Semantic source code annotation using MAE

Sarvan Nebojsa, Manojlovic Nina, Nikolic Milica

Inteligent systems

Faculty of Techical Sciences

University of Novi Sad

sarvan.nebojsa, ninam94, milka94ns (@gmail.com)

Segedinac Milan, Savic Goran

Inteligent systems

Faculty of Techical Sciences

University of Novi Sad

milan.segedinac, savic071 (@gmail.com)

*Abstract* — Nowadays, when public opinion is available all over the Internet, it is very important to collect and properly analyze all the data. This data can be very valuable when using in strategic, political or marketing purposes. In order to process the data correctly, one could rely on variety of tools for text mining. When in posses of all the necessary information, there are lots of techniques for deeper analysis. For better understanding and providing computers to deal with data as good as humans, general semantic of the text should be recognized. Semantic web is the most popular weapon when it comes to this. Words and phrases are represented by ontologies, which help computers determine text sentiment or meaning. While our primary task is mainly related to IT domain, more specifically to education in IT, all the work is based on techniques mentioned above. This paper refers to existing tool for annotating source code and explains in details how to use it in combination with our specific tool in order to get the best possible results. Although the paper is more concentrated on combination of these two tools, our software can also be used independently for other kinds of purposes.

Key words — text mining; text annotation; source code annotation; annotating tool; MAE

# INTRODUCTION

Looking at fact that more of 80% of available data is unstructured, usage of text mining as a way of extracting high quality information from text is very important. As humans, we do this really efficiently and almost unconsciously. We filter the entire context surrounding a word or phrase, pull out the relevant pieces, compare them against our past experiences, and use them to deepen our understanding of the content at hand. Machines have historically failed at this because they lacked that kind of filter – the ability to determine what is relevant and why. In order to do this, we should consider putting individual words into context [1, 2]. Combining classical text mining with some kind of text annotation should do the job.

Text annotation is the practice and the result of adding a note to a text, which may include highlights or underlining, comments, footnotes, tags and links. Text annotation enriches content with machine processable information by linking background information to extracted concepts. These concepts, found in a document or another piece of content, are unambiguously defined and related to each other within and outside the content. It turns the content into better manageable data source [3]. From this point, text annotations can be referred to metadata insofar as it is added post hoc and provides information about a text without fundamentally altering its original form [4].

As we have already noticed, semantic annotation is a helpful technique to understand the under-laying semantics of the document. The problem with semantic annotations is that these annotations are not universal, so that semantic annotation for a document in particular domain might have different meaning in another domain. Therefore, domain specific knowledge is used for semantic annotation and this domain specific information is provided by ontologies [3, 5]. Although ontologies are not main subject of this paper, specific domain should be accentuated. In case considered, semantic annotation is strictly limited to the field of IT, more specifically to annotating source code.

Dealing with source code annotation is way more distinct from just annotating regular text. It introduces new kind of complexity, where more structured forms are expected for defining and describing concepts those annotations are related to.

In following, organizational structure of this paper is given. In Section II, we are going to refer to related work to this topic, published so far. We will try to present and analyze some of techniques used and explain how our solution should overcome spotted disadvantages. Afterwards, one specific tool used for annotating source code - MAE annotation tool, will be presented and covered in details. Next section will fully describe our work. Firstly, we will mention all of modifications done to adjust the MAE tool for the purpose of our research and finally our software for working with MAE output will be fully specified. Concrete case study is presented and discussed in Section IV. Lastly, Section V gives an overview of the paper and some possible directions of how this system can be upgraded in the future.

