Originally published in: Dipper, Stefanie/Zeldes, Amir (Eds.):
Proceedings of the 14th Linguistic Annotation Workshop. December 12, 2020, Barcelona, Spain (Online).
– Stroudsburg: Association for Computational Linguistics, 2020. Pp. 167-173.

pyMMAX2: Deep Access to MMAX2 Projects from Python

Mark-Christoph Müller
Heidelberg Institute for Theoretical Studies gGmbH
Heidelberg, Germany
mark-christoph.mueller@h-its.org

Abstract

pyMMAX2 is an API for processing MMAX2 stand-off annotation data in Python. It provides a lightweight basis for the development of code which opens up the Java- and XML-based ecosystem of MMAX2 for more recent, Python-based NLP and data science methods. While pyMMAX2 is pure Python, and most functionality is implemented from scratch, the API re-uses the complex implementation of the essential business logic for MMAX2 annotation schemes by interfacing with the original MMAX2 Java libraries. pyMMAX2 is available for download at http://github.com/nlpAThits/pyMMAX2.

1 Introduction

MMAX2[1] (Müller and Strube, 2006) is a multi-level annotation tool implemented in Java, with a focus on rich, discourse-level features in small to mid-sized corpora. MMAX2 has been used in many different annotation projects (e.g. Desmet and Hoste (2010), Dipper et al. (2011), Liu (2011), Hendrickx et al. (2012), Schäfer et al. (2012), Martínez et al. (2016), Lapshinova-Koltunski et al. (2019), Faessler et al. (2020), and Uryupina et al. (2020)), and a considerable number of richly annotated data sets are available in the MMAX2 stand-off annotation format.

This short paper introduces pyMMAX2[2], an API for processing MMAX2 stand-off annotation data in Python (3.6 or higher). pyMMAX2 provides a lightweight basis for the development of code which opens up the Java- and XML-based ecosystem of MMAX2 for more recent, Python-based NLP and data science methods. This way, existing MMAX2 data sets are easily made available for processing in Python. At the same time, the creation of *new* MMAX2 annotation projects (either from raw or pre-annotated data) is facilitated, thus potentially lowering the entry barriers for using the tool for manual annotation. pyMMAX2 is written in pure Python, and most functionality is implemented from scratch, taking advantage of powerful libraries from the Python ecosystem. However, for the *annotation scheme* business logic, which constitutes an essential component of MMAX2, the API re-uses the complex original Java implementation, by transparently interfacing with the MMAX2 Java libraries at run time (cf. Section 2.2 below). It is in this sense that pyMMAX2 provides 'deep' access to MMAX2 projects.

pyMMAX2 is intended as a general-purpose API to support and encourage the development of modern, lightweight, and flexible Python code for accessing MMAX2 projects. As such, it replaces the built-in MMAX2 Project Wizard with its only rudimentary functionality, and the MMAX2 Discourse API (Müller and Strube, 2002) with its dependence on Java. Another way of creating MMAX2 data sets from pre-annotated data is by using one of the well-established annotation converters which support MMAX2 both as source and target format. These include e.g. PAULA (Chiarcos et al., 2008), the PEPPER framework (Zipser and Romary, 2010), or discoursegraphs (Neumann, 2015). For conversion among common, supported data formats, these tools are clearly the best choice. However, when it comes to processing pre-annotated data in a less common or entirely idiosyncratic format, more flexible

[1] https://github.com/nlpAThits/MMAX2

[2] Available for download at https://github.com/nlpAThits/pyMMAX2

The 14th Linguistic Annotation Workshop, pages 167–173
Barcelona, Spain (Online), December 12, 2020.

solutions are required, because adding support for a new format to e.g. PEPPER can quickly bring about development overhead which is unjustified for a one-off data conversion requirement.

When compared to newer tools like e.g. brat (Stenetorp et al., 2012), WebAnno (Yimam et al., 2013), and INCEpTION (Klie et al., 2018).[3], MMAX2 is quite different. These tools are all web-based, supporting distributed annotation, role-based annotation project management, and other advanced features. While there are clearly settings that can benefit from the infrastructure and control offered by these tools, including annotation projects with a large number of annotators, there is also a considerable overhead involved here. MMAX2, in contrast, is completely self-contained, light-weight, and requires no installation.

