# Webstruct Documentation

***Release 0.5***

**Scrapinghub Inc.**

**May 10, 2017**

# Contents

Webstruct is a library for creating statistical NER systems that work on HTML data, i.e. a library for building tools that extract named entities (addresses, organization names, open hours, etc) from webpages.

Contents:

CHAPTER 1

# Webstruct

## Overview

To develop a statistical NER system the following steps are needed:

1. define what entities you are interested in;
2. get some training/testing data (annotated webpages);
3. define what features should be extracted;
4. develop a statistical model that uses these features to produce the output.

Webstruct can:

- read annotated HTML data produced by WebAnnotator or GATE;
- transform HTML trees into "HTML tokens", i.e. text tokens with information about their position in HTML tree preserved;
- extract various features from these HTML tokens, including text-based features, html-based features and gazetteer-based features (GeoNames support is built-in);
- convert these features to the format sequence labelling toolkits accept;
- use Wapiti or CRFSuite CRF toolkit for entity extraction (helpers for other toolkits like seqlearn are planned);
- group extracted entites using an unsupervised algorithm;
- embed annotation results back into HTML files (using WebAnnotator format), allowing to view them in a web browser and fix using visual tools.

Unlike most NER systems, webstruct works on HTML data, not only on text data. This allows to define features that use HTML structure, and also to embed annotation results back into HTML.

# Installation

Webstruct requires Python 2.7 or Python 3.3+.

To install Webstruct, use pip:

```
$ pip install webstruct
```

Webstruct depends on the following Python packages:

- lxml for parsing HTML;
- scikit-learn >= 0.14.

Note that these dependencies are not installed automatically when `pip install webstruct` is executed.

There are also some optional dependencies:

- Webstruct has support for Wapiti sequence labelling toolkit; you'll need both the `wapiti` binary and python-wapiti wrapper (from github master) for the tutorial.
- For training data annotation you may use WebAnnotator Firefox extension.
- Code for preparing GeoNames gazetteers uses marisa-trie and pandas.

CHAPTER 2

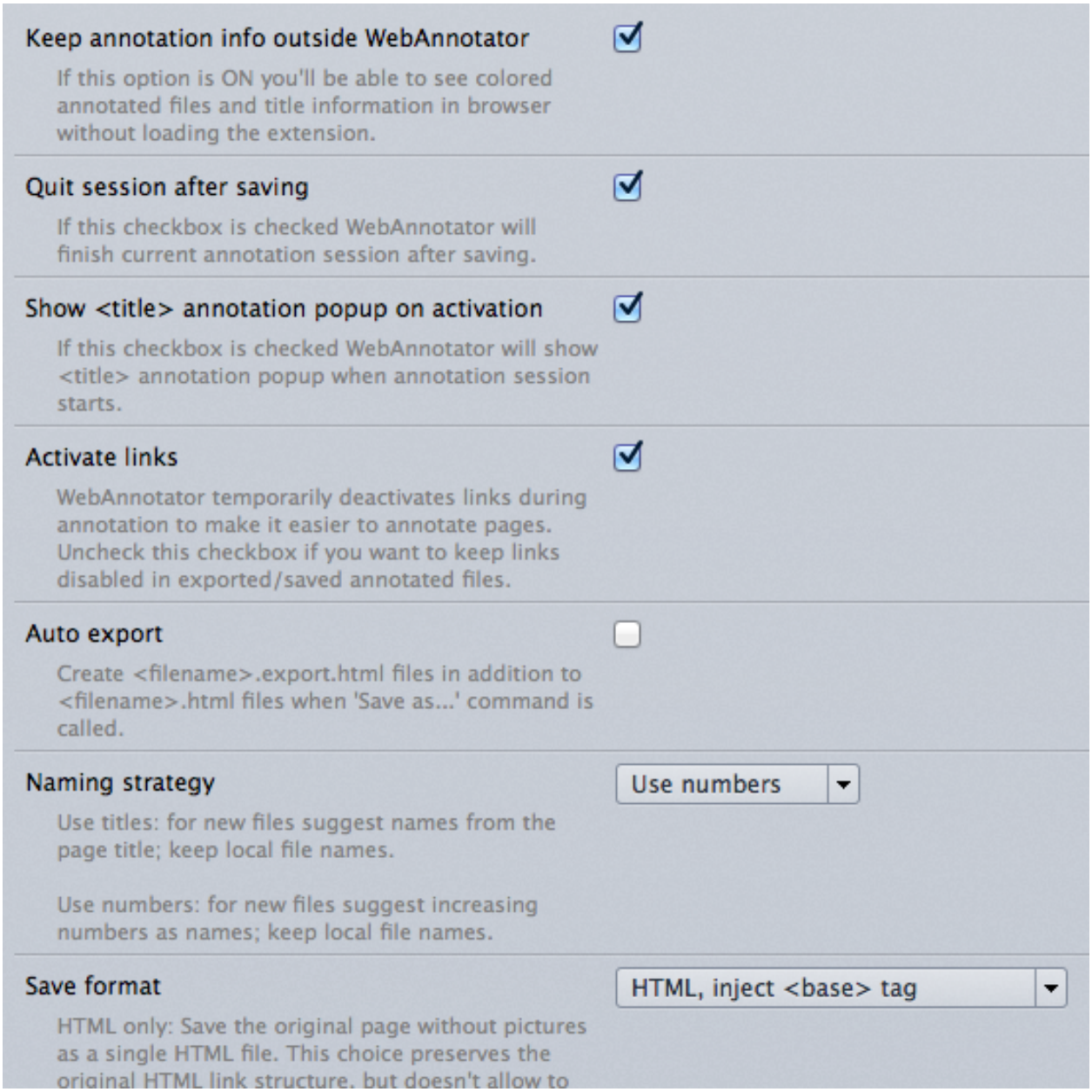
# Tutorial

This tutorial assumes you are familiar with machine learning.

## Get annotated data

First, you need the training/development data. We suggest to use WebAnnotator Firefox extension to annotate HTML pages.

Recommended WebAnnotator options:




Pro tip - enable WebAnnotator toolbar buttons:

Follow WebAnnotator manual to define named entities and annotate some web pages (nested WebAnnotator entities are not supported). Use "Save as.." menu item or "Save as" toolbar button to save the results; don't use "Export as".

After that you can load annotated webpages as lxml trees:

```
import webstruct
trees = webstruct.load_trees("train/*.html", webstruct.WebAnnotatorLoader())
```

See *HTML Loaders* for more info. GATE annotation format is also supported.

## From HTML to Tokens

To convert HTML trees to a format suitable for sequence prediction algorithm (like CRF, MEMM or Structured Perceptron) the following approach is used:

1. Text is extracted from HTML and split into tokens.
2. For each token a special *HtmlToken* instance is created. It contains information not only about the text token itself, but also about its position in HTML tree.

A single HTML page corresponds to a single input sequence (a list of HtmlTokens). For training/testing data (where webpages are already annotated) there is also a list of labels for each webpage, a label per HtmlToken.

To transform HTML trees into labels and HTML tokens use *HtmlTokenizer*.

```
html_tokenizer = webstruct.HtmlTokenizer()
X, y = html_tokenizer.tokenize(trees)
```

Input trees should be loaded by one of the WebStruct loaders. For consistency, for each tree (even if it is loaded from raw unannotated html) *HtmlTokenizer* extracts two arrays: a list of *HtmlToken* instances and a list of tags encoded using IOB2 encoding (also known as BIO encoding). So in our example `X` is a list of lists of *HtmlToken* instances, and `y` is a list of lists of strings.

## Feature Extraction

For supervised machine learning algorithms to work we need to extract features.

In WebStruct feature vectors are Python dicts `{"feature_name": "feature_value"}`; a dict is computed for each HTML token. How to convert these dicts into representation required by a sequence labelling toolkit depends on a toolkit used; we will cover that later.

To compute feature dicts we'll use *HtmlFeatureExtractor*.

