ctparse - Parse natural language time expressions in pytho Documentation

Release __version__ = '0.3.6'

Sebastian Mika

Nov 28, 2022

Contents:

CHAPTER 1

ctparse - Parse natural language time expressions in python

Documentation: https://ctparse.readthedocs.io.

1.1 Background

The package `ctparse` is a pure python package to parse time expressions from natural language (i.e. strings). In many ways it builds on similar concepts as Facebook's `duckling` package (https://github.com/facebook/duckling). However, for the time being it only targets times and only German and English text.

In principle `ctparse` can be used to **detect** time expressions in a text, however its main use case is the semantic interpretation of such expressions. Detecting time expressions in the first place can - to our experience - be done more efficiently (and precisely) using e.g. CRFs or other models targeted at this specific task.

`ctparse` is designed with the use case in mind where interpretation of time expressions is done under the following assumptions:

- All expressions are relative to some pre-defined reference times
- Unless explicitly specified in the time expression, valid resolutions are in the future relative to the reference time (i.e. `12.5.` will be the next 12th of May, but `12.5.2012` should correctly resolve to the 12th of May 2012).
- If in doubt, resolutions in the near future are more likely than resolutions in the far future (not implemented yet, but any resolution more than i.e. 3 month in the future is extremely unlikely).

The specific comtravo use-case is resolving time expressions in booking requests which almost always refer to some point in time within the next 4-8 weeks.

`ctparse` currently is language agnostic and supports German and English expressions. This might get an extension in the future. The main reason is that in real world communication more often than not people write in one language (their business language) but use constructs to express times that are based on their mother tongue and/or what they believe to be the way to express dates in the target language. This leads to text in German with English time expressions

and vice-versa. Using a language detection upfront on the complete original text is for obvious no solution - rather it would make the problem worse.

1.2 Example

```
from ctparse import ctparse
from datetime import datetime

# Set reference time
ts = datetime(2018, 3, 12, 14, 30)
ctparse('May 5th 2:30 in the afternoon', ts=ts)
```

This should return a `Time` object represented as `Time[0-29]{2018-05-05 14:30 (X/X)}`, indicating that characters `0-29` were used in the resolution, that the resolved date time is the 5th of May 2018 at 14:30 and that this resolution is neither based on a day of week (first `X`) nor a part of day (second `X`).

1.2.1 Latent time

Normally, `ctparse` will anchor time expressions to the reference time. For example, when parsing the time expression `8:00 pm`, ctparse will resolve the expression to 8 pm after the reference time as follows

```
parse = ctparse("8:00 pm", ts=datetime(2020, 1, 1, 7, 0), latent_time=True) # default
# parse.resolution -> Time(2020, 1, 1, 20, 00)
```

This behavior can be customized using the option `latent_time=False`, which will return a time resolution not anchored to a particular date

```
parse = ctparse("8:00 pm", ts=datetime(2020, 1, 1, 7, 0), latent_time=False)
# parse.resolution -> Time(None, None, None, 20, 00)
```

1.3 Implementation

`ctparse` - as `duckling` - is a mixture of a rule and regular expression based system + some probabilistic modeling. In this sense it resembles a PCFG.

1.3.1 Rules

At the core `ctparse` is a collection of production rules over sequences of regular expressions and (intermediate) productions.

Productions are either of type `Time`, `Interval` or `Duration` and can have certain predicates (e.g. whether a `Time` is a part of day like `'afternoon'`).

A typical rule than looks like this:

```
@rule(predicate('isDate'), dimension(Interval))
```

I.e. this rule is applicable when the intermediate production resulted in something that has a date, followed by something that is in interval (like e.g. in `'May 5th 9-10'`).

The actual production is a python function with the following signature:

```
@rule(predicate('isDate'), dimension(Interval))
def ruleDateInterval(ts, d, i):
  """
  param ts: datetime - the current refenrence time
  d: Time - a time that contains at least a full date
  i: Interval - some Interval
  """
  if not (i.t_from.isTOD and i.t_to.isTOD):
    return None
  return Interval(
    t_from=Time(year=d.year, month=d.month, day=d.day,
                hour=i.t_from.hour, minute=i.t_from.minute),
    t_to=Time(year=d.year, month=d.month, day=d.day,
              hour=i.t_to.hour, minute=i.t_to.minute))
```

This production will return a new interval at the date of `predicate('isDate')` spanning the time coded in `dimension(Interval)`. If the latter does code for something else than a time of day (TOD), no production is returned, e.g. the rule matched but failed.