# RELATED WORK

When firstly introduced to our task, the main goal was to find an existing tool for source code annotation, rather than writing it from the scratch. From this point of view, we could more concentrate on the particular task and ways of providing better solution instead of dealing with programming issues. However, among plenty annotation tools it was not that easy finding convenient one. At the very start, ELAN [6, 7] was our best choice. After more and more researching, we gave up on this idea, based on the fact that ELAN is principally designed for multimedia annotations and that modifying its source code would take more time than implementing our own annotation tool from beginning. Since this idea has been dropped, we turned to GATE [8, 9] software. This amazing software has very good characteristics and it represented our main choice for a very long time. When it comes to text mining, more precisely text extraction, preprocessing and analysis, GATE provides well performing results. More importantly, GATE is capable of automatic text annotation, relying on existing ontologies. Due to its very complex structure and extra task on making an ontology that reflects our data, GATE also did not represent the best choice. After directing to just one specific programming environment, we have introduced ourselves to *Eclipse plugin* [10] that deals with source code annotations. However, this plugin incorporates all attached annotations directly with source code, which makes the original source code messy and harder to understand. What we needed was a simple tool that allows manual text annotation, whether it is free plain text or source code. Beside just selecting specific lines or sections, freedom in annotating the code with annotations that fit the task most properly and setting our own annotating structure was what really matters. At the end, we have found MAE [11, 12] as the most suitable tool for our work. We did some extra modifying of original tool in order to respond our needs in annotating process. Later, we upgraded it and combined it with our application so that we can complete the task using these particular tools.

## MAE annotation tool

MAE (*Multi-document Annotation Environment*) is an annotation tool for natural language text annotation. MAE is written in Java, thus Java version 8 or later is required for MAE to run. It is available in executable binary (jar) file or as a release package (zip).

Right after running MAE tool, defining an annotation task is first thing to do in order to use this tool properly. Creating an annotation task for MAE is fairly straightforward. The format of the input is a simplified DTDs (*Document Type Definition*) used for specifying XML elements/tags. There are three main parts of task creation: the task name, the tag names, and the tag attributes. DTDs are plain text files with a *.dtd* extension – you can create them in any text editing program. Current version of MAE does not support full functionality of DTD declarations.

The task name is defined with the *!ENTITY* tag. If you wanted to create a task called “myTask”, then create the *!ENTITY* line with *name* and the name of the task in double quotes, which you should write as:

<!ENTITY name “myTask”>

This simply provides the name of the root tag element in the annotation XML output files.

Tag elements (defined by *!ELEMENT*) are used to define the names of the tags being used in your annotation task and their attributes. MAE recognizes two types of tags: extent tags (tags used to label spans of text in the document) and link tags (tags that identify a relationship between extent tags). However, you cannot have two tags with the same name, even if they are of different types. To define an extent tag, the line in DTD file should look like this:

<!ELEMENT ExtentTagName (#PCDATA)>

while a link tag should look like:

<!ELEMENT LinkTagName EMPTY>

where (*#PCDATA*) indicates that the *ExtentTagName* will be associated with some span of text in the document, while *EMPTY* indicates *LinkTagName* will be used for linking other tags to one another, but will not be associated with a particular span of text. When MAE reads in a task definition, each extent tag is assigned a color to visualize tag instance over the document one is annotating. Since link tags do not consume text spans, it is not necessary to visualize by manipulating text colors. So, link tags would not be given colors.

Attributes (defined by the *!ATTLIST*) contain the information associated with each tag. Some attributes are pre-defined by MAE – extent tags will always have *id, spans,* and *text* attributes, even if they are not defined in the DTD. Link tags will always have *id* and at least two arguments, named *from* and *to* by default. To define attributes, one must include the name of the element that they are associated to, followed by the name of the attribute and the type of the attribute, like so:

<!ATTLIST TagName attribute1 (YES|NO) #IMPLIED>

<!ATTLIST TagName attribute2 CDATA #IMPLIED>

We will talk about the types of attributes later in this paper. Before we will go over the details of the special attributes pre-defined in MAE.

