DiaParser

Release 1.1.3

Yu Zhang, Giuseppe Attardi

Mar 24, 2023

CONTENTS

1 Parsers **3**
- 1.1 Parser . . . 3
- 1.2 BiaffineDependencyParser . . . 4

2 Models **7**
- 2.1 BiaffineDependencyModel . . . 7

3 Modules **11**
- 3.1 Affine . . . 11
- 3.2 BertEmbedding . . . 11
- 3.3 LSTM . . . 12
- 3.4 CharLSTM . . . 13
- 3.5 Dropout . . . 14
- 3.6 MLP . . . 15

4 Utils **17**
- 4.1 Algorithms . . . 17
- 4.2 Dataset . . . 21
- 4.3 Field . . . 22
- 4.4 Transform . . . 26
- 4.5 Vocab . . . 31

5 tokenizer **33**
- 5.1 Tokenizer . . . 33

6 Indices and tables **35**

Index **37**

The `diaparser` package provides a state-of-the-art dependency parser based on the Biaffine Parser architecture and exploiting attentions from transformes..

CHAPTER

ONE

PARSERS

1.1 Parser

class diaparser.parsers.**Parser**(*args*, *model*, *transform*)

train(*train*, *dev*, *test*, *buckets=32*, *batch_size=5000*, *lr=0.002*, *mu=0.9*, *nu=0.9*, *epsilon=1e-12*, *clip=5.0*, *decay=0.75*, *decay_steps=5000*, *epochs=5000*, *patience=100*, *verbose=True*, ***kwargs*)

Parameters

- **lr** (*float*) – learnin rate of adam optimizer. Default: 2e-3.
- **mu** (*float*) – beta1 of adam optimizer. Default: .9.
- **nu** (*float*) – beta2 of adam optimizer. Default: .9.
- **epsilon** (*float*) – epsilon of adam optimizer. Default: 1e-12.
- **buckets** (*int*) – number of buckets. Default: 32.
- **epochs** (*int*) – number of epochs to train: Default: 5000.
- **patience** (*int*) – early stop after these many epochs. Default: 100.

predict(*data*, *pred=None*, *buckets=8*, *batch_size=5000*, *prob=False*, ***kwargs*)

Parses the data and produces a parse tree for each sentence. :param data: input to be parsed: either

- a str, that will be tokenized first with the tokenizer for the parser language
- a path to a file to be read, either in CoNLL-U format or in plain text if :param text: is supplied.
- a list of lists of tokens

Parameters

- **text** (*str*) – optional, specifies that the input data is in plain text in the specified language code.
- **pred** (*str or file*) – a path to a file where to write the parsed input in CoNLL-U fprmat.
- **bucket** (*int*) – the number of buckets used to group sentences to parallelize matrix computations.
- **batch_size** (*int*) – group sentences in batches.
- **prob** (*bool*) – whther to return also probabilities for each arc.

Returns

a Dataset containing the parsed sentence trees.

classmethod load(*name_or_path=''*, *lang='en'*, *cache_dir='/home/docs/.cache/diaparser'*, ***kwargs*)

Loads a parser from a pretrained model.

Parameters

- **name_or_path** (*str*) –
  - a string with the shortcut name of a pretrained parser listed in `resource.json` to load from cache or download, e.g., `'en_ptb.electra-base'`.
  - a path to a directory containing a pre-trained parser, e.g., *./<path>/model*.
- **lang** (*str*) – A language code, used in alternative to `name_or_path` to load the default model for the given language.
- **cache_dir** (*str*) – Directory where to cache models. The default value is *~/.cache/diaparser*.
- **kwargs** (*dict*) – A dict holding the unconsumed arguments that can be used to update the configurations and initiate the model.

Examples

```
>>> parser = Parser.load('en_ewt.electra-base')
>>> parser = Parser.load(lang='en')
>>> parser = Parser.load('./ptb.biaffine.dependency.char')
```

1.2 BiaffineDependencyParser

class diaparser.parsers.**BiaffineDependencyParser**(**args*, ***kwargs*)

The implementation of Biaffine Dependency Parser.

References

- Timothy Dozat and Christopher D. Manning. 2017. Deep Biaffine Attention for Neural Dependency Parsing.

MODEL

alias of *BiaffineDependencyModel*

train(*train*, *dev*, *test*, *buckets=32*, *batch_size=5000*, *punct=False*, *tree=False*, *proj=False*, *verbose=True*, ***kwargs*)

Parameters

- **train/dev/test** (*list[list] or str*) – Filenames of the train/dev/test datasets.
- **buckets** (*int*) – The number of buckets that sentences are assigned to. Default: 32.
- **batch_size** (*int*) – The number of tokens in each batch. Default: 5000.
- **punct** (*bool*) – If `False`, ignores the punctuations during evaluation. Default: `False`.
- **tree** (*bool*) – If `True`, ensures to output well-formed trees. Default: `False`.
- **proj** (*bool*) – If `True`, ensures to output projective trees. Default: `False`.

- **partial** (*bool*) – `True` denotes the trees are partially annotated. Default: `False`.
- **verbose** (*bool*) – If `True`, increases the output verbosity. Default: `True`.
- **kwargs** (*dict*) – A dict holding the unconsumed arguments that can be used to update the configurations for training.

evaluate(*data*, *buckets=8*, *batch_size=5000*, *punct=False*, *tree=True*, *proj=False*, *partial=False*, *verbose=True*, ***kwargs*)

Parameters

- **data** (*str*) – The data for evaluation, both list of instances and filename are allowed.
- **buckets** (*int*) – The number of buckets that sentences are assigned to. Default: 32.
- **batch_size** (*int*) – The number of tokens in each batch. Default: 5000.
- **punct** (*bool*) – If `False`, ignores the punctuations during evaluation. Default: `False`.
- **tree** (*bool*) – If `True`, ensures to output well-formed trees. Default: `False`.
- **proj** (*bool*) – If `True`, ensures to output projective trees. Default: `False`.
- **partial** (*bool*) – `True` denotes the trees are partially annotated. Default: `False`.
- **verbose** (*bool*) – If `True`, increases the output verbosity. Default: `True`.
- **kwargs** (*dict*) – A dict holding the unconsumed arguments that can be used to update the configurations for evaluation.

Returns

The loss scalar and evaluation results.

predict(*data*, *pred=None*, *buckets=8*, *batch_size=5000*, *prob=False*, *tree=True*, *proj=False*, *verbose=False*, ***kwargs*)

Parameters

- **data** (*list[list] or str*) – The data for prediction, both a list of instances and filename are allowed.
- **pred** (*str*) – If specified, the predicted results will be saved to the file. Default: `None`.
- **buckets** (*int*) – The number of buckets that sentences are assigned to. Default: 32.
- **batch_size** (*int*) – The number of tokens in each batch. Default: 5000.
- **prob** (*bool*) – If `True`, outputs the probabilities. Default: `False`.
- **tree** (*bool*) – If `True`, ensures to output well-formed trees. Default: `False`.
- **proj** (*bool*) – If `True`, ensures to output projective trees. Default: `False`.
- **verbose** (*bool*) – If `True`, increases the output verbosity. Default: `True`.
- **kwargs** (*dict*) – A dict holding the unconsumed arguments that can be used to update the configurations for prediction.