2 The pyMMAX2 API in a Nutshell

In MMAX2 parlance, the term *basedata* is used for tokenized text, which is the basis of every annotation project. Annotations for (potentially non-contiguous) sequences of basedata elements exist in the form of *markable* objects, which refer to the IDs of their underlying basedata elements by means of their `span` attribute, and which contain the actual annotations in the form of attribute-value pairs. Markables are organized in named *markable levels*, each of which can be dedicated to representing a particular (linguistic or other) phenomenon. For each markable level, there is a so-called *annotation scheme* which defines the attributes and permissible values resp. value combinations that can be assigned to markables on that level.
MMAX2 basedata, markables, and annotation schemes are all stored as XML files. For each MMAX2 project, references to these files are stored in a `.mmax` file and a `common_paths.xml`[4] file. Access to a MMAX2 project is established through a single *discourse* object, which serves as the single entry point from which all related data can be accessed.
With the exception of the annotation schemes (cf. Section 2.2), pyMMAX2 implements all of the above-mentioned objects (and several more) straightforwardly in pure Python, making use of tried and tested libraries like Beautiful Soup. This approach results in compact, fast, and easily extensible code. In general, the philosophy of pyMMAX2 is to offer lightweight Python objects with transparent functionality as a foundation for the development of more advanced annotation processing functionality.

2.1 Basedata Creation, Rendering, and Matching

Basedata elements are the smallest units that annotations can be associated with. They are individual tokens (mostly words and punctuation) in a simple XML format which assigns, as the only *required* attribute, an ID to each token, which is then used in markables' `span` attributes. In addition, basedata elements can have an arbitrary number of user-defined, XML-compliant attributes. These basedata-level attributes are expressly *not* intended for storing annotations (which are supposed to be on the level of markables), but they are used by the pyMMAX2 BASEDATA class for storing tokenization meta-information. This class contains the method `add_elements_from_string()` for creating basedata elements from raw Unicode strings. The method includes a basedata tokenizer which uses Unicode character categories[5] for determining token boundaries in the input string in a universal, language-independent way. This procedure is rather aggressive, creating considerably more and shorter tokens than 'standard' tokenizers, which are often only sensitive to white space and punctuation. Most notably, the basedata tokenizer will also split the input string at word-internal non-word characters, including hyphens. As a result, hyphenated words (e.g. noun compounds and other hyphenated multi-word expressions) will be spread across several contiguous basedata elements, allowing for more fine-grained annotation.[6] At the same time, the tokenizer keeps track of the original input string composition, including white space, and stores, for

[3] See also Neves and Ševa (2019) for an extensive overview.

[4] As the name implies, the latter file collects information that is used by more than one MMAX2 project. By default, each MMAX2 project expects a `common_paths.xml` file in the same folder as the `.mmax` file, but this can be overridden by supplying a different file at startup (cf. the example in Section 2.2 below).

[5] `http://www.unicode.org/reports/tr44/#General_Category_Values`

[6] This is particularly useful for expressions appearing in both *open* and *hyphenated* spellings. It prevents the arbitrary spelling difference from introducing two separate and one hyphenated basedata elements, which would be completely unrelated due to the atomic nature of basedata elements, while they actually represent the same lexemes.