1.3.2 Technical Background

Some observations on the problem:

- Each rule is a combination of regular expressions and productions.
- Consequently, each production must originate in a sequence of regular expressions that must have matched (parts of) the text.
- Hence, only subsequence of **all** regular expressions in **all** rules can lead to a successful production.

To this end the algorithm proceeds as follows:

1. Input a string and a reference time
2. Find all matches of all regular expressions from all rules in the input strings. Each regular expression is assigned an identifier.
3. Find all distinct sequences of these matches where two matches do not overlap nor have a gap inbetween
4. To each such subsequence apply all rules at all possible positions until no further rules can be applied - in which case one solution is produced

Obviously, not all sequences of matching expressions and not all sequences of rules applied on top lead to meaningful results. Here the **P**CFG kicks in:

- Based on example data (`corpus.py`) a model is calibrated to predict how likely a production is to lead to a/the correct result. Instead of doing a breadth first search, the most promising productions are applied first.
- Resolutions are produced until there are no more resolutions or a timeout is hit.
- Based on the same model from all resolutions the highest scoring is returned.

1.4 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install ctparse - Parse natural language time expressions in pytho, run this command in your terminal:

```
$ pip install ctparse
```

This is the preferred method to install ctparse - Parse natural language time expressions in pytho, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for ctparse - Parse natural language time expressions in pytho can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/comtravo/ctparse
```

Or download the tarball:

```
$ curl  -OL https://github.com/comtravo/ctparse/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use ctparse simply import the main `ctparse` function:

```
from datetime import datetime
from ctparse import ctparse

ctparse('today', datetime(2018, 7, 8), timeout=1)
```

The output for the above code is *2018-07-08 X:X (X/X) s=2.273 p=(149, 'ruleToday')*

For more details on the parameters please see the docstrings.

CHAPTER 4

Time Parse Dataset

The dataset included in `datasets/timeparse_corpus.json` contains a set of ~2000 human annotated time expression in english and german.

The dataset is a list of json records with the following fields:

- *text*: the text for the time expression
- *ref_time*: a timestamp in ISO 8601 format `YYYY-MM-DDTHH:MM:SS`
- *gold_parse*: the human annotation of the time expression. It can be a `Time` or `Interval`.
- *language*: a two-digit code indicating the language. In this dataset it is either "en" or "de".

For `Time`, the format is as follows:

```
Time[]{YYYY-MM-DD HH:MM (dow/tod)}
```

Where: - `YYYY` is a four-digit year or `X`, if year is missing - `MM` is a two-digit month or `X`, if month is missing - `DD` is a two-digit day or `X`, if day is missing - `HH` is a two-digit hour (24 hour clock) or `X`, if hour is missing - `MM` is a two-digit minute or `X`, if minute is missing - `dow` is an integer between 0 and 6 representing day of week or X, if missing (in the dataset, day of week is always missing) - `tod` is a string representing the time of day (such as earlymorning, morning, forenoon, noon, afternoon, evening, lateevening) or X if not specified.

Example:

```
Morning of the 11th June 2017
Time[]{2017-06-11 X:X (X/morning)}
```

For `Interval` the format is as follows:

```
Interval[]{<START_T> - <END_T>}
```

Where `<START_T>` and `<END_T>` are the beginning and end of the interval. `<START_T>` or `<END_T>` can be None if the interval is open-ended. They can be specified using the same representation for times, as described above:

```
YYYY-MM-DD HH:MM (dow/tod)
```

Example:

```
Wed, Oct 11 2017 8:30 PM - 9:47 PM
Interval[]{2017-10-11 08:30 (X/X) - 2017-10-11 09:47 (X/X)}
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Add Rules & Increase Coverage

If you find an expressions that `ctparse` can not resolve correctly but you feel it should do, you can adjust the existing rules or add a new one.

The following steps are probably a helpful guideline.

- Add your case to the `corpus.py` file and run the corpus tests using `py.test tests/test_corpus.py`. Now basically two things can happen:

    1. **The tests pass**, which means `ctparse` can correctly resolve the expression. It might not score it highest. To check this, rebuild the model and try parsing the expression again:

        ```
        make train
        ```

        To avoid issues with reloading, please restart the python interpreter after regenerating the model.

        If this fixes the issue please commit the updated `corpus.py` and the updated model as a pull request (PR) on GitHub, see this guide for more information on what pull requests are and how to create them https://help.github.com/articles/creating-a-pull-request/.

        The scoring can be influenced by adding more structurally identical examples to the corpus. Seeing more samples where a specific sequence of rule applications leads to the correct ranking will drive the model to favor these. This comes, however, at the potential price of downranking certain other production sequences. Although it would generally be considered more favorable to add varying test cases (e.g. in different languages, slight variation) to the corpus, the same string can also just be duplicated to achive this *implict up-weightning* effect. The examples that are intended to influence the scoring, as opposed to the ones used to develop new rules, are usually appended to the file `auto_corpus.py`.

2. **The tests fail**: if this is because not all tests in the corpus pass, i.e. you get an error message like the following:

