putput Documentation

Michael Perel, Arman Rahman

Oct 28, 2019

Contents

CHAPTER 1

putput package

1.1 Subpackages

1.1.1 putput.presets package

1.1.1.1 Submodules

1.1.1.2 putput.presets.displaCy module

putput.presets.displaCy.**preset**() → Callable

Configures the Pipeline for the 'DISPLACY' ENT format.

The ENT format: https://spacy.io/usage/visualizers#manual-usage

Returns A Callable that when called returns parameters for instantiating a Pipeline. This Callable can be passed into putput.Pipeline as the 'preset' argument.

Examples

```
>>> import json
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> pattern_def_path = Path(__file__).parent.parent.parent / 'tests' / 'doc' /
↪'example_pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset(preset(),
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_patterns_
↪map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for token_visualizer, group_visualizer in generator:
...     print(json.dumps(token_visualizer, sort_keys=True))
```

(continues on next page)

(continued from previous page)

```
...     print(json.dumps(group_visualizer, sort_keys=True))
...     break
{"ents": [{"end": 11, "label": "ADD", "start": 0},
        {"end": 17, "label": "ITEM", "start": 12},
        {"end": 29, "label": "ADD", "start": 18},
        {"end": 35, "label": "ITEM", "start": 30},
        {"end": 39, "label": "CONJUNCTION", "start": 36},
        {"end": 45, "label": "ITEM", "start": 40}],
"text": "can she get fries can she get fries and fries",
"title": "Tokens"}
{"ents": [{"end": 17, "label": "ADD_ITEM", "start": 0},
        {"end": 35, "label": "ADD_ITEM", "start": 18},
        {"end": 39, "label": "None", "start": 36},
        {"end": 45, "label": "None", "start": 40}],
        "text": "can she get fries can she get fries and fries",
        "title": "Groups"}
```

1.1.1.3 putput.presets.factory module

putput.presets.factory.**get_preset**(*preset: str*) → Callable

A factory that gets a 'preset' Callable.

Parameters **preset** – the preset's name.

Returns The return value of calling a preset's 'preset' function without arguments.

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> pattern_def_path = Path(__file__).parent.parent.parent / 'tests' / 'doc' /
↪'example_pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset('IOB2',
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_patterns_
↪map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
...     break
can she get fries can she get fries and fries
('B-ADD I-ADD I-ADD', 'B-ITEM', 'B-ADD I-ADD I-ADD', 'B-ITEM', 'B-CONJUNCTION',
↪'B-ITEM')
('B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM I-ADD_ITEM', 'B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM␣
↪I-ADD_ITEM',
 'B-None', 'B-None')
```

1.1.1.4 putput.presets.iob2 module

putput.presets.iob2.**preset**(*, *tokens_to_include: Optional[Sequence[str]] = None*, *tokens_to_exclude: Optional[Sequence[str]] = None*, *groups_to_include: Optional[Sequence[str]] = None*, *groups_to_exclude: Optional[Sequence[str]] = None*) → Callable

Configures the Pipeline for 'IOB2' format.

Adheres to IOB2: https://en.wikipedia.org/wiki/Inside%E2%80%93outside%E2%80%93beginning_(tagging).

This function should be used as the 'preset' argument of putput.Pipeline instead of the 'IOB2' str to specify which tokens and groups map to 'O'.

Parameters

- **tokens_to_include** – A sequence of tokens that should not be mapped to 'O'. Useful if the majority of tokens should be excluded. Cannot be used in conjunction with 'tokens_to_exclude'.
- **tokens_to_exclude** – A sequence of tokens that should map to 'O'. Useful if the majority of tokens should be included. Cannot be used in conjunction with 'tokens_to_include'.
- **groups_to_include** – A sequence of groups that should not be mapped to 'O'. Useful if the majority of groups should be excluded. Cannot be used in conjunction with 'groups_to_exclude'.
- **groups_to_exclude** – A sequence of groups that should map to 'O'. Useful if the majority of groups should be included. Cannot be used in conjunction with 'groups_to_include'.

Returns A Callable that when called returns parameters for instantiating a Pipeline. This Callable can be passed into putput.Pipeline as the 'preset' argument.

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> pattern_def_path = Path(__file__).parent.parent.parent / 'tests' / 'doc' /
↪'example_pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset(preset(tokens_to_include=('ITEM',), groups_to_
↪include=('ADD_ITEM',)),
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_patterns_
↪map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
...     break
can she get fries can she get fries and fries
('O O O', 'B-ITEM', 'O O O', 'B-ITEM', 'O', 'B-ITEM')
('B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM I-ADD_ITEM', 'B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM␣
↪I-ADD_ITEM', 'O', 'O')
```

1.1.1.5 putput.presets.luis module

putput.presets.luis.**preset**(*, *intent_map: Mapping[str, str] = None*, *entities: Optional[Sequence[str]] = None*) → Callable

Configures the Pipeline for LUIS test format.

Adheres to: https://docs.microsoft.com/en-us/azure/cognitive-services/luis/luis-tutorial-batch-testing.

This function should be used as the 'preset' argument of putput.Pipeline instead of the 'LUIS' str to specify intents and entities.