every basedata element, the number of leading white space characters <> 1 (default) in the element's spc attribute. This tokenization meta-data can be utilized in the following ways: First, it is available to the XSL style sheets which are used to build the MMAX2 GUI for annotation. Second, and more importantly, it is used by the render_string() method of the BASEDATA class, which accepts a list of basedata element IDs and returns the reconstructed original input string, a list containing the basedata elements' text, and a dictionary which maps string positions (as keys) to basedata IDs (as values). The following code snippet demonstrates this functionality.

```
from pymmax2 import pyMMAX2 as pmx
bd = pmx.Basedata('sample_basedata.xml', encoding='utf-8')
bd_ids = bd.add_elements_from_string('Self-isolation for 14 days is required!')
s,w,m = bd.render_string(for_ids=[bd_ids], mapping=True)
print("%s\n%s\n%s"%(s,w,m))

>> Self-isolation for 14 days is required!
>> ['Self', '-', 'isolation', 'for', '14', 'days', 'is', 'required', '!']
>> {0: 'word_0', 1: 'word_0', 2: 'word_0', 3: 'word_0', 4: 'word_1', ...
   34: 'word_7', 35: 'word_7', 36: 'word_7', 37: 'word_7', 38: 'word_8'}
```

The spc attribute default value of 1 makes sure that the method also works with older MMAX2 basedata. The BASEDATA class also supports tokenization-agnostic regular expression (RegEx) matching, which uses the above method to render the original string for all or selected basedata IDs, applies a list of one or more RegExs, and returns, for each RegEx, the IDs of the matched basedata elements. RegExs are required to have a named capturing group m (see example below), which is used to determine the relevant part of the match. Optionally, a descriptive label can be assigned to each RegEx, which is returned alongside all of its matches. The following code snippet demonstrates the matching functionality, where each RegEx yields exactly one match.

```
# This uses the previously defined bd and bd_ids objects
for m,p,l in bd.match_string([(r'(?P<m>\S-\S)','hyphen'),(r'(?P<m>\d)','numeric')], for_ids=[bd_ids]):
    print("%s\t%s"%(m,l))

>> [[['word_0', 'word_1', 'word_2']]] hyphen
>> [[['word_4']], [['word_4']]] numeric
```

2.2 Annotation Schemes & Validation

For each markable level in a MMAX2 project, there can be one *annotation scheme* which defines the attributes and permissible values resp. value combinations that can be assigned to markables on that level. Annotation schemes can be very complex and expressive, which makes them a central component of MMAX2. They can include dependencies between attributes, such that some attribute will only be valid (and available for manual annotation in the MMAX2 GUI) if some other attribute has a particular value. From the viewpoint of user ergonomics in an actual manual MMAX2 annotation setting, a well-designed annotation scheme can support annotators by prompting them to provide values for only those attributes that are permissible in the current situation, based on previous annotations of the same markable, and by doing so *one attribute at a time*. And, even more importantly, from the viewpoint of annotation data management, annotation schemes are essential for enforcing correctness and consistency of annotations by means of validation. Due to their essential role and complex annotation business logic, the handling of annotation schemes is not re-implemented in the pyMMAX2 API. Instead, a *hybrid* approach is used which accesses functionality from the original MMAX2 Java code base in the background. Technically, this is realized at Python execution time by instantiating and accessing MMAX2 Java classes by means of JPype[7]. JPype allows Python programs full access to Java class libraries, interfacing both the Python and Java virtual machines at the native level. The following code snippet demonstrates the functionality by opening a file from the ACL Anthology Corpus (Schäfer et al., 2012) which consists of ACL papers annotated for coreference. The original annotation was done with MMAX2 and the dataset is freely available[8].

[7] https://github.com/jpype-project/jpype/

[8] http://dfki.de/~uschaefer/C12-2103-dataset/C12-2103-dataset.zip

```
from pymmax2 import pyMMAX2 as pmx
import jpype
MMAX2_CP = PATH_TO_MMAX2_LIBS # path to all MMAX2-related jar files
jpype.startJVM(jpype.getDefaultJVMPath(), '-Djava.class.path='+MMAX2_CP)

pd = pmx.MMAX2Discourse('./C12-2103/annotation/C/C02-1001/C02-1001.mmax',
                        common_paths='./C12-2103/common/common_paths.xml',
                        mmax2_java_binding=jpype)
try:
    pd.load_markables()
except pmx.MultipleInvalidMMAX2AttributeExceptions as mive:
    print('%s exceptions, e.g.\n%s'%(str(mive.get_exception_count()),
                                     str(mive.get_exception_at(0)).strip()))
pd.info()

>> MMAX2 Project Info:
>> -------------------
>> .mmax file        : ./C12-2103/annotation/C/C02-1001/C02-1001.mmax
>> Basedata elements : 2471
>> Markable levels   :
>>  coref            : 216 markables [default: <>NP_Form:none, Sure:yes]
>>  sentence         : 126 markables [default: imported_tag_type:]
```

