PETRARCH Documentation

Release 1.0.0

Open Event Data Alliance

October 03, 2018

Contents

A Python Engine for Text Resolution And Related Coding Hierarchy part 2. This is the documentation for PETRARCH2, though PETRARCH is used throughtout this documentation as interchangeable with PETRARCH2. The difference between the programs lies in the coding engine rather than the API; more details can be seen in the Comparison.

> One of my students put it this way: "Francesco Petrarch was Kayne West. He jumps up on stage, says 'Yo, welcome to the Renaissance, bitches!' And then drops the mike." – Dorsey Armstrong *Great Minds of the Medieval World* (Great Courses Series), lecture 20

PETRARCH is a natural language processing tool for machine-coding events data. It is designed to process fully-parsed news summaries in Penn Treebank format, from which 'whom-did-what-to-whom' relations are extracted.

PETRARCH is the next-generation successor to the TABARI event-data coding software. A description of the differences between TABARI and PETRARCH-generation software is available here.

CHAPTER 1

Events Data

Over the last few decades, computational and social scientists have refined a process of systematically coding events from news summaries and event descriptions referred to as "event data." This process consists of two component parts:

1. First, is the collection of raw unstructured text including information about relevant events. For this step we have developed a web scraper that automatically pulls news stories form a white list of RSS feeds. Scraped stories are then stored in a MongoDB instance for easy future retrieval.
2. **Second in this process is the extraction of structured data from scraped unstructured texts using an event data coding system. In earlier work this was done using the TABARI system, but in this system has PETRARCH which works with fully-parsed inputs in the Penn TreeBank format, which we generate using the Stanford CoreNLP parser.**

The output of this process, event observations identified and extracted in a 'who-did-what-to-whom' format, is what we refer to as event data. At this most fundamental event event data consists of three component parts, `{SOURCE_ACTOR, ACTION_TYPE, TARGET_ACTOR}` as well as general attributes `{DATE_TIME, LOCATION}`:

Sample Raw Event Data Output:

```
| Date     | Source | CAMEO Code | Target | Cameo Event |
|:--------:|:------:|:----------:|:------:|:-------------------------------:
| 19980312 | ISRMIL | 190        | PALINS | (Use conventional military face) |
```

For more information on event data as well as event data related research see: http://eventdata.parusanalytics.com/

CHAPTER 2

Installing

If you do decide you want to work with Petrarch as a standalone program, it is possible to install:

1. Run `pip install git+https://github.com/openeventdata/petrarch2.git`

Some users may experience issues with this install command. Using

```
pip install git+https://github.com/openeventdata/petrarch2.git
--ignore-installed
```

may work instead.

This will install the program with a command-line hook. You can now run the program using:

```
petrarch2 <COMMAND NAME> [OPTIONS]
```

You can get more information using:

```
petrarch2 -h
```

StanfordNLP:

See the README about this.

CHAPTER 3

Running

Currently, you can run PETRARCH using the following command if installed:

```
petrarch2 batch [-i <INPUT FILE> ] [-o [<OUTPUT FILE>]
```

If not installed:

```
python petrarch2.py batch -i <INPUT FILE> -o <OUTPUT FILE>
```

You can see a sample of the input/output by running (assuming you're in the PETRARCH2 directory):

```
petrarch2 batch -i ./petrarch2/data/text/GigaWord.sample.PETR.xml -o test.txt
```

This will return a file named *evts.test.txt*.

There's also the option to specify a configuration file using the `-c <CONFIG FILE>` flag, but the program will default to using `PETR_config.ini`.

When you run the program, a `PETRARCH.log` file will be opened in the current working directory. This file will contain general information, e.g., which files are being opened, and error messages.

CHAPTER 4

Logged Warnings

As of September 2014, we are regularly running PETRARCH on hundreds of thousands of sentences from a diverse set of sources and it is not crashing. If you encounter a situation where it is crashing, please let us know, ideally with a copy of the parsed input text that caused the error.

Unexpected conditions where the program encountered a potentially fatal error are recorded in the log file with the word **WARNING**. These should be rare: in a couple of recent tests we coded 60,000 AFP sentences from the Gigaword corpus and found four such errors; in another test we coded about 360,000 records from BBC sources and had 43 errors. In short, these should be really, really rare and if you are getting them more frequently there is presumably some quirk in your processing pipeline or source texts that is giving you significantly different parsed input than we were working with.