Examples

```
>>> import json
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> from pprint import pprint
>>> import random
>>> random.seed(0)
>>> pattern_folder = Path(__file__).parent.parent.parent / 'tests' / 'doc'
>>> pattern_def_path = pattern_folder / 'example_pattern_definition_with_intents.
↪yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset('LUIS',
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_patterns_
↪map)
>>> for luis_result in p.flow(disable_progress_bar=True):
...     print(json.dumps(luis_result, sort_keys=True))
...     break
{"entities": [{"endPos": 16, "entity": "ITEM", "startPos": 12},
              {"endPos": 34, "entity": "ITEM", "startPos": 30},
              {"endPos": 44, "entity": "ITEM", "startPos": 40}],
 "intent": "ADD_INTENT",
 "text": "can she get fries can she get fries and fries"}
```

Parameters

- **intent_map** – A mapping from an utterance pattern string to a single intent. The value '__DISCARD' is reserved.
- **entities** – A sequence of tokens that are considered entities. To make all tokens entities, give a list with only the value '__ALL'. E.g. entities=['_ALL']

Returns A Callable that when called returns parameters for instantiating a Pipeline. This Callable can be passed into putput.Pipeline as the 'preset' argument.

1.1.1.6 putput.presets.stochastic module

putput.presets.stochastic.**preset**(*, *chance: int = 20*) → Callable

Randomly replaces words with synonyms from wordnet synsets.

Tags each word in the utterance with nltk's part of speech tagger. Using the part of speech, each word in the utterance is replaced with a randomly chosen word from the first synset with the same part of speech as the word to replace, subject to the specified chance. If no synset exists with the same part of speech, the original word will not be replaced.

Downloads nltk's wordnet, punkt, and averaged_perceptron_tagger if non-existent on the host.

Parameters **chance** – The chance between [0, 100] for each word to be replaced by a synonym.

Returns A Callable that when called returns parameters for instantiating a Pipeline. This Callable can be passed into putput.Pipeline as the 'preset' argument.

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> pattern_def_path = Path(__file__).parent.parent.parent / 'tests' / 'doc' /
↪'example_pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset(preset(chance=100),
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_patterns_
↪map,
...                          seed=0)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
...     break
can she acquire chips can she acquire french-fried_potatoes and french_fries
('[ADD(can she acquire)]', '[ITEM(chips)]',
 '[ADD(can she acquire)]', '[ITEM(french-fried_potatoes)]',
 '[CONJUNCTION(and)]', '[ITEM(french_fries)]')
('{[ADD(can she acquire)] [ITEM(chips)]}',
 '{[ADD(can she acquire)] [ITEM(french-fried_potatoes)]}',
 '{[CONJUNCTION(and)]}', '{[ITEM(french_fries)]}')
```

1.1.1.7 Module contents

1.2 Submodules

1.3 putput.combiner module

putput.combiner.**combine**(*utterance_combo: Sequence[Sequence[str]], tokens: Sequence[str], groups: Sequence[Tuple[str, int]], *, token_handler_map: Optional[Mapping[str, Callable[[str, str], str]]] = None, group_handler_map: Optional[Mapping[str, Callable[[str, Sequence[str]], str]]] = None, combo_options: Optional[putput.joiner.ComboOptions] = None*) → Tuple[int, Iterable[Tuple[str, Sequence[str], Sequence[str]]]]

Generates an utterance, handled tokens, and handled groups.

Parameters

- **utterance_combo** – An utterance_combo from pipeline.expander.expand.
- **tokens** – Tokens that have yet to be handled from pipeline.expander.expand.
- **groups** – Groups that have yet to be handled from pipeline.expander.expand.

- **token_handler_map** – A mapping between a token and a function with args (token, phrase generated by token) that returns a handled token. If 'DEFAULT' is specified as the token, the handler will apply to all tokens not otherwise specified in the mapping.
- **combo_options** – Options for randomly sampling the combination of 'utterance_combo'.

Returns The length of the Iterable, and the Iterable consisting of an utterance and handled tokens.

Examples

```
>>> def _iob_token_handler(token: str, phrase: str) -> str:
...     tokens = ['{}-{}'.format('B' if i == 0 else 'I', token)
...               for i, _ in enumerate(phrase.replace(" '", "'").split())]
...     return ' '.join(tokens)
>>> def _just_groups(group_name: str, _: Sequence[str]) -> str:
...     return '[{}]'.format(group_name)
>>> token_handler_map = {'DEFAULT': _iob_token_handler}
>>> group_handler_map = {'DEFAULT': _just_groups}
>>> combo_options = ComboOptions(max_sample_size=2, with_replacement=False)
>>> utterance_combo = (('can she get', 'may she get'), ('fries',))
>>> tokens = ('ADD', 'ITEM')
>>> groups = (('ADD_ITEM', 2),)
>>> sample_size, generator = combine(utterance_combo,
...                                  tokens,
...                                  groups,
...                                  token_handler_map=token_handler_map,
...                                  group_handler_map=group_handler_map,
...                                  combo_options=combo_options)
>>> sample_size
2
>>> for utterance, handled_tokens, handled_groups in generator:
...     print(utterance)
...     print(handled_tokens)
...     print(handled_groups)
can she get fries
('B-ADD I-ADD I-ADD', 'B-ITEM')
('[ADD_ITEM]',)
may she get fries
('B-ADD I-ADD I-ADD', 'B-ITEM')
('[ADD_ITEM]',)
```

1.4 putput.expander module

putput.expander.**expand**(*pattern_def: Mapping[KT, VT_co], *, dynamic_token_patterns_map: Optional[Mapping[str, Sequence[str]]] = None*) → Tuple[int, Iterable[Tuple[Sequence[Sequence[str]], Sequence[str], Sequence[Tuple[str, int]]]]]

Expands the pattern_def to prepare for combination.

Parameters

- **pattern_def** – A dictionary representation of the pattern definition.
- **dynamic_token_patterns_map** – The 'dynamic' counterpart to the 'static' section in the pattern definition. This mapping between token and token patterns is useful in scenarios

where tokens and token patterns cannot be known before runtime.

Returns The length of the Iterable, and the Iterable consisting of an utterance_combo, tokens (that have yet to be handled), and groups (that have yet to be handled).

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import _load_pattern_def
>>> pattern_def_path = Path(__file__).parent.parent / 'tests' / 'doc' / 'example_
↪pattern_definition.yml'
>>> pattern_def = _load_pattern_def(pattern_def_path)
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> num_utterance_patterns, generator = expand(pattern_def,
...                                            dynamic_token_patterns_map=dynamic_
↪token_patterns_map)
>>> num_utterance_patterns
1
>>> for utterance_combo, unhandled_tokens, unhandled_groups in generator:
...     print(utterance_combo)
...     print(unhandled_tokens)
...     print(unhandled_groups)
(('can she get', 'may she get'), ('fries',), ('can she get', 'may she get'), (
↪'fries',), ('and',), ('fries',))
('ADD', 'ITEM', 'ADD', 'ITEM', 'CONJUNCTION', 'ITEM')
(('ADD_ITEM', 2), ('ADD_ITEM', 2), ('None', 1), ('None', 1))
```