The existing MMAX2 project is accessed through a MMAX2DISCOURSE object, which is created by providing the `.mmax` file name and (optionally) a `common_paths` parameter which allows to use an alternative `common_paths.xml` file.[9] After the creation of the MMAX2DISCOURSE object, markables are loaded via an explicit call to its `load_markables()` method. The subsequent call to the `info()` method prints some project info to the console. It includes the default attribute-value pairs for each markable level, which are obtained by querying the MMAX2 Java code for annotation scheme handling in the background.[10]

When calling `load_markables()`, all markables on all markable levels will be loaded *and validated* against their respective annotation schemes. In line with the general pyMMAX2 philosophy mentioned in Section 2 above, the role of validation is to detect and inform about annotation inconsistencies, but it does not include fixing or consolidating them. This is mainly because, at least for a significant subset of possible annotation inconsistencies, there is no simple, *default* way of dealing with them. Validation is implemented on the level of the individual markable, because it is here that annotation inconsistencies occur. Markable attributes are set via the `set_attributes()` method, which will raise an INVALIDMMAX2ATTRIBUTEEXCEPTION if inconsistencies are detected. The provided attributes, however, will be set nonetheless. The raised exception provides details about the validation problem, making it the user's responsibility to decide how to handle it. In order to allow bulk loading and validation of markables (as in the `load_markables()` method), all exceptions raised while validating individual markables are collected, and if at least one exception was raised, a MULTIPLEINVALIDMMAX2ATTRIBUTEEXCEPTIONS exception will be raised in turn, which provides access to detailed information on the contained exceptions. The effect of validation can be observed when temporarily removing the possible value `ne` from the `np_form` attribute in the annotation scheme for the coref level. Running the above code again will produce the output below.

```
58 exceptions, e.g.
Level: coref, ID: markable_825
Supplied: {'coref_class': 'set_114', 'sure': 'yes', 'np_form': 'ne'}
Valid:    {'sure': 'yes'}
Extra:    {'coref_class': 'set_114', 'np_form': 'ne'}
```

Note how the `coref_class` attribute is not recognized as valid because it depends on the `np_form` attribute having a valid value itself.

2.3 Creating a New MMAX2 Project from Pre-Annotated Data

The LitCovid dataset[11] (Chen et al., 2020) contains almost 45,000 Covid-19-related documents. It has annotations in the BioC XML format (Comeau et al., 2013) for six different biomedical entity types. In

[9] For collections consisting of many homogeneous MMAX2 projects (like the ACL Anthology Corpus), providing a *global* `common_paths.xml` file in this way is strongly recommended, because it allows to maintain properties shared by all projects (e.g. adding and removing of markable levels, styles, customizations, etc.) in a single place.

[10] If no `mmax2_java_binding` parameter is provided, this information is not available.

[11] `https://ftp.ncbi.nlm.nih.gov/pub/lu/LitCovid/`

a nut shell, each document comes as a sequence of <passage> elements, each containing a <text> element and one or more <annotation> elements, which associate annotations with the passage text by means of character offsets and lengths. The following two code fragments demonstrate the creation of a new MMAX2 project from a single LitCovid document (ID 32393453). Two simple annotation schemes for the passages and the entities level, covering all defined values plus a none value as default, were created manually. The first code snippet creates some helper objects and a rudimentary MMAX2DISCOURSE object with two initially empty markable levels. In this case, no value is provided for mmax2_java_binding, so that no validation will be performed during project creation.

```
dirname = './litcovid-mmax2/'
mmax2project = pmx.MMAX2Project(dirname+'project', {'words':'words.xml'})
mmax2project.write()
cp_path = 'common_paths.xml'
cp = pmx.MMAX2CommonPaths(dirname+cp_path)
cp.write(overwrite=True)
pmx_disc = pmx.MMAX2Discourse(mmax2project.get_mmax2_path(full=True), common_paths=dirname+cp_path)
passages_level=pmx_disc.add_markable_level('passages',
               namespace='xmlns="www.pymmax2.org/NameSpaces/passages"', create_if_missing=True)
entities_level=pmx_disc.add_markable_level('entities',
               namespace='xmlns="www.pymmax2.org/NameSpaces/entities"', create_if_missing=True)
```

