txtDataNLP Framework – Documentation

Table of Contents

[1 Introduction 1](#_Toc43636766)

[2 Central Concepts and corresponding classes 2](#_Toc43636767)

[2.1 TextUnit 2](#_Toc43636768)

[2.2 Word 3](#_Toc43636769)

[2.3 Annotation 3](#_Toc43636770)

[2.4 Span 4](#_Toc43636771)

[2.5 IAnnotationObject 4](#_Toc43636772)

[2.6 AbstractAnnotator 4](#_Toc43636773)

[2.7 AbstractCreator 4](#_Toc43636774)

[2.8 Language 4](#_Toc43636775)

[3 Annotators 4](#_Toc43636776)

[3.1 What is an Annotator? 5](#_Toc43636777)

[3.2 WhitelistAnnotator 5](#_Toc43636778)

[3.3 PosPatternAnnotator 5](#_Toc43636779)

[3.4 SubsumedAnnotationsRemover 5](#_Toc43636780)

[3.5 RecursiveDictionaryAnnotator 5](#_Toc43636781)

[4 Common use cases 5](#_Toc43636782)

[4.1 Creating a simple dictionary-based annotator 5](#_Toc43636783)

[4.2 Creating a more complex annotator based on “POS-Rules” 5](#_Toc43636784)

[4.3 Working with annotations from a third-party Machine Learning model 5](#_Toc43636785)

[4.4 Implementing white- and blacklists on top of a third-party Machine Learning model 5](#_Toc43636786)

[4.5 Building a complex NLP-Pipeline that combines various Machine Learning and/or rule-based approaches 5](#_Toc43636787)

# Introduction

txtDataNLP is a light Java framework for Natural Language Processing (NLP) projects.

Typical use cases include:

* Combining various third-party NLP and Machine Learning (ML) tools into one setup.
* Implementing algorithms that combine output from various NLP/ML modules and return one coherent text analysis—even if their output is contradictory.
* Extraction of features from low-level NLP tools (tokenizers, taggers) for downstream ML models.
* Implementing dictionary-based, rule-based or linguistically motivated approaches to NLP.
* Implementing white- or backlists that often sit on top of ML approaches to deal with specific mistakes the ML models make.

txtDataNLP contains no ML algorithms or models itself but is was built to support the use of ML models in NLP and to provide infrastructure to combine various NLP/ML modules, use them together reliably and to fill the gaps that they commonly leave.

# Central classes and corresponding concepts

## TextUnit

In many aspects the TextUnit class is the central class of txtDataNLP. It represents a piece of text that is supposed to be analyzed and annotated. Typically, the text represented in one instance of TextUnit is a sentence, but this does not have to be the case. In some scenarios, a TextUnit might also represent different things, e.g. a search query, a headline, a sub-heading or even a complete paragraph or article.

Each TextUnit instance keeps track of three kinds of information:

1. The surface of the respective text, as a Java String. It can be retrieved with the getSurfaceText() method.
2. A list of words that represent the words the text contains, using the Word class, see below. Words can be retrieved with getWords().
3. A list of annotations, that represent interesting bits in the text, using the Annotation class, see below. All annotations of this TextUnit can be retrieved with getAnnotations().

The TextUnit class contains a toString method which is useful to get a quick idea of what is stored in an instance:

|  |
| --- |
| He lives in London and she in New York City.  He PRP (0-2)  lives VBZ (3-8)  in IN (9-11)  London NNP (12-18)  and CC (19-22)  she PRP (23-26)  in IN (27-29)  New NNP (30-33)  York NNP (34-38)  City NNP (39-43)  . . (43-44)  [12-18] 'London' Location  [30-43] 'New York City' Location |

The first line contains the surface text of the analyzed sentence. The following lines list each of the words that are present in the sentence, together with their part-of-speech (POS) tags and the start and end positions of each word in the surface text. Finally, the last two lines show annotations: Here two entities have been detected: “London” and “New York City”. These are given together with the position where they occur in the sentence and their semantic class: “Location”.

Note that TextUnit merely provides a container to store such information. The actual analysis necessary to determine word boundaries, POS tags and entities is done somewhere else and can be configured.

## Word

The Word class represents a word that occurs in a text. It contains various pieces of information about that word, for example its surface appearance, the position in characters where the word starts in the text in which it was found, the position where it ends etc.

Since words appear in text and text in txtDataNLP is typically represented with the TextUnit class, the words you will be working with often originate from TextUnit’s getWords() method.

Important methods of the Word class are:

* getSurface() Returns the surface structure of the word.
* getStarts() Returns the position where this word starts in the text in which it occurs.
* getEnds() Returns the position where this word ends in the text in which it occurs.

In a typical scenario the three just mentioned method should always return a value and never return null.

Other methods that might be set, or return null, depending on the scenario are:

* getRoot(), returns the root form of the word, e.g. “house” for “houses”.
* getPOS(), return the part-of-speech for the word, e.g. “nn” for “house”.
* getMorph(), returns morphological information for the word.

For these three methods, the values returned is highly dependent on the actual project setup. The project needs to be configured so that these fields are filled with relevant information. In many project scenarios, some or all of these fields might return null.

## Annotation

An annotation represents a piece of text that is for some reason interesting or relevant in a certain scenario. If you are for example looking to detect locations in a text and one sentence in that text is “He lives in London and she in New York City”, the most likely parts that will be annotated are “London” and “New York City”.

As the Word class, the Annotation class has the following three methods:

* getSurface() Returns the surface structure of the annotation.
* getStarts() Returns the position where this annotation starts in the text in which it occurs.
* getEnds() Returns the position where this annotation ends in the text in which it occurs.

Each Annotation class also has a dedicated slot where semantic information belonging to the annotation can be stored: IAnnotationObject, which can be retrieved with getAnnotationObject(), see below.

## Span

A Span represents a region in a text. It has no further semantics and is the base class for Word and Annotation. Therefore, it also has the getSurface()**,** getStarts() and getEnds() methods.

If a text span is supposed to carry a special meaning, it is likely that the Annotation class is a better fit.

## IAnnotationObject

Typically, annotations carry a certain meaning. If “London” is annotated somewhere in a text, we might want to note that this annotation is of type “Location”. Depending on the implementation scenario, we also might want to be able to retrieve certain other information of this annotation, e.g. the fact that London is in England, that it is the capital city and maybe its geo coordinates.

Each annotation has an IAnnotationObject, which can be retrieved with getAnnotationObject()**.** This functions as a container for any semantic information that one wants to associate with the annotation.

The interface IAnnotationObject only has two methods: getType() andsetType(), which are meant to be used to get and set the semantic type of the annotation (e.g. “Location” for “London”). The class SimpleAnnotationObject provides a straightforward implementation of IAnnotationObject. In many scenarios however classes implementing IAnnotationObject are significantly more complex, e.g. in cases where they link to an external ontology.

## AbstractAnnotator

## AbstractCreator

## Language

# Annotators

## What is an Annotator?

## WhitelistAnnotator

## PosPatternAnnotator

## SubsumedAnnotationsRemover

## RecursiveDictionaryAnnotator

# Common use cases

## Creating a simple dictionary-based annotator

## Creating a more complex annotator based on “POS-Rules”

## Working with annotations from a third-party Machine Learning model

## Implementing white- and blacklists on top of a third-party Machine Learning model

## Building a complex NLP-Pipeline that combines various Machine Learning and/or rule-based approaches