First, define your feature functions. A feature function should take an *HtmlToken* instance and return a feature dict; feature dicts from individual feature functions will be merged into the final feature dict for a token. Feature functions can ask questions about token itself, its neighbours (in the same HTML element), its position in HTML.

---

**Note:** WebStruct supports another kind of feature functions that work on multiple tokens; we don't cover them in this tutorial.

---

There are predefined feature functions in *webstruct.features*, but for this tutorial let's create some functions ourselves:

```
def token_identity(html_token):
    return {'token': html_token.token}

def token_isupper(html_token):
    return {'isupper': html_token.token.isupper()}

def parent_tag(html_token):
    return {'parent_tag': html_token.parent.tag}

def border_at_left(html_token):
    return {'border_at_left': html_token.index == 0}
```

Next, create *HtmlFeatureExtractor*:

```
feature_extractor = HtmlFeatureExtractor(
    token_features = [
        token_identity,
        token_isupper,
        parent_tag,
        border_at_left
    ]
)
```

and use it to extract feature dicts:

```
features = feature_extractor.fit_transform(X)
```

See *Feature Extraction* for more info about HTML tokenization and feature extraction.

## Using a Sequence Labelling Toolkit

WebStruct doesn't provide a CRF or Structured Perceptron implementation; learning and prediction is supposed to be handled by an external sequence labelling toolkit like Wapiti, CRFSuite or seqlearn.

Once feature dicts are extracted from HTML you should convert them to a format required by your sequence labelling tooklit and use this toolkit to train a model and do the prediction. For example, you may use DictVectorizer from scikit-learn to convert feature dicts into seqlearn input format.

WebStruct provides some helpers for Wapiti sequence labelling toolkit. To use Wapiti with WebStruct, you need

- **for training:** wapiti C++ library itself, including `wapiti` command-line utility (python-wapiti wrapper is not necessary);
- **for prediction:** python-wapiti wrapper, github version (C++ library is not necessary).

We'll use Wapiti in this tutorial.

### Defining a Model

Basic way to define CRF model is the following:

```
model = webstruct.create_wapiti_pipeline('mymodel.wapiti',
    token_features = [token_identity, token_isupper, parent_tag, border_at_left],
    train_args = '--algo l-bfgs --maxiter 50 --compact'
)
```

First *create_wapiti_pipeline()* argument is a file name Wapiti model will be save to after training. `train_args` is a string or a list with arguments passed to wapiti; check Wapiti manual for available options.

Under the hood *create_wapiti_pipeline()* creates a `sklearn.pipeline.Pipeline` with an *HtmlFeatureExtractor* instance followed by *WapitiCRF* instance. The example above is just a shortcut for the following:

```
model = Pipeline([
    ('fe', HtmlFeatureExtractor(
        token_features = [
            token_identity,
            token_isupper,
            parent_tag,
```

```
            border_at_left,
        ]
    )),
    ('crf', WapitiCRF(
        'mymodel.wapiti',
        train_args = '--algo l-bfgs --maxiter 50 --compact',
    )),
])
```

## Extracting Features using Wapiti Templates

Wapiti has "templates" support which allows to define richer features from the basic features, and to specify what to do with labels. Template format is described in Wapiti manual; you may also check CRF++ docs to get the templates idea - CRF++ and Wapiti template formats are very similar.

WebStruct allows to use feature names instead of numbers in Wapiti templates.

Let's define a template that will make Wapiti use first-order transition features, plus `token` text values in a +-2 window near the current token.

```
feature_template = '''
# Label unigram & bigram
*

# Nearby token unigrams
uLL:%x[-2,token]
u-L:%x[-1,token]
u-R:%x[ 1,token]
uRR:%x[ 2,token]
'''
```

**Note:** *create_wapiti_pipeline()* (via *WapitiCRF*) by default adds all features for the current token to template. That's why we haven't defined them in our template, and that's why we were fine without using template at all. In our example additional auto-generated lines would be

```
ufeat:token=%x[0,token]
ufeat:isupper=%x[0,isupper]
ufeat:parent_tag=%x[0,parent_tag]
ufeat:border_at_left=%x[0,border_at_left]
```

To make Wapiti use this template, pass it as an argument to *create_wapiti_pipeline()* (or *WapitiCRF*, whatever you use):

```
model = webstruct.create_wapiti_pipeline('mymodel.wapiti',
    token_features = [token_identity, token_isupper, parent_tag, border_at_left],
    feature_template = feature_template,
    train_args = '--algo l-bfgs --maxiter 50 --compact'
)
```

## Training

To train a model use its `fit` method:

```
model.fit(X, y)
```

`X` and `y` are return values of *`HtmlTokenizer.tokenize()`* (a list of lists of *`HtmlToken`* instances and a list of lists of string IOB labels).

If you use *`WapitiCRF`* directly then train it using *`WapitiCRF.fit()`* method. It accepts 2 lists: a list of lists of feature dicts, and a list of lists of tags:

```
crf.fit(features, y)
```

## Named Entity Recognition

Once you got a trained model you can use it to extract entities from unseen (unannotated) webpages. First, get some binary HTML data:

```
>>> import urllib2
>>> html = urllib2.urlopen("http://scrapinghub.com/contact").read()
```

Then create a *`NER`* instance initialized with a trained model:

```
>>> ner = webstruct.NER(model)
```

The `model` must provide a `predict` method that extracts features from HTML tokens and predicts labels for these tokens. A pipeline created with *`create_wapiti_pipeline()`* function fits this definition.

Finally, use *`NER.extract()`* method to extract entities:

```
>>> ner.extract(html)
[('Scrapinghub', 'ORG'), ..., ('Iturriaga 3429 ap. 1', 'STREET'), ...]
```

Generally, the steps are:

1. Load data using *`HtmlLoader`* loader. If a custom HTML cleaner was used for loading training data make sure to apply it here as well.
2. Use the same `html_tokenizer` as used for training to extract HTML tokens from loaded trees. All labels would be "O" when using *`HtmlLoader`* loader - `y` can be discarded.
3. Use the same `feature_extractor` as used for training to extract features.
4. Run `your_crf.predict()` method (e.g. *`WapitiCRF.predict()`*) on features extracted in (3) to get the prediction - a list of IOB2-encoded tags for each input document.
5. Build entities from input tokens based on predicted tags (check *`IobEncoder.group()`* and *`smart_join()`*).
6. Split entities into groups (optional). One way to do it is to use *`webstruct.grouping`*.

*`NER`* helper class combines HTML loading, HTML tokenization, feature extraction, CRF model, entity building and grouping.

## Entity Grouping

Detecting entities on their own is not always enough; in many cases what is wanted is to find the relationship between them. For example, "**street_name/STREET city_name/CITY zipcode_number/ZIPCODE** form an address", or "**phone/TEL** is a phone of **person/PER**".

The first approximation is to say that all entities from a single webpage are related. For example, if we have extracted some **organizaion/ORG** and some **phone/TEL** from a single webpage we may assume that the phone is a contact phone of the organization.

Sometimes there are several "entity groups" on a webpage. If a page contains contact phones of several persons or several business locations it is better to split all entities into groups of related entities - "person name + his/her phone(s)" or "address".

WebStruct provides an *unsupervised algorithm* for extracting such entity groups. Algorithm prefers to build large groups without entities of duplicate types; if a split is needed algorithm tries to split at points where distance between entities is larger.

Use *NER.extract_groups()* to extract groups of entities:

```
>>> ner.extract_groups(html)
[[...], ... [('Iturriaga 3429 ap. 1', 'STREET'), ('Montevideo', 'CITY'), ...]]
```

Sometimes it is better to allow some entity types to appear multuple times in a group. For example, a person (PER entity) may have several contact phones and faxes (TEL and FAX entities) - we should penalize groups with multiple PERs, but multiple TELs and FAXes are fine. Use `dont_penalize` argument if you want to allow some entity types to appear multiple times in a group:

```
ner.extract_groups(html, dont_penalize={'TEL', 'FAX'})
```

The simple algorithm WebStruct provides is by no means a general solution to relation detection, but give it a try - maybe it is enough for your task.