The second code snippet creates basedata and markables and assigns attributes to the latter. Markables on the passages level are directly created along with the basedata chunks they cover. The add_markable method creates a new markable and returns the tuple (True, new markable) *only* if no other markable with an identical span exists on the level, in which case it returns (False, existing markable).

```
with open('samplelitcovid.xml', 'r') as f:
    contents = f.read()
    soup = bs(contents, 'lxml')
    neg_offset=0
    for p in soup.find_all('passage'):
        p_text = p.find('text').text
        p_type = p.find('infon',{'key':'type'}).text.lower()
        new_m, pm = passages_level.add_markable([pd.add_basedata_elements_from_string(p_text)])
        if new_m:    pm.set_attributes({'type':p_type})
        mt,_,pos2id = pd.get_basedata().render_string(pm.get_spanlists(), mapping=True)
        for anno in p.find_all('annotation'):
            offset = int(anno.find('location')['offset'])
            length = int(anno.find('location')['length'])
            e_type = anno.find('infon',{'key':'type'}).text.lower()
            anno_span=[]
            for f in range(length):
                try:
                    if pos2id[f+offset-neg_offset] not in anno_span:
                        anno_span.append(pos2id[f+offset-neg_offset])
                except KeyError:
                    pass
            new_m, pm = entities_level.add_markable([anno_span])
            if new_m:    pm.set_attributes({'entity_type':e_type})
        neg_offset=neg_offset+(len(mt))
pd.get_basedata().write(to_path=dirname, dtd_base_path='"', overwrite=True)
passages_level.write(to_path=dirname, overwrite=True)
entities_level.write(to_path=dirname, overwrite=True)
pd.get_commonpaths().write(overwrite=True)
```

Finally, the newly created MMAX2 project can be re-loaded from scratch like the previous example in Section 2.1, including 'deep' access to the related annotation schemes. Calling the info() method will produce the following output, including correct default values for both markable levels.

```
MMAX2 Project Info:
-------------------
.mmax file        : ./litcovid-mmax2/project.mmax
Basedata elements : 179
Markable levels   :
 passages         : 2 markables [default: type:none]
 entities         : 4 markables [default: entity_type:none]
```

Acknowledgements

The work described in this paper was done as part of the project DeepCurate, which is funded by the German Federal Ministry of Education and Research (BMBF) (No. 031L0204) and the Klaus Tschira Foundation, Heidelberg, Germany. We thank the anonymous reviewers for their helpful suggestions.

References

Q. Chen, A. Allot, and Z. Lu. 2020. Keep up with the latest coronavirus research. *Nature*, 579(7798):193.

Christian Chiarcos, Stefanie Dipper, Michael Götze, Ulf Leser, Anke Lüdeling, Julia Ritz, and Manfred Stede. 2008. A flexible framework for integrating annotations from different tools and tag sets. *Trait. Autom. des Langues*, 49(2):217–246.

Donald C. Comeau, Rezarta Islamaj Dogan, Paolo Ciccarese, Kevin Bretonnel Cohen, Martin Krallinger, Florian Leitner, Zhiyong Lu, Yifan Peng, Fabio Rinaldi, Manabu Torii, Alfonso Valencia, Karin Verspoor, Thomas C. Wiegers, Cathy H. Wu, and W. John Wilbur. 2013. Bioc: a minimalist approach to interoperability for biomedical text processing. *Database J. Biol. Databases Curation*, 2013.

Bart Desmet and Véronique Hoste. 2010. Towards a balanced named entity corpus for dutch. In *LREC*. European Language Resources Association.