putput.expander.**expand_utterance_patterns_ranges_and_groups**(*utterance_patterns: Sequence[Sequence[str]]*, *group_map: Mapping[str, Sequence[str]]*) → Tuple[Sequence[Sequence[str]], Sequence[Sequence[Tuple[str, int]]]]

Expands ranges and groups in utterance patterns, ensuring each utterance pattern is unique.

Parameters

- **utterance_patterns** – utterance_patterns section of pattern_def.
- **group_map** – A mapping between a group name and the tokens that make up the group.

Returns A tuple of utterance patterns with group names and ranges replaced by tokens, and groups which are tuples of (group_name, number of tokens the group spans).

Examples

```
>>> utterance_patterns = [['WAKE', 'PLAY_ARTIST', '1-2']]
>>> group_map = {'PLAY_ARTIST': ('PLAY', 'ARTIST')}
>>> patterns, groups = expand_utterance_patterns_ranges_and_groups(utterance_
↪patterns, group_map)
>>> tuple(sorted(patterns, key=lambda item: len(item)))
(('WAKE', 'PLAY', 'ARTIST'), ('WAKE', 'PLAY', 'ARTIST', 'PLAY', 'ARTIST'))
```

(continues on next page)

(continued from previous page)

```
>>> tuple(sorted(groups, key=lambda item: len(item)))
((('None', 1), ('PLAY_ARTIST', 2)), (('None', 1), ('PLAY_ARTIST', 2), ('PLAY_
↪ARTIST', 2)))
```

putput.expander.**get_base_item_map**(*pattern_def: Mapping[KT, VT_co]*, *base_key: str*) → Mapping[str, Sequence[str]]

Returns base item map or an empty dictionary if one does not exist.

Parameters

- **pattern_def** – A dictionary representation of the pattern definition.
- **base_key** – Key in pattern_def corresponding to the base item map.

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import _load_pattern_def
>>> pattern_def_path = Path(__file__).parent.parent / 'tests' / 'doc' / 'example_
↪pattern_definition.yml'
>>> pattern_def = _load_pattern_def(pattern_def_path)
>>> get_base_item_map(pattern_def, 'groups')
{'ADD_ITEM': ('ADD', 'ITEM')}
>>> get_base_item_map(pattern_def, 'base_tokens')
{'PRONOUNS': ('she',)}
>>> get_base_item_map(pattern_def, 'not_a_key')
{}
```

1.5 putput.joiner module

class putput.joiner.**ComboOptions**(*, *max_sample_size: int*, *with_replacement: bool*)

Bases: object

Options for join_combo via random sampling.

max_sample_size
Ceiling for number of components to sample.

with_replacement
Option to include duplicates when randomly sampling. If True, will sample max_sample_size. If False, will sample up to 'max_sample_size' unique combinations.

Raises ValueError – If max_sample_size <= 0.

max_sample_size
Ceiling for number of components to sample.

with_replacement
Option to include duplicates when randomly sampling.

putput.joiner.**join_combo**(*combo: Sequence[Sequence[T]]*, ***, *combo_options: Optional[putput.joiner.ComboOptions] = None*) → Iterable[Sequence[T]]

Generates the product of a combo, subject to 'combo_options'.

If 'combo_options' is not specified, 'join_combo' returns an Iterable of the product of combo. If 'combo_options' is specified, 'join_combo' returns an Iterable of samples of the product of combo.

Sampling should be used to speed up the consumption of the returned Iterable as well as to control size of the product, especially in cases where oversampling/undersampling is desired.

Parameters

- **combo** – Sequences to join.
- **combo_options** – Options for randomly sampling.

Yields A joined combo.

Examples