Every tag need an identifier to be distinguishable and referable. Thus, MAE will give *id* attribute to all tags defined in the task. When annotators create tags, each tag will be given a numerical id combined with a tag specific prefix. By default, MAE will use the first letter of the tag name. If you want to specify your own prefix, you can explicitly define *id* attribute and add *prefix* field to your element attribute, like:

<!ATTLIST Verb id ID prefix = “VB” #REQUIRED>

It is highly recommended that you choose your tag names, and prefixes carefully so as to make the distinctions clear to yourself and the rest of annotators. Relying on MAE to automatically prefix your tag identifiers is especially not recommended if you have multiple tag types whose names begin with the same character.

As previously mentioned, all extent tags have an attribute called *spans,* which denotes the character offset indices of the tag is anchored on. However, it is possible for an extent tag to be “non-consuming”; for instance, when you try to annotate hidden or omitted entities on the text, there won’t be any character offsets the tag can be anchored. By default, MAE will not allow a tag to be non-consuming, but by defining an extent tag’s *spans* attribute as *#IMPLIED*, opposed to *#REQUIRED*, MAE will allow that tag to be non-consuming

In MAE, an attribute value must be one of four followed types: *CDATA, ID, IDREF* and closed value set. *ID* suggests that an attribute will work as the identifier for the tag. A tag can have only one *ID* type attribute, and in MAE the pre-defined *id* attribute will be the one. *CDATA* means an attribute has a free text value. *IDREF* defines that an attribute is referencing another tag. *IDREF* attribute must have an id of the referent. Finally, it is possible in MAE to have a set of options for an attribute value, rather than asking the annotators to fill in their own values each time. If you want to have a list of values, create the attribute and include a list of possible values enclosed with parentheses, delimited by “|*”,* like:

<!ATTLIST TagName attribute1 (YES|NO|MAYBE) #IMPLIED>

You can set an attribute to be mandatory or optional, using *#REQUIRED* and *#IMPLIED* respectively. Annotators are supposed to fill all required attributes as they work. If an annotation work file (XML) has unfulfilled mandatory attributes, MAE will warn the user about the missing parts when it loads up or saves the work.

MAE allows you to set default values for any attribute by placing the desired value in quotes at the end of the attribute definition, like:

<!ATTLIST TagName attribute1 (YES|NO) #IMPLIED “YES”>

<!ATTLIST TagName attribute2 CDATA #IMPLIED “default”>

It is important that if a list of options is defined in an attribute but the default value does not appear in the list, MAE will not provide that default value when creating a new tag. On the other side, by giving an attribute a default value, annotators might skip annotating the attribute by mistake. Especially for required or important attributes, it is highly recommended not to provide a default value only to reduce human errors.

Once a valid task definition is provided, annotators can start annotation task by loading up the definition, using *[File] [New Task Definition].* MAE will generate tables corresponding to tags defined in the DTD in the bottom half of the interface (Image 1). As mentioned, each extent tag type has been assigned with a color. On the link-tag side, instead of colors, MAE will italicize text spans of extent tags (arguments) related to each link tag type.

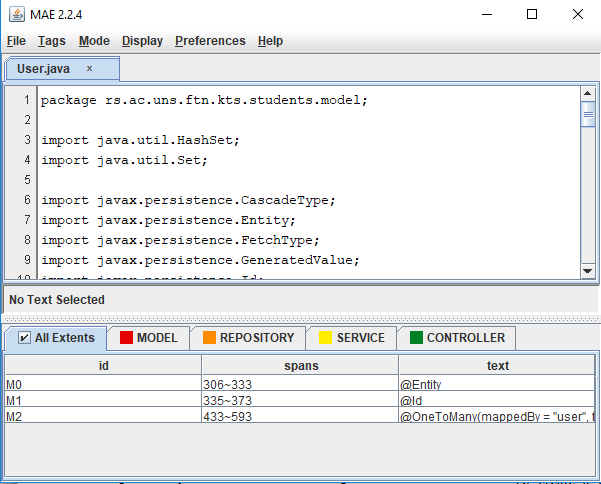


Image 1: MAE’s Graphic User Interface

MAE can load up a plain text file as a primary document for annotation or an existing XML annotation task. To load up a document use *[File][Open Document].* An existing XML file can be open as an annotation work, only when it matches the task definition name currently loaded (the root node of the XML). Otherwise MAE will open the XML as a plain text. The document will be showed up at the top half of the interface.