## Model Development

To develop the model you need to choose the learning algorithm, features, hyperparameters, etc. To do that you need scoring metrics, cross-validation utilities and tools for debugging what classifier learned. WebStruct helps in the following way:

1. Pipeline created by *create_wapiti_pipeline()* is compatible with cross-validation and grid search utilities from scikit-learn; use them to select model parameters and check the quality.

   One limitation of *create_wapiti_pipeline()* is that `n_jobs` in scikit-learn functions and classes should be 1, but other than that WebStruct objects should work fine with scikit-learn. Just keep in mind that for WebStruct an "observation" is a document, not an individual token, and a "label" is a sequence of labels for a document, not an individual IOB tag.

2. There is *webstruct.metrics* module with a couple of metrics useful for sequence classification. Currently they require seqlearn to be installed.

To debug what CRF learned you should use methods specific to a labelling toolkit. With Wapiti it would be `wapiti dump` console command and some UNIX utilities. For example, if we've saved our model to `mymodel.wapiti` file, and we want to check top positive features for `CITY` entity, we can execute the following in UNIX shell:

```
$ wapiti dump mymodel.wapiti | sort -nr -k4 | grep CITY | head -n 8
```

and get an output similar to this:

```
* Load model
* Dump model
*   B-CITY  I-CITY  2.74057
*   B-CITY  B-STATE 2.33235
*   I-STREET        B-CITY  1.98106
```

```
*   I-CITY  B-STATE 1.71408
u--L:street #       B-CITY  1.34199
u--L:west   #       I-CITY  1.32428
u--L:in     #       B-CITY  1.24937
u--L:-      #       B-CITY  1.11139
```

CHAPTER 3

# Reference

## HTML Loaders

Webstruct supports WebAnnotator and GATE annotation formats out of box; WebAnnotator is recommended.

Both GATE and WebAnnotator embed annotations into HTML using special tags: GATE uses custom tags like `<ORG>` while WebAnnotator uses tags like `<span wa-type="ORG">`.

*webstruct.loaders* classes convert GATE and WebAnnotator tags into __START_TAGNAME__ and __END_TAGNAME__ tokens, clean the HTML and return the result as a tree parsed by lxml:

```
>>> from webstruct import WebAnnotatorLoader
>>> loader = WebAnnotatorLoader()
>>> loader.load('0.html')
<Element html at ...>
```

Such trees can be processed with utilities from *webstruct.feature_extraction*.

### API

**class** webstruct.loaders.**WebAnnotatorLoader**(*encoding=None*, *cleaner=None*, *known_entities=None*)

Bases: *webstruct.loaders.HtmlLoader*

Class for loading HTML annotated using WebAnnotator.

---

**Note:** Use WebAnnotator's "save format", not "export format".

---

**load**(*filename*)

**loadbytes**(*data*)

**class** webstruct.loaders.**GateLoader**(*encoding=None*, *cleaner=None*, *known_entities=None*)

Bases: *webstruct.loaders.HtmlLoader*

Class for loading HTML annotated using GATE

```
>>> import lxml.html
>>> from webstruct import GateLoader
```

```
>>> loader = GateLoader(known_entities={'ORG', 'CITY'})
>>> html = b"<html><body><p><ORG>Scrapinghub</ORG> has an <b>office</b> in <CITY>
↪Montevideo</CITY></p></body></html>"
>>> tree = loader.loadbytes(html)
>>> lxml.html.tostring(tree).decode()
'<html><body><p> __START_ORG__ Scrapinghub __END_ORG__  has an <b>office</b> in  _
↪_START_CITY__ Montevideo __END_CITY__ </p></body></html>'
```

Note that you must specify known_entities when creating GateLoader. It should contain all entities which are present in data, even if you want to use only a subset of them for training. Use arguments of *HtmlLoader* to train a tagger which uses a subset of labels.

**load**(*filename*)

**loadbytes**(*data*)

**class** webstruct.loaders.**HtmlLoader**(*encoding=None*, *cleaner=None*)
Bases: object

Class for loading unannotated HTML files.

**load**(*filename*)

**loadbytes**(*data*)

webstruct.loaders.**load_trees**(*pattern*, *loader*, *verbose=False*)
Load HTML data using loader `loader` from all files matched by `pattern` glob pattern.

Example:

```
>>> trees = load_trees('path/*.html', HtmlLoader())
```

# Feature Extraction

## HTML Tokenization

*webstruct.html_tokenizer* contains *HtmlTokenizer* class which allows to extract text from a web page and tokenize it, preserving information about token position in HTML tree (token + its tree position = *HtmlToken*). *HtmlTokenizer* also allows to extract annotations from the tree (if present) and split them from regular text/tokens.

**class** webstruct.html_tokenizer.**HtmlToken**
HTML token info.

Attributes:

- `index` is a token index (in the `tokens` list)
- `tokens` is a list of all tokens in current html block
- `elem` is the current html block (as lxml's Element) - most likely you want `parent` instead of it
- `is_tail` flag indicates that token belongs to element tail

Computed properties:

- `token` is the current token (as text);
- `parent` is token's parent HTML element (as lxml's Element);
- `root` is an ElementTree this token belongs to.

**class** `webstruct.html_tokenizer.`**`HtmlTokenizer`**(*tagset=None*, *sequence_encoder=None*, *text_tokenize_func=None*, *kill_html_tags=None*, *replace_html_tags=None*, *ignore_html_tags=None*)

Class for converting HTML trees (returned by one of the *webstruct.loaders*) into lists of *HtmlToken* instances and associated tags. Also, it can do the reverse conversion.

Use *tokenize_single()* to convert a single tree and *tokenize()* to convert multiple trees.

Use *detokenize_single()* to get an annotated tree out of a list of *HtmlToken* instances and a list of tags.

**Parameters** **tagset** : set, optional

A set of entity types to keep. If not passed, all entity types are kept. Use this argument to discard some entity types from training data.

**sequence_encoder** : object, optional

Sequence encoder object. If not passed, *IobEncoder* instance is created.

**text_toknize_func** : callable, optional

Function used for tokenizing text inside HTML elements. By default, *HtmlTokenizer* uses *webstruct.text_tokenizers.tokenize()*.

**kill_html_tags: set, optional**

A set of HTML tags which should be removed. Contents inside removed tags is not removed. See *webstruct.utils.kill_html_tags()*

**replace_html_tags: dict, optional**

A mapping `{'old_tagname': 'new_tagname'}`. It defines how tags should be renamed. See *webstruct.utils.replace_html_tags()*

**ignore_html_tags: set, optional**

A set of HTML tags which won't produce *HtmlToken* instances, but will be kept in a tree. Default is `{'script', 'style'}`.

**`detokenize_single`**(*html_tokens*, *tags*)

Build annotated `lxml.etree.ElementTree` from `html_tokens` (a list of *HtmlToken* instances) and `tags` (a list of their tags).

Annotations are encoded as `__START_TAG__` and `__END_TAG__` text tokens (this is the format *webstruct.loaders* use).

**`tokenize`**(*trees*)

**`tokenize_single`**(*tree*)

Return two lists:

- a list a list of HtmlToken tokens;
- a list of associated tags.

For unannotated HTML all tags will be "O" - they may be ignored.

Example:

```
>>> from webstruct import GateLoader, HtmlTokenizer
>>> loader = GateLoader(known_entities={'PER'})
>>> html_tokenizer = HtmlTokenizer(replace_html_tags={'b': 'strong'})
>>> tree = loader.loadbytes(b"<p>hello, <PER>John <b>Doe</b></PER> <br> <PER>
↪Mary</PER> said</p>")
>>> html_tokens, tags = html_tokenizer.tokenize_single(tree)
>>> html_tokens
[HtmlToken(token='hello', parent=<Element p at ...>, index=0), HtmlToken...]
>>> tags
['O', 'B-PER', 'I-PER', 'B-PER', 'O']
>>> for tok, iob_tag in zip(html_tokens, tags):
...     print("%5s" % iob_tag, tok.token, tok.elem.tag, tok.parent.tag)
    O hello p p
B-PER John p p
I-PER Doe strong strong
B-PER Mary br p
    O said br p
```

For HTML without text it returns empty lists:

```
>>> html_tokenizer.tokenize_single(loader.loadbytes(b'<p></p>'))
([], [])
```

## Feature Extraction Utilitites

*webstruct.feature_extraction* contains classes that help with:

- converting HTML pages into lists of feature dicts and
- extracting annotations.