```
>>> random.seed(0)
>>> combo = (('hey', 'ok'), ('speaker', 'sound system'), ('play',))
>>> tuple(join_combo(combo))
(('hey', 'speaker', 'play'), ('hey', 'sound system', 'play'),
('ok', 'speaker', 'play'), ('ok', 'sound system', 'play'))
>>> combo_options = ComboOptions(max_sample_size=1, with_replacement=False)
>>> tuple(join_combo(combo, combo_options=combo_options))
(('ok', 'sound system', 'play'),)
```

1.6 putput.logger module

putput.logger.**get_logger**(*module_name: str*, ***, *level: int = 20*, *stream: IO[str] = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*) → logging.Logger

Returns a configured logger for the module.

Parameters

- **module_name** – __name__ for the calling module.
- **level** – Minimum logging level. Messages with this level or higher will be shown.
- **stream** – 'stream' argument to logging.StreamHandler, typically sys.stdout or sys.stderr.

Raises `ValueError` – If stream is not 'stderr' or 'stdout'.

1.7 putput.pipeline module

class putput.pipeline.**Pipeline**(*pattern_def_path: pathlib.Path, *, dynamic_token_patterns_map: Optional[Mapping[str, Sequence[str]]] = None, token_handler_map: Optional[Mapping[str, Callable[[str, str], str]]] = None, group_handler_map: Optional[Mapping[str, Callable[[str, Sequence[str]], str]]] = None, expansion_hooks_map: Optional[Mapping[str, Sequence[Callable[[Sequence[Sequence[str]], Sequence[str], Sequence[Tuple[str, int]]], Tuple[Sequence[Sequence[str]], Sequence[str], Sequence[Tuple[str, int]]]]]]] = None, combo_hooks_map: Optional[Mapping[str, Sequence[Callable]]] = None, combo_options_map: Optional[Mapping[str, putput.joiner.ComboOptions]] = None, seed: Optional[int] = None*)

Bases: object

Transforms a pattern definition into labeled data.

To perform this transformation, initialize 'Pipeline' and call 'flow'.

There are two ways to initialize 'Pipeline': by passing in desired arguments or through the use of a 'preset' in the method 'from_preset'. 'Presets' instantiate the 'Pipeline' with arguments that cover common use cases. As these arguments become attributes that the user can modify, using a 'preset' does not give up customizability.

Once 'Pipeline' has been initialized, calling the method 'flow' will cause labeled data to flow through 'Pipeline' to the user.

There are two stages in 'flow'. The first stage, 'expansion', expands the pattern definition file into an 'utterance_combo', 'tokens', and 'groups' for each utterance pattern. At the end of the first stage, if hooks in 'expansion_hooks_map' are specified for the current utterance pattern, they are applied in order where the output of a previous hook becomes the input to the next hook.

The second stage, 'combination', yields a sequence of 'utterance', 'handled_tokens', and 'handled_groups'. This stage applies handlers from 'token_handler_map' and 'group_handler_map' and is subject to constraints specified in 'combo_options_map'. At the end of the second stage, if hooks in 'combo_hooks_map' are specified for the current 'utterance_pattern', they are applied in order where the output of a previous hook becomes the input to the next hook.

Examples

Default behavior

```
>>> pattern_def_path = Path(__file__).parent.parent / 'tests' / 'doc' / 'example_
↪pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline(pattern_def_path, dynamic_token_patterns_map=dynamic_token_
↪patterns_map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
can she get fries can she get fries and fries
('[ADD(can she get)]', '[ITEM(fries)]', '[ADD(can she get)]', '[ITEM(fries)]',
↪'[CONJUNCTION(and)]',
```

(continues on next page)

(continued from previous page)

```
'[ITEM(fries)]')
('{ADD_ITEM([ADD(can she get)] [ITEM(fries)])}', '{ADD_ITEM([ADD(can she get)]␣
↪[ITEM(fries)])}',
'{None([CONJUNCTION(and)])}', '{None([ITEM(fries)])}')
can she get fries may she get fries and fries
('[ADD(can she get)]', '[ITEM(fries)]', '[ADD(may she get)]', '[ITEM(fries)]',
↪'[CONJUNCTION(and)]',
'[ITEM(fries)]')
('{ADD_ITEM([ADD(can she get)] [ITEM(fries)])}', '{ADD_ITEM([ADD(may she get)]␣
↪[ITEM(fries)])}',
'{None([CONJUNCTION(and)])}', '{None([ITEM(fries)])}')
may she get fries can she get fries and fries
('[ADD(may she get)]', '[ITEM(fries)]', '[ADD(can she get)]', '[ITEM(fries)]',
↪'[CONJUNCTION(and)]',
'[ITEM(fries)]')
('{ADD_ITEM([ADD(may she get)] [ITEM(fries)])}', '{ADD_ITEM([ADD(can she get)]␣
↪[ITEM(fries)])}',
'{None([CONJUNCTION(and)])}', '{None([ITEM(fries)])}')
may she get fries may she get fries and fries
('[ADD(may she get)]', '[ITEM(fries)]', '[ADD(may she get)]', '[ITEM(fries)]',
↪'[CONJUNCTION(and)]',
'[ITEM(fries)]')
('{ADD_ITEM([ADD(may she get)] [ITEM(fries)])}', '{ADD_ITEM([ADD(may she get)]␣
↪[ITEM(fries)])}',
'{None([CONJUNCTION(and)])}', '{None([ITEM(fries)])}')
```

With arguments

