py-processors Documentation

Release 3.2.3

Gus Hahn-Powell

Jun 14, 2018

Contents

CHAPTER 1

Overview

This page provides an overview of the available text processing pipelines.

CHAPTER 2

Annotating text

a `.annotate` call produces a `Document` with the following features:

- sentence segmentation
- tokenization
- part-of-speech tagging
- lemmatization
- named entity recognition
- syntactic dependency parsing

Annotation suitable to open domain text can be performed with either a `.annotate` or a `.fastnlp.annotate` call:

```
API = ProcessorsAPI(port=8886)
# annotate the provided text using FastNLPProcessor (a CoreNLP wrapper)
doc = API.fastnlp.annotate("My name is Inigo Montoya.  You killed my father.  Prepare␣
↪to die.")
```

For annotation tuned to the biomedical domain, use `.bionlp.annotate`:

```
API = ProcessorsAPI(port=8886)
doc = API.bionlp.annotate("In contrast, the EGFR T669A mutant increased both basal␣
↪EGFR and ERBB3 tyrosine phosphorylation that was not augmented by MEK inhibition.")
```

CHAPTER 3

API Reference

See *the API reference* for more details.

CHAPTER 4

Odin

Refer to the jupyter notebook for a tutorial on using Odin from `py-processors`.

The most comprehensive reference for the Odin event extraction framework is our manual, which includes a detailed description of our rule language (`rune`):

- http://arxiv.org/pdf/1509.07513v1.pdf

Planned additions to Odin

- Support in rules for comparisons using distributional semantic similarity (word vectors)
- Efficient matching gazetteers
- WordNet support in patterns
- Edit distance

CHAPTER 5

API Reference

This section of the documentation provides detailed information on functions, classes, and methods.

Server Communication

Communicating with the NLP server (`processors-server`) is handled by the following classes:

ProcessorsBaseAPI

class processors.api.**ProcessorsBaseAPI**(***kwargs*)

Bases: object

Manages a connection with processors-server and provides an interface to the API.

Parameters

- **port** (*int*) – The port the server is running on or should be started on. Default is 8886.
- **hostname** (*str*) – The host name to use for the server. Default is "localhost".
- **log_file** (*str*) – The path for the log file. Default is py-processors.log in the user's home directory.

annotate(*text*)

Produces a Document from the provided *text* using the default processor.

clu.**annotate**(*text*)

Produces a Document from the provided *text* using CluProcessor.

fastnlp.**annotate**(*text*)

Produces a Document from the provided *text* using FastNLPProcessor.

bionlp.**annotate**(*text*)

Produces a Document from the provided *text* using BioNLPProcessor.

annotate_from_sentences(*sentences*)
Produces a Document from *sentences* (a list of text split into sentences). Uses the default processor.

fastnlp.**annotate_from_sentences**(*sentences*)
Produces a Document from *sentences* (a list of text split into sentences). Uses FastNLPProcessor.

bionlp.**annotate_from_sentences**(*sentences*)
Produces a Document from *sentences* (a list of text split into sentences). Uses BioNLPProcessor.

corenlp.sentiment.**score_sentence**(*sentence*)
Produces a sentiment score for the provided *sentence* (an instance of Sentence).

corenlp.sentiment.**score_document**(*doc*)
Produces sentiment scores for the provided *doc* (an instance of Document). One score is produced for each sentence.

corenlp.sentiment.**score_segmented_text**(*sentences*)
Produces sentiment scores for the provided *sentences* (a list of text segmented into sentences). One score is produced for item in *sentences*.

odin.**extract_from_text**(*text*, *rules*)
Produces a list of Mentions for matches of the provided *rules* on the *text*. *rules* can be a string of Odin rules, or a url ending in *.yml* or *.yaml*.

odin.**extract_from_document**(*doc*, *rules*)
Produces a list of Mentions for matches of the provided *rules* on the *doc* (an instance of Document). *rules* can be a string of Odin rules, or a url ending in .yml or yaml.

ProcessorsAPI

class processors.api.**ProcessorsAPI**(***kwargs*)
Bases: *processors.api.ProcessorsBaseAPI*

Manages a connection with the processors-server jar and provides an interface to the API.

