A broad-coverage collection of portable NLP components for building shareable analysis pipelines

Richard Eckart de Castilho[1] **Iryna Gurevych**[1,2]
(1) Ubiquitous Knowledge Processing Lab (UKP-TUDA)
Dept. of Computer Science, Technische Universität Darmstadt
(2) Ubiquitous Knowledge Processing Lab (UKP-DIPF)
German Institute for Educational Research and Educational Information
http://www.ukp.tu-darmstadt.de

Abstract

Due to the diversity of natural language processing (NLP) tools and resources, combining them into processing pipelines is an important issue, and sharing these pipelines with others remains a problem. We present DKPro Core, a broad-coverage component collection integrating a wide range of third-party NLP tools and making them interoperable. Contrary to other recent endeavors that rely heavily on web services, our collection consists only of portable components distributed via a repository, making it particularly interesting with respect to sharing pipelines with other researchers, embedding NLP pipelines in applications, and the use on high-performance computing clusters. Our collection is augmented by a novel concept for automatically selecting and acquiring resources required by the components at runtime from a repository. Based on these contributions, we demonstrate a way to describe a pipeline such that all required software and resources can be automatically obtained, making it easy to share it with others, e.g. in order to reproduce results or as examples in teaching, documentation, or publications.

1 Introduction

Sharing is a central concept to scientific work and to software development. In science, information about experimental setups and results is shared with fellow researchers, not only to disseminate new insights, but also to allow others to validate results or to improve on them. In software development, libraries and component-based architectures are a central mechanism to promote the reuse of software. Portable software must operate in the same way across system platforms. In the context of scientific research, this is an important factor related to the reproducibility of results created from software-based experiments.

The NLP software landscape provides a wealth of reusable software in the form of NLP tools addressing language analysis at different levels from tokenization to sentiment analysis. These tools are combined into NLP pipelines that form essential parts of experiments in natural language research and beyond, e.g. in the emerging digital humanities. Therefore, it is essential that such pipelines can easily be shared between researchers, to reproduce results, to evolve experiments, and to allow for a better understanding of the exact details of an experiment (cf. Fokkens et al. (2013)).

Analyzing the current state of the art, we find that despite considerable effort that has been going into processing frameworks enabling interoperability, workbenches to build and run pipelines, and all kinds of online services, it is still not possible to create a readily shareable description of an NLP pipeline. A pipeline description is basically a configuration file referencing the components and resources used by the pipeline. Currently, these references are ambiguous, e.g. because they do not incorporate version information. This causes a reproducibility problem, e.g. when a pipeline is part of an experiment, because the use of a different version can easily lead to different results. A sharable description must be self-contained in the sense that it uniquely identifies all involved components and resources, permitting the execution environment for the pipeline to be set up reproducibly, in the best case automatically. Currently, the task of setting up the environment is largely left to the user and requires time and diligence.

Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT, pages 1–11,
Dublin, Ireland, August 23rd 2014.

In this paper, we present a novel concept for self-contained NLP pipeline descriptions supported by a broad-coverage collection of interoperable NLP components. Our approach is enabled by the combination of distributing portable NLP components and resources through a repository and by an auto-configuration mechanism allowing components to select suitable resources at runtime and to obtain them automatically from the repository. Our contributions facilitate the sharing of pipelines, e.g. as part of publications or examples in documentation, and allow users to maintain control by providing the ability to create backups of components and resources for a later reproduction of results.

Section 2 reflects on the state of the art and identifies the need for a broad-coverage component collection of portable components, as well as the need for self-contained pipeline descriptions. Section 3.1 describes a novel concept for the automatic selection and acquisition of resources. Section 3.2 presents a broad-coverage collection of portable components integrating this concept. Section 3.3 demonstrates a shareable workflow based on these contributions. Section 4 gives further examples of how our contributions could be applied. Finally, Section 5 summarizes the paper and suggests future research directions.

2 State of the art

In this section, we examine the current state of the art related to describing NLP pipelines, kinds of component collections, publishing of components and resources, and the selection of resources to use with a component. We start by defining the terminology used throughout the rest of this paper.

Definition of terminology We make a distinction between a *tool* and a *component*. Most NLP tools are standalone tools addressing one particular task, e.g. dependency parsing, relying on separate tokenizers, part-of-speech (POS) taggers, etc. These tools cannot be easily combined into pipelines, because their input/output formats are often not compatible and because they lack a uniform programming interface.

We speak of a *component* when a tool has been integrated into a *processing framework*, usually by implementing an adapter between the tool and the framework. The framework defines a uniform programming interface, data model, and processing model enabling interoperability between the components.