```
>>> import json
>>> import random
>>> def _just_tokens(token: str, _: str) -> str:
...     return '[{token}]'.format(token=token)
>>> def _just_groups(group_name: str, _: Sequence[str]) -> str:
...     return '[{group_name}]'.format(group_name=group_name)
>>> def _add_random_words(utterance: str,
...                       handled_tokens: Sequence[str],
...                       handled_groups: Sequence[str]
...                       ) -> Tuple[str, Sequence[str], Sequence[str]]:
...     utterances = utterance.split()
...     random_words = ['hmmmm', 'uh', 'um', 'please']
...     insert_index = random.randint(0, len(utterances))
...     random_word = random.choice(random_words)
...     utterances.insert(insert_index, random_word)
...     utterance = ' '.join(utterances)
...     return utterance, handled_tokens, handled_groups
>>> def _jsonify(utterance: str,
...              handled_tokens: Sequence[str],
...              handled_groups: Sequence[str]
...              ) -> str:
...     return json.dumps(dict(utterance=utterance,
...                            handled_tokens=handled_tokens,
...                            handled_groups=handled_groups),
...                       sort_keys=True)
>>> def _sample_utterance_combo(utterance_combo: Sequence[Sequence[str]],
...                             tokens: Sequence[str],
...                             groups: Sequence[Tuple[str, int]]
...                             ) -> Tuple[Sequence[Sequence[str]], Sequence[str],
↪ Sequence[Tuple[str, int]]]:
```

(continues on next page)

(continued from previous page)

```
...         TOKEN_INDEX = tokens.index('ADD')
...         utterance_combo_list = list(utterance_combo)
...         sampled_combos = tuple(random.sample(utterance_combo_list.pop(TOKEN_
↪INDEX), 1))
...         utterance_combo_list.insert(TOKEN_INDEX, sampled_combos)
...         utterance_combo = tuple(utterance_combo_list)
...         return utterance_combo, tokens, groups
>>> token_handler_map = {'ITEM': _just_tokens}
>>> group_handler_map = {'ADD_ITEM': _just_groups}
>>> expansion_hooks_map = {'ADD_ITEM, 2, CONJUNCTION, ITEM': (_sample_utterance_
↪combo,)}
>>> combo_hooks_map = {'ADD_ITEM, 2, CONJUNCTION, ITEM': (_add_random_words, _add_
↪random_words, _jsonify),
...                    'DEFAULT': (_jsonify,)}
>>> combo_options_map = {'DEFAULT': ComboOptions(max_sample_size=2, with_
↪replacement=False)}
>>> p = Pipeline(pattern_def_path,
...              dynamic_token_patterns_map=dynamic_token_patterns_map,
...              token_handler_map=token_handler_map,
...              group_handler_map=group_handler_map,
...              expansion_hooks_map=expansion_hooks_map,
...              combo_hooks_map=combo_hooks_map,
...              combo_options_map=combo_options_map,
...              seed=0)
>>> for json_result in p.flow(disable_progress_bar=True):
...     print(json_result)
{"handled_groups": ["[ADD_ITEM]", "[ADD_ITEM]", "{None([CONJUNCTION(and)])}", "
↪{None([ITEM])}"],
 "handled_tokens": ["[ADD(may she get)]", "[ITEM]", "[ADD(can she get)]", "[ITEM]
↪", "[CONJUNCTION(and)]",
                    "[ITEM]"],
 "utterance": "may she get fries please can she hmmmm get fries and fries"}
{"handled_groups": ["[ADD_ITEM]", "[ADD_ITEM]", "{None([CONJUNCTION(and)])}", "
↪{None([ITEM])}"],
 "handled_tokens": ["[ADD(may she get)]", "[ITEM]", "[ADD(may she get)]", "[ITEM]
↪", "[CONJUNCTION(and)]",
                    "[ITEM]"],
 "utterance": "may she get fries may she um um get fries and fries"}
```

With a preset

```
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset('IOB2',
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_patterns_
↪map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
...     break
can she get fries can she get fries and fries
('B-ADD I-ADD I-ADD', 'B-ITEM', 'B-ADD I-ADD I-ADD', 'B-ITEM', 'B-CONJUNCTION',
↪'B-ITEM')
('B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM I-ADD_ITEM', 'B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM␣
↪I-ADD_ITEM', 'B-None',
```

(continues on next page)

(continued from previous page)

```
'B-None')
```

combo_hooks_map
A mapping between an utterance pattern and hooks to apply after the combination phase. If 'DEFAULT' is specified as the utterance pattern, the hooks will apply to all utterance patterns not otherwise specified in the mapping. During, 'flow', hooks are applied in order where the output of the previous hook becomes the input to the next hook.

combo_options_map
A mapping between an utterance pattern and ComboOptions to apply during the combination phase. If 'DEFAULT' is specified as the utterance pattern, the options will apply to all utterance patterns not otherwise specified in the mapping.

dynamic_token_patterns_map
The dynamic counterpart to the static section in the pattern definition. This mapping between token and token patterns is useful in scenarios where tokens and token patterns cannot be known before runtime.

expansion_hooks_map
A mapping between an utterance pattern and hooks to apply after the expansion phase. If 'DEFAULT' is specified as the utterance pattern, the hooks will apply to all utterance patterns not otherwise specified in the mapping. During, 'flow', hooks are applied in order where the output of the previous hook becomes the input to the next hook.

flow(*, *disable_progress_bar: bool = False*) → Iterable[T_co]
Generates labeled data one utterance at a time.

Parameters **disable_progress_bar** – Option to display progress of expansion and combination stages as the Iterable is consumed.

Yields Labeled data.

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> pattern_def_path = Path(__file__).parent.parent / 'tests' / 'doc' /
↪'example_pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline(pattern_def_path, dynamic_token_patterns_map=dynamic_token_
↪patterns_map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
...     break
can she get fries can she get fries and fries
('[ADD(can she get)]', '[ITEM(fries)]', '[ADD(can she get)]', '[ITEM(fries)]',
'[CONJUNCTION(and)]', '[ITEM(fries)]')
('{ADD_ITEM([ADD(can she get)] [ITEM(fries)])}', '{ADD_ITEM([ADD(can she␣
↪get)] [ITEM(fries)])}',
'{None([CONJUNCTION(and)])}', '{None([ITEM(fries)])}')
```