Parameters

- **timeout** (*int*) – The number of seconds to wait for the server to initialize. Default is 120.
- **jvm_mem** (*str*) – The maximum amount of memory to allocate to the JVM for the server. Default is "-Xmx3G".
- **jar_path** (*str*) – The path to the processors-server jar. Default is the jar installed with the package.
- **kee_alive** (*bool*) – Whether or not to keep the server running when ProcessorsAPI instance goes out of scope. Default is false (server is shut down).
- **log_file** (*str*) – The path for the log file. Default is py-processors.log in the user's home directory.

start_server(*jar_path*, ***kwargs*)
Starts the server using the provided *jar_path*. Optionally takes hostname, port, jvm_mem, and timeout.

stop_server()
Attempts to stop the server running at self.address.

OdinAPI

class processors.api.**OdinAPI**(*address*)
Bases: object

API for performing rule-based information extraction with Odin.

Parameters **address** (*str*) – The base address for the API (i.e., everything preceding */api/..*)

OdinAPI

class processors.api.**OpenIEAPI**(*address*)
Bases: object

SentimentAnalysisAPI

class processors.sentiment.**SentimentAnalysisAPI**(*address*)
Bases: object

API for performing sentiment analysis

Parameters **address** (*str*) – The base address for the API (i.e., everything preceding */api/..*)

corenlp
processors.sentiment.CoreNLPSentimentAnalyzer – Service using [*CoreNLP*'s tree-based system](https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf) for performing sentiment analysis.

Data Structures

NLPDatum

class processors.ds.**NLPDatum**
Bases: object

Document

class processors.ds.**Document**(*sentences*)
Bases: *processors.ds.NLPDatum*

Storage class for annotated text. Based on [*org.clulab.processors.Document*](https://github.com/clulab/processors/blob/master/main/src/main/scala/org/clulab/processors/Document.scala)

Parameters **sentences** ([processors.ds.Sentence]) – The sentences comprising the *Document*.

id
str or None – A unique ID for the *Document*.

size
int – The number of *sentences*.

sentences
sentences – The sentences comprising the *Document*.

words
[str] – A list of the *Document*'s tokens.

tags
[str] – A list of the *Document*'s tokens represented using part of speech (PoS) tags.

lemmas
[str] – A list of the *Document*'s tokens represented using lemmas.

_entities
[str] – A list of the *Document*'s tokens represented using IOB-style named entity (NE) labels.

nes
dict – A dictionary of NE labels represented in the *Document* -> a list of corresponding text spans.

bag_of_labeled_deps
[str] – The labeled dependencies from all sentences in the *Document*.

bag_of_unlabeled_deps
[str] – The unlabeled dependencies from all sentences in the *Document*.

text
str or None – The original text of the *Document*.

bag_of_labeled_dependencies_using(*form*)
Produces a list of syntactic dependencies where each edge is labeled with its grammatical relation.

bag_of_unlabeled_dependencies_using(*form*)
Produces a list of syntactic dependencies where each edge is left unlabeled without its grammatical relation.

Sentence

class processors.ds.**Sentence**(***kwargs*)
Bases: *processors.ds.NLPDatum*

Storage class for an annotated sentence. Based on [*org.clulab.processors.Sentence*](https://github.com/clulab/processors/blob/master/main/src/main/scala/org/clulab/processors/Sentence.scala)

Parameters

- **text** (*str or None*) – The text of the *Sentence*.
- **words** (*[str]*) – A list of the *Sentence*'s tokens.
- **startOffsets** (*[int]*) – The character offsets starting each token (inclusive).
- **endOffsets** (*[int]*) – The character offsets marking the end of each token (exclusive).
- **tags** (*[str]*) – A list of the *Sentence*'s tokens represented using part of speech (PoS) tags.
- **lemmas** (*[str]*) – A list of the *Sentence*'s tokens represented using lemmas.
- **chunks** (*[str]*) – A list of the *Sentence*'s tokens represented using IOB-style phrase labels (ex. *B-NP*, *I-NP*, *B-VP*, etc.).
- **entities** (*[str]*) – A list of the *Sentence*'s tokens represented using IOB-style named entity (NE) labels.
- **graphs** (*dict*) – A dictionary of {graph-name -> {edges: [{source, destination, relation}], roots: [int]}}

text
str – The text of the *Sentence*.

startOffsets
[int] – The character offsets starting each token (inclusive).

endOffsets
[int] – The character offsets marking the end of each token (exclusive).

length
int – The number of tokens in the *Sentence*

graphs
dict – A dictionary (str -> *processors.ds.DirectedGraph*) mapping the graph type/name to a *processors.ds.DirectedGraph*.

basic_dependencies
processors.ds.DirectedGraph – A *processors.ds.DirectedGraph* using basic Stanford dependencies.

collapsed_dependencies
processors.ds.DirectedGraph – A *processors.ds.DirectedGraph* using collapsed Stanford dependencies.

dependencies
processors.ds.DirectedGraph – A pointer to the prefered syntactic dependency graph type for this *Sentence*.