Usually, the approach is the following:

1. Convert a web page to a list of *HtmlToken* instances and a list of annotation tags (if present). *HtmlTokenizer* is used for that.
2. Run a number of "token feature functions" that return bits of information about each token: token text, token shape (uppercased/lowercased/...), whether token is in <a> HTML element, etc. For each token information is combined into a single feature dictionary.

   Use *HtmlFeatureExtractor* at this stage. There is a number of predefined token feature functions in *webstruct.features*.
3. Run a number of "global feature functions" that can modify token feature dicts inplace (insert new features, change, remove them) using "global" information - information about all other tokens in a document and their existing token-level feature dicts. Global feature functions are applied sequentially: subsequent global feature functions get feature dicts updated by previous feature functions.

   This is also done by *HtmlFeatureExtractor*.

   `LongestMatchGlobalFeature` can be used to create features that capture multi-token patterns. Some predefined global feature functions can be found in `webstruct.gazetteers`.

**class** `webstruct.feature_extraction.`**`HtmlFeatureExtractor`**(*token_features*, *global_features=None*, *min_df=1*)

This class extracts features from lists of *HtmlToken* instances (*HtmlTokenizer* can be used to create such lists).

*fit()* / *transform()* / *fit_transform()* interface may look familiar to you if you ever used scikit-learn: *HtmlFeatureExtractor* implements sklearn's Transformer interface. But there is one twist: usually for sequence labelling tasks the whole sequences are considered observations. So in our case a single observation is a tokenized document (a list of tokens), not an individual token: *fit()* / *transform()* / *fit_transform()* methods accept lists of documents (lists of lists of tokens), and return lists of documents' feature dicts (lists of lists of feature dicts).

**Parameters** **token_features** : list of callables

List of "token" feature functions. Each function accepts a single `html_token` parameter and returns a dictionary wich maps feature names to feature values. Dicts from all token feature functions are merged by HtmlFeatureExtractor. Example token feature (it just returns token text):

```
>>> def current_token(html_token):
...     return {'tok': html_token.token}
```

*webstruct.features* module provides some predefined feature functions, e.g. *parent_tag* which returns token's parent tag.

Example:

```
>>> from webstruct import GateLoader, HtmlTokenizer,␣
↪HtmlFeatureExtractor
>>> from webstruct.features import parent_tag

>>> loader = GateLoader(known_entities={'PER'})
>>> html_tokenizer = HtmlTokenizer()
>>> feature_extractor = HtmlFeatureExtractor(token_
↪features=[parent_tag])

>>> tree = loader.loadbytes(b"<p>hello, <PER>John <b>Doe</b></
↪PER> <br> <PER>Mary</PER> said</p>")
>>> html_tokens, tags = html_tokenizer.tokenize_single(tree)
>>> feature_dicts = feature_extractor.transform_single(html_
↪tokens)
>>> for token, tag, feat in zip(html_tokens, tags, feature_
↪dicts):
...     print("%s %s %s" % (token.token, tag, feat))
hello O {'parent_tag': 'p'}
John B-PER {'parent_tag': 'p'}
Doe I-PER {'parent_tag': 'b'}
Mary B-PER {'parent_tag': 'p'}
said O {'parent_tag': 'p'}
```

**global_features** : list of callables, optional

List of "global" feature functions. Each "global" feature function should accept a single argument - a list of `(html_token, feature_dict)` tuples. This list contains all tokens from the document and features extracted by previous feature functions.

"Global" feature functions are applied after "token" feature functions in the order they are passed.

They should change feature dicts `feature_dict` inplace.

**min_df** : integer or Mapping, optional

Feature values that have a document frequency strictly lower than the given threshold are removed. If `min_df` is integer, its value is used as threshold.

TODO: if `min_df` is a dictionary, it should map feature names to thresholds.

**fit**(*html_token_lists*, *y=None*)

**fit_transform**(*html_token_lists*, *y=None*, ***fit_params*)

**transform**(*html_token_lists*)

**transform_single**(*html_tokens*)

## Predefined Feature Functions

webstruct.features.token_features.**bias**(*html_token*)

webstruct.features.token_features.**token_identity**(*html_token*)

webstruct.features.token_features.**token_lower**(*html_token*)

webstruct.features.token_features.**token_shape**(*html_token*)

webstruct.features.token_features.**token_endswith_dot**(*html_token*)

webstruct.features.token_features.**token_endswith_colon**(*html_token*)

webstruct.features.token_features.**token_has_copyright**(*html_token*)

webstruct.features.token_features.**number_pattern**(*html_token*)

webstruct.features.token_features.**prefixes_and_suffixes**(*html_token*)

**class** webstruct.features.token_features.**PrefixFeatures**(*lenghts=(2, 3, 4)*, *featname='prefix'*, *lower=True*)

**class** webstruct.features.token_features.**SuffixFeatures**(*lenghts=(2, 3, 4)*, *featname='suffix'*, *lower=True*)

webstruct.features.data_features.**looks_like_year**(*html_token*)

webstruct.features.data_features.**looks_like_month**(*html_token*)

webstruct.features.data_features.**looks_like_time**(*html_token*)

webstruct.features.data_features.**looks_like_weekday**(*html_token*)

webstruct.features.data_features.**looks_like_email**(*html_token*)

webstruct.features.data_features.**looks_like_street_part**(*html_token*)

webstruct.features.data_features.**looks_like_range**(*html_token*)

webstruct.features.block_features.**parent_tag**(*html_token*)

**class** webstruct.features.block_features.**InsideTag**(*tagname*)

webstruct.features.block_features.**borders**(*html_token*)

webstruct.features.block_features.**block_length**(*html_token*)

**class** webstruct.features.global_features.**DAWGGlobalFeature**(*filename*, *featname*, *format=None*)

Global feature that matches longest entities from a lexicon stored either in a `dawg.CompletionDAWG` (if `format` is None) or in a `dawg.RecordDAWG` (if `format` is not None).

**class** webstruct.features.global_features.**LongestMatchGlobalFeature**(*lookup_data*, *featname*)

**process_range**(*doc*, *start*, *end*, *matched_text*)

**class** webstruct.features.global_features.**Pattern**(**lookups*, ***kwargs*)
Global feature that combines local features.

### Gazetteer Support

**class** webstruct.gazetteers.features.**MarisaGeonamesGlobalFeature**(*filename*, *featname*, *format=None*)
Global feature that matches longest entities from a lexicon extracted from geonames.org and stored in a MARISA Trie.

webstruct.gazetteers.geonames.**read_geonames**(*filename*)
Parse geonames file to a pandas.DataFrame. File may be downloaded from http://download.geonames.org/export/dump/; it should be unzipped and in a "geonames table" format.

webstruct.gazetteers.geonames.**read_geonames_zipped**(*zip_filename*, *geonames_filename=None*)
Parse zipped geonames file.

webstruct.gazetteers.geonames.**to_dawg**(*df*, *columns=None*, *format=None*)
Encode `pandas.DataFrame` with GeoNames data (loaded using *`read_geonames()`* and maybe filtered in some way) to `dawg.DAWG` or `dawg.RecordDAWG`. `dawg.DAWG` is created if `columns` and `format` are both None.

webstruct.gazetteers.geonames.**to_marisa**(*df*, *columns=['country_code', 'feature_class', 'feature_code', 'admin1_code', 'admin2_code']*, *format='2s 1s 5s 2s 3s'*)
Encode `pandas.DataFrame` with GeoNames data (loaded using *`read_geonames()`* and maybe filtered in some way) to a `marisa.RecordTrie`.