Returns

A *Dataset* object that stores the predicted results.

classmethod build(*path*, *min_freq=2*, *fix_len=20*, ***kwargs*)

Build a brand-new Parser, including initialization of all data fields and model parameters.

Parameters

- **path** (*str*) – The path of the model to be saved.

- **min_freq** (*str*) – The minimum frequency needed to include a token in the vocabulary. Default: 2.
- **fix_len** (*int*) – The max length of all subword pieces. The excess part of each piece will be truncated. Required if using CharLSTM/BERT. Default: 20.
- **kwargs** (*dict*) – A dict holding the unconsumed arguments.

CHAPTER

TWO

MODELS

2.1 BiaffineDependencyModel

class diaparser.models.**BiaffineDependencyModel**(*n_words*, *n_feats*, *n_rels*, *feat='char'*, *n_word_embed=100*, *n_feat_embed=100*, *n_char_embed=50*, *bert=None*, *n_bert_layers=4*, *bert_fine_tune=False*, *mix_dropout=0.0*, *token_dropout=0.0*, *embed_dropout=0.33*, *n_lstm_hidden=400*, *n_lstm_layers=3*, *lstm_dropout=0.33*, *n_mlp_arc=500*, *n_mlp_rel=100*, *mask_token_id=0.0*, *mlp_dropout=0.33*, *use_hidden_states=True*, *use_attentions=False*, *attention_head=0*, *attention_layer=6*, *feat_pad_index=0*, *pad_index=0*, *unk_index=1*, ***kwargs*)

The implementation of Biaffine Dependency Parser.

References

- Timothy Dozat and Christopher D. Manning. 2017. Deep Biaffine Attention for Neural Dependency Parsing.

Parameters

- **n_words** (*int*) – The size of the word vocabulary.
- **n_feats** (*int*) – The size of the feat vocabulary.
- **n_rels** (*int*) – The number of labels in the treebank.
- **feat** (*str*) – Specifies which type of additional feature to use: `'char'` | `'bert'` | `'tag'`. `'char'`: Character-level representations extracted by CharLSTM. `'bert'`: BERT representations, other pretrained langugae models like XLNet are also feasible. `'tag'`: POS tag embeddings. Default: `'char'`.
- **n_word_embed** (*int*) – The size of word embeddings. Default: 100.
- **n_feat_embed** (*int*) – The size of feature representations. Default: 100.
- **n_char_embed** (*int*) – The size of character embeddings serving as inputs of CharLSTM, required if `feat='char'`. Default: 50.

- **bert** (*str*) – Specifies which kind of language model to use, e.g., `'bert-base-cased'` and `'xlnet-base-cased'`. This is required if `feat='bert'`. The full list can be found in transformers. Default: `None`.
- **n_bert_layers** (*int*) – Specifies how many last layers to use. Required if `feat='bert'`. The final outputs would be the weight sum of the hidden states of these layers. Default: 4.
- **bert_fine_tune** (*bool*) – Weather to fine tune the BERT model. Deafult: False.
- **mix_dropout** (*float*) – The dropout ratio of BERT layers. Required if `feat='bert'`. Default: .0.
- **token_dropout** (*float*) – The dropout ratio of tokens. Default: .0.
- **embed_dropout** (*float*) – The dropout ratio of input embeddings. Default: .33.
- **n_lstm_hidden** (*int*) – The size of LSTM hidden states. Default: 400.
- **n_lstm_layers** (*int*) – The number of LSTM layers. Default: 3.
- **lstm_dropout** (*float*) – The dropout ratio of LSTM. Default: .33.
- **n_mlp_arc** (*int*) – Arc MLP size. Default: 500.
- **n_mlp_rel** (*int*) – Label MLP size. Default: 100.
- **mlp_dropout** (*float*) – The dropout ratio of MLP layers. Default: .33.
- **use_hidden_states** (*bool*) – Wethre to use hidden states rather than outputs from BERT. Default: True.
- **use_attentions** (*bool*) – Wethre to use attention heads from BERT. Default: False.
- **attention_head** (*int*) – Which attention head from BERT to use. Default: 0.
- **attention_layer** (*int*) – Which attention layer from BERT to use; use all if 0. Default: 6.
- **feat_pad_index** (*int*) – The index of the padding token in the feat vocabulary. Default: 0.
- **pad_index** (*int*) – The index of the padding token in the word vocabulary. Default: 0.
- **unk_index** (*int*) – The index of the unknown token in the word vocabulary. Default: 1.

extra_repr()

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

forward(*words: Tensor*, *feats: Tensor*) → Tuple[Tensor, Tensor]

Parameters

- **words** (*LongTensor*) – `[batch_size, seq_len]`. Word indices.
- **feats** (*LongTensor*) – Feat indices. If feat is `'char'` or `'bert'`, the size of feats should be `[batch_size, seq_len, fix_len]`. if `'tag'`, the size is `[batch_size, seq_len]`.

Returns

The first tensor of shape `[batch_size, seq_len, seq_len]` holds scores of all possible arcs. The second of shape `[batch_size, seq_len, seq_len, n_labels]` holds scores of all possible labels on each arc.

Return type
Tensor, *Tensor*

loss(*s_arc: Tensor*, *s_rel: Tensor*, *arcs: Tensor*, *rels: Tensor*, *mask: Tensor*, *partial: bool = False*) → Tensor

Computes the arc and tag loss for a sequence given gold heads and tags.

Parameters

- **s_arc** (*Tensor*) – `[batch_size, seq_len, seq_len]`. Scores of all possible arcs.
- **s_rel** (*Tensor*) – `[batch_size, seq_len, seq_len, n_labels]`. Scores of all possible labels on each arc.
- **arcs** (*LongTensor*) – `[batch_size, seq_len]`. The tensor of gold-standard arcs.
- **rels** (*LongTensor*) – `[batch_size, seq_len]`. The tensor of gold-standard labels.
- **mask** (*BoolTensor*) – `[batch_size, seq_len]`. The mask for covering the unpadded tokens.
- **partial** (*bool*) – `True` denotes the trees are partially annotated. Default: `False`.

Returns
The training loss.

Return type
Tensor

decode(*s_arc: Tensor*, *s_rel: Tensor*, *mask: Tensor*, *tree: bool = False*, *proj: bool = False*) → Tuple[Tensor, Tensor]

Parameters

- **s_arc** (*Tensor*) – `[batch_size, seq_len, seq_len]`. Scores of all possible arcs.
- **s_rel** (*Tensor*) – `[batch_size, seq_len, seq_len, n_labels]`. Scores of all possible labels on each arc.
- **mask** (*BoolTensor*) – `[batch_size, seq_len]`. The mask for covering the unpadded tokens.
- **tree** (*bool*) – If `True`, ensures to output well-formed trees. Default: `False`.
- **proj** (*bool*) – If `True`, ensures to output projective trees. Default: `False`.