_entities
[str] – The IOB-style Named Entity (NE) labels corresponding to each token.

_chunks
[str] – The IOB-style chunk labels corresponding to each token.

nes
dict – A dictionary of NE labels represented in the *Document* -> a list of corresponding text spans (ex. {"PERSON": [phrase 1, ..., phrase n]}). Built from *Sentence._entities*

phrases
dict – A dictionary of chunk labels represented in the *Document* -> a list of corresponding text spans (ex. {"NP": [phrase 1, ..., phrase n]}). Built from *Sentence._chunks*

bag_of_labeled_dependencies_using(*form*)
Produces a list of syntactic dependencies where each edge is labeled with its grammatical relation.

bag_of_unlabeled_dependencies_using(*form*)
Produces a list of syntactic dependencies where each edge is left unlabeled without its grammatical relation.

Edge

class processors.ds.**Edge**(*source*, *destination*, *relation*)
Bases: *processors.ds.NLPDatum*

DirectedGraph

class processors.ds.**DirectedGraph**(*kind*, *deps*, *words*)
Bases: *processors.ds.NLPDatum*

Storage class for directed graphs.

Parameters

- **kind** (*str*) – The name of the directed graph.
- **deps** (*dict*) – A dictionary of {edges: [{source, destination, relation}], roots: [int]}
- **words** (*[str]*) – A list of the word form of the tokens from the originating *Sentence*.

_words
[str] – A list of the word form of the tokens from the originating *Sentence*.

roots
[int] – A list of indices for the syntactic dependency graph's roots. Generally this is a single token index.

edges
list[processors.ds.Edge] – A list of *processors.ds.Edge*

incoming
A dictionary of {int -> [int]} encoding the incoming edges for each node in the graph.

outgoing
A dictionary of {int -> [int]} encoding the outgoing edges for each node in the graph.

labeled
[str] – A list of strings where each element in the list represents an edge encoded as source index, relation, and destination index ("source_relation_destination").

unlabeled
[str] – A list of strings where each element in the list represents an edge encoded as source index and destination index ("source_destination").

graph
networkx.Graph – A *networkx.graph* representation of the *DirectedGraph*. Used by *shortest_path*

bag_of_labeled_dependencies_from_tokens(*form*)
Produces a list of syntactic dependencies where each edge is labeled with its grammatical relation.

bag_of_unlabeled_dependencies_from_tokens(*form*)
Produces a list of syntactic dependencies where each edge is left unlabeled without its grammatical relation.

Mention

class processors.odin.**Mention**(*token_interval*, *sentence*, *document*, *foundBy*, *label*, *labels=None*, *trigger=None*, *arguments=None*, *paths=None*, *keep=True*, *doc_id=None*)

Bases: *processors.ds.NLPDatum*

A labeled span of text. Used to model textual mentions of events, relations, and entities.

Parameters

- **token_interval** (Interval) – The span of the Mention represented as an Interval.
- **sentence** (*int*) – The sentence index that contains the Mention.
- **document** (Document) – The Document in which the Mention was found.
- **foundBy** (*str*) – The Odin IE rule that produced this Mention.
- **label** (*str*) – The label most closely associated with this span. Usually the lowest hyponym of "labels".
- **labels** (*list*) – The list of labels associated with this span.

- **trigger** (*dict or None*) – dict of JSON for Mention's trigger (event predicate or word(s) signaling the Mention).
- **arguments** (*dict or None*) – dict of JSON for Mention's arguments.
- **paths** (*dict or None*) – dict of JSON encoding the syntactic paths linking a Mention's arguments to its trigger (applies to Mentions produces from *type:"dependency"* rules).
- **doc_id** (*str or None*) – the id of the document

tokenInterval
processors.ds.Interval – An *Interval* encoding the *start* and *end* of the *Mention*.

start
int – The token index that starts the *Mention*.

end
int – The token index that marks the end of the Mention (exclusive).

sentenceObj
processors.ds.Sentence – Pointer to the *Sentence* instance containing the *Mention*.

characterStartOffset
int – The index of the character that starts the *Mention*.

characterEndOffset
int – The index of the character that ends the *Mention*.

type
Mention.TBM or Mention.EM or Mention.RM – The type of the *Mention*.

