simple and to maintain a high degree of flexibility and adaptability.

The ontology engineering started with the investigation of existing ontologies and models used in other applications. Using the Protégé software of Musen [2015] we modelled our ontology. Our main objective, for the ontology we are going to build, is simplicity and to keep it general to allow a great level of flexibility. Our Ontology consists of the classes `Claim` and `Premise` these two components are closer described in the related work chapter in the Section 2.2. Entities of the classes have properties for their start and end index and the actual string value. The `object properties` associated with the classes represent the possible relationships between the entities. The ontology includes the relationships `attacks`, `supports`, `semantically same` and `parts of same`. Lauscher et al. [2018c] define the `supports` relation as a direct relation where one component backs another component. The `attacks`, or in their wording *contradicts*, relations is the opposite of the `supports` relation. This relation is described

as bi-directional and exists if a component contradicts or attacks another component and vice versa. The `parts of same` relation was introduced by Lauscher et al. [2018c] and represents that components having this relation to each other, actually belong together and form one big component. Also, the `semantically same` relation has been introduced by them to represent the repetition of actually the same statement. If argumentative structures can be identified that are referring to mostly the same claim or premise, these components have this kind of relationship to each other. In addition, the inverse to every relationship is introduced and declared accordingly. Each inverse is itself a relation and contributes to a complete representation of all the relations that a class entity has.

This base ontology can be modified and extended to match personal, corpus-related and domain-specific needs. There is no restriction on the naming or number of classes or relationships. We want the system to implement a simple approach to manage the representation and declaration of the desired ontology and its mapping to the expected corpus schema.

3.3 Software Development

After the investigation of up to date research in the field was completed, we started the software development process. We decided to implement our pipeline using Python. The implementation of the software was set to follow the *scrum* process. With weekly meetings and sprints, we assure an adequate progression and a continuous reevaluation of the development state. We structure our code development in three phases and follow a bottom-up approach. The module phase, the connection phase and the test phase.

3.3.1 Module Phase

The pipeline we plan to build consists of two main modules, one generates PDF documents from an annotation corpus and the other one extracts the attachments and semantic annotation structure. These two models consist themselves of distinguishable sub-tasks, we call them sub-modules.

The module to generate PDF documents consists of four sub-modules:

Sub-module 1: Create the semantic annotation structure from the annotation file.

Sub-module 2: Generate the required LaTeX commands and annotation colour declaration according to the annotation structure.

Sub-module 3: Read the text file, embed the annotations and write the LaTeX file.

Sub-module 4: Attach the source files and embed the annotation structure to the PDF document.

The module to extract the data from the PDF documents has two sub-modules:

Sub-module 1: Extract the attached source files and the embedded annotation structure data.

Sub-module 2: Write an annotation file according to the specified configuration from the data of the extracted annotation structure.

The tasks described above are handled separately in this phase. With a divide and conquer approach we first find solutions for the isolated problems on a conceptual and code level. With test data, we implement a working example for every task.

3.3.2 Connection Phase

In this phase, we connect the single sub-modules and modules. The script is implemented, that enables the call of the program from the command line, interprets the given input arguments and handles their distribution to the right module. The previously implemented sub-modules are now getting connected within the module, which ensures a correct procedure order. Step by step, both modules reach their full capability to complete their assigned task. After the two modules are fully implemented, the command line interface of the pipeline is created. This interface builds the entry point of the system and handles the module calls according to the user's specifications on the command line and sets and propagates the inputs.

3.3.3 Test Phase

Upon completion of the connection phase, the test phase starts. We test the implemented pipeline both manually and automatically with a test suit implemented in the Python project. The automated testing again follows a bottom-up approach. Test data is created in the form of files, building the test input and to compare expected file outputs to the actual outputs. The test files consist of an exemplary text and annotation file from which the expected text file with the command embedding, the LaTeX file, the PDF document and the extracted annotation file are constructed as comparative data. Additionally, a text file with a mock-up URI and a text file with no URI exists in the test files for the test of the different target address retrievals. Within the test suit, run-time variable data is created to check the intermediate data produced by the code against these expected variable values.

Starting with unit tests, every functionality of the sub-modules is tested, ensuring a good function coverage. In the next step, the complete sub-modules are tested on their proper functionality. Once all sub-module tests are passed, we start the module tests. The functionality of the two complete main tasks of the pipeline, the creation of a PDF document from a pair of text and annotation files and the extraction of all carried data from a PDF file, is tested at once. Ultimately an integration test is run to test the complete pipeline. The integration test invokes the pipeline at its command line entry interface for both its main modules. With this test set-up, we show the proper functionality of our implementation in a manual and automated way. The manual test

is done by processing the complete corpus of Lauscher et al. [2018b] and producing the semantically enriched PDF documents for every pair of text and annotation file.

4

Implementation

This chapter elaborates on the implementation of the annotation pipeline. We start by giving a high-level overview of the pipeline's functionality and capability and continue to explain the separate process steps in detail. We then first show how to run the pipeline followed by what configurations can be set and how they are used during run time. Step by step we walk through the application's procedures. In the beginning, we explain how the metadata structure is created from the annotation file, and how the later embedded global target address is retrieved. Coming to the core functionality of generating the LaTeX file, we elaborate in separate sections in what way the colours and custom commands are created and specified, why and how the look-up structure is created on run time and finally the embedding of the text phrases into their specific commands and the conversion of special Unicode characters is explained. Next, the compilation process of the LaTeX file and the handling of the resulting PDF document and its attachments are shown. With this we conclude the explanation of the process to generate a document and move on to the elaboration of the extraction process. The description of the extraction is structured as follows: First the configuration possibilities, how they are used on run time and their effect on the process are shown. Then we explain how the attached and embedded data is extracted from the given PDF document. Finally, we show how the custom annotation file is created from the extracted data. The source code of the implemented system is available online.[1]

4.1 Pipeline Overview

The implemented system uses a variety of different packages and modules, such as for example *regex*[2], *PyPDF2*,[3] and *pylatexenc*.[4] The pipeline, written in Python, offers the possibility to automatically generate annotated PDF documents from annotation corpora. The pipeline takes a corpus structured in the format of Stenetorp et al. [2012] , like that of Lauscher et al. [2018b], as input. Processing the text file together with the annotation file, a LaTeXfile with highlighted argumentative structures and their respective

[1] *https://gitlab.ifi.uzh.ch/ddis/Students/Theses/2021-joel-watter*
[2] *https://docs.python.org/3/library/re.html*
[3] *https://pythonhosted.org/PyPDF2/*
[4] *https://pylatexenc.readthedocs.io/en/latest/latexencode/*

links to each other is produced. In addition, all annotations and relations are written to a metadata structure according to the predefined ontology and its mapping to the annotation schema.

The final PDF file is generated by the *pdfLatex* compiler, with the metadata structure written into the document information space of the PDF file itself. The result is a PDF file with human-understandable and visible annotations, their relationships to each other in tool-tips and in the comments area of the PDF viewer, the full annotation source as metadata, and with the original text and annotation file attached. The metadata can be extracted to create a custom annotation file and also the attached files are automatically extractable.

4.2 Run the Pipeline

The `annotation_pipeline` script serves as the entry point for the entire application. According to the specified input, the script handles the propagation of the input arguments to the right module of the pipeline as shown in Figure 4.1.

4.3 Configurations

At the beginning of the process to generate a document from corpus data, the program reads the configuration file with the help of Python's *configparser* library. The specified configurations are parsed into lists. These lists are then processed by the `read config` function, which creates a component data structure, a mapping structure, both in form of a dictionary. The remaining parameters, like for example the identifier for a section tag within the text file, are stored within a parameter dictionary.

To create the structures dynamically according to the specified configuration at run time, they are created as dictionaries filled with dictionaries. Each component type is described in the mapping structure with its corresponding keywords for the used annotation scheme of the corpus. By default, the mapping structure is equipped with an entry for the relations and their inverses, having an empty dictionary as their value, to distinguish them from the components. Within this sub dictionary, each relation and inverse is created with the corresponding mapping to the related keywords used in the corpus. The mapping structure is created in an atomic fashion and represents the translation of the expected annotation scheme of the original files into the ontology used in the pipeline. The defined mapping keywords from the corpus are used as keys in the dictionary structure. While this results in a manageable amount of redundant data, it increases the performance of searching for the dictionary entries. The component structure is prepared to hold the metadata and a dictionary is created for each component with the component name as the key.