Returns
Predicted arcs and labels of shape `[batch_size, seq_len]`.

Return type
Tensor, *Tensor*

CHAPTER

THREE

MODULES

3.1 Affine

class diaparser.modules.**Biaffine**(*n_in*, *n_out=1*, *bias_x=True*, *bias_y=True*)

extra_repr()

Set the extra representation of the module

To print customized extra information, you should re-implement this method in your own modules. Both single-line and multi-line strings are acceptable.

forward(*x*, *y*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

3.2 BertEmbedding

class diaparser.modules.**BertEmbedding**(*model*, *n_layers*, *n_out*, *stride=5*, *pad_index=0*, *dropout=0*, *requires_grad=False*, *mask_token_id=0*, *token_dropout=0.0*, *mix_dropout=0.0*, *use_hidden_states=True*, *use_attentions=False*, *attention_head=0*, *attention_layer=8*)

A module that directly utilizes the pretrained models in transformers to produce BERT representations. While mainly tailored to provide input preparation and post-processing for the BERT model, it is also compatible with other pretrained language models like XLNet, RoBERTa and ELECTRA, etc.

Parameters

- **model** (*str*) – Path or name of the pretrained models registered in transformers, e.g., `'bert-base-cased'`.
- **n_layers** (*int*) – The number of layers from the model to use. If 0, uses all layers.
- **n_out** (*int*) – The requested size of the embeddings. If 0, uses the size of the pretrained embedding model.

- **stride** (*int*) – A sequence longer than the limited max length will be splitted into several small pieces with a window size of `stride`. Default: 5.
- **pad_index** (*int*) – The index of the padding token in the BERT vocabulary. Default: 0.
- **dropout** (*float*) – The dropout ratio of BERT layers. Default: 0. This value will be passed into the *ScalarMix* layer.
- **requires_grad** (*bool*) – If `True`, the model parameters will be updated together with the downstream task. Default: `False`.

forward(*subwords*)

Parameters
subwords (*Tensor*) – `[batch_size, seq_len, fix_len]`.

Returns
BERT embeddings of shape `[batch_size, seq_len, n_out]`.

Return type
Tensor

class diaparser.modules.**ScalarMix**(*n_layers: int*, *dropout: float = 0.0*)

Computes a parameterised scalar mixture of N tensors, $mixture = \gamma * \sum_k(s_k * tensor_k)$ where $s = \mathrm{softmax}(w)$, with w and γ scalar parameters.

Parameters

- **n_layers** (*int*) – The number of layers to be mixed, i.e., N.
- **dropout** (*float*) – The dropout ratio of the layer weights. If dropout > 0, then for each scalar weight, adjust its softmax weight mass to 0 with the dropout probability (i.e., setting the unnormalized weight to -inf). This effectively redistributes the dropped probability mass to all other weights. Default: 0.

forward(*tensors*)

Parameters
tensors (*list[Tensor]*) – N tensors to be mixed.

Returns
The mixture of N tensors.

3.3 LSTM

class diaparser.modules.**LSTM**(*input_size*, *hidden_size*, *num_layers=1*, *bidirectional=False*, *dropout=0*)

LSTM is an variant of the vanilla bidirectional LSTM adopted by Biaffine Parser with the only difference of the dropout strategy. It drops nodes in the LSTM layers (input and recurrent connections) and applies the same dropout mask at every recurrent timesteps.

APIs are roughly the same as `LSTM` except that we only allows `PackedSequence` as input.

References

- Timothy Dozat and Christopher D. Manning. 2017. Deep Biaffine Attention for Neural Dependency Parsing.

Parameters

- **input_size** (*int*) – The number of expected features in the input.
- **hidden_size** (*int*) – The number of features in the hidden state *h*.
- **num_layers** (*int*) – The number of recurrent layers. Default: 1.
- **bidirectional** (*bool*) – If `True`, becomes a bidirectional LSTM. Default: `False`
- **dropout** (*float*) – If non-zero, introduces a *SharedDropout* layer on the outputs of each LSTM layer except the last layer. Default: 0.

forward(*sequence*, *hx=None*)

Parameters

- **sequence** (*PackedSequence*) – A packed variable length sequence.
- **hx** (*Tensor, Tensor*) – A tuple composed of two tensors *h* and *c*. *h* of shape `[num_layers*num_directions, batch_size, hidden_size]` holds the initial hidden state for each element in the batch. *c* of shape `[num_layers*num_directions, batch_size, hidden_size]` holds the initial cell state for each element in the batch. If *hx* is not provided, both *h* and *c* default to zero. Default: `None`.

Returns

The first is a packed variable length sequence. The second is a tuple of tensors *h* and *c*. *h* of shape `[num_layers*num_directions, batch_size, hidden_size]` holds the hidden state for *t=seq_len*. Like output, the layers can be separated using `h.view(num_layers, 2, batch_size, hidden_size)` and similarly for c. *c* of shape `[num_layers*num_directions, batch_size, hidden_size]` holds the cell state for *t=seq_len*.

Return type

PackedSequence, (*Tensor*, *Tensor*)

3.4 CharLSTM

class diaparser.modules.**CharLSTM**(*n_chars*, *n_word_embed*, *n_out*, *pad_index=0*)

CharLSTM aims to generate character-level embeddings for tokens. It summerizes the information of characters in each token to an embedding using a LSTM layer.

Parameters

- **n_char** (*int*) – The number of characters.
- **n_embed** (*int*) – The size of each embedding vector as input to LSTM.
- **n_out** (*int*) – The size of each output vector.
- **pad_index** (*int*) – The index of the padding token in the vocabulary. Default: 0.

forward(*x*)

Parameters
x (*Tensor*) – `[batch_size, seq_len, fix_len]`. Characters of all tokens. Each token holds no more than *fix_len* characters, and the excess is cut off directly.

Returns
The embeddings of shape `[batch_size, seq_len, n_out]` derived from the characters.

Return type
Tensor

3.5 Dropout

class diaparser.modules.**IndependentDropout**(*p=0.5*)

For N tensors, they use different dropout masks respectively. When $N - M$ of them are dropped, the remaining M ones are scaled by a factor of N/M to compensate, and when all of them are dropped together, zeros are returned.

Parameters
p (*float*) – The probability of an element to be zeroed. Default: 0.5.

Examples

```
>>> x, y = torch.ones(1, 3, 5), torch.ones(1, 3, 5)
>>> x, y = IndependentDropout()(x, y)
>>> x
tensor([[[1., 1., 1., 1., 1.],
         [0., 0., 0., 0., 0.],
         [2., 2., 2., 2., 2.]]])
>>> y
tensor([[[1., 1., 1., 1., 1.],
         [2., 2., 2., 2., 2.],
         [0., 0., 0., 0., 0.]]])
```

forward(**items*)

Parameters
items (*list[Tensor]*) – A list of tensors that have the same shape except the last dimension.

Returns
The returned tensors are of the same shape as *items*.

class diaparser.modules.**SharedDropout**(*p=0.5*, *batch_first=True*)

SharedDropout differs from the vanilla dropout strategy in that the dropout mask is shared across one dimension.