See also:

[*Odin* manual](https://arxiv.org/abs/1509.07513)

matches(*label_pattern*)
Test if the provided pattern, *label_pattern*, matches any element in *Mention.labels*.

overlaps(*other*)
Test whether other (token index or Mention) overlaps with span of this Mention.

copy(***kwargs*)
Copy constructor for this Mention.

words()
Words for this Mention's span.

tags()
Part of speech for this Mention's span.

lemmas()
Lemmas for this Mention's span.

_chunks()
chunk labels for this Mention's span.

_entities()
NE labels for this Mention's span.

`JSON` serialization/deserialization is handled via `processors.serialization.JSONSerializer`.

Interval

class processors.ds.**Interval**(*start*, *end*)
Bases: *processors.ds.NLPDatum*

Defines a token or character span

Parameters

- **start** (*str*) – The token or character index where the interval begins.
- **end** (*str*) – The 1 + the index of the last token/character in the span.

contains(*that*)
Test whether *that* (int or Interval) overlaps with span of this Interval.

overlaps(*that*)
Test whether this Interval contains another. Equivalent Intervals will overlap.

Annotators (Processors)

Text annotation is performed by communicating with one of the following annotators ("processors").

CluProcessor

class processors.annotators.**CluProcessor**(*address*)
Bases: processors.annotators.Processor

Processor for text annotation based on [*org.clulab.processors.clu.CluProcessor*](https://github.com/clulab/processors/blob/master/main/src/main/scala/org/clulab/processors/clu/CluProcessor.scala)

Uses the Malt parser.

FastNLPProcessor

class processors.annotators.**FastNLPProcessor**(*address*)
Bases: processors.annotators.Processor

Processor for text annotation based on [*org.clulab.processors.fastnlp.FastNLPProcessor*](https://github.com/clulab/processors/blob/master/corenlp/src/main/scala/org/clulab/processors/fastnlp/FastNLPProcessor.scala)

Uses the Stanford CoreNLP neural network parser.

BioNLPProcessor

class processors.annotators.**BioNLPProcessor**(*address*)
Bases: processors.annotators.Processor

Processor for biomedical text annotation based on [*org.clulab.processors.fastnlp.FastNLPProcessor*](https://github.com/clulab/processors/blob/master/corenlp/src/main/scala/org/clulab/processors/fastnlp/FastNLPProcessor.scala)

CoreNLP-derived annotator.

Sentiment Analysis

SentimentAnalyzer

class processors.sentiment.**SentimentAnalyzer**(*address*)
Bases: object

CoreNLPSentimentAnalyzer

class processors.sentiment.**CoreNLPSentimentAnalyzer**(*address*)
Bases: *processors.sentiment.SentimentAnalyzer*

Bridge to [*CoreNLP*'s tree-based sentiment analysis system](https://nlp.stanford.edu/~socherr/EMNLP2013_RNTN.pdf)

paths

DependencyUtils

class processors.paths.**DependencyUtils**
Bases: object

A set of utilities for analyzing syntactic dependency graphs.

build_networkx_graph(*roots*, *edges*, *name*)
Constructs a networkx.Graph

shortest_path(*g*, *start*, *end*)
Finds the shortest path in a *networkx.Graph* between any element in a list of start nodes and any element in a list of end nodes.

retrieve_edges(*dep_graph*, *path*)
Converts output of *shortest_path* into a list of triples that include the grammatical relation (and direction) for each node-node "hop" in the syntactic dependency graph.

simplify_tag(*tag*)
Maps part of speech (PoS) tag to a subset of PoS tags to better consolidate categorical labels.

lexicalize_path(*sentence*, *path*, *words=False*, *lemmas=False*, *tags=False*, *simple_tags=False*, *entities=False*, *limit_to=None*)
Lexicalizes path in syntactic dependency graph using Odin-style token constraints.

pagerank(*networkx_graph*, *alpha=0.85*, *personalization=None*, *max_iter=1000*, *tol=1e-06*, *nstart=None*, *weight='weight'*, *dangling=None*)
Measures node activity in a *networkx.Graph* using a thin wrapper around *networkx* implementation of pagerank algorithm (see *networkx.algorithms.link_analysis.pagerank*). Use with *processors.ds.DirectedGraph.graph*.

HeadFinder

class processors.paths.**HeadFinder**
Bases: object

Serialization

JSONSerializer

class processors.serialization.**JSONSerializer**
Bases: object

Utilities for serialization/deserialization of data structures.