Stefanie Dipper, Christine Rieger, Melanie Seiss, and Heike Zinsmeister. 2011. Abstract anaphors in german and english. In *DAARC*, volume 7099 of *Lecture Notes in Computer Science*, pages 96–107. Springer.

Erik Faessler, Luise Modersohn, Christina Lohr, and Udo Hahn. 2020. Progene - A large-scale, high-quality protein-gene annotated benchmark corpus. In *LREC*, pages 4585–4596. European Language Resources Association.

Iris Hendrickx, Amália Mendes, and Silvia Mencarelli. 2012. Modality in text: a proposal for corpus annotation. In *LREC*, pages 1805–1812. European Language Resources Association (ELRA).

Jan-Christoph Klie, Michael Bugert, Beto Boullosa, Richard Eckart de Castilho, and Iryna Gurevych. 2018. The INCEpTION platform: Machine-assisted and knowledge-oriented interactive annotation. In *Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations*, pages 5–9. Association for Computational Linguistics, Juni.

Ekaterina Lapshinova-Koltunski, Cristina España-Bonet, and Josef van Genabith. 2019. Analysing coreference in transformer outputs. In *DiscoMT@EMNLP*, pages 1–12. Association for Computational Linguistics.

Chaopeng Liu. 2011. Application of MMAX2 tool in chinese-english parallel corpus building. In *CSISE (3)*, volume 106 of *Advances in Intelligent and Soft Computing*, pages 697–700. Springer.

José Manuel Martínez Martínez, Ekaterina Lapshinova-Koltunski, and Kerstin Kunz. 2016. Annotation of lexical cohesion in english and german: Automatic and manual procedures. In *KONVENS*, volume 16 of *Bochumer Linguistische Arbeitsberichte*.

Christoph Müller and Michael Strube. 2002. An API for discourse-level access to xml-encoded corpora. In *LREC*. European Language Resources Association.

Christoph Müller and Michael Strube. 2006. Multi-level annotation of linguistic data with MMAX2. In Sabine Braun, Kurt Kohn, and Joybrato Mukherjee, editors, *Corpus Technology and Language Pedagogy: New Resources, New Tools, New Methods*, pages 197–214. Peter Lang, Frankfurt a.M., Germany.

Arne Neumann. 2015. discoursegraphs: A graph-based merging tool and converter for multilayer annotated corpora. In *NODALIDA*, volume 109 of *Linköping Electronic Conference Proceedings*, pages 309–312. Linköping University Electronic Press / Association for Computational Linguistics.

Mariana Neves and Jurica Ševa. 2019. An extensive review of tools for manual annotation of documents. *Briefings in Bioinformatics*, 12. bbz130.

Ulrich Schäfer, Christian Spurk, and Jörg Steffen. 2012. A fully coreference-annotated corpus of scholarly papers from the ACL anthology. In *Proceedings of COLING 2012: Posters*, pages 1059–1070, Mumbai, India, December. The COLING 2012 Organizing Committee.

Pontus Stenetorp, Sampo Pyysalo, Goran Topic, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. 2012. brat: a web-based tool for nlp-assisted text annotation. In *EACL*, pages 102–107. Association for Computational Linguistics.

Olga Uryupina, Ron Artstein, Antonella Bristot, Federica Cavicchio, Francesca Delogu, Kepa J. Rodriguez, and Massimo Poesio. 2020. Annotating a broad range of anaphoric phenomena, in a variety of genres: the arrau corpus. *Natural Language Engineering*, 26(1):95–128.

Seid Muhie Yimam, Iryna Gurevych, Richard Eckart de Castilho, and Chris Biemann. 2013. WebAnno: A flexible, web-based and visually supported system for distributed annotations. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 1–6, Sofia, Bulgaria, August. Association for Computational Linguistics.

Florian Zipser and Laurent Romary. 2010. A model oriented approach to the mapping of annotation formats using standards. In *Workshop on Language Resource and Language Technology Standards, LREC 2010*, La Valette, Malta, May.