Parameters

- **p** (*float*) – The probability of an element to be zeroed. Default: 0.5.
- **batch_first** (*bool*) – If `True`, the input and output tensors are provided as `[batch_size, seq_len, *]`. Default: `True`.

Examples

```
>>> x = torch.ones(1, 3, 5)
>>> nn.Dropout()(x)
tensor([[[0., 2., 2., 0., 0.],
         [2., 2., 0., 2., 2.],
         [2., 2., 2., 2., 0.]]])
>>> SharedDropout()(x)
tensor([[[2., 0., 2., 0., 2.],
         [2., 0., 2., 0., 2.],
         [2., 0., 2., 0., 2.]]])
```

forward(x)

Parameters

x (*Tensor*) – A tensor of any shape.

Returns

The returned tensor is of the same shape as x.

3.6 MLP

class diaparser.modules.**MLP**(*n_in*, *n_out*, *dropout=0*)

Applies a linear transformation together with LeakyReLU activation to the incoming tensor: $y = \mathrm{LeakyReLU}(xA^T + b)$

Parameters

- **n_in** (*Tensor*) – The size of each input feature.
- **n_out** (*Tensor*) – The size of each output feature.
- **dropout** (*float*) – If non-zero, introduce a *SharedDropout* layer on the output with this dropout ratio. Default: 0.

forward(x)

Parameters

x (*Tensor*) – The size of each input feature is *n_in*.

Returns

A tensor with the size of each output feature *n_out*.

CHAPTER

FOUR

UTILS

4.1 Algorithms

diaparser.utils.alg.**kmeans**(*x*, *k*, *max_it=32*)

KMeans algorithm for clustering the sentences by length.

Parameters

- **x** (*list[int]*) – The list of sentence lengths.
- **k** (*int*) – The number of clusters. This is an approximate value. The final number of clusters can be less or equal to *k*.
- **max_it** (*int*) – Maximum number of iterations. If centroids does not converge after several iterations, the algorithm will be early stopped.

Returns

The first list contains average lengths of sentences in each cluster. The second is the list of clusters holding indices of data points.

Return type

list[float], list[list[int]]

Examples

```
>>> x = torch.randint(10,20,(10,)).tolist()
>>> x
[15, 10, 17, 11, 18, 13, 17, 19, 18, 14]
>>> centroids, clusters = kmeans(x, 3)
>>> centroids
[10.5, 14.0, 17.799999237060547]
>>> clusters
[[1, 3], [0, 5, 9], [2, 4, 6, 7, 8]]
```

diaparser.utils.alg.**tarjan**(*sequence*)

Tarjan algorithm for finding Strongly Connected Components (SCCs) of a graph.

Parameters

sequence (*list*) – List of head indices.

Yields

A list of indices that make up a SCC. All self-loops are ignored.

Examples

```
>>> next(tarjan([2, 5, 0, 3, 1]))  # (1 -> 5 -> 2 -> 1) is a cycle
[2, 5, 1]
```

diaparser.utils.alg.**chuliu_edmonds**(*s*)

ChuLiu/Edmonds algorithm for non-projective decoding.

Some code is borrowed from tdozat's implementation. Descriptions of notations and formulas can be found in Non-projective Dependency Parsing using Spanning Tree Algorithms.

Notes

The algorithm does not guarantee to parse a single-root tree.

References

- Ryan McDonald, Fernando Pereira, Kiril Ribarov and Jan Hajic. 2005. Non-projective Dependency Parsing using Spanning Tree Algorithms.

Parameters
s (*Tensor*) – `[seq_len, seq_len]`. Scores of all dependent-head pairs.

Returns
A tensor with shape `[seq_len]` for the resulting non-projective parse tree.

Return type
Tensor

diaparser.utils.alg.**mst**(*scores*, *mask*, *multiroot=False*)

MST algorithm for decoding non-pojective trees. This is a wrapper for ChuLiu/Edmonds algorithm.

The algorithm first runs ChuLiu/Edmonds to parse a tree and then have a check of multi-roots, If `multiroot=True` and there indeed exist multi-roots, the algorithm seeks to find best single-root trees by iterating all possible single-root trees parsed by ChuLiu/Edmonds. Otherwise the resulting trees are directly taken as the final outputs.

Parameters

- **scores** (*Tensor*) – `[batch_size, seq_len, seq_len]`. Scores of all dependent-head pairs.
- **mask** (*BoolTensor*) – `[batch_size, seq_len]`. The mask to avoid parsing over padding tokens. The first column serving as pseudo words for roots should be `False`.
- **muliroot** (*bool*) – Ensures to parse a single-root tree If `False`.

Returns
A tensor with shape `[batch_size, seq_len]` for the resulting non-projective parse trees.

Return type
Tensor

Examples

```
>>> scores = torch.tensor([[[-11.9436, -13.1464,  -6.4789, -13.8917],
                            [-60.6957, -60.2866, -48.6457, -63.8125],
                            [-38.1747, -49.9296, -45.2733, -49.5571],
                            [-19.7504, -23.9066,  -9.9139, -16.2088]]])
>>> scores[:, 0, 1:] = float('-inf')
>>> scores.diagonal(0, 1, 2)[1:].fill_(float('-inf'))
>>> mask = torch.tensor([[False,  True,  True,  True]])
>>> mst(scores, mask)
tensor([[0, 2, 0, 2]])
```

diaparser.utils.alg.**eisner**(*scores*, *mask*)

First-order Eisner algorithm for projective decoding.

References

- Ryan McDonald, Koby Crammer and Fernando Pereira. 2005. Online Large-Margin Training of Dependency Parsers.

Parameters

- **scores** (*Tensor*) – `[batch_size, seq_len, seq_len]`. Scores of all dependent-head pairs.
- **mask** (*BoolTensor*) – `[batch_size, seq_len]`. The mask to avoid parsing over padding tokens. The first column serving as pseudo words for roots should be `False`.

Returns

A tensor with shape `[batch_size, seq_len]` for the resulting projective parse trees.

Return type

Tensor

Examples

```
>>> scores = torch.tensor([[[-13.5026, -18.3700, -13.0033, -16.6809],
                            [-36.5235, -28.6344, -28.4696, -31.6750],
                            [ -2.9084,  -7.4825,  -1.4861,  -6.8709],
                            [-29.4880, -27.6905, -26.1498, -27.0233]]])
>>> mask = torch.tensor([[False,  True,  True,  True]])
>>> eisner(scores, mask)
tensor([[0, 2, 0, 2]])
```

diaparser.utils.alg.**eisner2o**(*scores*, *mask*)

Second-order Eisner algorithm for projective decoding. This is an extension of the first-order one that further incorporates sibling scores into tree scoring.

References

- Ryan McDonald and Fernando Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms.