Visualization

JupyterVisualizer

.. autoclass:: processors.Visualization.JupyterVisualizer :show-inheritance:

CHAPTER 6

A walkthrough example

The following examples give an overview of how to use `py-processors`.

Getting started

For annotation and parsing, `py-processors` communicates with `processors-server` using a REST interface. The server can be run either via `java` directly or in a `docker` container. Let's look at how to connect to the server.

CHAPTER 7

Running the NLP server

Option 1: `processors-server.jar`

This method requires `java` and a compatible `processors-server.jar` for the server. An appropriate `jar` will be downloaded automatically if one is not found.

```
from processors import *
# The constructor requires you to specify a port for running the server.
API = ProcessorsAPI(port=8886)
```

NOTE: It may take a minute or so for the server to initialize as there are some large model files that need to be loaded.

Option 2: `docker` container

You can pull the official container from Docker Hub:

```
docker pull myedibleenso/processors-server:latest
```

You can check `py-processors` for the appropriate version to retrieve:

```
import processors
# print the recommended processors-server version
print(import processors.__ps_rec__)
```

Just replace `latest` in the command above with the appropriate version (`3.1.0` onwards).

The following command will run the container in the background and expose the service on port `8886`:

```
docker run -d -e _JAVA_OPTIONS="-Xmx3G" -p 127.0.0.1:8886:8888 --name procserv␣
↪myedibleenso/processors-server
```

For a more detailed example showcasing configuration options, take a look at this `docker-compose.yml` file. You'll need to map a local port to `8888` in the container.

Once the container is running, you can connect to it via `py-processors`:

```
from processors import *
# provide the local port that you mapped to 8888 on the running container
API = ProcessorsBaseAPI(hostname="127.0.0.1", port=8886)
```

Annotating text

Text can be annotated automatically with these linguistic attributes.

```
# try annotating some text using FastNLPProcessor (a CoreNLP wrapper)
doc = API.fastnlp.annotate("My name is Inigo Montoya.  You killed my father.  Prepare␣
↪to die.")

# you can also annotate text already segmented into sentences
doc = API.fastnlp.annotate_from_sentences(["My name is Inigo Montoya.", "You killed␣
↪my father.", "Prepare to die."])

# There should be 3 Sentence objects in this Document
doc.size

# A Document contains the words, pos tags, lemmas, named entities, and syntactic␣
↪dependencies of its component Sentences
doc.bag_of_labeled_deps

# We can access the named entities for the Document as a dictionary mapping an NE␣
↪label -> list of named entities
doc.nes

# A Sentence contains words, pos tags, lemmas, named entities, and syntactic␣
↪dependencies
doc.sentences[0].lemmas

# get the first sentence
s = doc.sentences[0]

# the number of tokens in this sentence
s.length

# the named entities contained in this sentence
s.nes

# generate labeled dependencies using "words", "tags", "lemmas", "entities", or token␣
↪index ("index")
s.bag_of_labeled_dependencies_using("tags")

# generate unlabeled dependencies using "words", "tags", "lemmas", "entities", or␣
↪token index ("index")
s.bag_of_unlabeled_dependencies_using("lemmas")

# play around with the dependencies directly
deps = s.dependencies

# see what dependencies lead directly to the first token (i.e. token 0 is the␣
↪dependent of what?)
deps.incoming[0]
```

```
# see what dependencies are originating from the first token (i.e. token 0 is the␣
↪head of what?)
deps.outgoing[0]

# find all shortest paths between "name" and either "Inigo" or "Montoya".
deps.shortest_paths(start=1, end=[3,4])

# find the shortest path between "name" and either "Inigo" or "Montoya".  Prefer a␣
↪path that involves a "nsubj" relation.
sp = deps.shortest_path(start=1, end=[3,4],
scoring_func=lambda path: 9000 if any(seg[1] == "nsubj" for seg in path) else 0)

# generate an Odin-like pattern with partial lexicalization
DependencyUtils.lexicalize_path(sentence=s, path=sp, lemmas=True, tags=True)

# limit lexicalization to tokens 1 and 4 (if present)
DependencyUtils.lexicalize_path(sentence=s, path=sp, lemmas=True, tags=True, limit_
↪to=[1,4])

# run PageRank on the dependency graph to find nodes with the most activity.
# SPOILER: When using reverse=True, the nodes with the highest weight are usually the␣
↪sentential predicate and its args
deps.pagerank(reverse=True)

# find out which nodes are most central to the dependency graph
deps.degree_centrality()

# retrieve the likely semantic head for a sentence.
from processors.paths import HeadFinder
doc2 = API.annotate("acute renal failure")
sentence = doc2.sentences[0]
# select the graph to examine (default is "stanford-collapsed") and
# optionally limit to a set of PoS tags (regex or str)
head_idx = sentence.semantic_head(graph_name="stanford-collapsed", valid_tags=None)
head_word = sentence.words[head_idx] if head_idx else None

# try using BioNLPProcessor
biodoc = api.bionlp.annotate("We next considered the effect of Ras monoubiquitination␣
↪on GAP-mediated hydrolysis")

# check out the bio-specific entities
biodoc.nes
```