classmethod from_preset(*preset: Union[str, Callable, Sequence[Union[str, Callable]]], *args, **kwargs*) → T_PIPELINE
Instantiates 'Pipeline' from a preset configuration.

There are two ways to use 'from_preset'. The simplest way is to use the preset's name. However, presets may have optional arguments that allow for more control. In that case, use a call to the preset's method, 'preset', with the desired arguments.

Parameters

- **preset** – A str that is the preset's name, a Callable that is the result of calling the preset's 'preset' function, or a Sequence of the two. The Callable form allows more control over the preset's behavior. If a Sequence is specified, the result of calling the presets' 'preset' function may only overlap in 'combo_hooks_map' and 'expansion_hooks_map'. If there is overlap, functions will be applied in the order of the Sequence.
- **args** – See __init__ docstring.
- **kwargs** – See __init__ docstring.

Raises `ValueError` – If presets or kwargs contain the same keys, and those keys are not 'combo_hooks_map' or 'expansion_hooks_map'.

Returns An instance of Pipeline.

Examples

Preset str

```
>>> from pathlib import Path
>>> from putput.pipeline import Pipeline
>>> pattern_def_path = Path(__file__).parent.parent / 'tests' / 'doc' /
↪'example_pattern_definition.yml'
>>> dynamic_token_patterns_map = {'ITEM': ('fries',)}
>>> p = Pipeline.from_preset('IOB2',
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_
↪patterns_map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
...     print(tokens)
...     print(groups)
...     break
can she get fries can she get fries and fries
('B-ADD I-ADD I-ADD', 'B-ITEM', 'B-ADD I-ADD I-ADD', 'B-ITEM', 'B-CONJUNCTION
↪', 'B-ITEM')
('B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM I-ADD_ITEM', 'B-ADD_ITEM I-ADD_ITEM I-ADD_
↪ITEM I-ADD_ITEM',
'B-None', 'B-None')
```

Preset function with arguments

```
>>> from putput.presets import iob2
>>> p = Pipeline.from_preset(iob2.preset(tokens_to_include=('ITEM',), groups_
↪to_include=('ADD_ITEM',)),
...                          pattern_def_path,
...                          dynamic_token_patterns_map=dynamic_token_
↪patterns_map)
>>> generator = p.flow(disable_progress_bar=True)
>>> for utterance, tokens, groups in generator:
...     print(utterance)
```

(continues on next page)

(continued from previous page)

```
...     print(tokens)
...     print(groups)
...     break
can she get fries can she get fries and fries
('O O O', 'B-ITEM', 'O O O', 'B-ITEM', 'O', 'B-ITEM')
('B-ADD_ITEM I-ADD_ITEM I-ADD_ITEM I-ADD_ITEM', 'B-ADD_ITEM I-ADD_ITEM I-ADD_
↪ITEM I-ADD_ITEM', 'O', 'O')
```

group_handler_map
A mapping between a group name and a function with args (group name, handled tokens) that returns a handled group. If 'DEFAULT' is specified as the group name, the handler will apply to all groups not otherwise specified in the mapping.

pattern_def_path
Read-only path to the pattern definition.

seed
Seed to control random behavior for Pipeline.

token_handler_map
A mapping between a token and a function with args (token, phrase to tokenize) that returns a handled token. If 'DEFAULT' is specified as the token, the handler will apply to all tokens not otherwise specified in the mapping.

1.8 putput.validator module

exception putput.validator.**PatternDefinitionValidationError**
Bases: Exception

Exception that describes an invalid pattern defintion.

putput.validator.**validate_pattern_def**(*pattern_def: Mapping[KT, VT_co]*) → None
Ensures the pattern definition is defined properly.

Parameters **pattern_def** – A dictionary representation of the pattern definition.

Raises *PatternDefinitionValidationError* – If the pattern definition file is invalid.

Examples

```
>>> from pathlib import Path
>>> from putput.pipeline import _load_pattern_def
>>> pattern_def_path = Path(__file__).parent.parent / 'tests' / 'doc' / 'example_
↪pattern_definition.yml'
>>> pattern_def = _load_pattern_def(pattern_def_path)
>>> validate_pattern_def(pattern_def)
```

1.9 Module contents

Package settings for putput.