Through integration with the framework, components become easily composeable into *pipelines*. A *pipeline* consists of *components* processing input documents one after the other and passing the output on to the next component. Each component adds *annotations* to the document, e.g. sentence and token boundaries, POS tags, syntactic constituents, or dependency relations, etc. These steps build upon each other, e.g. a component for dependency parsing requires at least sentences, tokens, and POS tags.

Many components are generic engines that require some *resource* (e.g. a *probabilistic model* or *knowledge base*) that configures them for a specific language, tagset, domain, etc. We use the term *resource selection* for the task of choosing a resource and configuring a component to use it. The task of obtaining the resource we call *resource acquisition*.

As Thompson et al. (2011) point out, achieving a consensus on the exact representation of different linguistic theories as annotations and thereby attaining full conceptual interoperability between components from different vendors is currently not considered feasible. Thus, frameworks leave the type system (kinds of annotations) unspecified. Therefore, the technical integration with a framework alone does not make the tools interoperable on the conceptual level (cf. Chiarcos et al. (2008)). As a consequence, multiple *component collections* exist, each providing interoperable components centered around a particular combination of processing framework and type system (e.g. Buyko and Hahn (2008), Kano et al. (2011), Wu et al. (2013)). Each of these defines its own concepts of tokens, sentence, syntactic structures, discourse structures, etc. Yet, even these type systems leave certain aspects underspecified, e.g. the various tagsets used to categorize parts-of-speech, syntactic constituents, etc.

2.1 Sharing pipelines

Processing frameworks offer a way to construct pipeline descriptions that instruct the framework to configure and execute a pipeline of NLP components.

GATE (Cunningham et al., 2002) and Apache UIMA (Ferrucci and Lally, 2004) are currently the most prominent processing frameworks. Both describe pipelines by means of XML documents. These refer to individual components by name and expect that the user has taken precautions that these components are

accessible by the framework. Neither framework includes provisions to automatically obtain the components or resources they require, e.g. from a repository (Section 2.2). In fact, the pipeline descriptions do not contain sufficient information to uniquely identify components. Components are referred to only by name, but not by version. The same is true for resources which are often referred to only by filename.

Thus, both of the major processing frameworks to not offer self-contained descriptions for pipelines. When such pipeline descriptions are shared with another person, the recipient requires additional information about which exact versions of tools and resources are needed to run the pipeline.

2.2 Publishing components and resources

In this section, we examine different approaches to publishing NLP components and resources so that they can be more easily found, accessed, or obtained in order to execute a particular pipeline.

Directories META-SHARE (Thompson et al., 2011) and the CLARIN Virtual Language Observatory (VLO) (Uytvanck et al., 2010) are two directories of language resources, including NLP tools and resources. These directories currently target primarily human users and offer rich metadata and a user interface to browse it or to find specific kinds of entries. However, these directories to not contain sufficient information to programmatically download the software or resources, or to access them as services.

Repositories Repositories are online services from which components and resources can be obtained.

The Central Repository (2014) is a repository within the Java-ecosystem used to distribute Java libraries and resources they require, so-called *artifacts*. It relies on concepts that have evolved around the Maven project (Sonatype Company, 2008). Meanwhile, these are supported by many build tools, development environments, and even by some programming languages (cf. Section 3.3). Several NLP tools (e.g. ClearNLP (2014), Stanford CoreNLP (Manning et al., 2014), MaltParser (Nivre et al., 2007), ClearTK (Ogren et al., 2009)) are already distributed via this medium, some including their resources.

There are many Maven repositories on the internet. They are organized as a loosely federated network. The Central Repository merely serves as the default point of contact built into clients. Repositories have the ability to access each other and to cache those artifacts required by their immediate users. This provides resilience against network failures or remote data loss. Artifacts can be addressed across the federation by a set of coordinates (*groupId*, *artifactId*, and *version*).

Another kind of repositories are plug-in repositories, such as those used by GATE (Cunningham et al., 2002). From these, the user can conveniently download and install components within the GATE workbench. These plug-in repositories are specific to GATE, whereas the Maven repositories are a generic infrastructure widely used by the Java community and that is supported by many tools and applications.

Online Workbenches While many NLP tools are offered as *portable software* for offline use, we observe a trend in recent years towards offering NLP tools as *web-services* for online use, sometimes as the only way to access them. Hinrichs et al. (2010) cite incompatibilities between the software and the user's machine and insufficiently powerful workstations as reasons for this approach. Another reason may be the ability to set up a *walled garden* in which the service provider is able to control the use of services, e.g. to academic researchers or to paying commercial customers.

