

The Tanl Pipeline

Giuseppe Attardi, Stefano Dei Rossi, Maria Simi

Dipartimento di Informatica, Università di Pisa

Largo B. Pontecorvo 3, I-56127 Pisa, Italy

E-mail: attardi@di.unipi.it, deirossi@di.unipi.it, simi@di.unipi.it

Abstract

Tanl (Natural Language Text Analytics) is a suite of tools for text analytics based on the software architecture paradigm of data pipelines. Tanl pipelines are data driven, i.e. each stage pulls data from the preceding stage and transforms them for use by the next stage. Since data is processed as soon as it becomes available, processing delay is minimized improving data throughput. The processing modules can be written in C++ or in Python and can be combined using few lines of Python scripts to produce full NLP applications. Tanl provides a set of modules, ranging from tokenization to POS tagging, from parsing to NE recognition. A Tanl pipeline can be processed in parallel on a cluster of computers by means of a modified version of Hadoop streaming. We present the architecture, its modules and some sample applications.

Introduction

Text analytics involves many tasks ranging from simple text collection, extraction, and preparation to linguistic syntactic and semantic analysis, cross reference analysis, intent mining and finally indexing and search. A complete system must be able to process textual data of any size and structure, to extract words, to classify documents into categories (taxonomies or ontologies), and to identify semantic relationships.

A full analytics application requires coordinating and combining several tools designed to handle specific subtasks. This may be challenging since many of the existing tools have been developed independently with different requirements and assumptions on how to process the data.

Several suites for NLP (Natural Language Processing) are available for performing syntactic and semantic data analysis, some as open source and other as commercial products. These toolsets can be grouped into two broad software architecture categories:

- *Integrated Toolkits*: these provide a set of classes and methods for each task, and are typically bound to a programming language. Applications are programmed using compilers and standard programming environments. Examples in this category are: LingPipe (LingPipe), OpenNlp (OpenNLP), NLTK (NLTK).
- *Component Frameworks*: these use generic data structures, described in a language independent formalism, and each tool consumes/produces such data; a special compiler transforms the data descriptions into types for the target programming language. Applications are built using specific framework tools. Examples in this category are: GATE (GATE), UIMA (UIMA).

Both GATE and UIMA are based on a *workflow software*

architecture, where the framework handles the workflow among the processing stages of the application, by means of a controller that passes data among the components invoking their methods. Each tool accepts and returns the same type of data and extends the data it receives by adding its own information, as shown using different colors in Figure 1: the Tokenizer adds annotations to represent the start and end of each token, the PosTagger adds annotations representing the POS for each token. Since the controller handles the whole processing in a single flow, each processing component receives the whole collection and returns the whole collection. If the collection is big, this might require large amounts of memory.

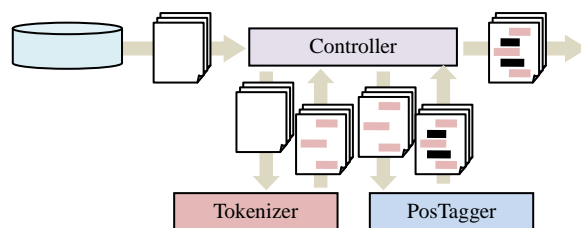


Figure 1: Workflow Software Architecture.

In this paper we present an alternative architecture based on the notion of *data pipeline*. The Tanl pipeline (Natural Language Text Analytics) uses both generic and specific data structures, and components communicate directly exchanging data through pipes, as shown in Figure 2. Since each tool pulls the data it needs from the previous stage of the pipeline, only the minimum amount of data passes through the pipeline, therefore reducing the memory footprint and improving the throughput. The figure shows single documents being passed along, but the granularity can be even smaller: for instance a module might just require single tokens or single sentences. This would be hard to handle with a workflow architecture, since the controller does not know which amount of data

each tool requires.