```
ctparse.py 527 WARNING  failure: target "Time[]{2019-X-X X:X (X/X)}" never␣
↪produced in "2019"
ctparse.py 532 WARNING  failure: "Time[]{2019-X-X X:X (X/X)}" not always␣
↪produced
```

- If the tests fail, run `ctparse` in debug mode to see what goes wrong:

```
import logging
from ctparse import ctparse
from ctparse.ctparse import logger
from datetime import datetime

logger.addHandler(logging.StreamHandler())
logger.setLevel(logging.DEBUG)

# Set reference time
ts = datetime(2018, 3, 12, 14, 30)
r = list(ctparse('May 5th', ts=ts, debug=True))
```

This gives you plenty of debugging output. First you will see the individual regular expressions that were matched (and the time this took):

```
================================================================================
-> matching regular expressions
regex: RegexMatch[0-3]{114:May}
regex: RegexMatch[4-5]{133:5}
regex: RegexMatch[4-7]{135:5th}
regex: RegexMatch[4-5]{134:5}
regex: RegexMatch[4-5]{148:5}
time in _match_regex: 1ms
================================================================================
```

Each line has the form `regex: RegexMatch[0-3]{114:May}` and describes the matched span in the text `[0-3]`, the ID of the matching expression `114` and the surface string that the expression matched `May`.

If relevant parts of your expression were not picked up, this is an indicator that you should either modify an existing regular expression or need to add a new rule (see below).

Next you see the unique sub-sequences constructed based on these regular expressions (plus again the time used to build them):

```
================================================================================
-> building initial stack
regex stack (RegexMatch[0-3]{114:May}, RegexMatch[4-7]{135:5th})
regex stack (RegexMatch[0-3]{114:May}, RegexMatch[4-5]{148:5})
regex stack (RegexMatch[0-3]{114:May}, RegexMatch[4-5]{134:5})
regex stack (RegexMatch[0-3]{114:May}, RegexMatch[4-5]{133:5})
time in _regex_stack: 0ms
initial stack length: 4
stack length after relative match length: 1
stack length after max stack depth limit: 1
================================================================================
```

This is followed by a summary of how many applicable rules there are per initial stack element:

```
================================================================================
-> checking rule applicability
of 75 total rules 20 are applicable in (RegexMatch[0-3]{114:May}, RegexMatch[4-7]
↪{135:5th})
time in _filter_rules: 0ms
================================================================================
================================================================================
-> checking rule applicability
of 75 total rules 20 are applicable in (RegexMatch[0-3]{114:May}, RegexMatch[4-5]
↪{148:5})
time in _filter_rules: 0ms
================================================================================
...
```

Again, if you do not see any sequence that captures all relevant parts of your input, you may need to modify the regular expressions or add new ones via rules.

Finally you see a list of productions that are applied to stack elements, where for each applicable rule the rule name and the new stack sequence are printed, e.g.:

```
--------------------------------------------------------------------------------
producing on (RegexMatch[0-3]{114:May}, RegexMatch[4-7]{135:5th}), score=-0.13
  ruleMonthMay -> (Time[0-3]{X-05-X X:X (X/X)}, RegexMatch[4-7]{135:5th}),␣
↪score=1.41
  ruleDOM2 -> (RegexMatch[0-3]{114:May}, Time[4-7]{X-X-05 X:X (X/X)}), score=1.38
added 2 new stack elements, depth after trunc: 2
--------------------------------------------------------------------------------
```

If no productions could be applied to a stack element the emitted results are printed:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
no rules applicable: emitting
=> Time[0-7]{2018-05-05 X:X (X/X)}, score=15.91,
--------------------------------------------------------------------------------
```

If the desired production does not show up, but the regular expressions look fine and the initial stack elements as well, try increasing the `max_stack_depth` parameter, i.e. run `ctparse(..., max_stack_depth=0)`. Also make sure that the `timeout` parameter is not set. Maybe `ctparse` is able to generate the resolution but it is too deep in the stack.