Parameters

- **scores** (*Tensor, Tensor*) – A tuple of two tensors representing the first-order and second-order scores repectively. The first (`[batch_size, seq_len, seq_len]`) holds scores of all dependent-head pairs. The second (`[batch_size, seq_len, seq_len, seq_len]`) holds scores of all dependent-head-sibling triples.
- **mask** (*BoolTensor*) – `[batch_size, seq_len]`. The mask to avoid parsing over padding tokens. The first column serving as pseudo words for roots should be `False`.

Returns

A tensor with shape `[batch_size, seq_len]` for the resulting projective parse trees.

Return type

Tensor

Examples

```
>>> s_arc = torch.tensor([[[ -2.8092,  -7.9104,  -0.9414,  -5.4360],
                           [-10.3494,  -7.9298,  -3.6929,  -7.3985],
                           [  1.1815,  -3.8291,   2.3166,  -2.7183],
                           [ -3.9776,  -3.9063,  -1.6762,  -3.1861]]])
>>> s_sib = torch.tensor([[[[ 0.4719,  0.4154,  1.1333,  0.6946],
                            [ 1.1252,  1.3043,  2.1128,  1.4621],
                            [ 0.5974,  0.5635,  1.0115,  0.7550],
                            [ 1.1174,  1.3794,  2.2567,  1.4043]],
                           [[-2.1480, -4.1830, -2.5519, -1.8020],
                            [-1.2496, -1.7859, -0.0665, -0.4938],
                            [-2.6171, -4.0142, -2.9428, -2.2121],
                            [-0.5166, -1.0925,  0.5190,  0.1371]],
                           [[ 0.5827, -1.2499, -0.0648, -0.0497],
                            [ 1.4695,  0.3522,  1.5614,  1.0236],
                            [ 0.4647, -0.7996, -0.3801,  0.0046],
                            [ 1.5611,  0.3875,  1.8285,  1.0766]],
                           [[-1.3053, -2.9423, -1.5779, -1.2142],
                            [-0.1908, -0.9699,  0.3085,  0.1061],
                            [-1.6783, -2.8199, -1.8853, -1.5653],
                            [ 0.3629, -0.3488,  0.9011,  0.5674]]]])
>>> mask = torch.tensor([[False,  True,  True,  True]])
>>> eisner2o((s_arc, s_sib), mask)
tensor([[0, 2, 0, 2]])
```

`diaparser.utils.alg.cky`(*scores*, *mask*)

The implementation of Cocke-Kasami-Younger (CKY) algorithm to parse constituency trees.

References

- Yu Zhang, Houquan Zhou and Zhenghua Li. 2020. Fast and Accurate Neural CRF Constituency Parsing.

Parameters

- **scores** (*Tensor*) – `[batch_size, seq_len, seq_len]`. Scores of all candidate constituents.
- **mask** (*BoolTensor*) – `[batch_size, seq_len, seq_len]`. The mask to avoid parsing over padding tokens. For each square matrix in a batch, the positions except upper triangular part should be masked out.

Returns

Sequences of factorized predicted bracketed trees that are traversed in pre-order.

Examples

```
>>> scores = torch.tensor([[[ 2.5659,  1.4253, -2.5272,  3.3011],
                            [ 1.3687, -0.5869,  1.0011,  3.3020],
                            [ 1.2297,  0.4862,  1.1975,  2.5387],
                            [-0.0511, -1.2541, -0.7577,  0.2659]]])
>>> mask = torch.tensor([[[False,  True,  True,  True],
                          [False, False,  True,  True],
                          [False, False, False,  True],
                          [False, False, False, False]]])
>>> cky(scores, mask)
[[(0, 3), (0, 1), (1, 3), (1, 2), (2, 3)]]
```

4.2 Dataset

class diaparser.utils.**Dataset**(*transform*, *data*, ***kwargs*)

Dataset that is compatible with `torch.utils.data.Dataset`. This serves as a wrapper for manipulating all data fields with the operating behaviours defined in *Transform*. The data fields of all the instantiated sentences can be accessed as an attribute of the dataset.

Parameters

- **transform** (Transform) – An instance of *Transform* and its derivations. The instance holds a series of loading and processing behaviours with regard to the specfic data format.
- **data** (*list[list] or str*) – A list of list of strings or a filename. This will be passed into `transform.load()`.
- **kwargs** (*dict*) – Keyword arguments that will be passed into `transform.load()` together with *data* to control the loading behaviour.

transform

An instance of *Transform*.

Type

Transform

sentences

A list of sentences loaded from the data. Each sentence includes fields obeying the data format defined in `transform`.

Type
list[Sentence]

4.3 Field

class diaparser.utils.**RawField**(*name*, *fn=None*)

Defines a general datatype.

A *RawField* object does not assume any property of the datatype and it holds parameters relating to how a datatype should be processed.

Parameters

- **name** (*str*) – The name of the field.
- **fn** (*function*) – The function used for preprocessing the examples. Default: None.

class diaparser.utils.**Field**(*name*, *pad=None*, *unk=None*, *bos=None*, *eos=None*, *lower=False*, *use_vocab=True*, *tokenize=None*, *fn=None*, *mask_token_id=0*)

Defines a datatype together with instructions for converting to Tensor. *Field* models common text processing datatypes that can be represented by tensors. It holds a *Vocab* object that defines the set of possible values for elements of the field and their corresponding numerical representations. The *Field* object also holds other parameters relating to how a datatype should be numericalized, such as a tokenization method.

Parameters

- **name** (*str*) – The name of the field.
- **pad_token** (*str*) – The string token used as padding. Default: None.
- **unk_token** (*str*) – The string token used to represent OOV words. Default: None.
- **bos_token** (*str*) – A token that will be prepended to every example using this field, or None for no *bos_token*. Default: None.
- **eos_token** (*str*) – A token that will be appended to every example using this field, or None for no *eos_token*.
- **lower** (*bool*) – Whether to lowercase the text in this field. Default: False.
- **use_vocab** (*bool*) – Whether to use a *Vocab* object. If False, the data in this field should already be numerical. Default: True.
- **tokenize** (*function*) – The function used to tokenize strings using this field into sequential examples. Default: None.
- **fn** (*function*) – The function used for preprocessing the examples. Default: None.

preprocess(*sequence*)

Loads a single example using this field, tokenizing if necessary. The sequence will be first passed to fn if available. If tokenize is not None, the input will be tokenized. Then the input will be lowercased optionally.

Parameters
sequence (*list*) – The sequence to be preprocessed.

Returns
A list of preprocessed sequence.

build(*dataset*, *min_freq=1*, *embed=None*)

Constructs a *Vocab* object for this field from the dataset. If the vocabulary has already existed, this function will have no effect.

Parameters

- **dataset** (Dataset) – A *Dataset* object. One of the attributes should be named after the name of this field.
- **min_freq** (*int*) – The minimum frequency needed to include a token in the vocabulary. Default: 1.
- **embed** (*Embedding*) – An Embedding object, words in which will be extended to the vocabulary. Default: None.

transform(*sequences: List[List[str]]*) → List[Tensor]

Turns a list of sequences that use this field into tensors.

Each sequence is first preprocessed and then numericalized if needed.

Parameters
sequences (*list[list[str]]*) – A list of sequences.