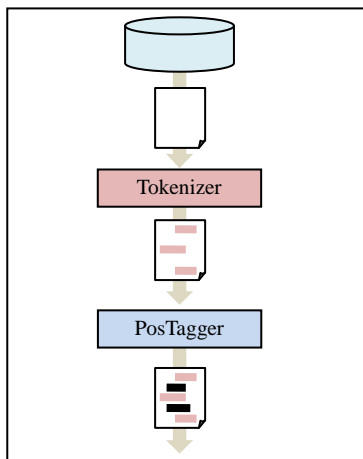


Figure 2: Tanl data pipeline.

Related work

We present an overview of some representative NLP toolsets and highlight the differences with the approach adopted for the Tanl pipeline.

Integrated Toolkits

NLTK (Natural Language Toolkit) is a suite of libraries and programs written in Python for symbolic and statistical natural language processing (Steven et al., 2009). For each task NLTK provides a specific API, implemented by several alternative modules. For example there are several chunker modules providing the `ChunkParserI` interface, classifier modules providing the `ClassifierI` interface, etc. Each interface specifies different data types, for instance the `ChunkParserI` interface operates on tokens represented as tuples (word, tag), the `ParserI` interface accepts a string and returns a `Tree`. Since many modules were developed independently, sometimes they provide their own API that extends the generic one. For instance one implementation of a dependency parser requires as input two lists, a list of tokens and a list of tags, another implementation operates on files, hence it creates an intermediate temporary file.

Workflow Frameworks

GATE (General Architecture for Text Engineering) is a Java framework organized according to three concepts: language resources, processing resources and the controller. A GATE application handles the following types of data:

- Features: a set of name/values pairs;
- Annotation: consists of a tuple (start, end, type, features), the start and end character positions in the text, the type of the annotation and the features associated to the annotation;
- Document: consists of a triple (content, annotations, features), where the content is the text of the document, annotations are the annotations in the document and features are those associated to the document.

- Corpus: a list of Documents.

In the following example, two processing resources are created (a `Tokenizer` and a `PosTagger`), a language resource is opened (a `Corpus`) and a controller is created of type `SerialAnalyserController`. The language and processing resources are supplied to the controller which supervises and coordinates the overall workflow: at each analysis step it passes data to a processing resource, gets back the enriched results and passes them along to the next step.

```
SerialAnalyserController sac =
    Factory.createResource(
        "SerialAnalyserController", ...);
FeatureMap params = Factory.newFeatureMap();
sac.add(Factory.createResource("Tokenizer",
    params));
sac.add(Factory.createResource("PosTagger",
    params));
Corpus corpus = ...;
sac.setCorpus(corpus);
sac.execute();
```

UIMA (Unstructured Information Management Architecture) is a general framework for the analysis of text and other media. The fundamental UIMA data model is called Common Analysis Structure (CAS): it provides data modeling, definition and retrieval facilities for the annotations stored in it. Annotations are defined in a hierarchically organized type system rooted in a basic type that contains the start and end position in the document as well as a set of features. Processing is performed by Analysis Engines (AE) according to a simple I/O logical interface model: each AE gets a CAS as input and produces a CAS as output. Typically each AE analyzes a CAS containing a document and adds more metadata to the CAS structure.

Each UIMA component, written in Java or C++, implements interfaces defined by the framework and provides self-describing metadata via XML descriptor files. An application can be created by joining together various components as shown in the following example.

```
AnalysisEngine tokAnnotator =
    produceAnalysisEngine(...);
AnalysisEngine posAnnotator =
    produceAnalysisEngine(...);

ArrayList<AnalysisEngineMetaData> mdl =
    new ArrayList<...>();
mdl.add(tokAnnot.getAnalysisEngineMetaData());
mdl.add(posAnnot.getAnalysisEngineMetaData());

CAS aCAS = createCAS(mdl);
aCAS.setDocumentText(getTextFromFile(...));

tokenAnnotator.process(aCAS);
posAnnotator.process(aCAS);
```

Two AEs are created, a tokenizer and a POS tagger. Then a CAS is created that contains both metadata from the tokenizer and the POS tagger. The CAS is given to both the AEs in sequence and each adds its own annotations. The framework manages the AEs and the data flow between them.

A CAS Consumer processes the CAS produced by an AE.