The specified identifier parameters, such as what identifies an annotation, relation or resource in the annotation file or with what tags are the start and end of a section within the text file marked, provide the pipeline with the information about how to interpret

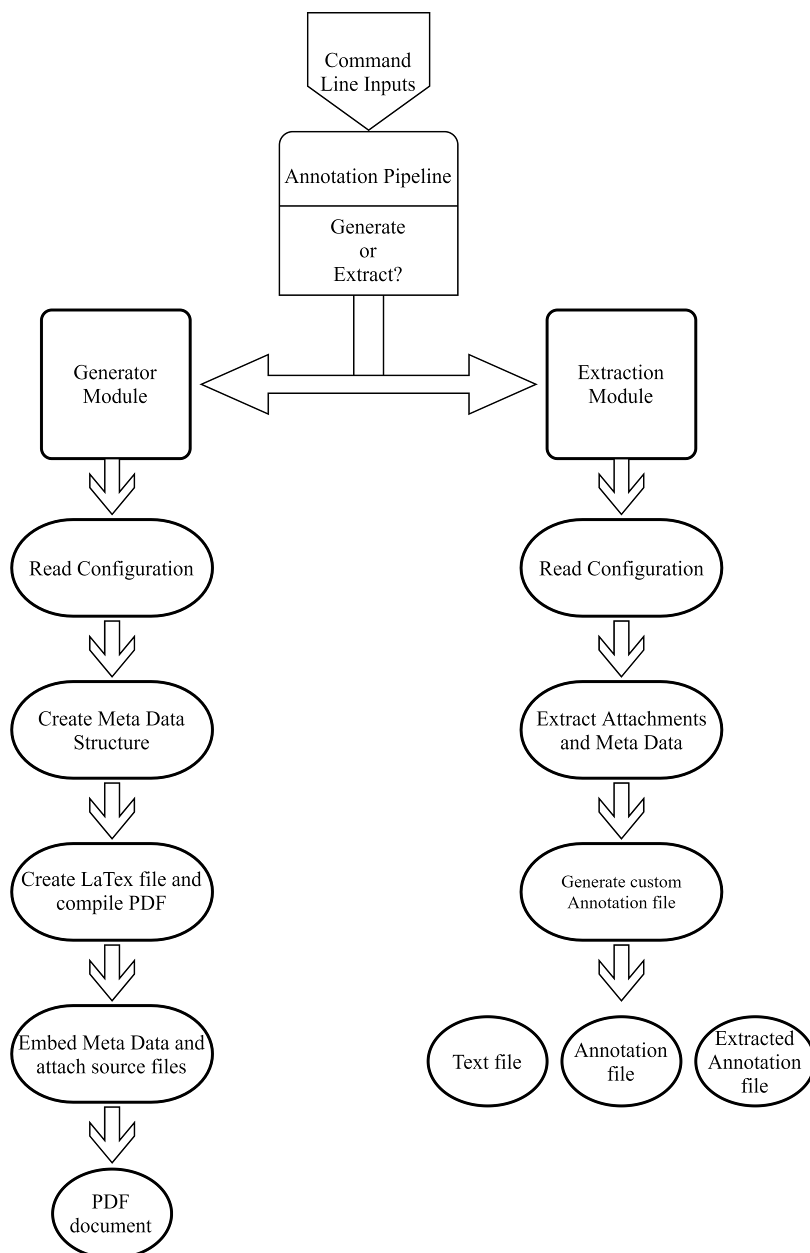

Figure 4.1: Structure of the pipeline

the annotation file entries and how to split the text file. The configuration can be customised and individualised either by using the `make_config_file.py` program in the annotator directory or by writing directly to the configuration file. The configuration file is structured in for main sections, `PARAMS`, `Component`, `Relations`, and `Inverse`. Identifiers for the annotations, relations, resources, and section levels in the text file

are specified in the `PARAMS` section. The components, referring to the different types an annotation can represent, and their mapping to the schema used in the annotation file are written to the `Component` section. Relations and their inverse have to be specified accordingly within their respective section. This structure must be maintained and the identifiers and mappings must be specified in the same way as in the default configuration to ensure the expected behaviour of the code.

The prepared structures and parameters are then returned to the main function and assigned to application variables using the function `assign_params`. After the configuration has been processed and returned to the main function, the library *glob* is used to capture all text files from the input directory.

For each file found, a path with the suffix `.ann` is first created by the library `pathlib` and then checked whether this file exists in the specified directory. If the file is found, the path to the text file and the path to its corresponding annotation file is stored as a tuple, building a file pair. These tuples are stored in a so-called `pairs_list`. The process for creating the PDF file is now started for each pair. The temporary directory `temp_dir` in the data directory of the application serves as the file holder during run time. The two files of the currently processed file pair to create the PDF document are copied directly into the temporary directory in order to attach them to the document later. Now the programme builds all the components needed for the file creation, starting with the metadata structure.

4.4 Generate the Meta Data Structure

After processing the configurations, the annotation file of the corpus is used to create the metadata file. This is done in an iterative process. The annotation file is read line by line and stored in a temporary list.

A template dictionary is generated from the mapping structure to later append a copy of it to every component entry. Each relation and its inversion is a key in this template and is assigned an empty list in which the respective component keys are later stored to represent the corresponding relation. In the next step, each annotation is processed. First, only the components are instantiated as dictionary entries. A component is one line or one annotation of the annotation file. Its identification tag starts with the specified `annotation_identifier`. It consists of its tag, the type it was assigned with from the used schema within the corpus, the start and end index of the string in the text file, and the string value it has in the text file. The annotations themselves are strings. The described structure above correlates to the BRAT standoff format [Stenetorp et al., 2012] like in the corpus of Lauscher et al. [2018b] and this is the expected format by the program .[5] An example of such an annotated component can be seen in the Listing 4.1. With different formats, it can not be guaranteed to successfully complete the creation process.

```
1 T9  background_claim 3349 3425  SSD is widely used in games, virtual
      reality and other realtime applications
```

[5] *https://brat.nlplab.org/standoff.html*

```
R2  supports Arg1:T7 Arg2:T9
```

Listing 4.1: Example of a component annotation (T9) and an annotated relation from the corpus of Lauscher et al. [2018b]

Using *regex*, the string of the annotation is tokenized and returned as a list. The first index of the token list, carrying the annotation tag like `T9` as in the example of Listing 4.1, will be the key for the respective component. This tag is what uniquely identifies the annotation within its annotation file and it will serve the same purpose inside our component structure for this specific document we generate. The following indices represent the type of the component, which is important for the mapping, the start and end index of the annotation in the original text file and the string value of the annotation.

The mapping dictionary and the specified annotation identifiers in the configuration file are used to look up the correct mapping of the scheme to our ontology. We transform the given type of the annotation from the annotation file according to our mapping to the respective type specified in the configuration. If the key, in form of the type of the annotation, exists within the mapping dictionary, the component key is assigned as the entry key to instantiate a new component. The component dictionary is updated with the new component entry and all its data from the annotation file and, as earlier mentioned, a copy of the template dictionary for the relations and inverses is added. Listing 4.2 shows an example of the instantiated component `T9` from the Listing 4.1. The dictionary with the key `Relations` in Listing 4.2, shows the appended template dictionary. Also, the different relations can be identified, for example, the entry with key `Supportedby` is the inverse of the `Supports` relation. With the default configuration, this ensures to store every relation from and to a component within every component itself, like it can be seen with the components `T9` and `T7`. If the type of an annotation from the annotation file can not be found as a key of the mapping dictionary, this annotation is ignored and an appropriate message is printed to the console.

```

component_dict = {'Claim': {'T9':
                            {'Start': '3349', 'End': '3425',
                             'Value': 'SSD is  widely  used in games,
                                       virtualreality  and  other
                                       realtime applications',
                             'Relations': {'Attackedby': [],
                                           'Attacks': [],
                                           'Parts_Of_Same': [],
                                           'Parts_of_sameto': [],
                                           'Semantically_Same': [],
                                           'Semantically_sameto': [],
                                           'Supportedby': ['T7'],
                                           'Supports': []}}
                            },
                  'Premise': {'T7':
                              {'Start': '3433', 'End': '3459',
                               'Value': 'its ease of implementation',
                               'Relations': {'Attackedby': [],
```

```
20                                                      'Attacks': [],
21                                                      'Parts_Of_Same': [],
22                                                      'Parts_of_sameto': [],
23                                                      'Semantically_Same': [],
24                                                      'Semantically_sameto': [],
25                                                      'Supportedby': [],
26                                                      'Supports': ['T9']}}},
27                             'Resource': {}}
```

Listing 4.2: Example of the component dictionary with two instantiated component and with the default configurations

Important to mention is the possibility to annotate resources. A component `Resources` is created per default from the configuration, as can be seen in the example in the Listing 4.2. A resource is a reference to another document. Such an annotation in the annotation file is marked by the corresponding `resource_identifier` in its tag has the type description `resource`, followed by the DOI link of the referenced document and the title of the document. This opens up the opportunity to annotate relations to components of other documents by specifying them with the additional referencing of the document they originate from. This source document of such annotations also needs to be annotated in the annotation file. Such relations are treated the same way as normal relations within the document but they come in a different form. Their identification tag is a concatenation of the tag of their respective resource and their original tag. For example, we assume for a moment that the annotation `T9` is originally from a specified resource with the tag `P3`. The resulting tag for such an annotation would be `P3:T9`.

Once all components are instantiated in the component dictionary, the relations are processed. This is again done in an iterative way. This time, the script only considers annotations starting with the specified relations identifier from the configuration. The relation `R2` in the Listing 4.1 is an example of such a reference, the relations identifier, in this case, is `R`. The type of the relation is identified with the second position and mapped according to the rules defined in the mapping dictionary. The tags of the involved arguments are split from the notation of the string entry of the list. The type of relation identifies what relation the first argument has to the second. The correlating component entries are updated in the component dictionary. With our example of the relation `R2` in the Listing 4.1, the relation specifies that `premise T7` supports `claim T9`. The representation of this relation in the default configuration setting is vice versa. The component `T7` gets updated with the identification tag of the component `T9` appended to the respective relations list, in this case, `Supports`. At the same time, the relations list for the inverse relation of the second argument component, which is for the inverse relation `Supportedby` and for the second component `T9` in this example, is updated with the respective identification tag of the first component. This example results in the component dictionary being updated to the structure shown in Listing 4.2. After the processing of the relations is completed, the component dictionary is filled with all annotation information and gets returned to the main function.