CHAPTER 2

About

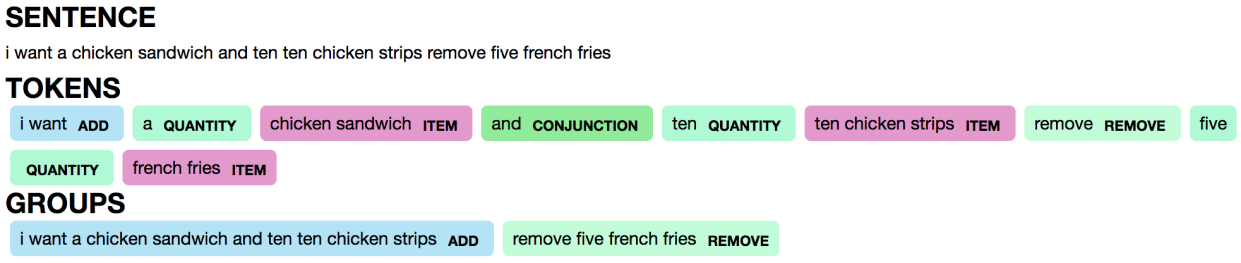
`putput` is a library that generates labeled data for chatbots. It's simple to use, highly customizable, and can handle big data generation on a consumer grade laptop. `putput` takes minutes to setup and seconds to generate millions of labeled data points.

`putput`'s labeled data could be used to:

- train a ML model when you do not have real data.
- augment training specific patterns in a ML model.
- test existing ML models for specific patterns.

`putput` provides an API to its `Pipeline` that specifies how to generate labeled data. It ships with presets that configure the `Pipeline` for common NLU providers such as LUIS and spaCy. `putput` excels at generating custom datasets, even for problems that have yet to be solved commercially and for which no publicly available datasets exist. For instance, checkout this jupyter notebook that uses `putput` to generate a dataset for **multi-intent** recognition and trains a LSTM with Keras to recognize multi-intent and entities.

Here is an example prediction from the LSTM trained with `putput` data:



Note that the trained LSTM can deal with real life complexity such as handling multi-intent ("add" and "remove" `groups`) and disambiguating between the same word in different contexts (the quantity "ten" vs. "ten" in the item "ten chicken strips").

CHAPTER 3

Installation

`putput` currently supports python >= 3.5. To install the production release, execute `pip install putput`.

CHAPTER 4

Samples

`putput` ships with several dockerized samples that show how to generate data.

- Clone the repo: `git clone https://github.com/michaelperel/putput.git`
- Move into the project directory: `cd putput`
- Ensure docker is running: `docker --version`
- Build the runtime environment: `docker build -t putput .`
- The project ships with several usage samples which you can execute, for example: `docker run putput smart_speaker` or `docker run putput restaurant`.

`putput` also ships with annotated jupyter notebooks in the `samples/` directory that use `putput` to solve real world NLU problems. Note: Github cannot correctly render certain graphics, so the notebooks should be viewed on nbviewer.

CHAPTER 5

Development

There are various checks that Travis (our CI server) executes to ensure code quality.
You can also run the checks locally:

1. Install the development dependencies via: `pip install -e .[dev]`
2. Run the linter: `python setup.py pylint`
3. Run the type checker: `python setup.py mypy`
4. Run the tests: `python setup.py test`

Alternatively, you can run all the steps via Docker: `docker build --target=build -t putput .`

CHAPTER 6

Usage

`putput` is a pipeline that works by reshaping the `pattern definition`, a user defined yaml file of patterns, into labeled data.

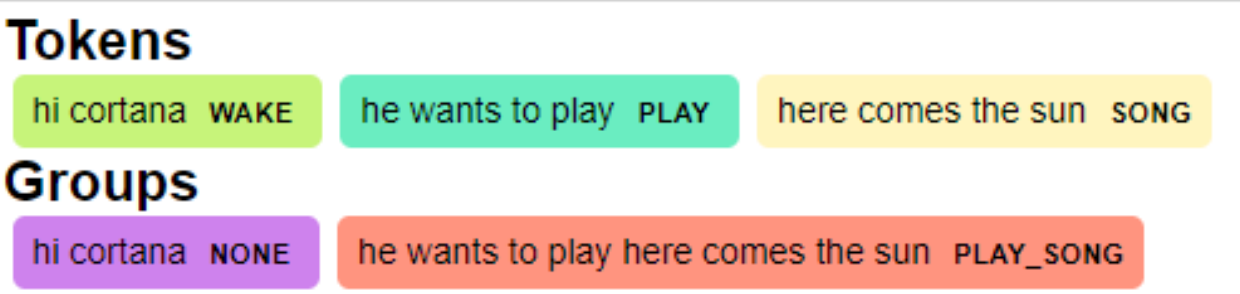
6.1 Example

Here is an example of a `pattern definition` that generates labeled data for a smart speaker.

```
base_tokens:
  - PERSONAL_PRONOUNS: [he, she]
  - SPEAKER: [cortana, siri, alexa, google]
token_patterns:
  - static:
    - WAKE:
      - [[hi, hey], SPEAKER]
    - PLAY:
      - [PERSONAL_PRONOUNS, [wants, would like], [to], [play]]
      - [[play]]
  - dynamic:
    - ARTIST
    - SONG
groups:
  - PLAY_SONG: [PLAY, SONG]
  - PLAY_ARTIST: [PLAY, ARTIST]
utterance_patterns:
  - [WAKE, PLAY_SONG]
  - [WAKE, PLAY_ARTIST]
  - [WAKE, 1-2, PLAY_SONG]
```

Focusing on the first `utterance_pattern`, `[WAKE, PLAY_SONG]`, `putput` would generate hundreds of `utterances`, `tokens`, and `groups` of the form:

`utterance` - hi cortana he wants to play here comes the sun



6.2 Pattern definition reference

In the `pattern definition`, the two most important sections are `token_patterns` and `utterance_patterns`. A `token_pattern` describes a sequence of components whose product constitutes a `token`. For instance, the sole `token_pattern` for the `WAKE token` is `[[hi, hey], [cortana, siri, alexa, google]]` (the `base_token`, `SPEAKER`, is replaced with its value `[cortana, siri, alexa, google]` at runtime). The product of this `token_pattern`:

- hi cortana
- hi siri
- hi alexa
- hi google
- hey cortana
- hey siri
- hey alexa
- hey google

represents the `WAKE token`.