For example one can collect all annotations of type Entity with the following code:

```
ArrayList<String> entities = new ...;
JFSIndexRepository idx =
    CAS_.getJFSIndexRepository();
Iterator<Entity> it =
    idx.getAnnotationIndex(Entity.type).iterator()
;

while (it.hasNext()) {
    Entity en = it.next();
    entities.add(en.getCoveredText());
}
```

UIMA additionally provides capabilities to wrap components as network services.

The JULIES NLP Toolsuite consists in a collection of UIMA components (JULIES NLP).

Tanl design

The Tanl architecture is based on a data pipeline paradigm (Figure 2) and allows integrating modules written in different languages.

The approach has the advantage that each component is directly connected to the other ones through pipes, so it is not necessary to wait until the end of one processing phase before starting the next one. As a tool produces the first result it is immediately passed to the next one through the pipeline without producing intermediate data structures.

Another advantage of this approach is that it is possible to compose a Tanl pipeline using a general scripting language, for example Python or Perl, instead of introducing ad-hoc tools.

Most of the Tanl tools exploit quantitative and statistical machine learning methods and they require an annotated training corpus for creating statistical models of the data.

Software architecture

Pipeline

The pipeline components can be distinguished into three basic types:

- *Source*: creates an initial pipe (e.g. a document reader, reading from a text file and creating a stream of tokens to be sent through the pipeline);
- *Transform*: receives data from one pipe and produces output on another pipe;
- *Sink*: consumes the output of a pipe.

For example a source can be created as an instance of `SentenceSplitter` and connected in pipe to an input stream:

```
ss = SentenceSplitter('ita.punkt').pipe(stdin)
```

The pipe can then be connected to other tools performing various tasks such as tokenization, POS tagging and parsing as follows:

```
wt = Tokenizer().pipe(ss)
pt = PosTagger('italian.pos').pipe(wt)
pa = Parser.create('italian.MLP').pipe(pt)
```

Each line in the above example represents a transformation stage in the pipeline. No processing of

data happens while the pipeline is being built.

Processing in the assembled pipeline only starts when a sink is connected to the pipeline and starts drawing items from it, in a fully data-driven process. Each stage in the pipeline, when requested for the next item, requests items from its preceding stage in order to produce the next output.

A sink can just be defined through a standard Python iterator pulling data from the last stage of the pipeline:

```
ret = ""
for s in pa:
    ret += string(s) + "\n"
return ret
```

Using a general purpose scripting language for composing the pipeline avoids the need for compilers and other development tools. Special processing can be added at any stage of the pipeline for whatever need with a few lines of code and exploiting the facilities of Python.

For example, if one needs to monitor what is happening at a certain stage in the pipeline, a tee component can be added for analyzing the items passing through that stage. Here is an example of a tee used for printing all items after the POS tagging stage:

```
pt = PosTagger('italian.pos').pipe(wt)
tee = Tee(printSink, pt)
pa = Parser.create('italian.MLP').pipe(tee)
```

The first argument to the Tee constructor is a sink through which items have to be pushed. However, since sinks behave in a pull mode, the Tee has to create an inversion of control, turning a sink into a pipe, by exploiting the functional mechanisms of Python.

```
class Tee:
    def __init__(self, func, arg):
        self.func = func
        self.arg = arg

    def next(self):
        aux = self.arg.next()
        func(aux)
        return aux
```

The next method of the Tee applies the function to the item (in this case the printing function) before passing the item itself to the next module in the pipeline.

An alternative solution would be to run sinks within separate threads and adopting an asynchronous streaming model, where each consumer processes data at its own pace. This however involves providing buffering in the components, partly defeating the purpose of a data driven pipeline, where data is produced only when requested and processed immediately.

A disadvantage of an integrated multi-language pipeline is dealing with debugging: since at the scripting language level the pipeline components are only visible as black boxes, it is hard to step into their execution for debugging code. An instrumented version of the Python runtime is required in order to start a process in debugging mode.

Metadata handling

As data passes through a pipeline, global or specific metadata might need to be collected. For example, Wikipedia articles contain metadata information such as

hypertext links, internal references as well as internal document structure such as titles and sections. These data are useful for certain modules of the pipeline, but unnecessary or unmanageable for others.