Argo (Rak et al., 2013) is a web-based workbench. It offers access to a collection of UIMA-based NLP-services that can be executed in different environments. Rak et al. mention in particular a cluster environment but also plan support for a number of cloud platforms. For this reason, we assume that most of the components are integrated into Argo as portable software that can be deployed to these platforms on-demand. Yet, it appears that the components are only accessible through Argo and that they are not distributed separately for use in other UIMA-based environments.

U-Compare (Kano et al., 2011) is a Java application for building and running UIMA-based pipelines and comparing their results. While some components accessible through the workbench run locally, many components are only stubs calling out to web-services running at different remote locations.

WebLicht (Hinrichs et al., 2010) is a distributed infrastructure of NLP services hosted at different locations. They exchange data in an XML format called TCF (*Text Corpus Format*). Pipelines can be built

and run using the web-based WebLicht workbench. Within this walled garden platform, authenticated academic users have access to resources that are free for academic research, but not otherwise.

Online Marketplaces AnnoMarket (Tablan et al., 2013) is another distributed infrastructure of NLP services based on GATE. It does not seem to offer a workbench to compose custom pipelines. Instead, it offers a set of pre-configured components and exposes them as web-services to be programmatically accessed. It is the only commercial offering in this overview that the user has to pay for.

Note on service-based approaches Service-based approaches have also been taken in other scientific domains to facilitate the creation of shareable and repeatable experiments, e.g. on platforms such as myExperiment (Goble et al., 2010). However, Gómez-Pérez et al. (2013) found service-based workflows to be subject to decay as services are updated and change their input/output formats, their results, or as they become temporarily unavailable due to network problems. We also expect they can become permanently unavailable, e.g. due to a lack of funding unless supported by a sound business model.

Furthermore, to our knowledge none of the offerings above allow the user to export their pipelines including all necessary software and resources, e.g. to make a backup or to deploy it on a private computing infrastructure, e.g. a private cloud or cluster system.

2.3 Component collections

We define a component collection as a set of interoperable components. The interoperability between the components is enabled by conventions that are typically rendered as a common annotation type system, a common API, or both.

Standalone NLP tools Most NLP tools are not comprehensive suites that cover all tasks from tokenization to e.g. coreference resolution, but are rather standalone tools addressing only a particular task, e.g. dependency parsing, relying on separate tokenizers, POS-taggers, etc. Examples of such standalone tools are MaltParser (Nivre et al., 2007) and HunPos (Halácsy et al., 2007). The major part of the analysis logic is implemented within the tool, such that they tend not to rely significantly on third-party libraries. However, many third-party resources can be found on the internet for popular standalone tools.

NLP tool suites Some vendors offer tool suites that cover multiple analysis tasks, e.g. ClearNLP, CoreNLP, and OpenNLP. They consist of a set of interoperable tools. Some even go so far as to include a proprietary processing framework and pipeline mechanism. For example, CoreNLP allows the user to implement custom analysis components and to register them with their framework. OpenNLP, on the other hand, provides UIMA-wrappers for their tools. These wrappers are configurable for different UIMA type systems, but unfortunately the configuration mechanism is not powerful enough to accommodate for the design of various major type systems.

We also refer to such tool suites as *single vendor collections*. As for standalone tools, the major part of the analysis logic is a part of the suite and tends not to rely significantly on third-party libraries. Also, again many third-parties offer resources for popular tool suites.

Special purpose collections Special purpose collections combine NLP tools into a comprehensive modular pipeline for a specific purpose.

The Apache cTAKES project (Savova et al., 2010) offers a UIMA-based pipeline for the analysis of medical records which includes components from ClearNLP, OpenNLP, and more for the basic language analysis. These third-party components are used in conjunction with resources created specifically for the domain of medical records. Higher-level tasks use original components from the project, e.g. to identify drugs or relations specific to the medical domain.

Broad-coverage collections Broad-coverage collections cover multiple analysis tasks, but they do not focus on a specific purpose. Instead, they provide the user with a choice for each analysis task by integrating tools from different vendors capable of doing the same task. Because the languages supported by each tool differ, this allows the collection to cover more languages than individual tools or even tool suites alone. Additionally, broad-coverage collections allow comparing different tools against each other.

The U-Compare workbench (Kano et al., 2011) focusses specifically on the ability to compare tools against each other. It offers a GUI for building analysis pipelines and comparing their results. U-Compare also offers a collection of UIMA-based components centered around the U-Compare type system. It started integrating analysis tools primarily from the biomedical domain, but many more tools were integrated as part of the META-NET project (Thompson et al., 2011). This makes the collection accessible through U-Compare one of the largest collections of interoperable NLP components available.