In the next step, the DOI link for the global target address is retrieved. While embedding the annotations later, each will be assigned with a global target address to enable

cross-referencing of annotations from different documents. This target address consists of the retrieved DOI link of the document we currently create and the identification tag of the component, separated by a hashtag. A DOI link is by definition unique and also the identifier tag of a component is unique within one document. Therefore, by concatenating the DOI link with the identifier tag, we can ensure a globally unique address for every single annotation.

The script expects the source document itself as resource <resource_identifier>00 within the currently handled annotation file. At run time, the pipeline first searches the component dictionary for this 00-resource. If it is found, the address is generated directly from the respective DOI. If a key error occurs, meaning the document is not annotated, the program attempts to find the first complete and isolated link within the text document using *regex*. Often the link for the document itself is at the beginning of a paper or book. To prevent a regex match of any link within the text, the script searches for an isolated DOI link. If a match is found, this link is used as the target link. If no link can be found in the text file either, the address is concatenated with the phrase `DOI Placeholder`. This address determination is done in the main pipeline function by calling the corresponding function of the script and storing the address in a variable to pass it to the LaTeX file generator module.

4.5 Create the LaTeX-File

At the core of the pipeline, a LaTeX file is created with the data from the input text file, the target address, the section and subsection identifiers specified in the configuration file, and the previously generated component dictionary, in order to bring the annotations onto the PDF file and visualise them. The `make_latex` function takes care of all the necessary prerequisites before the file can be written. We first look at the definition of the colours needed and specified for the highlighting of the annotations. In the next step, we present, how the custom commands are created and defined for every type of component followed by the creation of the lookup structure. The last subsection shows how the actual writing of the LaTeX file takes place.

4.5.1 Colour Definition

For every type of component, a colour is needed to highlight the annotation in the text and make the different types of annotations distinguishable. A set of colours is created with the `declare_colours` function. Based on the number of different component types in the component dictionary, an auxiliary function is called with the input parameter representing how many colours we need to declare. According to the input amount, floating-point numbers between 0 and 1 are generated. These floating-point numbers are linearly distributed and generated using the *numpy* library. For each previously generated number, the corresponding RGB values are retrieved from the `hsv` colour map, which is imported with the *matplotlib.pyplot* library. `hsv` stands for `hue saturation value`. This colour map has a circular shape with the radial degree on the circle spec-

ifying the colour, the saturation defines the intensity of the colour and the value is to set how light or dark the overall tone is. The `hsv` was intentionally chosen because of its circular shape to provide high contrast for clear differentiation between annotations.

The colour values are then returned to the colour definition function where the string is concatenated for the LaTeX colour definition. The colour is named after the corresponding component type and the entire definition string is appended to the list returned to `make_latex` once all colours are defined. Listing 4.3 shows an example of such a colour definition in the LaTeX file for the component type `Resource`.

```
\definecolor{Resource}{rgb}{0.0, 0.062501968751969, 1.0}
```

Listing 4.3: LaTeX colour definition resulting in a intense and vibrant marine blue

4.5.2 Custom Commands

Similar to the definition of the colours, the user-defined LaTeX commands have to be created by calling the function `build_latex_commands`. Within this function, a new command with the name of the respective component type is defined for each component type.

The commands are a combination of a *\pdfmarkupcomment* command of the *pdfcomment* LaTeX package, which is itself embedded into a *\hypertarget* command of the *hypperref* LaTeX package. The *hypperref* package provides a variety of ways to hyperlink text passages in a PDF document. These links can lead to an external target like a website as well as to targets within the document, similar to the *\label* command of LaTeXitself. The *pdfcomment* and its command *\pdfmarkupcomment* are used to highlight an embedded text passage, in our case the components we want to annotate, equip it with a tooltip and create a comment that is linked to this embedded text passage. The content of the comment is the type and component identifier tag as the author and all the relations the component has, ordered according to the relation type, and with the string value of the related component. The tooltip has the same content as the comment.

In our PDF file, we intentionally use only the target command. Because an annotation can and will be related to many other annotations in our document, there is no reasonable way to link a passage of text to any number of targets. Instead, each annotation will get assigned a unique global address as its target like described in Section 4.4. This address is the first argument the custom command receives as input. The second argument, which is later passed to the custom command, represents the `id` of the comment, which gives the comment its identifier tag to be able to search and filter for this comment later in the PDF viewer. Next, the `author` argument is passed to the command. This will represent the title of the comment and is going to be filled with the type of the component and the identifier tag of the component while the embedding takes place. The fourth argument is the text in the document to be highlighted. The fifth and last argument represents the content of the comment itself. The colour for the highlighting is predefined with the corresponding colour belonging to the type of component. An example of three command definitions is presented in Listing 4.4. The commands are

concatenated and appended to a list. After the creation of all necessary commands is completed, the list of commands is returned to the function `make_latex`.

```
\newcommand{\Claim}[5]{\hypertarget{#1}{\pdfmarkupcomment[id=#2,author={#3},subject={Annotation},color=Claim,opacity=0.5,markup=Highlight]{#4}{#5}}}

\newcommand{\Premise}[5]{\hypertarget{#1}{\pdfmarkupcomment[id=#2,author={#3},subject={Annotation},color=Premise,opacity=0.5,markup=Highlight]{#4}{#5}}}

\newcommand{\Resource}[5]{\hypertarget{#1}{\pdfmarkupcomment[id=#2,author={#3},subject={Annotation},color=Resource,opacity=0.5,markup=Highlight]{#4}{#5}}}
```

Listing 4.4: Example of the command definitions with the default configuration

4.5.3 Write the LaTeX File

The last requirement before we can start writing the LaTeX file is to create the lookup dictionary. The function `make_lookup_for_indc` mirrors the component dictionary with the modification that the start index of each component serves as the key to the component. The purpose of the look-up dictionary is to increase the performance when selecting the relevant components for a certain text passage according to the current index.

Now that all the necessary structures have been prepared, the actual writing process of the LaTeX file begins. The original text file, like with the corpus of Lauscher et al. [2018b] for example *A01.txt*, is opened and read. A new `.tex` file in write mode is created and opened in the temporary directory, with the return value in the form of the path to the file already set.

At the beginning of the text processing, the *regex* library is used to determine the start and end indices of the title and abstract parts. These are handled separately in order to include the text parts in the correct LaTeX representation. The beginning of the `.tex` file is the same for each file and is written to the file first. This header part is needed in order to make the LaTeX file compile later and consists of the document class and all used packages. The documents belong to the class *article*. The required packages are *xcolor*, *pdfcomment* and *hyperref*. In the next step, all prepared colour definitions are written into the file, as well as the predefined user-defined commands.

After all used packages, colours and commands are declared, the statement *begin*{*document*} is written into the file and marks the beginning of the actual content. First, the title text is wrapped in its LaTeX representation (*title*{*title text*}) and the command *maketitle* is specified to ensure a correct representation of the title after compilation. After the title, the abstract is packed in its representation. This is done by first writing the statement *begin*{*abstract*}, followed by the actual text of the abstract and finished with the *end*{*abstract*} statement. The script now processes each section and subsection (and subsubsection if present) separately. Based on the tags specified in the configuration

for the sections and subsections, each tag occurrence in the text is identified with *regex*. This returns a list with the start and end indices of the section titles.

A loop starts for every found occurrence of a section identifier tag. By comparing the currently processed match to the specified section identifiers from the configuration file, the script determines what actions to take and what type of section was matched. When the currently processed match is equal to a starting tag of a section, we encountered a new starting point of a section title. In that case, the text that spans within the index range of the end of the current match to the start of the next match is the text value of a section title. The corresponding title text is retrieved from the original text and wrapped in its required LaTeX representation like for example for a subsection: *subsection*{*section title text*}.

If the current match is equal to an end identifier, the following text is the content of a previously defined section. The position of this text passage is defined by the index range from the end index of the current match to the start index of the next match, which in turn will again be a start identifier. If there is no next match in the match list, which means the code is now going to process the last section, the next match variable is set to `none`. The text passage of this iteration, therefore, spans from the end index of the last match to the end of the entire text file.

After the index span for the current text passage of interest has been identified, it gets retrieved from the whole text file according to its index. This is done to process every section content separately and step by step. For each piece of text, the function `command_embedding` is called, which is explained in Subsection 4.5.4. The input to the function is the text string itself, the end index of the current match, the start index of the next match, the lookup structure, the target address template (the retrieved DOI link or the placeholder if there could no DOI be found), and the component dictionary.

4.5.4 Command Embedding

The function `command_embedding` is the core of the annotation process. It takes care of checking and identifying the exact string parts that need to be converted into their corresponding custom commands according to the extracted components and their relationships in the dictionary. To find the relevant components, all the keys, representing the start index of every component, are extracted from the lookup dictionary and sorted. The script uses the *bisect* library for a fast and efficient extraction of the first and last item from the key list whose start index is within the bounds of the specified start and end parameters of the whole currently processed text part. A loop begins to process each annotation separately within the current function call. The current index is initialised with the start parameter and updated in each iteration. Since the function formats and embeds only a part of the original text file, using the annotation data that refers to the whole text, it is of great importance to keep track of and have access to the current index in the context of the text part that is a part of the whole text.