Serializing to/from `json`

Once you've annotated text, you can serialize it to `json` for later loading.

```
# serialize to/from JSON!
json_file = "serialized_doc_example.json"
ross_doc = api.fastnlp.annotate("We don't make mistakes, just happy little accidents.
↪")

# serialize to JSON
with open(json_file, "w") as out:
```

```
    out.write(ross_doc.to_JSON())

# load from JSON
with open(json_file, "r") as jf:
    d = Document.load_from_JSON(json.load(jf))
```

Perform sentiment analysis

You can perform sentiment analysis using CoreNLP's tree-based system.

```
# get sentiment analysis scores
review = "The humans are dead."
doc = API.fastnlp.annotate(review)

# try Stanford's tree-based sentiment analysis
# you'll get a score for each Sentence
# scores are between 1 (very negative) - 5 (very positive)
scores = API.sentiment.corenlp.score_document(doc)

# you can pass text directly
scores = API.sentiment.corenlp.score_text(review)

# ... or a single sentence
score = API.sentiment.corenlp.score_sentence(doc.sentences[0])

# ... or from text already segmented into sentences
lyrics = ["My sugar lumps are two of a kind", "Sweet and white and highly refined",
↪"Honeys try all kinds of tomfoolery", "to steal a feel of my family jewelry"]
scores = API.sentiment.corenlp.score_segmented_text(lyrics)
```

Rule-based information extraction (IE) with Odin

If you're unfamiliar with writing Odin rules, see our manual for a primer on the language: http://arxiv.org/pdf/1509.07513v1.pdf

```
# Do rule-based IE with Odin!
# see http://arxiv.org/pdf/1509.07513v1.pdf for details
example_rule = """
- name: "ner-person"
  label: [Person, PossiblePerson, Entity]
  priority: 1
  type: token
  pattern: |
    [entity="PERSON"]+
    |
    [tag=/^N/]* [tag=/^N/ & outgoing="cop"] [tag=/^N/]*
"""

example_text = """
Barack Hussein Obama II is the 44th and current President of the United States and␣
↪the first African-American to hold the office.
He is a Democrat.
Obama won the 2008 United States presidential election, on November 4, 2008.
```

```
He was inaugurated on January 20, 2009.
"""

# take a look at the .label, .labels, and .text attributes of each mention
mentions = API.odin.extract_from_text(example_text, example_rule)
# visualize the structure of a mention as colored output in the terminal
for m in mentions: print(m)

# Alternatively, you can provide a rule URL.  The URL should end with .yml or .yaml.
rules_url = "https://raw.githubusercontent.com/clulab/reach/
↪508697db2217ba14cd1fa0a99174816cc3383317/src/main/resources/edu/arizona/sista/demo/
↪open/grammars/rules.yml"

mentions = API.odin.extract_from_text(example_text, rules_url)

# You can also perform IE with Odin on a Document.
barack_doc = API.annotate(example_text)
mentions = API.odin.extract_from_document(barack_doc, rules_url)

# mentions can be serialized as well
mentions_json_file = "mentions.json"

with open(mentions_json_file, "w") as out:
    out.write(JSONSerializer.mentions_to_JSON(mentions))

# loading from a file is also handled via JSONSerializer
with open(mentions_json_file, "r") as jf:
    mentions = JSONSerializer.mentions_from_JSON(json.load(jf))
```

CHAPTER 8

OpenIE for concept recognition

coming soon

CHAPTER 9

Jupyter notebook visualizations

py-processors supports some custom notebook-based visualizations, but you'll need to install the extra [jupyter] module in order to use them:

```
pip install "py-processors[jupyter]"
```