ClearTK (Ogren et al., 2009) is actually a machine-learning framework based on Apache UIMA. However, it also integrates various NLP tools from different vendors and for this reason we list it under the broad-coverage collections. The tools are integrated to provide features for the machine-learning algorithms. The main reason for ClearTK not to use components from other existing UIMA component collections may have been the lack of a comprehensive UIMA component collection for NLP at the time ClearTK was in its early stages.

Note on cross-collection interoperability An alternative to broad-coverage collections that integrate many tools and make them interoperable would be to achieve cross-collection interoperability. That means, many vendors would provide small collections or even individual components and the end users would combine them into pipelines as desired. However, even within a framework like UIMA or GATE, a) some conventions, like a common type system, would need to be respected, b) extensive mapping between the individual components would be required, or c) the components would need to be adaptable to arbitrary type systems through configuration. Until at least one of these points has been resolved in a user-friendly way, we consider broad-coverage collections to be the most convenient solution for the user. The insights gained in building the broad-coverage collections may eventually contribute to finding solutions for these problems.

2.4 Resource selection and acquisition

Many NLP tools are generic, language-independent engines that are parametrized for a particular language with a resource, e.g. a probabilistic model, a set of rules, or another knowledge base. We call this *resource selection*. The selection can happen manually or automatically.

Manual selection is required, for example, in ClearTK or GATE. Components that require a resource offer a parameter pointing to the location from where this resource can be loaded, typically a location on the local file system. This entails that the resource is either bundled with the component or that the user must find and download the resource to the local machine. We call this step *resource acquisition*.

U-Compare (Kano et al., 2011), on the other hand, offers some components preconfigured with resources for certain languages. In particular, components that call out to remote web-services tend to support multiple languages. Based on the language they are invoked for, the service employs a particular resource. However, in this case the users cannot invoke the service with a custom resource from their local machine. Portable components that are bundled with U-Compare also allow for custom resources.

2.5 Need for shareable pipelines based on portable components

Current workflow descriptions are inconvenient to share with others because they are not self-contained. They do not uniquely identify components and resources. The responsibility to obtain, and install components and resources is largely left to the user. Web-based workbenches and marketplaces provide some remedy in this aspect as they remove the need for any local installation by the user. However, such online service-based approaches have been found to be a cause of workflow decay (Gómez-Pérez et al., 2013).

In consequence, we find that a shareable pipeline should rely on portable software and resources that can be automatically obtained from a repository. Once obtained, these remain within the control of the user, e.g. to create backups, or to run them on alternative environments, such as a private compute cluster. In the latter case, the use of remote services would likely cause a performance bottleneck. To make such an approach to shareable pipelines attractive, it must be supported by a broad-coverage collection from which pipelines can be assembled for various tasks.

3 Contributions

3.1 Automatic selection and acquisition of resources

We present a novel approach to the configuration of components with resources based on the data being processed. Resources are stored in a repository from where a component can obtain them on demand. The approach is based on a set of coordinates to address resources: *tool*, *language*, *variant*, and *version*.

In many cases, this removes the need for the user to explicitly configure the resource to be used. By overriding specific coordinates (mainly *variant*), the user can choose between different resources. Additionally, the user can disable resource-resolution via coordinates and instruct the component to use a model at a specific location, e.g. to use custom model from the local file system.

As an example, consider a part-of-speech-tagger component being used to process English text:

- *tool* – this coordinate is uniquely identified by the component being used, e.g. *opennlp-tagger*.
- *language* – this coordinate is obtained from the data being processed by the tagger, e.g. *en* or *de*.
- *variant* – as there can be multiple applicable resources per language, this coordinate is used to choose one of the resources. A default variant is provided by the component, possibly a different variant depending on the language, e.g. *fast* or *accurate*.
- *version* – resources are versioned, just as components are. New versions of a resource are created to fix bad data, to extend the data on which the resource is based, or to make it compatible with a new version of a tool. We note that generally, the versioning of tools and resources is independent of each other: a resource may be compatible with multiple versions of a tool and multiple versions of a resource may be compatible with one specific version of a tool. Furthermore, some vendors do not version resources properly or at all. For example, by comparing hash values, we observed that from version to version only some of the models packaged with CoreNLP change, while others remain identical. We also found the models (and even binaries) of TreeTagger (Schmid, 1994) to change from time to time without any apparent change in version. As a consequence, we decided to consequently use a time-based versioning scheme for resources. The independence between tool and resource versions also has another effect: users find it hard to manually select a resource version compatible with the tool version they use. Thus, we maintain a list of resources and default versions with each component and use it to fill in the *version* coordinate.

To operationalize this concept, we translate these coordinates into Maven coordinates and use these to resolve the resource against the Maven repository infrastructure. For example the coordinates *[tool: opennlp-tagger, language: en, variant: maxent, version: 20120616.1]* would be translated into *[groupId: de.tudarmstadt.ukp.dkpro.core, artifactId: de.tudarmstadt.ukp.dkpro.core.opennlp-model-tagger-en-maxent, version: 20120616.1]*.