Text parts that do not belong to an annotation are appended directly to the final string by the index difference between the current index and the beginning of the annotation now being processed. The final global target address is now created for the annotation.

This final global target address is the concatenation of the global address template, a hashtag character (#), and the identifier tag of the component, for example, `T9` as seen in the Listing 4.2. The global address template is either the retrieved DOI linke or the placeholder string (`DOI Placeholder`) as explained in Section 4.4. The code checks if the global address template has a URL format to declare it explicitly for LATEX with the *url*{*url string*} statement. If the placeholder format is used, no special declaration is required. The call of the LATEX custom command is now prepared.

The corresponding command for the annotation is automatically set by the respective type of the processed component and the parameters are filled with the data from the entry of the lookup dictionary. For the comment and the tooltip, the text needs to be concatenated from all relations of the component. This is done with a loop where every relation entry of the component is taken and checked if such a relation exists. The relation type descriptions are only added if such a relation exists in order to keep the comments as lean and clean as possible. If there exists a relation, the referenced relational annotations are added with a preceding tab for a clearer presentation of the identifier tag of the related component and the text value of the related component is added followed by a line break. A tab is specified by the *textHT* statement and a line break is added with the *textLF* statement. The content and format of such a concatenated comment are shown in Listing 4.5.

The full command string for this component and its resulting text annotation is assembled at the end of the loop iteration for this component and then appended to the embedded string. An example of such an embedded command is presented in Listing 4.5. The current index is set equal to the final index of the annotation just processed for the next iteration. As soon as all annotations of the current text part have been processed, the complete string is returned to the function `make_latex`.

```
1 ... \Claim{\url{http://dx.doi.org/10.1515/libr.2002.169\#T9}}{T9}{Claim:
    T9}{SSD is widely used in games, virtual reality and other realtime
    applications}{Supports: \textLF \textHT T6: Skeleton Subspace
    Deformation SSD is the predominant approach to character skinning at
    present \textLF Supportedby: \textLF \textHT T7: its ease of
    implementation \textLF \textHT T8: low cost of computing} due to ...
```

Listing 4.5: Example of an embedded command for the component `T9` from the file pair *A01.txt* and *A01.ann* from the corpus of Lauscher et al. [2018b]

4.5.5 Unicode Conversion

Special Unicode characters can cause problems when compiling a LATEX file if they are not handled individually. For example, the character ω causes the compilation to abort if the character is written into the file as it is. To counteract this problem, each text first goes through the `unicode_to_latex` function before it is actually written to the LATEX file.

This conversion is done at the very end, as otherwise the indices noted in the annotations would no longer match the indices of the text passages since the conversion replaces a single character with several characters. The conversion function uses the

library *pylatexenc* and its method *latexencode*.[6] This method automatically replaces all special Unicode characters with the corresponding LaTeX notation and wraps them in the command `\ensuremath{}` if necessary. It has been shown that this command wrapping causes problems for some characters when embedded in a *hyperref* and *pdfcomment* command. For this reason, the script removes the `\ensuremath{}` commands immediately after the conversion and replaces them with the `$<LaTeX notation>$` notation. So the example from before is returned as `$\omega$`. Besides this conversion, the function also takes care of the correct *escaping* of special characters that are not converted by the library because they have their meaning in LaTeX through their normal representation, such as an ampersand, but we need them as plain text for the compile process to work. The converted string is then returned to the function `make_latex` where it is written to the file.

After all embeddings and conversions described above have been done for each part of the text file, the LaTeX file is closed with the statement `\end{document}`. The script now creates a `.log` and a `.aux` file with the same name as the LaTeX file in the temporary directory where all files for the PDF are located. These files are needed in the compilation process to store the run information and reference data of the processed LaTeX file. Before the file creation module is completed, all used and created files are closed and the filename of the LaTeX file to be compiled in the next step is returned to the main function of the `corpus_to_pdf_convertor` script.

4.6 Compile the LaTeX File

The files are now ready to be compiled into a PDF document. The current working directory is stored and then it is changed to the directory of the temporary folder. This is done to directly have access to the LaTeX file of interest and all other files within this directory that are needed during the compilation. *pdflatex* is used to compile the document. The statements needed to run the compilation, out of the current script on the command line, are gathered in a list to pass it afterwards to the `subprocess`. The specified statements are *pdflatex* to invoke the compiler, the *-interaction* flag to specify how the compile process should be run, the *nonstopmode* statement to skip warning messages during the process, and finally the name of the LaTeX file to compile for example *A01.tex*.

By running a `subprocess` on the command line, the specified commands from the just created list are executed on the LaTeX file in the temporary directory. The compilation process is performed three times for each file. This is necessary to correctly link all comments and references within the document. During the first compilation process, the found targets, highlighted sections and comments are written to the `.aux` file. In the next *pdflatex* calls, the auxiliary file is read first in order to display the text sections correctly and to establish the links from the text to the corresponding comment and vice versa.

[6] *https://pylatexenc.readthedocs.io/en/latest/latexencode/*

After the PDF file is created, its local script path, still within the temporary directory, is concatenated and saved and the current working directory is returned to the normal state. Next, the attachments in the form of the source text and annotation file are added to the PDF file together with the component dictionary structure.

4.7 PDF Attachments

One goal of this work is to create a complete human-readable document with all potentially required data in the form of annotations and text origin. To fulfil this requirement, the files from which the PDF document itself originates are attached to the document that has just been created as well as the component structure that has been built. Immediately after creating the PDF document, the script opens the file again in binary read mode. With the help of the *PyPDF2* library, a `PDFFileReader` object is created from this opened file.[7] At the same time, a `PDFFileWriter` object is instantiated. All pages from the `FileReader` object are appended to the `FileWriter` object.

The cloning of the original PDF document is necessary to embed the component structure later as metadata into the file structure of the PDF document. The component structure is embedded in the document as metadata with the entry tag: `/Relations`. This type of embedding guarantees the possibility of a fast and efficient extraction of the metadata from the document. While this method of embedding the data is quite convenient, the *PyPDF2* library does not provide any functionality to add metadata to a `PDFFileReader` object. This is only possible with a `PDFFileWriter` object, so the content must be copied from the `Reader` into a `Writer` object. Before the components are added to the metadata area, the source text file and source annotation file, from which the PDF document was created, are appended to the PDF document. This is done by calling the function `make_attachment`. The function was created from the original `addAttachment` function of the *PyPDF2* library with slight adaptations to allow multiple file attachments [8]. We extended and changed the function to check for the presence of the `/EmbeddedFiles` key within the `/Names` key of the PDF `_root_object`. The code we added to the function is located on the lines 9 to 13 in Listing 4.7. The function takes the `PDFWriter` object, the filename of the attachment and the read data of the file as input. The data is packed into a `DecodedStreamObject` of the *PyPDF2* library, which in turn has the type of a `/EmbeddedFile` represented in its dictionary structure. The `DecodedStreamObject` is now packed into a `DictionaryObject`, which is subsequently added to another `DictionaryObject`. This just described embedding and wrapping process of such a file that needs to be attached is done in the code shown in Listing 4.6.

```
1 def make_attachment(myPdfFileWriterObj, fname, fdata):
2     # This piece of code has partly been taken from: https://
      stackoverflow.com/a/59085770
3     # The differences and additions have been implemented by myself
4
```

[7] *https://pypi.org/project/PyPDF2/*
[8] *https://stackoverflow.com/a/59085770*

```
5       # The entry for the file
6       file_entry = DecodedStreamObject()
7       file_entry.setData(fdata)
8       file_entry.update({NameObject("/Type"): NameObject("/EmbeddedFile")})
9
10      # The Filespec entry
11      efEntry = DictionaryObject()
12      efEntry.update({ NameObject("/F"):file_entry })
13
14      filespec = DictionaryObject()
15      filespec.update({NameObject("/Type"): NameObject("/Filespec"),
    NameObject("/F"): createStringObject(fname),NameObject("/EF"): efEntry
    })
```

Listing 4.6: Code of the object wrapping for the PDF attachment (code source)

The objects are encapsulated to ensure seamless and correct insertion into the root structure of the PDF document. Now the function checks in the next step whether the required keys already exist in the root structure of the PDF document or whether they have to be newly created. First, the key `/Names` is checked within the PDF file structure, followed by the key `/EmbeddedFiles`. The check of the key `/EmbeddedFiles` was implemented specifically because the PDF documents we create with our tool do not automatically have this entry after their creation. Our added code is located on the lines 9 to 13 in the Listing 4.7. If both keys are already present, the current file is not the first attachment and will therefore be appended to the existing `ArrayObject`.