Within the `token_patterns` section, there are `static` and `dynamic` sections. `static` means all of the `token_patterns` can be specified before the application runs. `dynamic` means the `token_patterns` will be specified at runtime. In our example, `WAKE` is defined underneath `static` because all ways to awake the smart speaker are known before runtime. `ARTIST` and `SONG`, however, are defined underneath `dynamic` because the artists and songs in your music catalog may change frequently. The values for these `tokens` can be passed in as arguments to `Pipeline` at runtime.

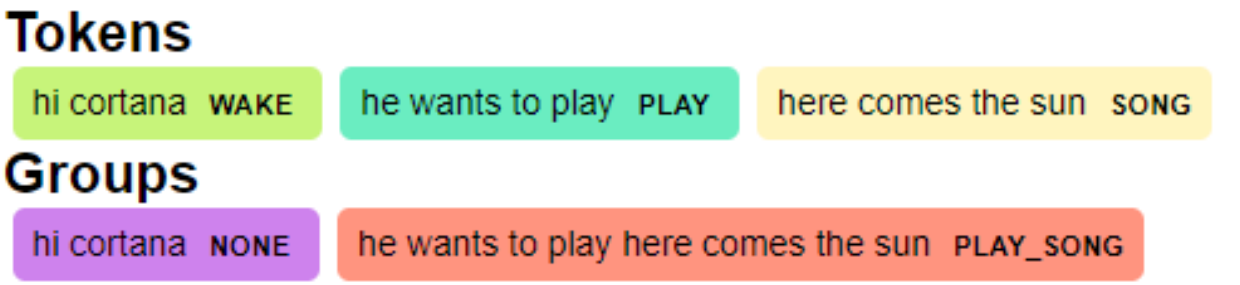
Within each `token_pattern`, `base_tokens` may be used to keep yourself from repeating the same components. For instance, in our example, we could potentially use `PERSONAL_PRONOUNS` in many different places, so we'd like to only have to define it once.

An `utterance_pattern` describes the product of `tokens` that make up an `utterance`. For instance, the first `utterance_pattern`, `[WAKE, PLAY, SONG]`, is a product of all of the products of `token_patterns` for `WAKE`, `PLAY`, and `SONG` (the `group`, `PLAY_SONG`, is replaced with its value `[PLAY, SONG]`). Example `utterances` generated from this `utterance_pattern` would be:

- hi cortana play here comes the sun
- hi cortana he would like to play here comes the sun

Within each `utterance_pattern`, `groups` may be used to keep yourself from repeating the same `tokens`. For instance, in our example, we could potentially use `PLAY_SONG` in many different places, so we'd like to only have

to define it once. Unlike `base_tokens`, `putput` keeps track of `groups`. For instance, recall one potential output corresponding to the `utterance_pattern`, `[WAKE, PLAY_SONG]`:



Since `PLAY_SONG` is the only `group` in the `utterance_pattern`, the `WAKE` `token` is assigned the `group` `NONE` whereas the `PLAY` and `SONG` `tokens` are assigned the `group` `PLAY_SONG`.

Thinking in terms of commercial NLU providers, `groups` could be used to match to `intents` and `tokens` could be used to match `entities`.

`utterance_patterns` and `groups` support range syntax. Looking at the last `utterance_pattern`, `[WAKE, 1-2, PLAY_SONG]`, we see the range, 1-2. Putput will expand this `utterance_pattern` to two `utterance_patterns`, `[WAKE, PLAY_SONG]` and `[WAKE, WAKE, PLAY_SONG]`. Ranges are inclusive and may also be specified as a single number, which would expand into one `utterance_pattern`.

Finally, `groups` may be defined within `groups`. For instance:

```
- groups:
  - PLAY_SONG: [PLAY, SONG]
  - WAKE_PLAY_SONG: [WAKE, PLAY_SONG, 10]
```

is valid syntax.

6.3 Pipeline

After defining the `pattern definition`, the final step to generating labeled data is instantiating `putput`'s `Pipeline` and calling `flow`.

```
dynamic_token_patterns_map = {
    'SONG': ('here comes the sun', 'stronger'),
    'ARTIST': ('the beatles', 'kanye west')
}
p = Pipeline(pattern_def_path, dynamic_token_patterns_map=dynamic_token_patterns_map)
for utterance, tokens, groups in p.flow():
    print(utterance)
    print(tokens)
    print(groups)
```

`flow` yields results one `utterance` at a time. While the results could be the tuple `(utterance, tokens, groups)` for each iteration, they could also be customized by specifying arguments to `Pipeline`. Some common use cases are limiting the size of the output, oversampling/undersampling `utterance_patterns`, specifying how `tokens` and `groups` are tokenized, etc. Customization of the `Pipeline` is extensive and is covered in the `Pipeline`'s docs. Common `preset` configurations are covered in the `preset` docs.

Python Module Index

p

Index