Mind that some vendors already distribute resources for their tools via Maven repositories (cf. Section 2.2), but they do so at their own coordinates, e.g. at *[groupId: com.clearnlp, artifactId: clearnlp-general-en-dep, version: 1.2]* and these resources can become of a significant size.[1] To avoid republishing resources at coordinates matching our naming scheme, the artifact at the translated coordinates serves only as a proxy that does not contain the resource itself. Instead, it contains a redirection to the artifact containing the actual resource. This allows us to maintain a common coordinate scheme for all resources while being able to incorporate existing third-party resources. It also allows us to maintain additional metadata, e.g. about tagsets. When vendors do not distribute their resources via Maven, we package them and distribute them via our own public repository – if their license does not prohibit this.

3.2 The DKPro Core broad-coverage component collection or portable components

We present *DKPro Core*, a broad-coverage collection of NLP components based on the UIMA processing framework. Our collection relies only on portable software and resources and it is distributed via the Maven repository infrastructure. It also served as a use-case and test-bed for the development of our resource selection mechanism (Section 3.1). DKPro Core is provided as open-source software.[2]

[1] For example, the ClearNLP dependency parser model for general English (version 1.2) has about 721 MB.

[2] `https://code.google.com/p/dkpro-core-asl/`

Task	Components	Languages
Language identification	2	de, en, es, fr, +65
Tokenization and sentence boundary detection	5	de, en, es, fr, +25
Lemmatization	7	de, en
Stemming	1	de, en, es, fr, +11
Part-of-speech tagging	9	de, en, es, fr, +14
Morphological analysis	2	de, en, fr, it, +1
Named entity recognition	2	de, en, es, nl
Chunking	1	en
Constituency parsing	3	de, en, fr, zh, +1
Dependency parsing	5	de, en, es, fr, +7
Coreference analysis	1	en
Semantic role labelling	1	en
Spell checking and grammar checking	3	de, en, es, fr, +25

Figure 1: Analysis tasks covered by the DKPro Core component collection

The collection targets users with a strong interest in the ability to programmatically assemble pipelines, e.g. as part of dynamic scientific experiments or within NLP-enabled applications. For this reason, our collection employs the Apache uimaFIT library (Apache UIMA Community, 2013) to allow the implementation of pipelines with only a few lines of code (cf. Section 3.3).

Table 1 provides an overview over the analysis tasks currently covered by the collection.[3] Additionally, our collection provides diverse input/output modules that support different file formats ranging from simple text, over various corpus formats (CoNLL, TIGER-XML, BNC-XML, TCF, etc.), to tool-specific formats (IMS Open Corpus Workbench (Evert and Hardie, 2011), TGrep2 (Rohde, 2005), several UIMA-specific formats, etc.). These enable the processing of corpora from many sources and the further processing of results with specialized tools.

We primarily integrate third-party tools with the UIMA framework and include only few original components, mainly for reading and writing the different supported data formats. Our work focusses on the concerns related to interoperability and usability, such as the resource selection mechanism (Section 3.1).

It is our policy to integrate only third-party tools that are properly versioned and that are distributed via the Central Repository, generally including their full source code.[4] As a considerable portion of the tools we integrate do not initially meet this requirement, we regularly reach out to the respective communities and either help them publishing their tools the Central Repository or offer to do so on their behalf. The DKPro Core components themselves are also distributed via the Central Repository.

3.3 Self-contained executable pipeline example

We define a self-contained pipeline description as uniquely identifying all required components and resources. Assuming that the results of the pipeline are fully defined by these and by the input data, such a self-contained pipeline should allow for reproducible results. In particular, the results must not influenced by the platform the pipeline is run on.

We take a step further making self-contained pipelines also convenient for the users by removing the need to manually obtain and install the required components and resources. To do so, we rely on a generic bootstrapping mechanism which is capable of extracting the information about the required artifacts from the pipeline description and of obtaining them automatically from a repository.

We achieve this goal most illustratively through a combination of these ingredients: our auto-configuration mechanism (Section 3.1) which removes the need for explicit configuration and which identifies and fetches the required resources from the repository at runtime; our component collection (Section 3.2) that is published through a Maven repository; Groovy (2014) and its Grape[5] subsystem serving as a bootstrapping mechanism to fetch the components from the repository; uimaFIT providing a concise way of assembling a pipeline of UIMA components in Groovy and making it executable.

Listing 1 demonstrates such a self-contained and executable pipeline. The example consists of three

[3] Unfortunately, we cannot give a full account of the actually integrated third-party tools here, due to the lack of space.