After attaching both the text and annotation files, the component dictionary is converted into a string using the *json* library and its `dumps` function. By using *json*, the character string of the component dictionary can be easily converted back into a dictionary structure when extracting it from a PDF file. The string is added to a dictionary with the key `/Relations` and this dictionary is then appended to the root object of the PDF file using the `addMetadata` function of the `PdfFileWriter` object, as shown in Listing 4.8.

```
1       if "/Names" not in myPdfFileWriterObj._root_object.keys():
2           # No files attached yet. Create the entry for the root, as it
    needs a reference to the Filespec
3           embeddedFilesNamesDictionary = DictionaryObject()
4           embeddedFilesNamesDictionary.update({NameObject("/Names"):
    ArrayObject([createStringObject(fname), filespec])})
5
6           embeddedFilesDictionary = DictionaryObject()
7           embeddedFilesDictionary.update({NameObject("/EmbeddedFiles"):
    embeddedFilesNamesDictionary})
8           myPdfFileWriterObj._root_object.update({NameObject("/Names"):
    embeddedFilesDictionary})
9       elif "/EmbeddedFiles" not in myPdfFileWriterObj._root_object["/Names"
    ].keys():
10          embeddedFilesNamesDictionary = DictionaryObject()
11          embeddedFilesNamesDictionary.update({NameObject("/Names"):
    ArrayObject([createStringObject(fname), filespec])})
12
```

```
13          myPdfFileWriterObj._root_object["/Names"].update({NameObject("/
     EmbeddedFiles"): embeddedFilesNamesDictionary})
14      else:
15          # There are files already attached. Append the new file.
16          myPdfFileWriterObj._root_object["/Names"]["/EmbeddedFiles"]["/
     Names"].append(createStringObject(fname))
17          myPdfFileWriterObj._root_object["/Names"]["/EmbeddedFiles"]["/
     Names"].append(filespec)
```

Listing 4.7: Attach the files to the PDF (code source)

```
1           annotation_string = json.dumps(comp_dict)
2           meta_dict = {'/Relations': annotation_string}
3           fw.addMetadata(meta_dict)
```

Listing 4.8: Add the meta data to the PDF structure

The PDF document for this pair of text and annotation file is now complete and can be saved in the specified output directory. Before the iteration is finished, open files are closed and the entire temporary directory is emptied. The creation of the next document from a new pair of text and annotation files can begin.

4.8 The Extraction

The PDF documents generated by the pipeline offer great value through their annotations alone because they enable a human reader to directly see all annotations within their context and provide an overview of every relation this annotation has to other annotations of the document. To further increase the usability of the documents from the perspective of a computer scientist or researcher in general, the extraction module provides a convenient way to extract the metadata and attachments of all generated PDF files within a given directory at once. That way the original source files are directly accessible without the need to search for the corpus the PDF originated from or have it handed out separately. With the stored metadata of the component structure, a customised annotation file can be processed using the model specified in the configurations. The command to run the extraction via the command line is described in Section 4.2.

4.8.1 The Extraction Configuration

The extraction module has a configuration file that defines how the extracted metadata is to be handled and assigned. The configuration file can be edited directly or by using the script `make_extraction_config.py` in an IDE. The specified configurations carry the reverse mapping information and tell the extraction script, how the custom annotation file should be written from the extracted metadata of the component structure. The default configuration is tailored to the naming and ontology used in the corpus of Lauscher et al. [2018b] and provides a copy of the original annotation file, except for the order of the lines, the identification tag numbers of the existing relations and the missing

differentiation between *background_claim* and *own_claim*. The configuration specifies the translation between the model used in the generated PDF and the model desired in the resulting annotation file. The configuration file for the extractor has two main entries, one for mapping the components and one for mapping the relations. The script expects a one-to-one mapping.

With the configuration file, it can be controlled, which relations and components are written to the generated annotation file. If, for example, it is desired to write the inverses of the relations into the annotation file, a corresponding mapping can be added to the configuration. Components and relations that are not specified in the configuration are ignored at run time. Changing the mapping configuration will naturally result in a different number of components and relations than in the original annotation file.

4.8.2 Read the Extraction Configuration

The extraction module starts its process by reading the configuration file with the function `get_config`. The reading of the file is done with the help of the library *configparser*. A component and a relationship mapping dictionary are instantiated. The script is hard-coded to access the keys `Component_Mapping` and `Relation_Mapping` of the configuration read object. These master keys must be preserved and must not be changed if the configuration file is edited. With two loops, one for the components and one for the relations, the entries are written into the respective dictionaries. The names of the entities in the extracted metadata form the keys with their mapping name as value. After the second loop is completed, the two dictionaries are returned to the main function.

The extraction process continues. Using the *glob* library, the specified input directory is searched for all files with the suffix `.pdf`. The extraction starts for each PDF file found in the directory. The file is opened in binary read mode and a `PdfFileReader` object is created with the *PyPDF2* library. The attachments are first extracted by calling the function `get_attachment`.

4.8.3 Attachment Retrieval

To retrieve the attached files of the PDF document, the `trailer` property of the reader object is accessed at its root. This root catalogue behaves like a dictionary and the function stores the file names found within the key sequence `[/names][/EmbeddedFiles][/names]`. The list stored at this key sequence in the file data structure contains always the filename of the attached file followed by the data associated with this attached file at the next index of the list. The file name and the associated data to reconstruct the whole attached file are stored in variables. The variable carrying the associated data stream for the file is again a dictionary, and the script gets the actual string data from the key sequence `[/EF][/F]` with the method `getData`. For each file, the data is written to the attachment dictionary, using the file name as the key. At the end of this process, the attachment dictionary contains an entry for every found attached file, with the name of the file as key and the complete string that will be written to this file as the value

associated with the key. This attachment dictionary is returned to the main function once all attached files have been processed.

To keep the file processing clean, the script now tries to create a directory within the output folder with the name of the currently processed PDF document. The files from the attachment dictionary are written to this newly created directory and saved.

The metadata of the component structure, that has been embedded into the PDF file structure in the creation process, is extracted from the `documentInfo` property of the `PdfFileReader` object. The string dictionary is fetched from the structure with the keyword `/Relations`. With the function `loads` of the *json* library, the string is converted back into a dictionary object. Specifying the file name, the new directory for the resulting files, the component dictionary just retrieved, the component mapping and the relations mapping as input parameters, the function `make_ann_file` is called.

4.8.4 Generate a Custom Annotation File

To clearly distinguish the annotation file created by the extractor, the file name, for example `A01`, is concatenated with `_extract.ann` to `A01_extracte.ann`. This new file is opened in write mode to create it and the script starts to loop over the component types in the component dictionary to work its way through the component structure. With the default configuration these types are `Claim`, `Premise`, and `Resource`. The script retrieves from the mapping dictionary the description of the component types for the new annotation file, like for example if a `Premise` in the new annotations should be called `Data`, `Data` is the new description from the mapping. Now each entry, the components themselves, of this component type for example `Claim` is handled. All components are written to the annotation file in the same format, consisting of their identification tag, the mapped description, the start and end index and their string value. The component type `Resource` differs from this format. Resources have no index entries and consist of the concatenated string of their identification tag, the description from the mapping, by default they are called `resource`, the corresponding DOI link and the title of the resource.

For each component, all its relationships are processed directly. Since the relations are not stored with their identification tag inside the component dictionary structure and represented as a property of a component, the script initiates a counter at the beginning of this function to assign every relation of the new annotation file with a unique number inside the scope of this file. This relation counter is incremented after every encountered relation. For every entry of every relations list of the handled component, a relation in the new annotation file results. The written relation is a string concatenated from the identification tag, the relation description, the first argument component, and the second argument component. The relations identification tag results from the concatenation of the letter `R` and the current relations count. The description of the relationship in the new annotation file again corresponds to the specification in the relation mapping dictionary. The first argument component is the identification tag of the currently handled component, the second argument component is represented by the identification tag from within the relations list of the handled component. The result is a string of the

format shown in Listing 4.9, which is going to be written to the new annotation file.

```
R2  supports Arg1:T9 Arg2:T6
```

Listing 4.9: Example of an extracted relation that is written to the new annotation file

After the extracted annotation file has been completed and closed, the extraction process for the treated PDF document is finished. All extracted files are located in the created folder in the specified output directory and the script continues the extraction for the next document.

Within this chapter, we have shown and explained what our implemented pipeline is capable of and how every task, sub-task, and process is accomplished. To further visualise the product of our pipeline, we provide the screenshot Figure 4.2.