These visualizations make use of our fork of displaCy, You can now visualize a Sentence graph as an SVG image using visualization.JupyterVisualizer.display_graph():

```
from processors.visualization import JupyterVisualizer as viz
# run this snippet within a jupyter notebook
text = "To be loved by unicorns is the greatest gift of all."
doc = API.annotate(text)
viz.display_graph(doc.sentences[0], graph_name="stanford-collapsed")
```

Mentions can also be visualized in a notebook:

```
# run this snippet within a jupyter notebook
rules = """
rules:
  - name: "ner-location"
    label: [Location, PossibleLocation, Entity]
    priority: 1
    type: token
    pattern: |
      [entity="LOCATION"]+ | Twin Peaks

  - name: "ner-person"
    label: [Person, PossiblePerson, Entity]
    priority: 1
    type: token
    pattern: |
     [entity="PERSON"]+

  - name: "ner-org"
    label: [Organization, Entity]
```

```
    priority: 1
    type: token
    pattern: |
      [entity="ORGANIZATION"]+

  - name: "ner-date"
    label: [Date]
    priority: 1
    type: token
    pattern: |
      [entity="DATE"]+

  - name: "missing"
    label: Missing
    pattern: |
      trigger = [lemma=go] missing
      theme: Person = <xcomp nsubj
      date: Date? = prep_on
"""
mentions = API.odin.extract_from_text("FBI Special Agent Dale Cooper went missing on␣
↪June 10, 1991.  He was last seen in the woods of Twin Peaks. ", rules=rules)

for m in mentions: viz.display_mention(m)
```

CHAPTER 10

Other ways of initializing the server

Using a custom `processors-server`

When initializing the API, you can specify a path to a custom `processors-server.jar` using the `jar_path` parameter:

```
from processors import *

API = ProcessorsAPI(port=8886, jar_path="path/to/processors-server.jar")
```

Alternatively, you can set an environment variable, `PROCESSORS_SERVER`, with the path to the `jar` you wish to use. In your `.bashrc` (or equivalent), add this line with the path to the `jar` you wish to use with `py-processors`:

```
export PROCESSORS_SERVER="path/to/processors-server.jar"
```

Remember to `source` your `profile`:

```
source path/to/your/.profile
```

`py-processors` will now prefer this jar whenever a new API is initialized.

NOTE: If you decide that you no longer want to use this enivronment variable, remember to both remove it from your profile and run `unset PROCESSORS_SERVER` *from the shell.*

Allocating memory

By default, the server will be run with 3GB of RAM. You might be able to get by with a little less, though. You can start the server with a different amount of memory with the `jvm_mem` parameter:

```
from processors import *
# run the sever with 2GB of memory
API = ProcessorsAPI(port=8886, jvm_mem="-Xmx2G")
```

NOTE: This won't have any effect if the server is already running on the given port.

CHAPTER 11

Keeping the server running

If you've launched the server via `java`, `py-processors` will by default attempt to shut down the server whenever an API instance goes out of scope (ex. your script finishes or you exit the interpreter).

If you'd prefer to keep the server alive, you'll need to initialize the API with `keep_alive=True`:

```
from processors import *

API = ProcessorsAPI(port=8886, keep_alive=True)
```

This is useful if you're sharing access to the server on a network, or if you have a bunch of independent tasks and would prefer to avoid waiting for the server to initialize again and again.

CHAPTER 12

FAQ

I want the latest `processors-server.jar`

In that case, take a look over here.

Something is already running on port xxxx, but I don't know what. Help!

Try running the following command:

```
lsof -i :<portnumber>
```

You can then kill the responsible process using the reported `PID`

Does `py-processors` produce a (server) log?

Yep! By default, it will write to `~/py-processors.log`. You can specify a different location when initializing the API:

```
from processors import *

API = ProcessorsAPI(port=8886, log_file="my/desired/log/file.log")
```

CHAPTER 13

Changes

- v3.2.2:
  - Improved `jar` download
  - Removed `six` dependency
  - More compact `json`
- v3.2.1:
  - Fix to `limit_to` param of `DependencyUtils.lexicalize_path`
- v3.2.0:
  - `ProcessorsAPI` now inherits from `ProcessorsBaseAPI`
  - `ProcessorsBaseAPI` can be used with a `docker` backend
  - Updated documentation
  - Updated requirements for building documentation
  - More tests covering syntactic dependencies
- v3.1.0:
  - Upgraded `processors-server` version to v3.1.0
  - Added support for `CluProcessor`
  - `odin` variables can now be used in imports
  - Log file no longer prefixed with `.`
  - `jupyter` notebook visualizations now treated as an extra module that can be installed via `pip install "py-processors[jupyter]"`
- v3.0.3:
  - `jupyter` notebook visualizations for `Sentence` graphs (dependency parses) and `Mention` structure
    * `visualization.JupyterVisualizer.display_graph()`