[4] An exception to this rule are tools that need to be integrated as binaries because they are not implemented in Java.

[5] Groovy Adaptable Packaging Engine: `http://groovy.codehaus.org/Grape`

Listing 1: Executable pipeline implemented in Groovy

```
1  @Grab(group='de.tudarmstadt.ukp.dkpro.core',
2        module='de.tudarmstadt.ukp.dkpro.core.textcat-asl', version='1.6.1')
3  @Grab(group='de.tudarmstadt.ukp.dkpro.core',
4        module='de.tudarmstadt.ukp.dkpro.core.stanfordnlp-gpl', version='1.6.1')
5  @Grab(group='de.tudarmstadt.ukp.dkpro.core',
6        module='de.tudarmstadt.ukp.dkpro.core.maltparser-asl', version='1.6.1')
7  @Grab(group='de.tudarmstadt.ukp.dkpro.core',
8        module='de.tudarmstadt.ukp.dkpro.core.io.text-asl', version='1.6.1')
9  @Grab(group='de.tudarmstadt.ukp.dkpro.core',
10       module='de.tudarmstadt.ukp.dkpro.core.io.conll-asl', version='1.6.1')
11
12 import de.tudarmstadt.ukp.dkpro.core.textcat.*;
13 import de.tudarmstadt.ukp.dkpro.core.stanfordnlp.*;
14 import de.tudarmstadt.ukp.dkpro.core.maltparser.*;
15 import de.tudarmstadt.ukp.dkpro.core.io.text.*;
16 import de.tudarmstadt.ukp.dkpro.core.io.conll.*;
17 import static org.apache.uima.fit.factory.AnalysisEngineFactory.*;
18 import static org.apache.uima.fit.factory.CollectionReaderFactory.*;
19 import static org.apache.uima.fit.pipeline.SimplePipeline.*;
20
21 runPipeline(
22   createReaderDescription(TextReader,
23     TextReader.PARAM_SOURCE_LOCATION, args[0]),
24   createEngineDescription(LanguageIdentifier),
25   createEngineDescription(StanfordSegmenter),
26   createEngineDescription(StanfordPosTagger),
27   createEngineDescription(MaltParser),
28   createEngineDescription(Conll2006Writer,
29     Conll2006Writer.PARAM_TARGET_LOCATION, args[1]));
```

sections. Lines 1-10 identify the components used in the pipeline by name and version. Lines 12-19 are necessary boilerplate code making the components accessible within the Groovy script. Lines 21-29 employ uimaFIT to assemble and run a pipeline consisting of components from our collection.

When the Groovy script representing the pipeline is executed, it downloads all required artifacts. Afterwards, these artifacts remain on the user's system and they can be used again for a subsequent execution of the script. The user may also create backups of these artifacts or transfer them to a different system. Thus, in contrast to pipelines that rely on online services, our approach allows the user to maintain control over the involved software and resources.

The example pipeline given in Listing 1 can indeed be run on any computer that has Groovy installed. It is a life example of a self-contained NLP pipeline shared as part of a scientific publication. By means of this example, we demonstrate that we have reached our goal of providing a concept for shareable pipelines based on portable components and resources.

Due to its conciseness, we consider the Groovy script to provide the most illustrative example of the benefits provided by our contributions. However, there are alternative ways to operationalize our concepts. Alternatively we could use a Jython (2014) script and jip[6] to resolve the Maven dependencies, Java and Maven, or a variety of other JVM-based languages and build tools supporting Maven repositories.

4 Applications

The DKPro Core collection has already been successfully used for linguistic pre-processing in various tasks, including, but not limited to, temporal tagging (Strötgen and Gertz, 2010), text segmentation based on topic models (Riedl and Biemann, 2012), and textual entailment (Noh and Padó, 2013).

The portable components and resources from our collection can be integrated into online workbenches and can be run on cloud platforms by users that find this convenient. Combined with our concept for executable pipelines, users can be enabled to export self-contained pipelines from such workbenches and to archive them for later reproduction. Additionally, users can und run the pipelines on private hardware, possibly on sensitive data which users do not feel comfortable submitting to the cloud. We believe that service-based offerings should be based as much as possible on portable software, and we focussed in this paper on improving the availability and convenience of using such portable NLP software. Thus, we consider our approach not to be competing with service-based approaches but rather as complementing them.