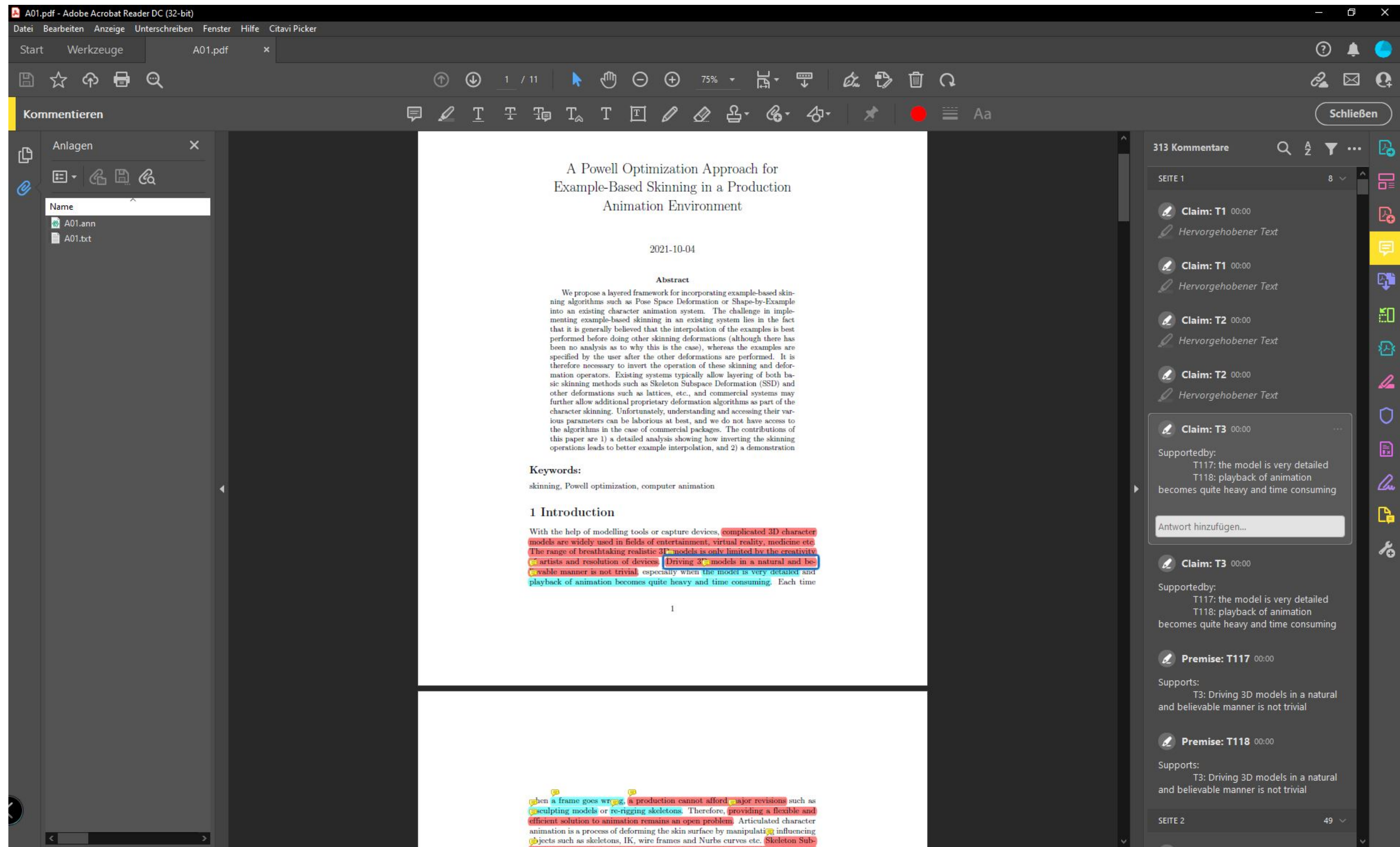

Figure 4.2: Example of the generated PDF document for the file pair A01 from the corpus of Lauscher et al. [2018b]

5

Limitations and Future Work

The pipeline we implemented can be considered as a proof of concept. Corpora can be transformed into PDF documents with visual annotations within the context of the whole text but there remain limitations to its capability and flexibility. This chapter elaborates on the persisting limitations of our system and the auspicious future work possible for and with the use of our pipeline.

5.1 Limitations

The system we built is flexible in terms of the ontology used to interpret the annotations within an annotation file but limited in the depth this ontology can have. The ontology the system is capable to map has one equal depth level of both classes and relations, sub- or superclasses and relations can not directly be represented and therefore also inheritance is not directly specifiable to the system. Another limitation of the system is, that ontologies must be specified manually within the configuration file. Specifying a large ontology can result in a time-consuming task because there is no possibility to import an ontology file automatically.

In terms of wide applicability, the pipeline was built to work with and was tested on the annotation corpus from Lauscher et al. [2018b]. The used format of this corpus and expected by the pipeline is the *BRAT* format of Stenetorp et al. [2012]. The system is therefore limited to handle annotation files written in this format and will not produce an expected output with different file notations. If needed, the system is adaptable to handle different formats of a file's notation but it would need to be implemented specifically. In addition, the extraction module of the pipeline is tailored to extract attached files and metadata from documents created with the pipeline's generator module. The locations and structures to store and retrieve this data in the PDF document are set by definition and can not be changed.

Further, the pipeline was implemented to work with text files written in the English alphabet. Although every text passage written to the produced LaTeX file is first piped through a conversion function to translate special Unicode characters to their respective LaTeX representation, this function is limited to the capabilities of the used Python package *pylatexenc* to recognise and convert such characters.[1] It cannot be guaranteed

[1] *https://pylatexenc.readthedocs.io/en/latest/*

that all existing special characters will be converted and that the pipeline will also work for texts written in an alphabet other than English, such as Cyrillic or Arabic. Unconverted (special) characters can potentially break the LaTeX compilation process and it is nearly impossible to catch and cover every existing character.

The representation of all features and annotations of the generated PDF documents work best with the Adobe Acrobat Reader and have also only been tested with the free version of this PDF viewer.[2] With other dedicated PDF viewers or included solutions and previewers like in a web browser, the correct representation and functionality of the generated PDF documents can not be guaranteed. This is due to the different implementations and capabilities of different PDF viewers and how and what they actually access within a PDF file. Adapting the produced PDF document to every existing PDF viewer to ensure full functionality would be out of scope and potentially also not possible. The machine readability and possibility to extract the attached files and stored metadata to use them is independent of the used PDF viewer.

As mentioned before, we see our implementation as a proof of concept and not yet as a finished software product that can be distributed and used on a large scale of data. With our proof of concept, we hope to serve as an inspiration and expect more work with and around our system.

5.2 Future Work

The pipeline we implemented with this thesis builds the starting point for further development of its core concept to create all in one PDF documents from meticulously created annotation corpora. The improvement of our system to reduce inflexibilities and limitations such as mentioned in Section 5.1 is certainly a starting point for future work in our research area. We intend to improve our pipeline in the future, namely with the functionality to automatically process given ontology files and extend the capabilities in handling annotations with references to other documents. To make it even more simple to exchange the used ontology at run time, we see a big improvement possibility to enable the specification of just an `.owl` file. The pipeline then automatically handles the translation of the found ontology within this file to a format that is useable within the program. Additionally, we can think of an extension to our pipeline. This extension would take generated PDF files as input and searches them for cross-referenced relations. By defined mapping rules for different found relations, the documents annotation files could automatically be equipped with additional cross-references. For example if an annotation `T4` from a document `A` supports the annotation `T22` of document `B` and this in turn supports annotation `T17` of document `c`, maybe `T4` also supports `T17` and vice versa. With such an extension, the potential could be created to start with a few cross-referenced documents which itself have only a few such annotations, and exponentially grow the number of links between the documents and annotations.

With an increasing number of PDF documents created with our pipeline, we see

[2] *https://www.adobe.com/ch_de/acrobat/pdf-reader.html*

great potential for the unique global target address we introduced and assigned to every annotation as explained at the end of Section 4.4. With this address, every annotation is uniquely identifiable. Future work can focus on utilising this address to relate semantic argument annotations across documents to each other. Since this concept and the way we introduce it to cross-link annotations among documents is to the best of our knowledge a new approach, manual addition of such relations to annotations of other documents in the annotation files might be needed at the beginning. But we hope that with the increase in the availability of such data, we could imagine the possibility to come up with automated approaches to predict and generate such cross-references. Future work could potentially investigate the feasibility to use this data to build kind of a knowledge graph of arguments not limited to the boundaries of the document they originate from in terms of relations and look whether the produced data can be used and incorporated in ideas such as the one proposed by Rahwan et al. [2007]. If possible, this may lead to a practical concept idea to process mined arguments and produce data for a knowledge database on a large scale, providing future technologies like the Semantic or Argument Web with ready to use information.

Driving forward our work could potentially open up a wide range of new possibilities to aid the development of technologies in NLP and the Semantic Web thematic and build a cornerstone in generating big amounts of data which can build the basis for such data-hungry applicationsRahwan et al. [2007].

6

Lessons Learned

Besides the effective deliverables, there remains more from the work on a bachelor thesis. During the time of six months, experiences are made and knowledge is gathered around a research topic on a deeper level than ever before. This chapter is going to elaborate on what challenges we faced, the knowledge we acquired in different areas and lessons we learned in general.

6.1 Knowledge Acquisition

At the beginning of this thesis, we first needed to get familiar with the concepts and principles of the research area in which we found ourselves. Starting with the basic principles of argument annotation and argument mining, we have continued to learn about mapping and linking this data using ontologies. We explored the functionality of ontologies and their instances in the context of the development of an Argument Web as described by Rahwan et al. [2007] and Bex et al. [2013], which in turn is a specialised subset of the Semantic Web vision. Further, we learned how an ontology is engineered, from both a theoretical and a practical standpoint. With this process, we also have seen for ourselves how difficult it is to design an ontology, that gives a feasible general approach, not too specific to be only used in a certain domain but also not too general so the necessary details are represented.

To better understand the challenges and opportunities of automated processes, we learned about the complex principles of machine learning, how they are applied and used in automated argument mining and what this means for the further development of a Semantic Web. Having established the understanding of the theoretical fundamentals, we analysed existing applications, systems and tools in each field and had to realise the immense value a future Argument Web could provide.

In addition to all the new knowledge we have acquired in the different research areas, we learned how unpredictable scientific work can be. During our work, the use case and value of an Argument Web became evident to us. This increased our motivation, even more, to contribute to the development of such technologies. Now knowing the importance and potential of technologies in the field of the Semantic Web and the tightly coupled research on Ontologies, we are excited to learn more about this field of research.