## Model Creation Helpers

*`webstruct.model`* contains convetional wrappers for creating NER models.

**class** webstruct.model.**NER**(*model*, *loader=None*, *html_tokenizer=None*, *entity_colors=None*)
Class for extracting named entities from HTML.

Initialize it with a trained `model`. `model` must have `predict` method that accepts lists of *`HtmlToken`* sequences and returns lists of predicted IOB2 tags. *`create_wapiti_pipeline()`* function returns such model.

**extract**(*bytes_data*)
Extract named entities from binary HTML data `bytes_data`. Return a list of `(entity_text, entity_type)` tuples.

**extract_from_url**(*url*)
A convenience wrapper for *`extract()`* method that downloads input data from a remote URL.

**extract_raw**(*bytes_data*)
Extract named entities from binary HTML data `bytes_data`. Return a list of `(html_token, iob2_tag)` tuples.

**extract_groups**(*bytes_data*, *dont_penalize=None*)
Extract groups of named entities from binary HTML data `bytes_data`. Return a list of lists of `(entity_text, entity_type)` tuples.

Entites are grouped using algorithm from *webstruct.grouping*.

**extract_groups_from_url**(*url*, *dont_penalize=None*)
A convenience wrapper for *extract_groups()* method that downloads input data from a remote URL.

**build_entity**(*html_tokens*)
Join tokens to an entity. Return an entity, as text. By default this function uses *webstruct.utils.smart_join()*.

Override it to customize *extract()*, *extract_from_url()* and *extract_groups()* results. If this function returns empty string or None, entity is dropped.

**annotate**(*bytes_data*, *url=None*, *pretty_print=False*)
Return annotated HTML data in WebAnnotator format.

**annotate_url**(*url*, *pretty_print=False*)
Return annotated HTML data in WebAnnotator format; input is downloaded from `url`.

## Metrics

*webstruct.metrics* contains metric functions that can be used for model developmenton: on their own or as scoring functions for scikit-learn's cross-validation and model selection.

webstruct.metrics.**avg_bio_f1_score**(*y_true*, *y_pred*)
Macro-averaged F1 score of lists of BIO-encoded sequences `y_true` and `y_pred`.

A named entity in a sequence from `y_pred` is considered correct only if it is an exact match of the corresponding entity in the `y_true`.

It requires https://github.com/larsmans/seqlearn to work.

webstruct.metrics.**bio_classification_report**(*y_true*, *y_pred*)
Classification report for a list of BIO-encoded sequences. It computes token-level metrics and discards "O" labels.

webstruct.metrics.**bio_f_score**(*y_true*, *y_pred*)
F-score for BIO-tagging scheme, as used by CoNLL.

This F-score variant is used for evaluating named-entity recognition and related problems, where the goal is to predict segments of interest within sequences and mark these as a "B" (begin) tag followed by zero or more "I" (inside) tags. A true positive is then defined as a BI* segment in both y_true and y_pred, with false positives and false negatives defined similarly.

Support for tags schemes with classes (e.g. "B-NP") are limited: reported scores may be too high for inconsistent labelings.

**Parameters** **y_true** : array-like of strings, shape (n_samples,)

Ground truth labeling.

**y_pred** : array-like of strings, shape (n_samples,)

Sequence classifier's predictions.

**Returns** **f** : float

F-score.

## Entity Grouping

Often it is not enough to find all entities on a webpage. For example, one may want to extract separate "entity groups" with combined information about individual offices from a page that has contact details of several offices. An "entity group" may consist of the name of the office along with office address (street, city, zipcode) and contacts (phones, faxes) in this case. *webstruct.grouping* module provides a simple unsupervised algorithm to group extracted entities into clusters. It works this way:

1. Each HTML token is assigned a position (an integer number). Position increases with each token and when HTML element changes.
2. Distances between subsequent entities are calculated.
3. If a distance between 2 subsequent entities is greater than a certain threshold then new "cluster" is started.
4. Clusters are scored - longer clusters get larger scores, but clusters with several entities of the same type are penalized (unless user explicitly asked not to penalize this entity type). Total clustering score is calculated as a sum of scores of individual clusters.
5. Threshold value for the final clustering is selected to maximize total clustering score (4). Each input page gets its own threshold.

webstruct.grouping.**choose_best_clustering**(*html_tokens*, *tags*, *score_func=None*, *score_kwargs=None*)

Select a best way to split `html_tokens` and `tags` into clusters of named entities. Return `(threshold, score, clusters)` tuple.

`clusters` in the resulting tuple is a list of clusters; each cluster is a list of named entities: `(html_tokens, tag, distance)` tuples.

`html_tokens` and `tags` could be a result of *webstruct.model.NER.extract_raw()*.

If `score_func` is None, *choose_best_clustering()* uses *default_clustering_score()* to compute the score of a set of clusters under consideration (optimization objective). You can pass your own scoring function to change the heuristic used. Your function must have 2 positional parameters: `clusters` and `threshold` (and any number of keyword arguments) and return a score (number) which should be large if the clustering is good and small or negative if it is bad.

`score_kwargs` is a dict of keyword arguments passed to scoring function. For example, if you use default `score_func`, the goal is to group contact information, and you want to allow several phones (TEL) and faxes (FAX) in the same group, pass `score_kwargs={'dont_penalize': {'TEL', 'FAX'}}`.

webstruct.grouping.**default_clustering_score**(*clusters*, *threshold*, *dont_penalize=None*)

Heuristic scoring function for clusters:

- larger clusters get bigger scores;
- clusters that have multiple entities of the same tag are penalized (unless the tag is in `dont_penalize` set);
- total score is computed as a sum of scores of all clusters.

`dont_penalize` is a set of tags for which duplicates are not penalized. It is empty by default.

## Wapiti Helpers

*webstruct.wapiti* module provides utilities for easier creation of Wapiti models, templates and data files.

**class** webstruct.wapiti.**WapitiCRF**(*model_filename=None*, *train_args=None*, *feature_template='# Label unigrams and bigrams:n*n'*, *unigrams_scope='u'*, *tempdir=None*, *unlink_temp=True*, *verbose=True*, *feature_encoder=None*, *dev_size=0*)

Bases: *webstruct.base.BaseSequenceClassifier*

Class for training and applying Wapiti CRF models.

For training it relies on calling original Wapiti binary (via subprocess), so "wapiti" binary must be available if you need "fit" method.

Trained model is saved in an external file; its filename is a first parameter to constructor. This file is created and overwritten by *WapitiCRF.fit()*; it must exist for WapitiCRF.transform() to work.

For prediction WapitiCRF relies on python-wapiti library.

**WAPITI_CMD = 'wapiti'**
Command used to start wapiti

**fit**(*X*, *y*, *X_dev=None*, *y_dev=None*, *out_dev=None*)
Train a model.

**Parameters** **X** : list of lists of dicts

Feature dicts for several documents.

**y** : a list of lists of strings

Labels for several documents.

**X_dev** : (optional) list of lists of feature dicts

Data used for testing and as a stopping criteria.

**y_dev** : (optional) list of lists of labels

Labels corresponding to X_dev.

**out_dev** : (optional) string

Path to a file where tagged development data will be written.

**predict**(*X*)
Make a prediction.

**Parameters** **X** : list of lists

feature dicts

**Returns** **y** : list of lists

predicted labels

**run_wapiti**(*args*)
Run wapiti binary in a subprocess

**score**(*X*, *y*)
Macro-averaged F1 score of lists of BIO-encoded sequences y_true and y_pred.

A named entity in a sequence from y_pred is considered correct only if it is an exact match of the corresponding entity in the y_true.

**class** webstruct.wapiti.**WapitiFeatureEncoder**(*move_to_front=('token', )*)
Bases: BaseEstimator, TransformerMixin

Utility class for preparing Wapiti templates and converting sequences of dicts with features to the format Wapiti understands.