Workflow systems like GATE or UIMA store these as annotations in a global structure like the CAS. Since in Tanl items are passed along the pipe, there is no place to store global data.

Our solution consists in storing metadata in the tokens, using a field called context. A context contains a set of key/value pairs representing metadata. Contexts can be nested, referring to a parent context, for representing nested structures such as sections within documents or XML trees. Tokens that belong to the same context share the same context object, so that memory overhead is reduced, even using a distributed representation rather than a global structure.

Implementation

Enumerators and Tokens

Each module of the Tanl pipeline consumes a stream produced by a previous module and produces a stream. Streams consist of tokens or combinations of tokens, e.g. sentences which are sequence of tokens.

Tokens are the basic data structure type that all components manipulate. The token data structure was designed to be extensible, so that each tool can add to it its own annotations, which are passed along to later stages.

A token contains a string that represents its form and an arbitrary number of attributes and links. Attributes are simple key/value pairs, while links are labeled arcs referring to other tokens through their id:

```
struct Token {
    string form;    ///< word form
    Attributes attributes;
    Links links;
    Context* context; ///< context
};

struct Link {
    int target;    ///< the ID of the target
    string label;  ///< the label for the link
};
```

In the implementation of attributes though, we avoid the naive solution of using a hash table, since this would entail a significant cost for each token: instead the token only contains the attribute values and an index of attribute names is used to retrieve an attribute by name. The index is shared among all the tokens in a Corpus.

A Corpus represents the common aspects of a collection of documents, including the tongue, the list of token attributes, the attribute name index, the file format and it also provides methods for writing and reading sentences from corpus documents.

A stream is defined through a generic class Enumerator that provides methods to advance to the next item and to access it:

```
template <class T>
class Enumerator {
```

```
public:
    virtual bool MoveNext() = 0;

    virtual T Current() = 0;

    virtual void Reset() {}
};
```

Listing 1: Generic Enumerator interface

Each module provides an interface for connecting to a pipeline:

```
template <class Tin, Tout>
struct IPipe {
    Enumerator<Tout>* pipe(Enumerator<Tin>&);
};
```

Listing 2: Pipeline interface

Language Integration

C++ modules can be invoked from scripting languages by means of wrappers created with SWIG (SWIG), an automated tool for generating wrappers directly from code. In particular Tanl provides predefined wrappers for Python.

C++ Enumerators as Python iterators

SWIG allows exposing C++ objects and methods to Python, but an even tighter integration is provided that allows operating on C++ objects in a more convenient way. In particular the standard Python iterator constructs, for instance for `x in pipe: ...x...`, can be used to process pipeline streams. Since the Python iterator protocol consists of a single method `next()` and termination is obtained by raising an exception, a magic trick is required in the SWIG code in order to conform to this protocol:

```
%exception Tanl::Enumerator<Item>::next() {
    $action
    if (!result) {
        PyErr_SetObject(PyExc_StopIteration,
                        Py_None);
        return NULL;
    }
};
```

This SWIG notation is used to add a few lines to the wrapper for method `next()` that will raise the required exception.

Memory management

Since objects are passed between C++ and Python, stored within wrappers, memory must be managed properly so that objects are released when no longer in use. This is normally handled by telling to SWIG which objects must remain under control by Python. Python uses reference counting, and when an object is no longer accessible, it automatically calls its C++ destructor.

However there are cases where this mechanism is not sufficient, for example when a pipe is created like this:

```
pp = Parser.create("model").pipe(sr)
```

both a parser object and a parser proxy that wraps it are created. Then a pipe is created which refers to the parser

object and assigned to the variable `pp`. The parser should survive as long as `pp` exists. However Python destroys the parser proxy, since it has no references to it, and calls the parser C++ destructor. In order to avoid this, a reference count is added to the C++ parser object, reflecting the number of Python objects referring to it. The C++ destructor is only invoked when this count goes to 0. In order to maintain this counter, it must be incremented when the pipe is created from Python. This can be done in SWIG with the following rule:

```
%exception Parser::pipe {
    $action
    arg1->incRef();    // arg1 is the parser object
}
```

When the count of the pipe proxy reaches zero, Python calls the pipe destructor.