5.1.1 Adding a rule

When adding rules try to follow these guidelines:

1. Be as general as possible: instead of writing one long regular expression that matches only a specific case, check whether you can rather divide your pattern in production parts + some regular expressions. For example, if you have a very specific way to speficy the year of a date in mind, it might do no harm to just allow anything that with `predicate('hasDate')` plus your specific year expression, i.e.

```
@rule(predicate('hasDate'), r'your funky year')
```

2. Keep your regex as general as possible, but avoid regular expressions that are likely to generate many "false positives". Often that can be prevented by using positive or negative lookaheads and lookbehinds to keep the context sane (see Lookaround on the excellent regular-expression.info site).

3. Make sure your production covers corner cases and matches the `ctparse` opinion to resolve to times in the near future but - unless explicit – never in the past (relative to the reference time). Also make sure it favors the

close future over the further future.

5.2 Other Types of Contributions

5.2.1 Report Bugs

Report bugs at https://github.com/comtravo/ctparse/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.2.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.2.4 Write Documentation

ctparse - Parse natural language time expressions in pytho could always use more documentation, whether as part of the official ctparse - Parse natural language time expressions in pytho docs, in docstrings, or even on the web in blog posts, articles, and such.

5.2.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/comtravo/ctparse/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.3 Get Started!

Ready to contribute? Here's how to set up *ctparse* for local development.

1. Fork the *ctparse* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/ctparse.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv ctparse
$ cd ctparse/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests:

   $ make lint $ make testlcoverage

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.7. - 3.10.