**fit** (*X*, *y=None*)
X should be a list of lists of dicts with features. It can be obtained, for example, using *HtmlFeatureExtractor*.

**prepare_template** (*template*)
Prepare Wapiti template by replacing feature names with feature column indices inside `%x[row,col]` macros. Indices are compatible with `WapitiFeatureEncoder.transform()` output.

```
>>> we = WapitiFeatureEncoder(['token', 'tag'])
>>> seq_features = [{'token': 'the', 'tag': 'DT'}, {'token': 'dog', 'tag': 'NN
↪'}]
>>> we.fit([seq_features])
WapitiFeatureEncoder(move_to_front=('token', 'tag'))
>>> we.prepare_template('*:Pos-1 L=%x[-1, tag]\n*:Suf-2 X=%m[ 0,token,".?.?$"]
↪')
'*:Pos-1 L=%x[-1,1]\n*:Suf-2 X=%m[0,0,".?.?$"]'
```

Check these links for more info about template format:

- http://wapiti.limsi.fr/manual.html
- http://crfpp.googlecode.com/svn/trunk/doc/index.html#templ

**transform_single** (*feature_dicts*)
Transform a sequence of dicts `feature_dicts` to a list of Wapiti data file lines.

**unigram_features_template** (*scope='*'*)
Return Wapiti template with unigram features for each of known features.

```
>>> we = WapitiFeatureEncoder(['token', 'tag'])
>>> seq_features = [{'token': 'the', 'tag': 'DT'}, {'token': 'dog', 'tag': 'NN
↪'}]
>>> we.fit([seq_features])
WapitiFeatureEncoder(move_to_front=('token', 'tag'))
>>> print(we.unigram_features_template())

# Unigrams for all custom features
*feat:token=%x[0,0]
*feat:tag=%x[0,1]

>>> print(we.unigram_features_template('u'))

# Unigrams for all custom features
ufeat:token=%x[0,0]
ufeat:tag=%x[0,1]
```

webstruct.wapiti.**create_wapiti_pipeline** (*model_filename=None*, *token_features=None*, *global_features=None*, *min_df=1*, ***crf_kwargs*)
Create a scikit-learn Pipeline for HTML tagging using Wapiti. This pipeline expects data produced by *HtmlTokenizer* as an input and produces sequences of IOB2 tags as output.

Example:

```
import webstruct
from webstruct.features import EXAMPLE_TOKEN_FEATURES

# load train data
html_tokenizer = webstruct.HtmlTokenizer()
train_trees = webstruct.load_trees(
    "train/*.html",
```

```
    webstruct.WebAnnotatorLoader()
)
X_train, y_train = html_tokenizer.tokenize(train_trees)

# train
model = webstruct.create_wapiti_pipeline(
    model_filename = 'model.wapiti',
    token_features = EXAMPLE_TOKEN_FEATURES,
    train_args = '--algo l-bfgs --maxiter 50 --nthread 8 --jobsize 1 --stopwin 10
↪',
)
model.fit(X_train, y_train)

# load test data
test_trees = webstruct.load_trees(
    "test/*.html",
    webstruct.WebAnnotatorLoader()
)
X_test, y_test = html_tokenizer.tokenize(test_trees)

# do a prediction
y_pred = model.predict(X_test)
```

webstruct.wapiti.**prepare_wapiti_template**(*template*, *vocabulary*)

Prepare Wapiti template by replacing feature names with feature column indices inside `%x[row,col]` macros:

```
>>> vocab = {'token': 0, 'tag': 1}
>>> prepare_wapiti_template('*:Pos-1 L=%x[-1, tag]\n*:Suf-2 X=%m[ 0,token,".?.?$"]
↪', vocab)
'*:Pos-1 L=%x[-1,1]\n*:Suf-2 X=%m[0,0,".?.?$"]'
```

It understands which lines are comments:

```
>>> prepare_wapiti_template('*:Pos-1 L=%x[-1, tag]\n# *:Suf-2 X=%m[ 0,token,".?.?$
↪"]', vocab)
'*:Pos-1 L=%x[-1,1]\n# *:Suf-2 X=%m[ 0,token,".?.?$"]'
```

Check these links for more info about template format:

- http://wapiti.limsi.fr/manual.html
- http://crfpp.googlecode.com/svn/trunk/doc/index.html#templ

## CRFsuite Helpers

CRFsuite backend for webstruct based on python-crfsuite and sklearn-crfsuite.

**class** webstruct.crfsuite.**CRFsuitePipeline**(*fe*, *crf*)

Bases: `Pipeline`

A pipeline for HTML tagging using CRFsuite. It combines a feature extractor and a CRF; they are available as `fe` and `crf` attributes for easier access.

In addition to that, this class adds support for X_dev/y_dev arguments for `fit()` and `fit_transform()` methods - they work as expected, being transformed using feature extractor.

webstruct.crfsuite.**create_crfsuite_pipeline**(*token_features=None*, *global_features=None*, *min_df=1*, ***crf_kwargs*)

Create *CRFsuitePipeline* for HTML tagging using CRFsuite. This pipeline expects data produced by *HtmlTokenizer* as an input and produces sequences of IOB2 tags as output.

Example:

```
import webstruct
from webstruct.features import EXAMPLE_TOKEN_FEATURES

# load train data
html_tokenizer = webstruct.HtmlTokenizer()
train_trees = webstruct.load_trees(
    "train/*.html",
    webstruct.WebAnnotatorLoader()
)
X_train, y_train = html_tokenizer.tokenize(train_trees)

# train
model = webstruct.create_crfsuite_pipeline(
    token_features = EXAMPLE_TOKEN_FEATURES,
)
model.fit(X_train, y_train)

# load test data
test_trees = webstruct.load_trees(
    "test/*.html",
    webstruct.WebAnnotatorLoader()
)
X_test, y_test = html_tokenizer.tokenize(test_trees)

# do a prediction
y_pred = model.predict(X_test)
```

## WebAnnotator Utilities

*webstruct.webannotator* provides functions for working with HTML pages annotated with WebAnnotator Firefox extension.

webstruct.webannotator.**to_webannotator**(*tree*, *entity_colors=None*, *url=None*)

Convert a tree loaded by one of WebStruct loaders to WebAnnotator format.

If you want a predictable colors assignment use `entity_colors` argument; it should be a mapping `{'entity_name': (fg, bg, entity_idx)}`; entity names should be lowercased. You can use *EntityColors* to generate this mapping automatically:

```
>>> from webstruct.webannotator import EntityColors, to_webannotator
>>> # trees = ...
>>> entity_colors = EntityColors()
>>> wa_trees = [to_webannotator(tree, entity_colors) for tree in trees]
```

**class** webstruct.webannotator.**EntityColors**(***kwargs*)

`{"entity_name": ("fg_color", "bg_color", entity_index)}` mapping that generates entries for new entities on first access.

**classmethod from_htmlbytes**(*html_bytes*, *encoding=None*)

**classmethod from_htmlfile** (*path*, *encoding=None*)
Load the color mapping from WebAnnotator-annotated HTML file

## BaseSequenceClassifier

**class** webstruct.base.**BaseSequenceClassifier**
Bases: BaseEstimator, TransformerMixin

**score** (*X*, *y*)
Macro-averaged F1 score of lists of BIO-encoded sequences y_true and y_pred.

A named entity in a sequence from y_pred is considered correct only if it is an exact match of the corresponding entity in the y_true.

## Miscellaneous

### Utils

**class** webstruct.utils.**BestMatch** (*known*)
Bases: object

Class for finding best non-overlapping matches in a sequence of tokens. Override *get_sorted_ranges()* method to define which results are best.