Returns
A list of tensors transformed from the input sequences.

compose(*sequences*)

Composes a batch of sequences into a padded tensor.

Parameters
sequences (*list[Tensor]*) – A list of tensors.

Returns
A padded tensor converted to proper device.

class diaparser.utils.**SubwordField**(**args*, *fix_len=0*, ***kwargs*)

A field that conducts tokenization and numericalization over each token rather the sequence.

This is customized for models requiring character/subword-level inputs, e.g., CharLSTM and BERT.

Parameters
fix_len (*int*) – A fixed length that all subword pieces will be padded to. This is used for truncating the subword pieces that exceed the length. To save the memory, the final length will be the smaller value between the max length of subword pieces in a batch and *fix_len*.

Examples

```
>>> from transformers import AutoTokenizer
>>> tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
>>> field = SubwordField('bert',
                         pad=tokenizer.pad_token,
                         unk=tokenizer.unk_token,
                         bos=tokenizer.cls_token,
                         eos=tokenizer.sep_token,
                         fix_len=20,
                         tokenize=tokenizer.tokenize)
```

(continues on next page)

(continued from previous page)

```
>>> field.vocab = tokenizer.get_vocab()  # no need to re-build the vocab
>>> field.transform([['This', 'field', 'performs', 'token-level', 'tokenization
↪']])[0]
tensor([[  101,     0,     0],
        [ 1188,     0,     0],
        [ 1768,     0,     0],
        [10383,     0,     0],
        [22559,   118,  1634],
        [22559,  2734,     0],
        [  102,     0,     0]])
```

build(*dataset*, *min_freq=1*, *embed=None*)

Constructs a *Vocab* object for this field from the dataset. If the vocabulary has already existed, this function will have no effect.

Parameters

- **dataset** (*Dataset*) – A *Dataset* object. One of the attributes should be named after the name of this field.
- **min_freq** (*int*) – The minimum frequency needed to include a token in the vocabulary. Default: 1.
- **embed** (*Embedding*) – An Embedding object, words in which will be extended to the vocabulary. Default: None.

transform(*sequences*)

Turns a list of sequences that use this field into tensors.

Each sequence is first preprocessed and then numericalized if needed.

Parameters

sequences (*list[list[str]]*) – A list of sequences.

Returns

A list of tensors transformed from the input sequences.

class diaparser.utils.**BertField**(*name*, *tokenizer*, ***kwargs*)

A field that is dealt by a transformer.

Parameters

- **name** (*str*) – name of the field.
- **tokenizer** (*AutoTokenizer*) – the tokenizer for the transformer.
- **fix_len** (*int*) – A fixed length that all subword pieces will be padded to. This is used for truncating the subword pieces that exceed the length. To save the memory, the final length will be the smaller value between the max length of subword pieces in a batch and *fix_len*.

Examples

```
>>> tokenizer = BertField.tokenizer('bert-base-cased')
>>> field = BertField('bert',
                      tokenizer,
                      fix_len=20)
>>> field.transform([['This', 'field', 'performs', 'token-level', 'tokenization
↪']])[0]
tensor([[  101,     0,     0],
        [ 1188,     0,     0],
        [ 1768,     0,     0],
        [10383,     0,     0],
        [22559,   118,  1634],
        [22559,  2734,     0],
        [  102,     0,     0]])
```

build(*dataset*)

Pretrained: nothing to be done.

classmethod tokenizer(*name*)

Create an instance of tokenizer from either path or name. :param name: path or name of tokenizer.

class diaparser.utils.**ChartField**(*name*, *pad=None*, *unk=None*, *bos=None*, *eos=None*, *lower=False*, *use_vocab=True*, *tokenize=None*, *fn=None*, *mask_token_id=0*)

Field dealing with constituency trees.

This field receives sequences of binarized trees factorized in pre-order, and returns charts filled with labels on each constituent.

Examples

```
>>> sequence = [(0, 5, 'S'), (0, 4, 'S|<>'), (0, 1, 'NP'), (1, 4, 'VP'), (1, 2, 'VP|
↪<>'),
                (2, 4, 'S+VP'), (2, 3, 'VP|<>'), (3, 4, 'NP'), (4, 5, 'S|<>')]
>>> field.transform([sequence])[0]
tensor([[ -1,  37,  -1,  -1, 107,  79],
        [ -1,  -1, 120,  -1, 112,  -1],
        [ -1,  -1,  -1, 120,  86,  -1],
        [ -1,  -1,  -1,  -1,  37,  -1],
        [ -1,  -1,  -1,  -1,  -1, 107],
        [ -1,  -1,  -1,  -1,  -1,  -1]])
```

build(*dataset*, *min_freq=1*)

Constructs a *Vocab* object for this field from the dataset. If the vocabulary has already existed, this function will have no effect.

Parameters

- **dataset** (*Dataset*) – A *Dataset* object. One of the attributes should be named after the name of this field.
- **min_freq** (*int*) – The minimum frequency needed to include a token in the vocabulary. Default: 1.

- **embed** (*Embedding*) – An Embedding object, words in which will be extended to the vocabulary. Default: None.

transform(*sequences*)

Turns a list of sequences that use this field into tensors.

Each sequence is first preprocessed and then numericalized if needed.

Parameters
sequences (*list[list[str]]*) – A list of sequences.

Returns
A list of tensors transformed from the input sequences.

4.4 Transform

class diaparser.utils.**Transform**

A Transform object corresponds to a specific data format. It holds several instances of data fields that provide instructions for preprocessing and numericalizing, etc.

training

Sets the object in training mode. If False, some data fields not required for predictions won't be returned. Default: True.

Type
bool

save(*path*, *sentences*)

path (str of file): file where to write sentences or None to use stdout.

class diaparser.utils.**CoNLL**(*ID=None*, *FORM=None*, *LEMMA=None*, *CPOS=None*, *POS=None*, *FEATS=None*, *HEAD=None*, *DEPREL=None*, *PHEAD=None*, *PDEPREL=None*, *reader=<built-in function open>*)

The CoNLL object holds ten fields required for CoNLL-X data format. Each field can be binded with one or more *Field* objects. For example, FORM can contain both *Field* and *SubwordField* to produce tensors for words and subwords.

ID

Token counter, starting at 1.

FORM

Words in the sentence.

LEMMA

Lemmas or stems (depending on the particular treebank) of words, or underscores if not available.

CPOS

Coarse-grained part-of-speech tags, where the tagset depends on the treebank.

POS

Fine-grained part-of-speech tags, where the tagset depends on the treebank.

FEATS

Unordered set of syntactic and/or morphological features (depending on the particular treebank), or underscores if not available.

HEAD

Heads of the tokens, which are either values of ID or zeros.

DEPREL

Dependency relations to the HEAD.

PHEAD

Projective heads of tokens, which are either values of ID or zeros, or underscores if not available.

PDEPREL

Dependency relations to the PHEAD, or underscores if not available.

References

- Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X Shared Task on Multilingual Dependency Parsing.

classmethod toconll(*tokens*)

Converts a list of tokens to a string in CoNLL-X format. Missing fields are filled with underscores.