Similarly, the parser counter must be decremented when the pipe proxy that embeds the parser gets destroyed. This is done with:

```
%feature("unRef")
Tanl::Enumerator<Tanl::Sentence*>
"$this->Dispose();"
```

which will call the following method on the pipe:

```
void ParserPipe::Dispose() { parser.decRef();
                             delete this; }
```

that will decrement/release the parser before deleting the pipe. Finally, in order to keep in synch the reference count of the parser proxy, it must be updated whenever Python creates a reference to it, by using these SWIG rules:

```
%feature("ref") Parse "$this->incRef();"
%feature("unref") Parse "$this->decRef();"
```

A reference count mechanism is also required to manage Context objects used in tokens, since their lifetime duration is independent from that of the token where they appear.

Map Reduce

A Tanl pipeline can be processed in parallel using the Map/Reduce pattern, for instance using Apache Hadoop (Hadoop). The data to be processed is partitioned into subsets, each of which is assigned for processing to a node in the cluster.

The mapper and reducer functions are normally written in Java, but the framework also provides a facility called Hadoop streaming that allows running any executable as a mapper or reducer.

Unfortunately the standard implementation of Hadoop streaming does not ensure that the outputs of each mapper are combined by the reducer preserving the original order. To overcome this problem we modified the implementation (Tanl Hadoop Streaming) by adding a sequence number to each document passed to the mapper and introducing a reducer that uses these numbers for recombining the documents in the original order.

Pipeline Modules

The following modules are currently available as part of the Tanl pipeline:

- **Sentence Splitter:** splits the text into sentences, producing an enumerator of strings, each representing a sentence. The module is written in Python and is based on the Punkt Tokenizer from the NLTK suite, which implements the technique by Kiss and Strunk (Kiss & Strunk, 2006).
- **Word Tokenizer:** deals with the segmentation of a sentence into tokens, producing a stream of vector of tokens. The module consists of C++ code produced using Quex (Quex), a generator of lexical analyzers, capable of handling Unicode characters.
- **Word Aggregator:** combines polyrematic expressions of common use into a single token (e.g. "a meno che" becomes "a_meno_che").
- **POS Tagger:** enriches the structure Token representing a token within a sentence with attributes representing the PoS and lemma. Two alternative taggers are available: one based on TreeTagger (Schmid, 1994) and one based on Hunpos (Halácsy et al., 2007), an open source reimplementation of TnT (Brants, 2000).
- **Morph Splitter:** splits the POS of each token into separate POS and morphology attributes and also splits clitic forms into two or more tokens (e.g. the verb "avercelo" becomes "aver- ce- lo").
- **Parser:** parses sentences producing dependency parse trees. The module takes as input a stream of vectors of tokens, and produces a stream of sentences. It uses DeSR, a state-of-the-art multilingual dependency parser based on the Shift/Reduce paradigm (Attardi, 2006; Attardi et al., 2007; Attardi et al., 2009).

A few semantic analysis modules are also available; as the previous modules, they consume and produce a stream of vectors of tokens, adding specific semantic attributes to the structure Token:

- **Named Entity Tagger:** identifies and classifies atomic elements such as person names, organizations, locations, temporal expressions, quantities, percentages etc.
- **SuperSense Tagger:** assigns a semantic category to nouns, adjectives and verbs, corresponding to the WordNet lexicographer class labels (Fellbaum, 1998).

Both tools use a Maximum Entropy classifier provided in the Tanl library.

The Tanl Indexer produces a special full-text search index enriched with syntactic and semantic information. The index is organized in multiple layers, so that at each document position a stack of values is present. Each layer represents a different token attribute, e.g. form, lemma, POS, NE, SuperSense and dependency relations. The index also maintains information on sentence boundaries so that the search can return sentences matching queries rather than documents. An additional inverted index is also present that allows searching for pairs of word in a given syntactic relation. A special query language allows expressing queries involving not just words, but any attributes of tokens and in particular dependency paths in the parse trees. Typical boolean, proximity and phrase

operators allow forming even more complex queries.