**find_ranges** (*tokens*)

**get_sorted_ranges** (*ranges*, *tokens*)

**class** webstruct.utils.**LongestMatch** (*known*)
Bases: *webstruct.utils.BestMatch*

Class for finding longest non-overlapping matches in a sequence of tokens.

```
>>> known = {'North Las', 'North Las Vegas', 'North Pole', 'Vegas USA', 'Las Vegas
↪', 'USA', "Toronto"}
>>> lm = LongestMatch(known)
>>> lm.max_length
3
>>> tokens = ["Toronto", "to", "North", "Las", "Vegas", "USA"]
>>> for start, end, matched_text in lm.find_ranges(tokens):
...     print(start, end, tokens[start:end], matched_text)
0 1 ['Toronto'] Toronto
2 5 ['North', 'Las', 'Vegas'] North Las Vegas
5 6 ['USA'] USA
```

*LongestMatch* also accepts a dict instead of a list/set for a known argument. In this case dict keys are used:

```
>>> lm = LongestMatch({'North': 'direction', 'North Las Vegas': 'location'})
>>> tokens = ["Toronto", "to", "North", "Las", "Vegas", "USA"]
>>> for start, end, matched_text in lm.find_ranges(tokens):
...     print(start, end, tokens[start:end], matched_text)
2 5 ['North', 'Las', 'Vegas'] North Las Vegas
```

**get_sorted_ranges** (*ranges*, *tokens*)

webstruct.utils.**flatten**(*sequence*) → list
Return a single, flat list which contains all elements retrieved from the sequence and all recursively contained sub-sequences (iterables).

Examples:

```
>>> [1, 2, [3,4], (5,6)]
[1, 2, [3, 4], (5, 6)]
>>> flatten([[[1,2,3], (42,None)], [4,5], [6], 7, (8,9,10)])
[1, 2, 3, 42, None, 4, 5, 6, 7, 8, 9, 10]
```

webstruct.utils.**get_combined_keys**(*dicts*)

```
>>> sorted(get_combined_keys([{'foo': 'egg'}, {'bar': 'spam'}]))
['bar', 'foo']
```

webstruct.utils.**get_domain**(*url*)

```
>>> get_domain("http://example.com/path")
'example.com'
>>> get_domain("https://hello.example.com/foo/bar")
'example.com'
>>> get_domain("http://hello.example.co.uk/foo?bar=1")
'example.co.uk'
```

webstruct.utils.**html_document_fromstring**(*data*, *encoding=None*)
Load HTML document from string using lxml.html.HTMLParser

webstruct.utils.**kill_html_tags**(*doc*, *tagnames*, *keep_child=True*)

```
>>> from lxml.html import fragment_fromstring, tostring
>>> root = fragment_fromstring('<div><h1>head 1</h1></div>')
>>> kill_html_tags(root, ['h1'])
>>> tostring(root).decode()
'<div>head 1</div>'
```

```
>>> root = fragment_fromstring('<div><h1>head 1</h1></div>')
>>> kill_html_tags(root, ['h1'], False)
>>> tostring(root).decode()
'<div></div>'
```

webstruct.utils.**merge_dicts**(**dicts*)

```
>>> sorted(merge_dicts({'foo': 'bar'}, {'bar': 'baz'}).items())
[('bar', 'baz'), ('foo', 'bar')]
```

webstruct.utils.**replace_html_tags**(*root*, *tag_replaces*)
Replace lxml elements' tag.

```
>>> from lxml.html import fragment_fromstring, document_fromstring, tostring
>>> root = fragment_fromstring('<h1>head 1</h1>')
>>> replace_html_tags(root, {'h1': 'strong'})
>>> tostring(root).decode()
'<strong>head 1</strong>'
```

```
>>> root = document_fromstring('<h1>head 1</h1> <H2>head 2</H2>')
>>> replace_html_tags(root, {'h1': 'strong', 'h2': 'strong', 'h3': 'strong', 'h4
↪': 'strong'})
>>> tostring(root).decode()
'<html><body><strong>head 1</strong> <strong>head 2</strong></body></html>'
```

webstruct.utils.**run_command**(*args*, *verbose=True*)

Execute a command in a subprocess, terminate it if exception occurs, raise CalledProcessError exception if command returned non-zero exit code.

If `verbose == True` then print output as it appears using "print". Unlike `subprocess.check_call` it doesn't assume that stdout has a file descriptor - this allows printing to work in IPython notebook.

Example:

```
>>> run_command(["python", "-c", "print(1+2)"])
3
>>> run_command(["python", "-c", "print(1+2)"], verbose=False)
```

webstruct.utils.**smart_join**(*tokens*)

Join tokens without adding unneeded spaces before punctuation:

```
>>> smart_join(['Hello', ',', 'world', '!'])
'Hello, world!'

>>> smart_join(['(', '303', ')', '444-7777'])
'(303) 444-7777'
```

webstruct.utils.**substrings**(*txt*, *min_length*, *max_length*, *pad=''*)

```
>>> substrings("abc", 1, 100)
['a', 'ab', 'abc', 'b', 'bc', 'c']
>>> substrings("abc", 2, 100)
['ab', 'abc', 'bc']
>>> substrings("abc", 1, 2)
['a', 'ab', 'b', 'bc', 'c']
>>> substrings("abc", 1, 3, '$')
['$a', 'a', '$ab', 'ab', '$abc', 'abc', 'abc$', 'b', 'bc', 'bc$', 'c', 'c$']
```

webstruct.utils.**train_test_split_noshuffle**(**arrays*, ***options*)

Split arrays or matrices into train and test subsets without shuffling.

It allows to write

```
X_train, X_test, y_train, y_test = train_test_split_noshuffle(X, y, test_
↪size=test_size)
```

instead of

```
X_train, X_test = X[:-test_size], X[-test_size:]
y_train, y_test = y[:-test_size], y[-test_size:]
```

**Parameters** ***arrays** : sequence of lists

**test_size** : float, int, or None (default is None)

If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, test size is set to 0.25.

**Returns** **splitting** : list of lists, length=2 * len(arrays)

List containing train-test split of input array.

### Examples

```
>>> train_test_split_noshuffle([1,2,3], ['a', 'b', 'c'], test_size=1)
[[1, 2], [3], ['a', 'b'], ['c']]
>>> train_test_split_noshuffle([1,2,3,4], ['a', 'b', 'c', 'd'], test_size=0.5)
[[1, 2], [3, 4], ['a', 'b'], ['c', 'd']]
```

webstruct.utils.**alphanum_key**(*s*)
Key func for sorting strings according to numerical value.

webstruct.utils.**human_sorted**()
`sorted` that uses *alphanum_key()* as a key function

## Text Tokenization

**class** webstruct.text_tokenizers.**DefaultTokenizer**

**tokenize**(*text*)

**class** webstruct.text_tokenizers.**WordTokenizer**
This tokenizer is copy-pasted version of TreebankWordTokenizer that doesn't split on @ and ':' symbols and doesn't split contractions:

```
>>> from nltk.tokenize.treebank import TreebankWordTokenizer
>>> s = '''Good muffins cost $3.88\nin New York. Email: muffins@gmail.com'''
>>> TreebankWordTokenizer().tokenize(s)
```

['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York.', 'Email', ':', 'muffins', '@', 'gmail.com'] >>> WordTokenizer().tokenize(s) ['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York.', 'Email:', 'muffins@gmail.com']

```
>>> s = '''Shelbourne Road,'''
>>> WordTokenizer().tokenize(s)
['Shelbourne', 'Road', ',']
```

```
>>> s = '''population of 100,000'''
>>> WordTokenizer().tokenize(s)
['population', 'of', '100,000']
```

```
>>> s = '''Hello|World'''
>>> WordTokenizer().tokenize(s)
['Hello', '|', 'World']
```

```
>>> s2 = '"We beat some pretty good teams to get here," Slocum said.'
>>> WordTokenizer().tokenize(s2)
['``', 'We', 'beat', 'some', 'pretty', 'good',
```

```
'teams', 'to', 'get', 'here', ',', "'", 'Slocum', 'said', '.']
>>> s3 = '''Well, we couldn't have this predictable,
... cliche-ridden, \"Touched by an
... Angel\" (a show creator John Masius
... worked on) wanna-be if she didn't.'''
>>> WordTokenizer().tokenize(s3)
['Well', ',', 'we', "couldn't", 'have', 'this', 'predictable',
 ',', 'cliche-ridden', ',', '``', 'Touched', 'by', 'an',
 'Angel', "'", '(', 'a', 'show', 'creator', 'John', 'Masius',
 'worked', 'on', ')', 'wanna-be', 'if', 'she', "didn't", '.']
```