5.5 Tips

To run a subset of tests:

```
$ py.test tests.test_ctparse
```

5.6 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run on the `master` branch:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
$ make release
```

You will need a username and password to upload to pypi

CHAPTER 6

ctparse

6.1 ctparse package

6.1.1 Subpackages

ctparse.time package

Submodules

ctparse.time.corpus module

ctparse.time.rules module

Module contents

6.1.2 Submodules

6.1.3 ctparse.corpus module

class ctparse.corpus.**TimeParseEntry**(*text*, *ts*, *gold*)

Bases: tuple

gold

Alias for field number 2

text

Alias for field number 0

ts

Alias for field number 1

ctparse.corpus.**load_timeparse_corpus**(*fname: str*) → Sequence[ctparse.corpus.TimeParseEntry]

Load a corpus from disk.

For more information about the format of the time parse corpus, refer to the documentation.

ctparse.corpus.**make_partial_rule_dataset**(*entries: Sequence[ctparse.corpus.TimeParseEntry], scorer: ctparse.scorer.Scorer, timeout: Union[float, int], max_stack_depth: int, relative_match_len: float = 1.0, progress: bool = False*) → Iterable[Tuple[List[str], bool]]

Build a data set from an iterable of TimeParseEntry.

The text is run through ctparse and all parses (within the specified timeout, max_stack_depth and scorer) are obtained. Each parse contains a sequence of rules (see `CTParse.rules`) used to produce that parse.

A dataset is generated by taking every possible partial rule and assigning to it a boolean indicating if that partial sequence did lead to a successful parse.

If *progress* is `True`, display a progress bar.

Example:

rule sequence: [r1, r2, r3] parse_is_correct: True

[r1] -> True [r1, r2] -> True [r1, r2, r3] -> True

ctparse.corpus.**parse_nb_string**(*gold_parse: str*) → Union[ctparse.types.Time, ctparse.types.Interval, ctparse.types.Duration]

Parse a Time, Interval or Duration from their no-bound string representation.

The no-bound string representations are generated from `Artifact.nb_str`.

ctparse.corpus.**run_corpus**(*corpus: Sequence[Tuple[str, str, Sequence[str]]]*) → Tuple[List[List[str]], List[bool]]

Load the corpus (currently hard coded), run it through ctparse with no timeout and no limit on the stack depth.

The corpus passes if ctparse generates the desired solution for each test at least once. Otherwise it fails.

While testing this, a labeled data set (X, y) is generated based on *all* productions. Given a final production p, based on initial regular expression matches r_0, ..., r_n, which are then subsequently transformed using production rules p_0, ..., p_m, will result in the samples

[r_0, ..., r_n, p_0, 'step_0'] [r_0, ..., r_n, p_0, p_1, 'step_1'] ... [r_0, ..., r_n, p_0, ..., p_m, 'step_m']

All samples from one production are given the same label which indicates if the production was correct.

To build a similar datasets without the strict checking, use *make_partial_rule_dataset*

ctparse.corpus.**run_single_test**(*target: str*, *ts: str*, *test: str*) → None

Run a single test case and raise an exception if the target was never produced.

Below *max_stack_depth* might be increased if tests fail.

Parameters

- **target** (*str*) – Target to produce
- **ts** (*str*) – Reference time as *string*
- **test** (*str*) – Test case

6.1.4 ctparse.count_vectorizer module

class ctparse.count_vectorizer.**CountVectorizer**(*ngram_range: Tuple[int, int]*)

Bases: object

fit(*documents: Sequence[Sequence[str]]*) → ctparse.count_vectorizer.CountVectorizer

Learn a vocabulary dictionary of all tokens in the raw documents.

Parameters **documents** (*Sequence[Sequence[str]]*) – Sequence of documents, each as a sequence of tokens

Returns The updated vectorizer, i.e. this updates the internal vocabulary

Return type *CountVectorizer*

fit_transform(*documents: Sequence[Sequence[str]]*) → Sequence[Dict[int, int]]

Learn the vocabulary dictionary and return a term-document matrix. Updates the internal vocabulary state of the vectorizer.

Parameters **documents** (*Sequence[Sequence[str]*) – Sequence of documents, each as a sequence of tokens

Returns Document-term matrix.

Return type Sequence[Dict[int, int]]

transform(*documents: Sequence[Sequence[str]]*) → Sequence[Dict[int, int]]

Create term-document matrix based on pre-generated vocabulary. Does *not* update the internal state of the vocabulary.

Parameters **documents** (*Sequence[Sequence[str]]*) – Sequence of documents, each as a sequence of tokens

Returns Document-term matrix.

Return type Sequence[Dict[int, int]]

6.1.5 ctparse.ctparse module

class ctparse.ctparse.**CTParse**(*resolution: ctparse.types.Artifact, production: Tuple[Union[int, str], ...], score: float*)

Bases: object

ctparse.ctparse.**ctparse**(*txt: str*, *ts: Optional[datetime.datetime] = None*, *timeout: Union[int, float] = 1.0*, *debug: bool = False*, *relative_match_len: float = 1.0*, *max_stack_depth: int = 10*, *scorer: Optional[ctparse.scorer.Scorer] = None*, *latent_time: bool = True*) → Optional[ctparse.ctparse.CTParse]

Parse a string *txt* into a time expression

Parameters

- **ts** (*datetime.datetime*) – reference time
- **timeout** (*float*) – timeout for parsing in seconds; timeout=0 indicates no timeout
- **debug** – if True do return iterator over all resolution, else return highest scoring one (default=False)
- **relative_match_len** (*float*) – relative minimum share of characters an initial regex match sequence must cover compared to the longest such sequence found to be considered for productions (default=1.0)

- **max_stack_depth** (*int*) – limit the maximal number of highest scored candidate productions considered for future productions (default=10); set to 0 to not limit
- **latent_time** – if True, resolve expressions that contain only a time (e.g. 8:00 pm) to be the next matching time after reference time *ts*

Returns Optional[CTParse]

ctparse.ctparse.**ctparse_gen**(*txt: str*, *ts: Optional[datetime.datetime] = None*, *timeout: Union[int, float] = 1.0*, *relative_match_len: float = 1.0*, *max_stack_depth: int = 10*, *scorer: Optional[ctparse.scorer.Scorer] = None*, *latent_time: bool = True*) → Iterator[Optional[ctparse.ctparse.CTParse]]