To create a consuming tag, first select text span using mouse cursor. With the text span highlighted, right click on the text to open a context menu of possible entity tag types. Selecting the desired type from the menu will create a tag instance, and it will immediately populate the corresponding table in the bottom. MAE will automatically generate *id* value and fill in *spans* and *text.* When a new tag is created, MAE will assign any default values for its attributes, if specified in DTD. Then annotators can annotate attribute values, using the table in the bottom. If an attribute has a closed set of possible values, MAE will use a drop-down menu for the attribute, allowing annotators to select one of the valid values. Double-click on a cell will start editing its value, if the cell is editable. Otherwise the corresponding tag is highlighted in the document area. Deletion operation is only achievable through the context menu after selecting desired tags, either from the table or on the text. Currently, MAE does not support auto-save or recovery feature, so it is highly recommended to make saves frequently.

# OUR WORK

In previous section we have introduced an existing annotation tool – MAE and fully described it in its original form. This part will also describe MAE, but now from the point of our, modified form. All of modifications will be mentioned and explained. Also, reasons for those modifications will be given. Later, we’ll present how the output of the modified MAE tool can be integrated and used for purpose of source code annotation.

## MAE modifications

As already mentioned, main goal of this research is annotation of the source code for educational purposes. MAE does the job when it comes to annotating source code, but for our specific task it failed in giving good results. Rather than doing all the work from the beginning, we have decided to use the existing solution, and upgrade it. First of all, it’s important to emphasize that we didn’t modified the source code on its own, but only made changes of their original DTD file.

As described in section II, MAE has quite complex structure. In order to simplify it, we left out multiple parts of its DTD file, which are surplus and irrelevant for our particular problem. These parts mainly refer to link tags and their attributes. What have left of original DTD was later modified in a way to support structure of files needed to be annotated. The files are the part of student project with unique structure that will be described in details in the next section.

Image 2 represents DTD file when modified.

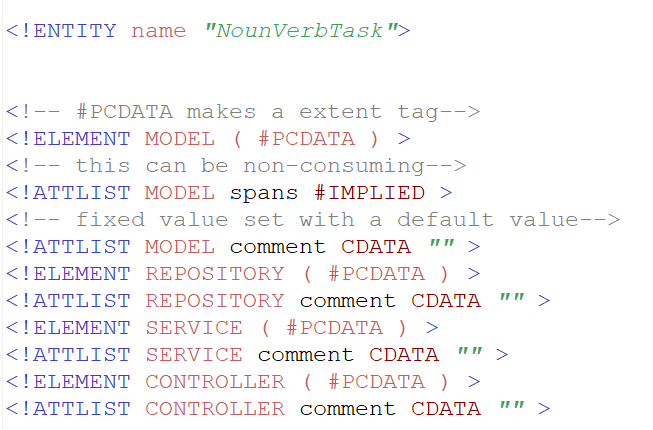


Image 2: Look of the modified .*dtd* file

DTD contains of four main elements – MODEL, REPOSITORY, SERVICE and CONTROLLER. Each element has the same attribute list, as follows: *id*, *spans*, *text*, *and comment*. First three attributes are already described in previous section. The last-mentioned attribute is added to expand the meaning of each element, more precisely to give the element semantic connotation. This attribute is represented by free text input field, and is left empty for an annotator to fill it. There is no specific standard for that, but we established the convention in order to have unique structured data that can be easily processed. Element titles were not chosen randomly, but in a way to follow the problem definition. Reasons for declaring those titles and also *comment* field structure will be given in section IV. Now, let’s turn to specification of our specific software and how it deals with the output that modified MAE tool provides.