6.2 Technologies

During this thesis, we worked with various technologies. Some were previously unknown to us and others we did not yet know from a developer's point of view. This section describes what knowledge and know-how we gathered along the implementation process of our system, focusing on software and file formats. We put our focus in this section on the two cornerstones of our application, the work with LaTeX and the PDF file format.

6.2.1 LaTeX

LaTeX plays an important role in our system. We were familiar with using predefined templates and slightly adapting some settings inside them but during this thesis, we started working with LaTeX on a completely new level. At first, we needed to get a better understanding of how LaTeX functions, how and why packages are declared and where they have to be located in the local file system of our computer, such that they are correctly recognised during the compilation.

In the beginning, we had to think about how to realise our requirements, visualise the annotations, show their respective relations to other annotations, and give them a unique target address. We investigated what possibilities are provided in LaTeX to highlight and colour text phrases and how we can use them for our purpose. In this process we learned how to specify new colours used in a LaTeX file and found the perfect package, *pdfcomment*, to meet our requirements of highlighting and referencing related annotations at once. To associate a target link with our annotation we utilised the *hyperref* package and realised, that commands can be encapsulated within each other in LaTeX . This in turn let us think about an even more generic way of bringing our text data into its desired format. We researched how custom commands can be built and invoked in LaTeX and implemented the dynamic creation of those commands, wrapping the commands of the *pdfcomment* and *hypperref* packages into one command and utilising the self-declared colours. Besides these detailed and implementation based skills, we also learned what declarations and statements are crucial for a clean compilation of a LaTeX file at the end.

Having a deeper understanding of LaTeX is a great advantage for writing this thesis and for future academic work in general.

6.2.2 PDF

The PDF standard as a file type was well known to us from the beginning. They are great to read and every time something has to be handed in and should not be changeable, we convert it to a PDF document. When it comes to working with the PDF standard, we were fairly inexperienced. It was our understanding that most operations possible with a PDF document require a premium software licence. Working with PDF documents in a code and accessing its structure opened up a lot of previously unknown capabilities of this file format to us.

To achieve our goal of creating a PDF that combines human- and machine-readable data in one file, we investigated the possibilities the PDF format provides. While the incorporation of human-readable data in the form of the text is obvious, we searched for ways to conveniently put our created annotation structure data into the document as some sort of metadata. Adobe provides the Extensible Metadata Platform (XMP) which allows to store metadata in a PDF document.[1] With further study of this XMP space its capabilities, especially for media files, became clear to us. The fact, that a complete XML file with a rather strict namespace regulation was needed to put our metadata inside the XMP space, we looked for other ways to satisfy our requirement. We finally settled for the method to place our data within the PDF file's inner root structure as an object, from where it can also be extracted by a machine.

To meet our other requirement we have learned how to attach a file to a PDF document, also within its inner root structure. The possibility of attaching files to a PDF and putting metadata into its inner structure further expands its role as a data carrier. We realised that the seemingly closed file type turned out to be very accessible when handled correctly in code. We hardly would have considered the PDF file format as a viable or convenient file type to work within the past. But by familiarizing ourselves with the PDF file format and knowing of its possibilities and capabilities, we look forward to make use of and incorporate such files more in upcoming projects.

[1] *https://www.adobe.com/products/xmp.html*

7

Conclusions

After the study of the related work in the fields of the Semantic and Argument Web, ontology engineering, and Argument Mining, we have come up with a not yet addressed use case with great potential. In time-consuming and complex manual tasks, annotation corpora are created like the one from the work of Lauscher et al. [2018b]. One particularly important use case of such corpora is to serve as training data for Machine Learning models [Lippi and Torroni, 2015]. From this point of view, these corpora provide a great value to a machine but, due to the missing context of the annotations within such a corpus and the format they have, we identified a significant lack of usability for a human reader. We designed a system with a pipeline character to address this problem. During the design process, we highly focused on the flexibility of our system to be able to be adaptable to any ontology.

We implemented the argument annotator pipeline that is capable of creating PDF documents from annotation corpora and also extract stored files and metadata embedded during the creation process. The creation, as well as the extraction, can separately be configured to map a desired ontology. The PDF documents produced have visual annotations of the argumentative structures within the text. The annotations provide a tooltip and a comment in the dedicated comment section of the PDF viewer. The comments and the annotations are linked and can be highlighted by clicking on them. The built-in functionality of the PDF viewer allows for a fast search, filtering and lookup of every comment with the name tag of the respective annotation. The comments and tooltips display of what type a certain annotation is. With the default configured ontology, these annotations are either a `claim` or a `premise`. Further, the comments show every relation this annotation has to other annotations within the text and the corresponding text value these related annotations have. The data structure of the annotations produced on run time is stored as metadata within the file structure of the PDF document such that it can be extracted at a later point in time if necessary. The original source annotation and text files are attached to the PDF document. The attached files can be accessed manually within the PDF viewer. The extraction tool of the pipeline automatically retrieves the stored metadata from the documents and creates an annotation file based on the specified configuration. This can be useful to automatically create an annotation file with a specified desired model. With this process, also the attached files are extracted.

To the best of our knowledge, the pipeline we built is the first application to convert annotation corpora to PDF documents with visual annotations. Our implementation

paths the way to increase the value of such corpora, to incorporate structured and classified argument annotation data, and their usability. Our system makes these annotations conveniently accessible and comprehensible to a human reader. With the visual semantic annotations and the presentation of the existing relations to other argumentative structures in the text, annotations can be recognised and comprehended within their surrounding context inside the text. In addition, the attached files and stored metadata with the possibility to automatically extract them, make the PDF document to an all in one solution with complete data transparency and accessibility.

As a result of this thesis, we converted the complete corpus of Lauscher et al. [2018b], with its 40 pairs of text and annotation files, to PDF documents with our pipeline. We came to the conclusion, that our envisioned concept works and produces usable and valuable output. These documents come as the mentioned all in one solution. The produced PDF documents are perfectly fitted to be distributed on the web, carrying all their relevant data with them and ready to be cross-referenced to other such documents in the future. If we can establish a standard with our approach, in the future every scientific document will have such rich annotations as the ones we produce. To understand a topic of a document will no longer be a sole question of domain familiarity and reading comprehension, but a task of interpreting the already visualised, related and connected arguments directly presented to you.

References

Al Khatib, K., Ghosal, T., Hou, Y., de Waard, A., and Freitag, D. (2021). Argument mining for scholarly document processing: Taking stock and looking ahead. In *Proceedings of the Second Workshop on Scholarly Document Processing*, pages 56–65, Online. Association for Computational Linguistics.

Bex, F., Lawrence, J., Snaith, M., and Reed, C. (2013). Implementing the argument web. *Communications of the ACM*, 56:66–73.

Green, N. (2014). Towards creation of a corpus for argumentation mining the biomedical genetics research literature. In *Proceedings of the first workshop on argumentation mining*, pages 11–18.

Groza, T., Handschuh, S., and Kim, H. L. (2006). Salt: Semantically annotated latex.

Groza, T., Handschuh, S., Möller, K., and Decker, S. (2007). Salt-semantically annotated latex for scientific publications. In *European Semantic Web Conference*, pages 518–532. Springer.

Lauscher, A., Glavaš, G., and Eckert, K. (2018a). Arguminsci: A tool for analyzing argumentation and rhetorical aspects in scientific writing. Association for Computational Linguistics.

Lauscher, A., Glavaš, G., and Ponzetto, S. P. (2018b). An argument-annotated corpus of scientific publications. In *Proceedings of the 5th Workshop on Argument Mining*, pages 40–46.

Lauscher, A., Glavas, G., Ponzetto, S. P., and Eckert, K. (2018c). Annotating arguments in scientific publications.

Lawrence, J. and Reed, C. (2019). Argument mining: A survey. *Computational Linguistics*, 45(4):765–818.

Lippi, M. and Torroni, P. (2015). Argument mining: A machine learning perspective. In *International Workshop on Theory and Applications of Formal Argumentation*, pages 163–176. Springer.

Musen, M. A. (2015). The protégé project: a look back and a look forward. *AI matters*, 1(4):4–12.

Rahwan, I. and Reed, C. (2009). *The Argument Interchange Format*, pages 383–402. Springer US, Boston, MA.

Rahwan, I., Zablith, F., and Reed, C. (2007). Laying the foundations for a world wide argument web. *Artificial Intelligence*, 171(10):897–921. Argumentation in Artificial Intelligence.

Stab, C. and Gurevych, I. (2014). Identifying argumentative discourse structures in persuasive essays. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 46–56.

Stab, C., Kirschner, C., Eckle-Kohler, J., and Gurevych, I. (2014). Argumentation mining in persuasive essays and scientific articles from the discourse structure perspective. In *ArgNLP*, pages 21–25.

Stede, M., Schneider, J., and Hirst, G. (2018). *Argumentation Mining*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Stenetorp, P., Pyysalo, S., Topić, G., Ohta, T., Ananiadou, S., and Tsujii, J. (2012). Brat: a web-based tool for nlp-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107.

Toulmin, S. (1958). *The Uses of Argument*. Cambridge University Press.

Vázquez Orta, I. and Giner, D. (2009). Writing with conviction: The use of boosters in modelling persuasion in academic discourses.

Walton, D. (2009). Argumentation theory: A very short introduction. In *Argumentation in artificial intelligence*, pages 1–22. Springer.