Generate parses for the string *txt*.

This function is equivalent to ctparse, with the exception that it returns an iterator over the matches as soon as they are produced.

6.1.6 ctparse.loader module

Utility to load default model in ctparse

ctparse.loader.**load_default_scorer**() → ctparse.scorer.Scorer

Load the scorer shipped with ctparse.

If the scorer is not found, the scorer defaults to *DummyScorer*.

6.1.7 ctparse.nb_estimator module

class ctparse.nb_estimator.**MultinomialNaiveBayes**(*alpha: float = 1.0*)

Bases: object

Implements a multinomial naive Bayes classifier. For background information (and what has inspired this, see e.g. https://scikit-learn.org/stable/...

...modules/generated/sklearn.naive_bayes.MultinomialNB.html)

fit(*X: Sequence[Dict[int, int]]*, *y: Sequence[int]*) → ctparse.nb_estimator.MultinomialNaiveBayes

Fit a naive Bayes model from a count of feature matrix

Parameters

- **X** (*Sequence[Dict[int, int]]*) – Sequence of sparse {feature_index: count} dictionaries
- **y** (*Sequence[int]*) – Labels +1/-1

Returns The fitted model

Return type *MultinomialNaiveBayes*

predict_log_probability(*X: Sequence[Dict[int, int]]*) → Sequence[Tuple[float, float]]

Calculate the posterior log probability of new sample X

Parameters **X** (*Sequence[Dict[int, int]]*) – Sequence of data to predict on as sparse {feature_index: count} dictionarie

Returns Tuple of (negative-class, positive-class) log likelihoods

Return type Sequence[Tuple[float, float]]

6.1.8 ctparse.nb_scorer module

This module cointains the implementation of the scorer based on naive bayes.

class ctparse.nb_scorer.**NaiveBayesScorer**(*nb_model: ctparse.pipeline.CTParsePipeline*)

Bases: *ctparse.scorer.Scorer*

classmethod from_model_file(*fname: str*) → ctparse.nb_scorer.NaiveBayesScorer

score(*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*) → float

Produce a score for a partial production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the partial parse that needs to be scored

score_final(*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*, *prod: ctparse.types.Artifact*) → float

Produce the final score for a production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the PartialParse object that generated the production
- **prod** – the production

ctparse.nb_scorer.**save_naive_bayes**(*model: ctparse.pipeline.CTParsePipeline*, *fname: str*) → None

Save a naive bayes model for NaiveBayesScorer

ctparse.nb_scorer.**train_naive_bayes**(*X: Sequence[Sequence[str]]*, *y: Sequence[bool]*) → ctparse.pipeline.CTParsePipeline

Train a naive bayes model for NaiveBayesScorer

6.1.9 ctparse.partial_parse module

class ctparse.partial_parse.**PartialParse**(*prod: Tuple[ctparse.types.Artifact, ...]*, *rules: Tuple[Union[int, str], ...]*)

Bases: object

apply_rule(*ts: datetime.datetime*, *rule: Callable[[...], Optional[ctparse.types.Artifact]]*, *rule_name: Union[str, int]*, *match: Tuple[int, int]*) → Optional[ctparse.partial_parse.PartialParse]

Check whether the production in rule can be applied to this stack element.

If yes, return a copy where this update is incorporated in the production, the record of applied rules and the score.

Parameters

- **ts** – reference time
- **rule** – a tuple where the first element is the production rule to apply
- **rule_name** – the name of the rule
- **match** – the start and end index of the parameters that the rule needs.

classmethod from_regex_matches(*regex_matches: Tuple[ctparse.types.RegexMatch, ...]*) → ctparse.partial_parse.PartialParse

Create partial production from a series of RegexMatch

This usually is called when no production rules (with the exception of regex matches) have been applied.

6.1.10 ctparse.pipeline module

class ctparse.pipeline.**CTParsePipeline**(*transformer: ctparse.count_vectorizer.CountVectorizer*, *estimator: ctparse.nb_estimator.MultinomialNaiveBayes*)

Bases: object

fit(*X: Sequence[Sequence[str]], y: Sequence[int]*) → ctparse.pipeline.CTParsePipeline

Fit the transformer and then fit the Naive Bayes model on the transformed data

Returns Returns the fitted pipeline

Return type *CTParsePipeline*

predict_log_proba(*X: Sequence[Sequence[str]]*) → Sequence[Tuple[float, float]]

Apply the transforms and get probability predictions from the estimator

Parameters **X** (*Sequence[Sequence[str]]*) – Sequence of documents, each as sequence of tokens. In ctparse case there are just the names of the regex matches and rules applied

Returns For each document the tuple of negative/positive log probability from the naive bayes model

Return type Sequence[Tuple[float, float]]

6.1.11 ctparse.rule module

ctparse.rule.**dimension**(*dim: Type[ctparse.types.Artifact]*) → Callable[[ctparse.types.Artifact], bool]

ctparse.rule.**predicate**(*pred: str*) → Callable[[ctparse.types.Artifact], bool]

ctparse.rule.**regex_match**(*r_id: int*) → Callable[[ctparse.types.Artifact], bool]

ctparse.rule.**rule**(**patterns*) → Callable[[Any], Callable[[...], Optional[ctparse.types.Artifact]]]

ctparse.rule.**ruleAbsorbFromInterval**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleAbsorbOnTime**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleAfterTime**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleAfterTomorrow**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleAtDOW**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleBeforeTime**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleBeforeYesterday**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDDMM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDDMMYYYY**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOM1**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOM2**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOMDate**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOMMonth**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOMMonth2**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOWDOM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOWDate**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOWNextWeek**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOWPOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOYDate**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDOYYear**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateDOM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateDOW**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateDOWInterval**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateDate**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateInterval**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDatePOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateTOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDateTimeDateTime**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDigitDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDurationAndDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDurationDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDurationHalf**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleDurationInterval**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleEOM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleEOY**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleEarlyLatePOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleHHMM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleHHMMmilitary**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleHHOClock**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleHalfAfterHH**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleHalfBeforeHH**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleIntervalConjDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleIntervalDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleLatentDOM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleLatentDOW**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleLatentDOY**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleLatentPOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleMMDD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleMidnight**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleMonthDOM**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleMonthOrdinal**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleMonthYear**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleNamedDOW**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleNamedHour**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleNamedMonth**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleNamedNumberDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleNextDOW**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleNow**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**rulePOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**rulePODDate**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**rulePODInterval**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**rulePODPOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**rulePODTOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleQuarterAfterHH**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleQuarterBeforeHH**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleTODDate**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleTODPOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleTODTOD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleThisDOW**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleTimeDuration**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleToday**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleTomorrow**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleYYYYMMDD**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleYear**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

ctparse.rule.**ruleYesterday**(*ts: datetime.datetime*, **args*) → Optional[ctparse.types.Artifact]

6.1.12 ctparse.scorer module

This module contains the Scorer abstraction that can be used to implement scoring strategies for ctparse.

class ctparse.scorer.**DummyScorer**
Bases: *ctparse.scorer.Scorer*

A scorer that always return a 0.0 score.

score(*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*) → float
Produce a score for a partial production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the partial parse that needs to be scored

score_final(*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*, *prod: ctparse.types.Artifact*) → float
Produce the final score for a production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the PartialParse object that generated the production
- **prod** – the production

class ctparse.scorer.**RandomScorer**(*rng: Optional[random.Random] = None*)
Bases: *ctparse.scorer.Scorer*

score(*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*) → float
Produce a score for a partial production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the partial parse that needs to be scored

score_final(*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*, *prod: ctparse.types.Artifact*) → float
Produce the final score for a production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the PartialParse object that generated the production
- **prod** – the production

class ctparse.scorer.**Scorer**
Bases: object

Interface for scoring parses generated by ctparse

score (*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*) → float
Produce a score for a partial production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the partial parse that needs to be scored

score_final (*txt: str*, *ts: datetime.datetime*, *partial_parse: ctparse.partial_parse.PartialParse*, *prod: ctparse.types.Artifact*) → float
Produce the final score for a production.

Parameters

- **txt** – the text that is being parsed
- **ts** – the reference time
- **partial_parse** – the PartialParse object that generated the production
- **prod** – the production

6.1.13 ctparse.timers module

Utilities for tracking time spent in functions.

Although this module is not part of the public API, it is used in various parts of the ctparse package.