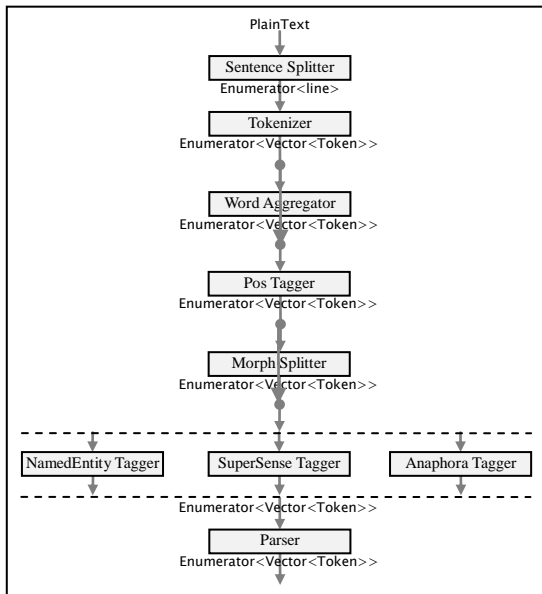


Figure 3: Sample Tanl pipeline.

Figure 3 shows an example of a full Tanl pipeline built with some of the available modules.

Applications

As a case study for the Tanl suite we annotated two significant subset of Wikipedia: the English Wikipedia, consisting in over 3 million articles for a total of 29.320.747 sentences and the Italian Wikipedia, consisting of over 660.000 articles for a total of 5.507.225 sentences. The Wikipedia is challenging both in terms of size and in terms of the variety of material and topics covered. DeepSearch and Yahoo! Correlator are two applications that use the annotated Wikipedia.

DeepSearch

DeepSearch (DeepSearch) is a Wikipedia search engine that exploits syntactic and semantic annotations extracted from Wikipedia articles. The extended query language allows expressing queries that involve various attributes in the annotation.

For example “Who killed Caesar?”, can be answered by sentences where Caesar is the object of the verb ‘to kill’: this can be expressed in our special query language as a query for the word ‘Cesar’ occurring as the dependent of a dependency labeled as ‘OBJ’ and whose head is a word with lemma ‘kill’.

Similarly “What Edison did not invent?” can be answered retrieving sentences where ‘Edison’ is the subject of a verb of category ‘Creation’ (one of the Super Senses), with a negation as a modifier of the verb.

Yahoo! Correlator

Yahoo! Correlator (Yahoo! Correlator) is a search engine and content aggregator that extracts and organizes information from text, collects and displays related names, concepts, places, and events correlated to user queries. The online demo is based on an annotated version of the English Wikipedia processed with earlier versions of the Tanl pipeline tools.

The main result page shows a synthetic page assembled from several Wikipedia entries matching the search, grouped using the Wikipedia category structure. Additional pages display names of people, places on a map, concepts or events in a timeline related to those found in answers to the query.

Dependency Parser

A dependency parser can be built with a few lines of scripting similar to those presented in Section 0. This can be turned into a Web service for processing multiple requests by creating the transforms just once:

```
ss = SentenceSplitter('italian.punkt')
tk = Tokenizer()
pt = PosTagger('italian.ttagger')
ms = MorphSplitter()
pa = Parser.create('italian.MLP')
```

A pipe is created connecting these modules each time a request is received to parse a string text:

```
p1 = [text]
p2 = ss.pipe(p1)
p3 = tk.pipe(p2)
p4 = pt.pipe(p3)
p5 = ms.pipe(p4)
p6 = pa.pipe(p5)
ret = ""
for s in p6:
    ret += c.toString(s) + "\n"
return ret
```

A Web service actually running this code is available at <http://paleo.di.unipi.it/parse> (Tanl Parser). The parser used is the DeSR dependency parser, which uses a MultiLayer Perceptron model and produces parse trees annotated using the Tanl Dependency Notation (Tanl Dependency Notation). The output parse trees are displayed graphically in HTML or can be obtained in the CoNLL X format (CoNLL X Format).

TornadoWeb (TornadoWeb) is used as an application server for Python.