    * visualization.Jupyter.Visualizer.display_mention()
  - Display errors for broken Odin rules
  - Upgraded processors-server version to v3.0.2
  - Added HeadFinder and HeadFinder.semantic_head()
  - pagerank bug fix related to outdated parameter
  - Make Sentence, DirectedGraph, and Mention hashable
- v3.0.2:
  - Added support to reverse directed graph for pagerank, resulting in predicate and argument nodes having higher weights
  - DependencyUtils.directed_relation()
  - Added DirectedGraph.degree_centrality(), DirectedGraph.in_degree_centrality(), and DirectedGraph.out_degree_centrality()
  - pagerank now returns a collections.Counter
  - Optionally limit DependencyUtils.lexicalize_path() to specific token indices
- v3.0.1:
  - Fixed an off-by-one error for character offsets when creating mentions
- v3.0.0:
  - PageRank for dependency graphs via networkx
  - Find shortest paths in dependency graphs via networkx
  - Updated api to match v.3.0.0 of processors-server
  - Support for Odin @Mention.argname
  - json compatible with org.clulab.processors
- v2.9.7:
  - .nes support for IOB format
- v2.9.6:
  - Simplified path checks for jar
- v2.9.5:
  - Docstring for ProcessorsAPI
  - Updated Odin rules url for tests
  - Check version of jar against recommended version
  - Cleanup (kwargs usage, path resolution, etc.)
- v2.9.4:
  - jar is now downloaded when first used
- v2.9.3:
  - Pass hostname to command for starting server
  - Fixed an Odin test and added needed resources
- v2.9.2:

  - Added `.stop_server()` test
- v2.9.1:
  - Create `Mention` from `Mention.trigger` when `trigger` is not `None`
- v2.9:
  - `Mention.arguments` bug fix related to creating Mentions for each arg corresponding to each role
  - `Dependencies.incoming` and `Dependencies.outgoing` bug fixes
  - Implemented custom **eq** and **ne** for core data structures
- v2.8:
  - `Dependencies` bug fix related to initializing from `json`
  - Added `keep_alive` boolean parameter to `ProcessorsAPI` constructor to provide a way to keep the server running when instance goes out of scope
  - Compatibility fixes for 2.x
  - Updated api to match v.2.7 of `processors-server`
    - handle pre-segmented text (preserve provided sentence segmentation in `.annotate` and `.sentiment.corenlp.score_segmented_text` calls)
- v2.7:
  - Added `Mention` class and support for rule-based information extraction with Odin
  - Updated api to match v.2.5 of `processors-server`
- v2.6:
  - Added interface to CoreNLP's tree-based sentiment analysis
  - Rewrote `json` serialization and loading to mirror changes in `processors-server` v2.2
- v2.4:
  - Added support for `json` serialization
- v2.1:
  - Added interface to `BioNLPProcessor`
  - Download latest `processors-server.jar` as part of installation
- v1.0:
  - Basic functionality (interface to `FastNLPProcessor`)

CHAPTER 14

What is it?

`py-processors` is a Python wrapper for the CLU Lab's `processors` NLP library. `py-processors` relies on `processors-server`.

Though compatible* with Python 2.x, this library was developed with 3.x in mind.

CHAPTER 15

Requirements

The server component can be run either via `docker` or directly with `java`.

Option 1

- `docker` and the `myedibleenso/processors-server` container

Option 2

- Java 8
- `processor-server` (v3.1.0)
  - this dependency will be retrieved automatically during installation
- At least 2GB of RAM free for the server (I recommend 3GB+)

CHAPTER 16

Installation

`py-processors` can be installed via `pip`. The library also has a `jupyter` extras module which adds widgets/visualizations to `juypter` notebooks.

basic installation

```
pip install py-processors
```

basic + `jupyter` notebook widgets

```
pip install py-processors[jupyter]
```

bleeding edge

```
pip install git+https://github.com/myedibleenso/py-processors.git
```

CHAPTER 17

How to use it?

See *the walkthrough example*

Index

Symbols

A

B

C

D

E

F

G

H

I