exception ctparse.timers.**CTParseTimeoutError**
Bases: Exception

Exception raised by the *timeout* function.

ctparse.timers.**timeit** (*f: Callable[[...], T]*) → Callable[[...], Tuple[T, float]]
Wrapper to time a function.

The wrapped function is modified so that it returns a tuple *(f(args), t)* where *t* the time in seconds the function call took to run.

Example

def fun(x): return x * x

result, exec_time = timeit(fun)(3)

ctparse.timers.**timeout** (*timeout: Union[float, int]*) → Callable[[], None]
Generate a functions that raises an exceptions if a timeout has passed.

Example

sentinel = timeout(1.0) time.sleep(0.5) sentinel() # Do nothing time.sleep(0.6) sentinel() # Raises CTParseTimeoutException

Parameters **timeout** – time in seconds. If it is equal to zero, it means to never raise an exception.

Returns A function that raises a *CTParseTimeoutException* if *timeout* seconds have expired.

6.1.14 ctparse.types module

class ctparse.types.**Artifact**
Bases: object

nb_str() → str
Return a string representation without the bounds information.

update_span(**args*) → T

class ctparse.types.**Duration**(*value: int*, *unit: ctparse.types.DurationUnit*)
Bases: *ctparse.types.Artifact*

classmethod from_str(*text: str*) → ctparse.types.Duration

class ctparse.types.**DurationUnit**
Bases: enum.Enum

An enumeration.

DAYS = 'days'

HOURS = 'hours'

MINUTES = 'minutes'

MONTHS = 'months'

NIGHTS = 'nights'

WEEKS = 'weeks'

class ctparse.types.**Interval**(*t_from: Optional[ctparse.types.Time] = None*, *t_to: Optional[ctparse.types.Time] = None*)
Bases: *ctparse.types.Artifact*

end

classmethod from_str(*text: str*) → ctparse.types.Interval

isDateInterval

isTimeInterval

start

class ctparse.types.**RegexMatch**(*id: int*, *m: regex.regex.compile*)
Bases: *ctparse.types.Artifact*

class ctparse.types.**Time**(*year: Optional[int] = None*, *month: Optional[int] = None*, *day: Optional[int] = None*, *hour: Optional[int] = None*, *minute: Optional[int] = None*, *DOW: Optional[int] = None*, *POD: Optional[str] = None*)
Bases: *ctparse.types.Artifact*

dt

end

classmethod from_str(*text: str*) → ctparse.types.Time

hasDOW
at least a day of week

hasDOY
at least a day of year

hasDate
at least a date

hasPOD
at least a part of day

hasTime
at least a time to the hour

isDOM
isDayOfMonth <=> a dd but no month

isDOW
isDayOfWeek <=> DOW is the 0=Monday index; fragile test, as the DOW could be accompanied by e.g. a full date etc.; in practice, however, the production rules do not do that.

isDOY
isDayOfYear <=> a dd.mm but not year

isDate
isDate - only a date, not time

isDateTime
a date and a time

isHour
only has an hour

isMonth

isPOD
isPartOfDay <=> morning, etc.; fragile, tests only that there is a POD and neither a full date nor a full time

isTOD
isTimeOfDay - only a time, not date

isYear
just a year

start

6.1.15 Module contents

ctparse - parse time expressions in strings

CHAPTER 7

Credits

7.1 Development Lead

- Sebastian Mika <sebastian.mika@comtravo.com>

7.2 Contributors

- Gabriele Lanaro <gabriele.lanaro@comtravo.com>

CHAPTER 8

History

8.1 0.3.4 (2022-07-28)

- Add fuzzy matching on longer literals
- [internal] De-tangle corpus tests into isolated test cases
- Allow spaces around separators in ruleDDMMYYYY and ruleYYYYMMDD

8.2 0.3.3 (2022-07-18)

- Add rule for straight forward US formatted dates (*ruleYYYYMMDD*)
- Added rule *ruleYearMonth*
- Added corpus cases for some open issues that now pass
- Changed all internal imports to be absolute (i.e. *from ctparse.x* instead of *from .x*)
- Dropped *tox* (now using github actions)

8.3 0.3.2 (2022-07-18)

- Drop support for python 3.6, update dev requirements

8.4 0.3.1 (2021-07-07)

- Add support for python 3.9 on travis and in manifest; update build config

8.5 0.3.0 (2021-02-01)

- Removed latent rules regarding times (latent rules regarding dates are still present)
- Added latent_time option to customize the new behavior, defauld behavior is backwards-compatible

8.6 0.2.1 (2020-05-27)

- Update development dependencies
- Add flake8-bugbear and fixed issues

8.7 0.2.0 (2020-04-23)

- Implemented new type *Duration*, to handle lengths of time
- Adapted the dataset to include *Duration*
- Implemented basic rule to merge *Duration*, *Time* and *Interval* in simple cases.
- Created a make target to train the model *make train*

8.8 0.1.0 (2020-03-20)

- Major refactor of code underlying predictive model
- Based on a contribution from @bharathi-srini: replace naive bayes from sklearn by own implementation
- Thus remove dependencies on numpy, scipy, scikit-learn
- Predictions are much faster: 97/s in the old vs. 239/s in the new code base
- Performance identical
- Deprecate support for python 3.5, add 3.8
- Add more strict type checking rules (mypy.ini)
- Force black code formatting, make this a linter step, "black" all code

8.9 0.0.47 (2020-02-28)

- Allow overlapping matches of regular expression when generating inital stack of "tokens"

8.10 0.0.46 (2020-02-26)

- Implemented heuristics to detect (albeit imperfectly) military times

8.11 0.0.44 (2019-11-05)

- Released time corpus
- Implemented training model using ctparse corpus

8.12 0.0.43 (2019-11-01)

- Added slash as a general separator
- Added ruleTODTOD (to support expression like afternoon/evening)

8.13 0.0.42 (2019-10-30)

- Removed nb module
- Fix for two digit years
- Freshly retrained model binary file

8.14 0.0.41 (2019-10-29)

- Fix run_corpus refactoring bug
- Implemented retraining utilities

8.15 0.0.40 (2019-10-25)

- update develop dependencies
- remove unused Protocol import from typing_extensions

8.16 0.0.39 (2019-10-24)

- split ctparse file into several different modules
- added types to public interface
- introduced the Scorer abstraction to implement richer scoring strategies

8.17 0.0.38 (2018-11-05)

- Added python 3.7 to supported versions (fix on travis available)

8.18 0.0.8 (2018-06-07)

- First release on PyPI.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

L

M

N

P

R

S

T

U

W