Some issues:

```
>>> WordTokenizer().tokenize("Phone:855-349-1914")
['Phone', ':', '855-349-1914']
```

```
>>> WordTokenizer().tokenize("Copyright © 2014 Foo Bar and Buzz Spam. All Rights␣
↪Reserved.")
['Copyright', '\xc2\xa9', '2014', 'Wall', 'Decor', 'and', 'Home', 'Accents', '.',
↪'All', 'Rights', 'Reserved', '.']
```

```
>>> WordTokenizer().tokenize("Powai Campus, Mumbai-400077")
['Powai', 'Campus', ',', 'Mumbai", "-", "400077']
```

```
>>> WordTokenizer().tokenize("1 5858/ 1800")
['1', '5858', '/', '1800']
```

```
>>> WordTokenizer().tokenize("Saudi Arabia-")
['Saudi', 'Arabia', '-']
```

**open_quotes = <_sre.SRE_Pattern object>**

**rules = [(<_sre.SRE_Pattern object>, u'‘'), (<_sre.SRE_Pattern object>, u'“'), (<_sre.SRE_Pattern object>, u"”"), (<_sr**

**tokenize**(*text*)

webstruct.text_tokenizers.**tokenize**(*self*, *text*)

## Sequence Encoding

**class** webstruct.sequence_encoding.**InputTokenProcessor**(*tagset=None*)

**classify**(*token*)

```
>>> tp = InputTokenProcessor()
>>> tp.classify('foo')
('token', 'foo')
>>> tp.classify('__START_ORG__')
('start', 'ORG')
>>> tp.classify('__END_ORG__')
('end', 'ORG')
```

**class** webstruct.sequence_encoding.**IobEncoder**(*token_processor=None*)

Utility class for encoding tagged token streams using IOB2 encoding.

Encode input tokens using `encode` method:

```
>>> iob_encoder = IobEncoder()
>>> input_tokens = ["__START_PER__", "John", "__END_PER__", "said"]
>>> iob_encoder.encode(input_tokens)
[('John', 'B-PER'), ('said', 'O')]
```

Get the result in another format using `encode_split` method:

```
>>> input_tokens = ["hello", "__START_PER__", "John", "Doe", "__END_PER__", "__
↪START_PER__", "Mary", "__END_PER__", "said"]
>>> tokens, tags = iob_encoder.encode_split(input_tokens)
>>> tokens, tags
(['hello', 'John', 'Doe', 'Mary', 'said'], ['O', 'B-PER', 'I-PER', 'B-PER', 'O'])
```

Note that IobEncoder is stateful. This means you can encode incomplete stream and continue the encoding later:

```
>>> iob_encoder = IobEncoder()
>>> iob_encoder.encode(["__START_PER__", "John"])
[('John', 'B-PER')]
>>> iob_encoder.encode(["Mayer", "__END_PER__", "said"])
[('Mayer', 'I-PER'), ('said', 'O')]
```

To reset internal state, use `reset method`:

```
>>> iob_encoder.reset()
```

Group results to entities:

```
>>> iob_encoder.group(iob_encoder.encode(input_tokens))
[(['hello'], 'O'), (['John', 'Doe'], 'PER'), (['Mary'], 'PER'), (['said'], 'O')]
```

Input token stream is processed by `InputTokenProcessor()` by default; you can pass other token processing class to customize which tokens are considered start/end tags.

**encode**(*input_tokens*)

**encode_split**(*input_tokens*)
The same as `encode`, but returns `(tokens, tags)` tuple

**classmethod group**(*data*, *strict=False*)
Group IOB2-encoded entities. `data` should be an iterable of `(info, iob_tag)` tuples. `info` could be any Python object, `iob_tag` should be a string with a tag.

Example:

```
>>>
>>> data = [("hello", "O"), (",", "O"), ("John", "B-PER"),
...         ("Doe", "I-PER"), ("Mary", "B-PER"), ("said", "O")]
>>> for items, tag in IobEncoder.iter_group(data):
...     print("%s %s" % (items, tag))
['hello', ','] O
['John', 'Doe'] PER
['Mary'] PER
['said'] O
```

By default, invalid sequences are fixed:

```
>>> data = [("hello", "O"), ("John", "I-PER"), ("Doe", "I-PER")]
>>> for items, tag in IobEncoder.iter_group(data):
...     print("%s %s" % (items, tag))
['hello'] O
['John', 'Doe'] PER
```

Pass 'strict=True' argument to raise an exception for invalid sequences:

```
>>> for items, tag in IobEncoder.iter_group(data, strict=True):
...     print("%s %s" % (items, tag))
Traceback (most recent call last):
...
ValueError: Invalid sequence: I-PER tag can't start sequence
```

**iter_encode**(*input_tokens*)

**classmethod iter_group**(*data*, *strict=False*)

**reset**()
Reset the sequence

## Webpage domain inferring

Module for getting a most likely base URL (domain) for a page. It is useful if you've downloaded HTML files, but haven't preserved URLs explicitly, and still want to have cross-validation done right. Grouping pages by domain name is a reasonable way to do that.

WebAnnotator data has either <base> tags with original URLs (or at least original domains), or a commented out base tags.

Unfortunately, GATE-annotated data doesn't have this tag. So the idea is to use a most popular domain mentioned in a page as a page's domain.

webstruct.infer_domain.**get_base_href**(*tree*)
Return href of a base tag; base tag could be commented out.

webstruct.infer_domain.**get_tree_domain**(*tree*, *blacklist=set(['flickr.com', 'pinterest.com', 'youtube.com', 'google.com', 'fonts.com', 'paypal.com', 'twitter.com', 'fonts.net', 'addthis.com', 'facebook.com', 'googleapis.com', 'linkedin.com'])*, *get_domain=<function get_domain>*)
Return the most likely domain for the tree. Domain is extracted from base tag or guessed if there is no base tag. If domain can't be detected an empty string is returned.

webstruct.infer_domain.**guess_domain**(*tree*, *blacklist=set(['flickr.com', 'pinterest.com', 'youtube.com', 'google.com', 'fonts.com', 'paypal.com', 'twitter.com', 'fonts.net', 'addthis.com', 'facebook.com', 'googleapis.com', 'linkedin.com'])*, *get_domain=<function get_domain>*)
Return most common domain not in a black list.

CHAPTER 4

# Changes

## 0.5 (2017-05-10)

- webstruct.model.NER now uses `requests` library to make HTTP requests;
- changed default headers used by webstruct.model.NER;
- new `webstruct.infer_domain` module useful for proper cross-validation;
- webstruct.webannotator.to_webannotator got an option to add `<base>` tag with the original URL to the page;
- fixed a warning in webstruct.gazetteers.geonames.read_geonames;
- add a few more country names to countries.txt list.

## 0.4.1 (2016-11-28)

- fixed a bug in NER.extract().

## 0.4 (2016-11-26)

- sklearn-crfsuite is used as a CRFsuite wrapper, CRFsuiteCRF class is removed;
- comments are preserved in HTML trees because recent Firefox puts `<base>` tags to a comment when saving pages, and this affects WebAnnotator;
- fixed 'dont_penalize' argument of webstruct.NER.extract_groups_from_url;
- new webstruct.model.extract_entity_groups utility function;
- HtmlTokenizer and HtmlToken are moved to their own module (webstruct.html_tokenizer);
- test improvements;

## 0.3 (2016-09-19)

There are many changes from previous version: API is changed, Python 3 is supported, better gazetteers support, CRFsuite support, etc.

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## W

# Index

## T



## U



## W