## Our software

This software is, above all, made as an upgrade of MAE tool. However, it can be used independently for similar kinds of problems. The main goal is to do mapping of features given by MAE into specific form suitable for later analysis. This chapter provides information about software implementation, both as a guide of how to use it.

Software is written in Java programming language, so it is basically Java application. Input is the output of the MAE tool, more specifically, the XML files generated as a result of annotating process. XML contains all kind of knowledge of a file that was annotated using MAE. In our case, annotated file consists of source code, instead of plain text. *TEXT* tag contains this kind of information. It is followed by tag *TAGS* which represents all of user defined annotations. As mentioned in Chapter II (A), these tags consist of attributes needed for describing particular annotation. One of the main reasons for creating this tool is exactly the fact that the most important attribute is not in proper form for further analysis. Concretely, attribute *span* contains range between first and last character position of annotated area in the file that has been annotated. The way we need it to be is some kind of format that provides information about line in which annotated area starts and ends, both as columns with same meaning. So, our first task was to do the mapping from one to another appropriated form. This particular task required parsing XML files first, so we could gain originally annotated source code as well as access *span* attribute atall. One may not have all of the original files on its file system, so one way of accessing it would be to extract it from *TEXT* tag and save it as another temporary file. After having those values, one should call inbuilt *LineFromChar* function, as follows:

LineFromChar(File tempFile, int startChar, int EndChar)

where *tempFile* refers to location of temporary file, *startChar* represent position of first written character while *endChar* is position of last written character in annotated area. Return value of this function is a String that contains all the knowledge about lines and columns in which annotated text starts and ends. Due to numerous annotations with same title that can be found in multiple different files, output of our tool is represented as a collection of sections, looking as follows:

Section(String fileName, String content, String sectionType, int lineStart, int colStart, int colEnd, int lineEnd, String comment)

where *fileName* is a name of original source code file which contains this particular annotation, *content* is a piece of source code covered by annotation, *lineStart*, *lineEnd*, *colStart* and *colEnd* are respectively line in which annotated text starts, line in which annotated text end, column in which annotated text starts and column in which annotated text ends. *SectionType* as well as *comment* parameter are strictly tied to our case study, thus will be explained in details in Chapter IV.

Very this collection is mapped to JSON format, as it is the most suitable detected form for latter tasks. The whole process is shown on Image 3.

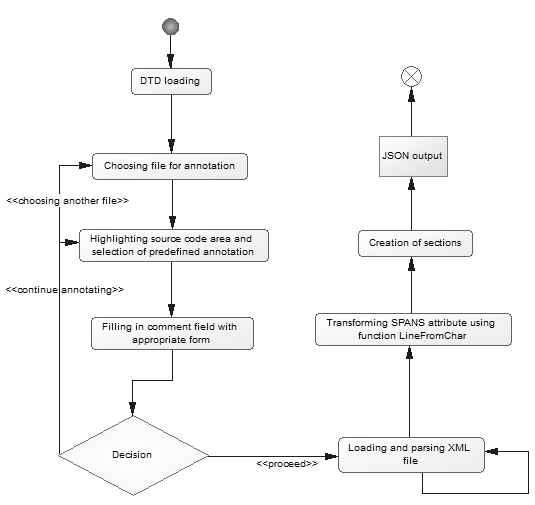


Image 3: Annotation process shown on Activity Diagram

# CASE STUDY

All of the work is done as a part of major project which purpose is to determine student’s knowledge in IT domain, based on eye movements. For this task Java Spring project is used as a case study, so model for predefined annotations and annotations themselves are specified for Java source code. Structure of project that is used is shown on Image 4.