Walton, D., Reed, C., and Macagno, F. (2008). *Argumentation schemes*. Cambridge University Press.

Weinstein, M. (1990). Towards an account of argumentation in science. *Argumentation an international journal on reasoning*, 4(3).

Zhou, H., Song, N., Cheng, H., and Wang, X. (2019). Argument ontology for describing scientific articles: A statistical study based on articles from two research areas. *Proceedings of the Association for Information Science and Technology*, 56(1):855–857.

A

Appendix

A.1 User Guide

The ontology currently used is specified in the configuration file. The two modules of the application, one for generating documents and the other for extracting the metadata they contain, use separate configurations. The configuration of the generator module instructs the system how to interpret the different types of annotations and relations in the `.ann` file and in which way to display them in the metadata and in the PDF document. The configuration of the extractor module maps the translation from the ontology used in the creation process of a document to an annotation file. With the default settings, the extracted annotations incorporate the same information as the source annotation file. By changing these configurations, the resulting annotation file can be converted to different formats and naming conventions.

The configurations for the generator have four main entries. The parameters are for the identification of the tags in the text and annotation file and do not directly belong to the representation of the ontology. In the `Component` entry, classes are declared. One or multiple type descriptions from within an annotation file can be mapped to a class. A type description must only point to one class to ensure a clear identification of the mapping. At the `Relations` entry, the declaration of possible relationships follows the same rules of cardinality as the classes. The inverse relations specified under the `Inverse` entry must always refer only to one relation.

The specified entries in the `PARAMS` main entry of the configuration file for the generator are hardcoded and expected by the program. Only a change of the respective value of the property is possible if needed. The purpose of these parameters is to declare certain interpretation principles to the system. The `annotation_identifier`, `relation_identifier`, and `resource_identifier` tell the system how to interpret the identification tags of the annotation file.
In the default setting, the `annotation_identifier` is set to `T` and the `relation_identifier` is set to `R`, in that case an annotation starting with the identifier tag `T9`, is recognised as a component annotation and `R3` is identified as a relation. The `section_identidentifier` and `subsection_identifier` are to identify the respective delimiter tags in the text file to catch the indices of the respective title passages. These two parameters have first specified the beginning delimiter tag followed by the

ending delimiter tag. Every text structure deeper is considered a `subsubsection`. The `subsubsec_start` specifies beginning delimiter tags for such deeper text structures and the `subsubsec_end` specifies the ending delimiter tags.

The configuration file for the extractor contains two main entries, the `Component_Mapping` and the `Relations_Mapping`. This configuration file is simpler and just requires, for every component and relation in the model used to generate the PDF document we are currently extracting from, a one to one mapping with the description it should have in the newly created annotation file.

Relationships and type descriptions in an annotation file that are not declared and mapped in the configurations are disregarded and will neither be annotated in the resulting document nor referenced in the metadata. The same applies to the extraction process. Declaring classes and relationships for and from description tags not present in the processed corpus has no effect on the resulting documents besides a slight growth of the metadata structure and a change in the colours generated to highlight the annotated text passages.

With this approach, our application supplies a fast and simplistic way to represent, extend and modify the ontology used in every process run. An automated import for ontology files is not yet provided at this point in time.

It is for both configuration files of great importance, to maintain the overall structure of the main entries and to keep the cardinalities straight. If too many or too few entries are given to an entry, an expected application behaviour can not be guaranteed. The Listing A.1 shows the default configuration file of the generator and Listing A.2 the one of the extractor.

```
1  [PARAMS]
2  annotation_identifier = T
3  relation_identifier = R
4  resource_identifier = P
5  section_identifiers = <H1>,</H1>
6  subsection_identifiers = <H2>,</H2>
7  subsubsec_start = <H3>,<H4>
8  subsubsec_end = </H3>,</H4>
9
10 [Component]
11 claim = background_claim, own_claim
12 premise = data
13 resource = resource
14
15 [Relations]
16 attacks = contradicts
17 supports = supports
18 semantically_same = semantically_same
19 parts_of_same = parts_of_same
20
21 [Inverse]
22 attacks = Attackedby
23 supports = Supportedby
24 semantically_same = Semantically_sameto
```

```
25 parts_of_same = Parts_of_sameto
```

Listing A.1: The default configuration file of the generator module

```
1 [Component_Mapping]
2 claim = claim
3 premise = data
4 resource = resource
5
6 [Relation_Mapping]
7 attacks = contradicts
8 supports = supports
9 semantically_same = semantically_same
10 parts_of_same = parts_of_same
```

Listing A.2: The default configuration file of the extractor module

A.2 Run the Pipeline

The pipeline can either be run directly inside an IDE or from the command line. The pipeline call expects at least one and up to four input arguments.

Input directory: This is the first argument and not optional, it specifies the input directory for the pipeline call. An example call from the command line with just the input directory is shown in Listing A.3.

```
1     C:\some\path\2021-joel-watter\src> Python Annotation_pipeline.py
      "C:\some\input\directory"
2
```

Listing A.3: Call the pipeline only with a specified input directory on Windows

−−od Output directory: This second argument is optional to declare the output directory. An example call from the command line with an input and an output directory is shown in Listing A.4.

```
1     C:\some\path\2021-joel-watter\src> Python Annotation_pipeline.py
      "C:\some\input\directory" --od "C:\some\output\directory"
2
```

Listing A.4: Call the pipeline with an input and an output directory on Windows

−−extract: This flag is again optional and specified as the last argument. It can also be declared with just an input directory specification proceeding it. If it is absent, the pipeline will make a run to generate PDF documents from found corpus data in the directory. When this flag is declared, the pipeline is going to extract the data from found PDF files in the given input directory. In Listing A.5 an example call from the command line with just an input directory and the extraction flag is shown.

```
C:\some\path\2021-joel-watter\src> Python Annotation_pipeline.py
"C:\some\input\directory" --extract

```

Listing A.5: Call the pipeline for an extraction run with only a specified input directory on Windows

If no output directory is specified, the pipeline will store the produced output in generic directories inside its `data` directory. The local path for those directories are either `2021-joel-watter/src/data/PDF_Output` or `2021-joel-watter/src/data/Extraction_Output`.

To run the pipeline from the command line, it is necessary to first navigate to the directory of the `Annotation_pipeline.py` script as shown in Listing A.6.

```
C:\Users\somePath> cd C:\some\path\2021-joel-watter\src
```

Listing A.6: Navigate to the pipeline directory on Windows

When running the pipeline within the IDE, the arguments need to be specified in a list and given as a parameter for the call of the `main()` function of the `Annotation_pipeline.py` script. An example of such a configuration is shown in Listing A.7.

```
if __name__ == "__main__":

    args = ['C:\some\input\directory', '--od', 'C:\some\output\directory
', '--extract']
    main(args)
```

Listing A.7: Example run configuration within the IDE on a Windows machine

The implemented test suit is meant to be run from within the IDE.

A.3 Installation

By cloning the GitLab repository at *https://gitlab.ifi.uzh.ch/ddis/Students/Theses/2021-joel-watter* the code for the pipeline can be accessed.

To run, the pipeline requires Python 3.7 or higher to be installed.[1]

The pipeline was built using the PyCharm IDE.[2] For better visualisation of LaTeX specific lines of code, we recommend to install the TeXiFy IDEA plugin. [3]

For the compilation of the LaTeX files, the installation of s TexLive distribution with its extended (one package bigger than basic) is required.[4]

[1] *https://www.python.org/*
[2] *https://www.jetbrains.com/de-de/pycharm/*
[3] *https://plugins.jetbrains.com/plugin/9473-texify-idea*
[4] *https://www.tug.org/texlive/acquire-netinstall.html*

The required packages in python are listed in the `requirements.txt` file within the project directory. These can automatically be installed by using entering the command in Listing A.8 into the command line.

```
pip install -r /path/to/requirements.txt
```

Listing A.8: Command to install the required Python packages

List of Figures

4.1 Structure of the pipeline 19
4.2 Example of the generated PDF document for the file pair A01 from the corpus of Lauscher et al. [2018b] 35

List of Listings

4.1 Example of a component annotation (T9) and an annotated relation from the corpus of Lauscher et al. [2018b] . . . 20
4.2 Example of the component dictionary with two instantiated component and with the default configurations . . . 21
4.3 LaTeX colour definition resulting in a intense and vibrant marine blue . . 24
4.4 Example of the command definitions with the default configuration 25
4.5 Example of an embedded command for the component `T9` from the file pair *A01.txt* and *A01.ann* from the corpus of Lauscher et al. [2018b] . . . 27
4.6 Code of the object wrapping for the PDF attachment (code source) 29
4.7 Attach the files to the PDF (code source) . . . 30
4.8 Add the meta data to the PDF structure . . . 31
4.9 Example of an extracted relation that is written to the new annotation file 34

A.1 The default configuration file of the generator module . . . 50
A.2 The default configuration file of the extractor module . . . 51
A.3 Call the pipeline only with a specified input directory on Windows 51
A.4 Call the pipeline with an input and an output directory on Windows . . . 51
A.5 Call the pipeline for an extraction run with only a specified input directory on Windows . . . 52
A.6 Navigate to the pipeline directory on Windows . . . 52
A.7 Example run configuration within the IDE on a Windows machine 52
A.8 Command to install the required Python packages . . . 53