[6] https://pypi.python.org/pypi/jip

Our concept of automatically selecting and acquiring resources can be immediately transferred to other component collections. Although our component collection is based on UIMA, this aspect has been implemented independent of the processing framework. Having experienced the convenience offered by this concept, we believe that integrating a pluggable resource resolving mechanism directly into processing frameworks such as GATE or UIMA would be beneficial. A pluggable mechanism would be important because we expect that the underlying repository infrastructures and coordinate systems are likely to evolve over time. For example, we could envisage an integrated resolving mechanism that allows combining the rich metadata offered by directories such as META-SHARE or the Virtual Language Observatory with the ability to automatically acquire software and resources offered by Maven or with the ability of invoking NLP tools as services such as via AnnoMarket.

Our concept of rendering self-contained pipelines as executable scripts facilitates the sharing of pipelines. This can be either only the script which then downloads its dependencies upon execution, or the dependencies can be resolved beforehand and included with the script. The concise pipeline description is also useful for examples in teaching, in documentation, or on community platforms like Stack Overflow.[7] We offer Groovy- and Jython-based quick-start examples for the DKPro Core collection to new users.

5 Summary and future work

In this paper, we have presented a novel concept for implementing shareable NLP pipelines supported by a broad-coverage collection of interoperable NLP components. Our approach is enabled by the combination of distributing portable NLP components and resources through a repository infrastructure and by an auto-configuration mechanism allowing components to select suitable resources at runtime and to obtain them automatically from the repository.

We have demonstrated that our contributions enable a concise and self-contained pipeline description, which can easily be shared, e.g. as examples in teaching, documentation, or publications. The reliance on portable artifacts allow the user to maintain control, e.g. by creating backups of the involved artifacts to reproduce results at a later time, even if the original repository may no longer be available.

In the future, we plan to investigate a mechanism to automatically detect misalignments between resources and components within a pipeline to provide the user with an indication when suboptimal results may occur and what may cause them. This is necessary because components in the collection are interoperable at the level of annotation types, whereas tagsets and tokenization are simply passed through. While this is a common approach, it leads to the situation that the results may be negatively affected due to diverging tokenizations used while generate the resources for the components. Also, the automatic resource selection mechanism may currently choose resources with incompatible tagsets, e.g. a POS-tagger model producing tagset X while a subsequent dependency parser would require tagset Y.

We also plan to extend the resource selection process to support additional metadata. Eventually, the *variant* coordinate should be replaced by a more fine-grained mechanism to select resources based e.g. on the domain, tagset, or other characteristics.

Acknowledgements

The project was partially funded by means of the German Federal Ministry of Education and Research (BMBF) under the promotional reference 01UG1110D, and partially by the Volkswagen Foundation as part of the Lichtenberg-Professorship Program under grant No. I/82806. The authors take the responsibility for the contents.

References

Apache UIMA Community. 2013. Apache uimaFIT guide and reference, version 2.0.0. Technical report, Apache UIMA.

[7] `http://stackoverflow.com` (Last accesses: 2014-02-14)

Ekaterina Buyko and Udo Hahn. 2008. Fully embedded type systems for the semantic annotation layer. In *ICGL 2008 - Proceedings of First International Conference on Global Interoperability for Language Resources*, pages 26–33, Hong Kong.

Central Repository. 2014. The Central Repository. URL `http://search.maven.org` (Last accessed: 2014-03-19), March. Sonatype Inc. (`http://www.sonatype.org/central`).

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 tagsets. *Traitement Automatique des Langues*, 49(2):271–293.

ClearNLP. 2014. Version 2.0.2 - fast and robust NLP components implemented in Java. URL `http://opennlp.apache.org` (Last accessed: 2014-03-19), January.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: an architecture for development of robust HLT applications. In *Proceedings of 40th Annual Meeting of the Association for Computational Linguistics*, pages 168–175, Philadelphia, Pennsylvania, USA, July. Association for Computational Linguistics.

Stefan Evert and Andrew Hardie. 2011. Twenty-first century corpus workbench: Updating a query architecture for the new millennium. In *Proceedings of the Corpus Linguistics 2011 conference*, Birmingham, UK, July. University of Birmingham.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348.

Antske Fokkens, Marieke van Erp, Marten Postma, Ted Pedersen, Piek Vossen, and Nuno Freire. 2013. Offspring from reproduction problems: What replication failure teaches us. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1691–1701, Sofia, Bulgaria, August. Association for Computational Linguistics.

Carole A Goble, Jiten Bhagat, Sergejs Aleksejevs, Don Cruickshank, Danius Michaelides, David Newman, Mark Borkum, Sean Bechhofer, Marco Roos, Peter Li, et al. 2010. myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic acids research*, 38(suppl 2):W677–W682.

José Manuel Gómez-Pérez, Esteban Garcıa-Cuesta, Jun Zhao, Aleix Garrido, José Enrique Ruiz, and Graham Klyne. 2013. How reliable is your workflow: Monitoring decay in scholarly publications. In *Proceedings of the 3rd Workshop on Semantic Publishing (SePublica 2013) at 10th Extended Semantic Web Conference*, page 75, Montpellier, France, May.