Parameters

tokens (*list[str] or list[tuple]*) – This can be either a list of words or word/pos pairs.

Returns

A string in CoNLL-X format.

Examples

```
>>> print(CoNLL.toconll(['She', 'enjoys', 'playing', 'tennis', '.']))
1       She     _       _       _       _       _       _       _       _
2       enjoys  _       _       _       _       _       _       _       _
3       playing _       _       _       _       _       _       _       _
4       tennis  _       _       _       _       _       _       _       _
5       .       _       _       _       _       _       _       _       _
```

classmethod isprojective(*sequence*)

Checks if a dependency tree is projective. This also works for partial annotation.

Besides the obvious crossing arcs, the examples below illustrate two non-projective cases which are hard to detect in the scenario of partial annotation.

Parameters

sequence (*list[int]*) – A list of head indices.

Returns

`True` if the tree is projective, `False` otherwise.

Examples

```
>>> CoNLL.isprojective([2, -1, 1])  # -1 denotes un-annotated cases
False
>>> CoNLL.isprojective([3, -1, 2])
False
```

classmethod istree(*sequence*, *proj=False*, *multiroot=False*)

Checks if the arcs form an valid dependency tree.

Parameters

- **sequence** (*list[int]*) – A list of head indices.
- **proj** (*bool*) – If `True`, requires the tree to be projective. Default: `False`.
- **multiroot** (*bool*) – If `False`, requires the tree to contain only a single root. Default: `True`.

Returns

`True` if the arcs form an valid tree, `False` otherwise.

Examples

```
>>> CoNLL.istree([3, 0, 0, 3], multiroot=True)
True
>>> CoNLL.istree([3, 0, 0, 3], proj=True)
False
```

load(*data*, *proj=False*, *max_len=None*, ***kwargs*)

Loads the data in CoNLL-X format. Also supports for loading data from CoNLL-U file with comments and non-integer IDs.

Parameters

- **data** (*list[list] or str*) – A list of instances or a filename.
- **proj** (*bool*) – If `True`, discards all non-projective sentences. Default: `False`.
- **max_len** (*int*) – Sentences exceeding the length will be discarded. Default: `None`.

Returns

A list of `CoNLLSentence` instances.

class diaparser.utils.**Tree**(*WORD=None*, *POS=None*, *TREE=None*, *CHART=None*)

The Tree object factorize a constituency tree into four fields, each associated with one or more *Field* objects.

WORD

Words in the sentence.

POS

Part-of-speech tags, or underscores if not available.

TREE

The raw constituency tree in `nltk.tree.Tree` format.

CHART

The factorized sequence of binarized tree traversed in pre-order.

classmethod totree(*tokens*, *root=''*)

Converts a list of tokens to a `nltk.tree.Tree`. Missing fields are filled with underscores.

Parameters

- **tokens** (*list[str] or list[tuple]*) – This can be either a list of words or word/pos pairs.
- **root** (*str*) – The root label of the tree. Default: ''.

Returns

A `nltk.tree.Tree` object.

Examples

```
>>> print(Tree.totree(['She', 'enjoys', 'playing', 'tennis', '.'], 'TOP'))
(TOP (_ She) (_ enjoys) (_ playing) (_ tennis) (_ .))
```

classmethod binarize(*tree*)

Conducts binarization over the tree.

First, the tree is transformed to satisfy Chomsky Normal Form (CNF). Here we call `chomsky_normal_form()` to conduct left-binarization. Second, all unary productions in the tree are collapsed.

Parameters

tree (*nltk.tree.Tree*) – The tree to be binarized.

Returns

The binarized tree.

Examples

```
>>> tree = nltk.Tree.fromstring('''
                                (TOP
                                  (S
                                    (NP (_ She))
                                    (VP (_ enjoys) (S (VP (_ playing) (NP (_␣
↪tennis)))))
                                    (_ .)))
                                ''')
>>> print(Tree.binarize(tree))
(TOP
  (S
    (S|<>
      (NP (_ She))
      (VP
        (VP|<> (_ enjoys))
        (S+VP (VP|<> (_ playing)) (NP (_ tennis)))))
    (S|<> (_ .))))
```

classmethod factorize(*tree*, *delete_labels=None*, *equal_labels=None*)

Factorizes the tree into a sequence. The tree is traversed in pre-order.

Parameters

- **tree** (*nltk.tree.Tree*) – The tree to be factorized.
- **delete_labels** (*set[str]*) – A set of labels to be ignored. This is used for evaluation. If it is a pre-terminal label, delete the word along with the brackets. If it is a non-terminal label, just delete the brackets (don't delete childrens). In EVALB, the default set is: {'TOP', 'S1', '-NONE-', ',', ':', '``', "''", '.', '?', '!', ''} Default: None.
- **equal_labels** (*dict[str, str]*) – The key-val pairs in the dict are considered equivalent (non-directional). This is used for evaluation. The default dict defined in EVALB is: {'ADVP': 'PRT'} Default: None.

Returns

The sequence of the factorized tree.

Examples

```
>>> tree = nltk.Tree.fromstring('''
                                (TOP
                                  (S
                                    (NP (_ She))
                                    (VP (_ enjoys) (S (VP (_ playing) (NP (_␣
↪tennis)))))
                                    (_ .)))
                                ''')
>>> Tree.factorize(tree)
[(0, 5, 'TOP'), (0, 5, 'S'), (0, 1, 'NP'), (1, 4, 'VP'), (2, 4, 'S'), (2, 4, 'VP
↪'), (3, 4, 'NP')]
>>> Tree.factorize(tree, delete_labels={'TOP', 'S1', '-NONE-', ',', ':', '``', "
↪''", '.', '?', '!', ''})
[(0, 5, 'S'), (0, 1, 'NP'), (1, 4, 'VP'), (2, 4, 'S'), (2, 4, 'VP'), (3, 4, 'NP
↪')]
```

classmethod build(*tree*, *sequence*)

Builds a constituency tree from the sequence. The sequence is generated in pre-order. During building the tree, the sequence is de-binarized to the original format (i.e., the suffixes |<> are ignored, the collapsed labels are recovered).

Parameters

- **tree** (*nltk.tree.Tree*) – An empty tree that provides a base for building a result tree.
- **sequence** (*list[tuple]*) – A list of tuples used for generating a tree. Each tuple consits of the indices of left/right span boundaries and label of the span.

Returns

A result constituency tree.

Examples

```
>>> tree = Tree.totree(['She', 'enjoys', 'playing', 'tennis', '.'], 'TOP')
>>> sequence = [(0, 5, 'S'), (0, 4, 'S|<>'), (0, 1, 'NP'), (1, 4, 'VP'), (1, 2,
↪'VP|<>'),
                (2, 4, 'S+VP'), (2, 3, 'VP|<>'), (3, 4, 'NP'), (4, 5, 'S|<>')]
>>> print(Tree.build(tree, sequence))
(TOP
  (S
    (NP (_ She))
    (VP (_ enjoys) (S (VP (_ playing) (NP (_ tennis)))))
    (_ .)))
```

load(*data*, *max_len=None*, ***kwargs*)

Parameters

- **data** (*list[list] or str*) – A list of instances or a filename.
- **max_len** (*int*) – Sentences exceeding the length will be discarded. Default: None.