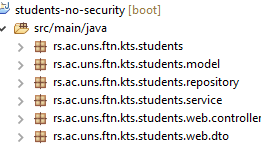


Image 4: Structure of a project used for case study

Based on project structure one can easily notice few obvious sections which we declared as predefined annotations. Followed by names of the packages, those annotations are titled as: *MODEL, REPOSITORY, SERVICE* and *COTROLLER.* Some of the classes does not carry any additional meaning, thus they are not relevant for this specific task and were not annotated. Each annotation has the same attributes which were explained in details in previous chapters. The most specific of them all is *comment* attribute, which is tightly connected to this particular case study. Depending on type of section, particularly if it is annotation in code, or part of a method, comment field can have three forms. Comment is a plain text field which should be written in one of the following formats:

* METHOD\_NAME;ENTITY , if section is referring to the whole method. In form above, *method\_name* represents name of the method which is being annotated, and *entity* stands for entity that the previous mentioned method is related to, mostly return value of specific method.
* METHOD\_NAME;ENTITY;SUBMETHOD\_DESCRIPTION , if section covers a piece of source code that is nested inside of a more complex method. *Submethod\_description* is a name (similar to the method name) that shortly describes logic behind highlighted part of method.
* ANNOTATION\_NAME;ENTITY;”annotation” , if section is reffering to an existing annotation. A*nnotation\_name* is the name of annotation (in our case Java annotation) and *entity* represents entity which has been annotated by previously mentioned Java annotation. The third part of the comment is a constant value *“annotation”* which indicates that a section is placed over Java annotation.

The form of this field is not standardized, but should follow mentioned convention.

Image 5 shows an example of all of the three forms of comment field. Part of the image marked with 1, is related to the first case where annotation represents the whole method. Blue box, numbered with 2, gives an overview of a piece of source code which is taken as a section, but is located within some other, more complex method. Number 3 is Java annotation section.

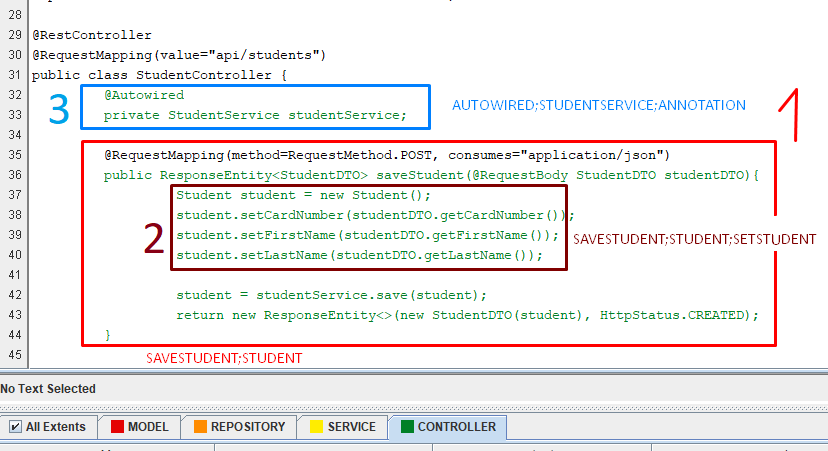


Image 5: Possible forms of *comment* field

As mentioned before, annotated source code is sent to further processing, and its output is represented by JSON file format. The snippet of the file is shown on Image 6.