Performance

The parse service described above is capable of parsing sentences of a dozen tokens in 10-20 milliseconds.

A batch pipeline from pure text to parse trees can process typically four Wikipedia articles per second. As a consequence, by parallelizing the process on a dozen of nodes, the whole Italian Wikipedia can be processed in about 4 hours.

Conclusions

We presented the software architecture underlying the Tanl suite. The benefits of the pipeline can be summarized as follows:

- Data pipeline: modules share a common data model based on a flexible and extensible representation of tokens which are passed along the pipe;
- Processing on demand: processing is data-driven and each stage pulls data as needed from the previous stages;
- Data granularity: the blocks of data traversing the pipeline are smaller than in the other toolsets. This reduces memory requirements and improves latency.

- Efficiency: core algorithms are written in C++;
- Flexibility: Python wrappers allow configuring pipelines using simple scripts and activating or monitoring the pipelines by inserting intermediate stages;
- Parallelism: collections can be partitioned and several pipes can be run in parallel on a cluster using a modified version of Hadoop Streaming (Tanl Hadoop).

Acknowledgments

Francesco Tamberi and Antonio Fuschetto participated in the development of the Tanl pipeline; Felice Dell'Orletta contributed to the DeSR parser.

Partial support was provided by Yahoo! Research and by Fondazione Cassa di Risparmio di Pisa.

References

- G. Attardi. Experiments with a Multilanguage non-projective dependency parser. In *Proc. of the Tenth CoNLL*. 2006.
- G. Attardi, A. Chanev, M. Ciaramita, F. Dell'Orletta and M. Simi. Multilingual Dependency Parsing and Domain Adaptation using DeSR. *Proc. the CoNLL Shared Task Session of EMNLP-CoNLL 2007*. Prague, CZ. 2007.
- G. Attardi, F. Dell'Orletta, M. Simi, J. Turian. Accurate Dependency Parsing with a Stacked Multilayer Perceptron. *Proc. of Workshop Evalita 2009*. 2009.
- T. Brants. TnT—A Statistical Part-of-Speech Tagger, *Proc. of ANLP-NAACL Conf.* 2000.
- C. Fellbaum, WordNet An Electronic Lexical Database. MIT Press, 1998.
- P. Halácsy, A. Kornai, C. Oravecz. HunPos – an open source trigram tagger, In *Proc. of the Demo and Poster Sessions of the 45th Annual Meeting of the ACL*, Prague, Czech Republic 209–212 (2007)
- T. Kiss and J. Strunk. Unsupervised multilingual sentence boundary detection, *Computational Linguistics*. Cambridge, USA: MIT Press, 2006, vol. 3-(4).
- H. Schmid. Probabilistic Part-of-Speech Tagging Using Decision Trees. In *Proc. of the International Conference on New Methods in Language Processing*, 44-49 (1994)
- B. Steven, E. Klein and E. Loper. *Natural Language Processing with Python*. O'Reilly Media Inc. 2009.
- CoNLL X Format:
<http://depparse.uvt.nl/depparse-wiki/DataFormat>
- DeepSearch: <http://semawiki.di.unipi.it/demo.html>
- GATE: <http://gate.ac.uk/>
- Hadoop: <http://hadoop.apache.org/>
- JULIE NLP:
http://www.julielab.de/Resources/Software/NLP_Tool_s.html
- LingPipe: <http://alias-i.com/lingpipe/>
- NLTK: <http://www.nltk.org/>
- OpenNLP: <http://opennlp.sourceforge.net/>
- Quex: <http://quex.sourceforge.net/>
- SWIG: <http://www.swig.org/>
- Tanl Dependency Notation:
http://medialab.di.unipi.it/wiki/Tanl_Tagsets
- Tanl Hadoop Streaming:
<http://medialab.di.unipi.it/wiki/Hadoop>
- Tanl Parser: <http://paleo.di.unipi.it/parse>
- TornadoWeb: <http://www.tornadoweb.org/>
- UIMA: <http://incubator.apache.org/uima/>
- Yahoo! Correlator: <http://sandbox.yahoo.com/Correlator>