Returns
A list of TreeSentence instances.

4.5 Vocab

class diaparser.utils.**Vocab**(*counter*, *min_freq=1*, *specials=[]*, *unk_index=0*)

Defines a vocabulary object that will be used to numericalize a field.

Parameters

- **counter** (*Counter*) – Counter object holding the frequencies of each value found in the data.
- **min_freq** (*int*) – The minimum frequency needed to include a token in the vocabulary. Default: 1.
- **specials** (*list[str]*) – The list of special tokens (e.g., pad, unk, bos and eos) that will be prepended to the vocabulary. Default: [].
- **unk_index** (*int*) – The index of unk token. Default: 0.

itos

A list of token strings indexed by their numerical identifiers.

stoi

A defaultdict object mapping token strings to numerical identifiers.

CHAPTER

FIVE

TOKENIZER

5.1 Tokenizer

class tokenizer.**Tokenizer**(*lang*, *dir='/home/docs/.cache/diaparser'*, *verbose=True*)

Interface to Stanza tokenizers. Args. lang (str): conventional language identifier. dir (str): directory for caching models. verbose (Bool): print download progress.

format(*sentences*)

Convert sentences to CoNLL format.

reader()

Reading function that returns a generator of CoNLL-U sentences.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

B

BertEmbedding (*class in diaparser.modules*), 11
BertField (*class in diaparser.utils*), 24
Biaffine (*class in diaparser.modules*), 11
BiaffineDependencyModel (*class in diaparser.models*), 7
BiaffineDependencyParser (*class in diaparser.parsers*), 4
binarize() (*diaparser.utils.Tree class method*), 29
build() (*diaparser.parsers.BiaffineDependencyParser class method*), 5
build() (*diaparser.utils.BertField method*), 25
build() (*diaparser.utils.ChartField method*), 25
build() (*diaparser.utils.Field method*), 23
build() (*diaparser.utils.SubwordField method*), 24
build() (*diaparser.utils.Tree class method*), 30

C

CharLSTM (*class in diaparser.modules*), 13
CHART (*diaparser.utils.Tree attribute*), 28
ChartField (*class in diaparser.utils*), 25
chuliu_edmonds() (*in module diaparser.utils.alg*), 18
cky() (*in module diaparser.utils.alg*), 20
compose() (*diaparser.utils.Field method*), 23
CoNLL (*class in diaparser.utils*), 26
CPOS (*diaparser.utils.CoNLL attribute*), 26

D

Dataset (*class in diaparser.utils*), 21
decode() (*diaparser.models.BiaffineDependencyModel method*), 9
DEPREL (*diaparser.utils.CoNLL attribute*), 27

E

eisner() (*in module diaparser.utils.alg*), 19
eisner2o() (*in module diaparser.utils.alg*), 19
evaluate() (*diaparser.parsers.BiaffineDependencyParser method*), 5
extra_repr() (*diaparser.models.BiaffineDependencyModel method*), 8
extra_repr() (*diaparser.modules.Biaffine method*), 11

F

factorize() (*diaparser.utils.Tree class method*), 29
FEATS (*diaparser.utils.CoNLL attribute*), 26
Field (*class in diaparser.utils*), 22
FORM (*diaparser.utils.CoNLL attribute*), 26
format() (*tokenizer.Tokenizer method*), 33
forward() (*diaparser.models.BiaffineDependencyModel method*), 8
forward() (*diaparser.modules.BertEmbedding method*), 12
forward() (*diaparser.modules.Biaffine method*), 11
forward() (*diaparser.modules.CharLSTM method*), 13
forward() (*diaparser.modules.IndependentDropout method*), 14
forward() (*diaparser.modules.LSTM method*), 13
forward() (*diaparser.modules.MLP method*), 15
forward() (*diaparser.modules.ScalarMix method*), 12
forward() (*diaparser.modules.SharedDropout method*), 15

H

HEAD (*diaparser.utils.CoNLL attribute*), 26

I

ID (*diaparser.utils.CoNLL attribute*), 26
IndependentDropout (*class in diaparser.modules*), 14
isprojective() (*diaparser.utils.CoNLL class method*), 27
istree() (*diaparser.utils.CoNLL class method*), 28
itos (*diaparser.utils.Vocab attribute*), 31

K

kmeans() (*in module diaparser.utils.alg*), 17

L

LEMMA (*diaparser.utils.CoNLL attribute*), 26
load() (*diaparser.parsers.Parser class method*), 3
load() (*diaparser.utils.CoNLL method*), 28
load() (*diaparser.utils.Tree method*), 31
loss() (*diaparser.models.BiaffineDependencyModel method*), 9

LSTM (*class in diaparser.modules*), 12

M

MLP (*class in diaparser.modules*), 15
MODEL (*diaparser.parsers.BiaffineDependencyParser attribute*), 4
mst() (*in module diaparser.utils.alg*), 18

P

Parser (*class in diaparser.parsers*), 3
PDEPREL (*diaparser.utils.CoNLL attribute*), 27
PHEAD (*diaparser.utils.CoNLL attribute*), 27
POS (*diaparser.utils.CoNLL attribute*), 26
POS (*diaparser.utils.Tree attribute*), 28
predict() (*diaparser.parsers.BiaffineDependencyParser method*), 5
predict() (*diaparser.parsers.Parser method*), 3
preprocess() (*diaparser.utils.Field method*), 22

R

RawField (*class in diaparser.utils*), 22
reader() (*tokenizer.Tokenizer method*), 33

S

save() (*diaparser.utils.Transform method*), 26
ScalarMix (*class in diaparser.modules*), 12
sentences (*diaparser.utils.Dataset attribute*), 21
SharedDropout (*class in diaparser.modules*), 14
stoi (*diaparser.utils.Vocab attribute*), 31
SubwordField (*class in diaparser.utils*), 23

T

tarjan() (*in module diaparser.utils.alg*), 17
toconll() (*diaparser.utils.CoNLL class method*), 27
Tokenizer (*class in tokenizer*), 33
tokenizer() (*diaparser.utils.BertField class method*), 25
totree() (*diaparser.utils.Tree class method*), 28
train() (*diaparser.parsers.BiaffineDependencyParser method*), 4
train() (*diaparser.parsers.Parser method*), 3
training (*diaparser.utils.Transform attribute*), 26
Transform (*class in diaparser.utils*), 26
transform (*diaparser.utils.Dataset attribute*), 21
transform() (*diaparser.utils.ChartField method*), 26
transform() (*diaparser.utils.Field method*), 23
transform() (*diaparser.utils.SubwordField method*), 24
Tree (*class in diaparser.utils*), 28
TREE (*diaparser.utils.Tree attribute*), 28

V

Vocab (*class in diaparser.utils*), 31

W

WORD (*diaparser.utils.Tree attribute*), 28