Groovy. 2014. Version 2.2.2 - A dynamic language for the Java platform. URL `http://groovy.codehaus.org` (Last accessed: 2014-03-19, February.

Péter Halácsy, András Kornai, and Csaba Oravecz. 2007. Hunpos – an open source trigram tagger. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions*, pages 209–212, Prague, Czech Republic, June. Association for Computational Linguistics.

Marie Hinrichs, Thomas Zastrow, and Erhard Hinrichs. 2010. WebLicht: Web-based LRT Services in a Distributed eScience Infrastructure. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 489–493, Valletta, Malta, May. European Language Resources Association (ELRA).

Jython. 2014. Jython: Python for the Java Platform. URL `http://www.jython.org` (Last accessed: 2014-03-19.

Yoshinobu Kano, Makoto Miwa, Kevin Bretonnel Cohen, Lawrence E. Hunter, Sophia Ananiadou, and Jun'ichi Tsujii. 2011. U-Compare: A modular NLP workflow construction and evaluation system. *IBM Journal of Research and Development*, 55(3):11:1–11:10, May.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland, June. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

Tae-Gil Noh and Sebastian Padó. 2013. Using UIMA to structure an open platform for textual entailment. In Peter Klügl, Richard Eckart de Castilho, and Katrin Tomanek, editors, *Proceedings of the 3rd Workshop on Unstructured Information Management Architecture (UIMA@GSCL 2013)*, pages 26–33, Darmstadt, Germany, Sep. CEUR-WS.org.

Philip V. Ogren, Philipp G. Wetzler, and Steven J. Bethard. 2009. ClearTK: a framework for statistical natural language processing. In Christian Chiarcos, Richard Eckart de Castilho, and Manfred Stede, editors, *Proceedings of the Biennial GSCL Conference 2009, 2nd UIMA@GSCL Workshop*, pages 241–248, Potsdam, Germany, September. Gunter Narr Verlag.

Rafal Rak, Andrew Rowley, Jacob Carter, and Sophia Ananiadou. 2013. Development and analysis of nlp pipelines in argo. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 115–120, Sofia, Bulgaria, August. Association for Computational Linguistics.

Martin Riedl and Chris Biemann. 2012. Text segmentation with topic models. *JLCL*, 27(1):47–69.

Douglas LT Rohde. 2005. Tgrep2 user manual version 1.15. *Massachusetts Institute of Technology. http://tedlab.mit.edu/dr/Tgrep2*.

Guergana K. Savova, James J. Masanz, Philip V. Ogren, Jiaping Zheng, Sunghwan Sohn, Karin C. Kipper-Schuler, and Christopher G. Chute. 2010. Mayo clinical text analysis and knowledge extraction system (cTAKES): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association*, 17(5):507–513.

Helmut Schmid. 1994. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing*, pages 44–49, Manchester, UK.

Sonatype Company. 2008. *Maven: The Definitive Guide*. O'Reilly Media, September. ISBN: 9780596517335.

Jannik Strötgen and Michael Gertz. 2010. HeidelTime: High Quality Rule-Based Extraction and Normalization of Temporal Expressions. In *Proceedings of the 5th International Workshop on Semantic Evaluation*, pages 321–324, Uppsala, Sweden, July. Association for Computational Linguistics.

Valentin Tablan, Kalina Bontcheva, Ian Roberts, Hamish Cunningham, and Marin Dimitrov. 2013. Annomarket: An open cloud platform for nlp. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 19–24, Sofia, Bulgaria, August. Association for Computational Linguistics.

Paul Thompson, Yoshinobu Kano, John McNaught, Steve Pettifer, Teresa Attwood, John Keane, and Sophia Ananiadou. 2011. Promoting interoperability of resources in meta-share. In *Proceedings of the Workshop on Language Resources, Technology and Services in the Sharing Paradigm*, pages 50–58, Chiang Mai, Thailand, November. Asian Federation of Natural Language Processing.

Dieter Van Uytvanck, Claus Zinn, Daan Broeder, Peter Wittenburg, and Mariano Gardellini. 2010. Virtual Language Observatory: The portal to the language resources and technology universe. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, pages 900–903, Valletta, Malta, may. European Language Resources Association (ELRA).

Stephen Wu, Vinod Kaggal, Dmitriy Dligach, James Masanz, Pei Chen, Lee Becker, Wendy Chapman, Guergana Savova, Hongfang Liu, and Christopher Chute. 2013. A common type system for clinical natural language processing. *Journal of Biomedical Semantics*, 4(1):1.