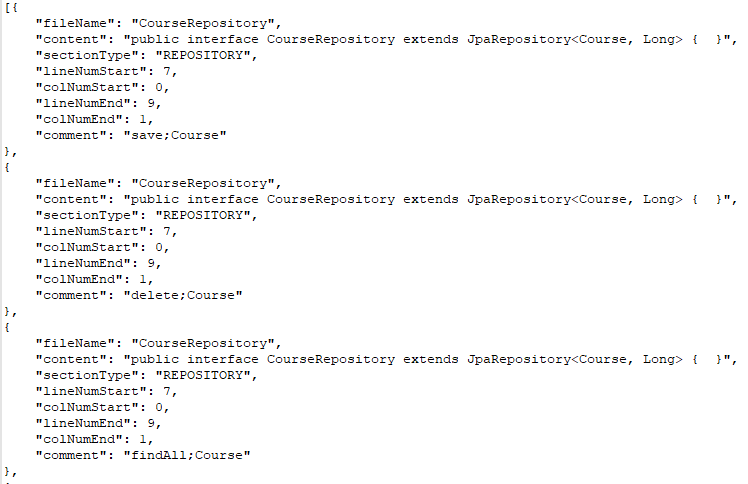


Image 6: Snippet of JSON output file

# CONCLUSION

This chapter gives an overview of all of the work done. It turns back to MAE’s major features and functionalities, our tool for processing and working with output of MAE annotation tool as well as the results of our work. Later in the chapter we will go through advantages and disadvantages of the tool and also all of the constraints will be mentioned. At the end we will give an idea of improving current solution and some more future work will be discussed.

Among plenty of tools for text annotation available nowadays, we have chosen MAE for supporting specific needs we had. The main goal was to find softwarethat can easily be used for annotating text, and more importantly annotating source code. Once the source code has been annotated, results are held in XML files which all integrated represents an input to our tool. This step implies parsing XML files and applying set of functions on original form in order to transform it in more convenient form for further analysis. These functions map ordinal number of character in annotated section to beginning and ending line and column number, and also preserve original file name from which section was taken. The result is represented in JSON file format.

When it comes to text annotation there is a variety of tool choices, which is not a case for annotating source code. At the same time this fact represents the biggest MAE advantage. Another reason for choosing MAE is that it is an independent tool, which hold and handle files for annotation by itself, rather than incorporating it in original source code which could change its original structure and make it more inconspicuous. Annotating source code turns out to be a big problem because of many different programming languages source code can be written into. Thus, specifying a software for one certain programming language could be both advantage and disadvantage at the same time. From that point of view this fact represents disadvantage of our system and also its huge constraint, since the software is restricted only to annotating Java Spring project. On the other hand, for ones working in this specific environment, our tool is making job much easier.

Regarding to all the work done, we have implemented only the parts tightly connected to our specific task. Beside functionalities described in this paper, there is still a lot of possibilities for enhancement in the field of text mining, extraction and interpretation of the data. All of this represents motivation for further directions in future work.

##### References

[1] Bo Pang and Lilian Lee, *Opinion Mining and Sentiment Analysis,* 2008.

[2] Vishal Gupta and Gurpreet S. Lehal, *A Survey of Text Mining Techniques and Application,* Journal of Emerging Technologies in Web Intelligence, 2009.

[3] Simone Teufel, Jean Carletta and Marc Moens, *An annotation scheme for discourse-level argumentation in research articles,* HCRC Language Technology Group and Human Communication Research Centre, Division of Informatics, University of Edinburgh

[4] Oscar Corcho, *Ontology based document annotation: trends and open research problems,* Intelligent Software Components (iSOCO), Spain

[5] Xian Wu, Lei Zhang and Yong Yu, *Exploring Social Annotations for the Semantic Web*

[6] Peter Wittenburg, Hennie Brugman, Albert Russel, Alex Klassmann and Han Sloetjes, *ELAN: a Professional Framework for Multimodality Research*, Max Planck Institute for Psycholinguistics, The Netherlands

[7] Maddalena Tacchetti, *User Guide for ELAN Linguistic Annotator*, 2017

[8] René Witte, Qiangquiang Li, Yonggang Zhang and Juergen Rilling, *Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach*, 2008.

[9] Ian H. Witten, Katherine J. Don, Michael Dewsnip and Valentin Tablan, *Text mining in a digital library,* 2003.

[10] Eclipse official documentation, Resource Marker, Available online: <https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm>

[11] Amber Stubbs, *MAE and MAI: Lightweight Annotation and Adjudication Tools,* Portland, 2011.

[12] Kyeongmin Rim, *MAE2: Portable Annotation Tool for General Natural Language Use,* Slovenia, 2016.