The one common error – not included in those counts – is the `Dateline` pattern, which is a particular pattern in the parse tree that occurs when the parsed material starts with a dateline such as "Beirut:" or "Beijing (Xinhua News Agency):" rather than the actual start of the sentence. We probably aren't catching all dateline errors with this pattern but it gets a lot of them, and if you are seeing frequent occurrences of this warning you need to modify your pre-filters to remove the datelines.

The remaining errors are due to very odd sentence constructions which either have confused CoreNLP so that the phrase structure is incorrect, or otherwise were not anticipated it the PETRARCH processing. Some of this can be fixed if brought to our attention, but some of it is on the side of CoreNLP, which we aren't even going to attempt to touch.

CHAPTER 5

Contents:

PETRARCH2 v. PETRARCH

PETRARCH has been totally redone. The logic now more strongly follows the tree structure provided to us by the TreeBank parse.

The verb dictionary has been completely reworked. Because of the tree-like nature of the new logic, the old linear patterns were insufficient. Patterns have now been formatted to follow the following rules:

1. All patterns match exactly one verb
2. Patterns are minimal in complexity
3. **Nouns, noun phrases, and prepositional phrases are annotated** (For more on this see the dictionary documentation)

Internally, Petrarch does not store verb codes as their CAMEO versions, rather as a hex code that has been translated from CAMEO into a new scheme that better represents the relationship between verb codes.

CoreNLP parsing abilities have been depracated in Petrarch, due to the difficulty of maintaining these across different OSs and systems. Instead we recommend other options in the README file.

PETRARCH2

This page contains some general notes about PETRARCH such as how the data is stored internally, how the configuration file is organized, and an outline of how PETRARCH differs from the previous-generation coder, TABARI.

Miscellaneous Operating Details

While PETRARCH is able to handle chunks of text as input, such as the first four sentences of a news story, the functional processing unit is the individual sentence. As can be seen in the section below, the data *is* organized within the program at the story level, but both the StanfordNLP and event coding process occurs stricly at the sentence level.

Command Line Interface

Primary options

`batch` Run the PETRARCH parser with all options specified in the config file. If combined with `-c`, configuration will be read from that file; default config file is `PETR_config.ini`.

parse **NOTE:** This command is deprecated in PETRARCH2. Run the PETRARCH parser specifying files in the command line

The following options can be used in the command line

-i, --inputs	File, or directory of files, to parse.
-o, --output	Output file for parsed events
-P, --parsed	Input has already been parsed: all input records contain StanfordNLP-parsed <Parse>...</Parse> block. Defaults to `False`.
-c, --config	Filepath for the PETRARCH configuration file. Defaults to `PETR_config.ini`.

Configuration File

The configuration file for PETRARCH currently has three sections: `Dictionaries`, `Options`, and `StanfordNLP`. An example config file is outlined below. This is the same setup as the default configuration used within PETRARCH.

```
[Dictionaries]
# See the PETRreader.py file for the purpose and format of these files
verbfile_name    = CAMEO.091003.master.verbs
#actorfile_list   = Phoenix.Countries.140227.actors.txt, Phoenix.Internatnl.140130.actors.txt, Phoeni
actorfile_list  = Phoenix.Countries.140227.actors.txt
agentfile_name   = Phoenix.140422.agents.txt
discardfile_name = Phoenix.140227.discards.txt
issuefile_name   = Phoenix.issues.140225.txt

[Options]
# textfile_list is a comma-delimited list of text files to code. This list has priority if
#               both a textfile_list and textfile_name are present
textfile_list = GigaWord.sample.PETR.txt
#textfile_list = AFP0808-01.txt, AFP0909-01.txt, AFP1210-01.txt
# textfile_name is the name of a file containing a list of names of files to code, one
# file name per line.
#textfile_name  = PETR.textfiles.benchmark.txt

# eventfile_name is the output file for the events
eventfile_name = events.PETR-Devel.txt


# INTERFACE OPTIONS: uncomment to activate
# Default: set all of these false, which is equivalent to an A)utocode in TABARI

# code_by_sentence: show events after each sentence has been coded; default is to
#                   show events after all of the sentences in a story have been coded
code_by_sentence = True
# pause_by_sentence: pause after the coding of each sentence. Entering 'Return' will
#                    cause the next sentence to be coded; entering any character will
#                    cause the program to exit. Default is to code without any pausing.
pause_by_sentence = True
# pause_by_story: pause after the coding of each story.
#pause_by_story = True


# CODING OPTIONS:
# Defaults are more or less equivalent to TABARI
```

```
# new_actor_length: Maximum length for new actors extracted from noun phrases if no
#                   actor or agent generating a code is found. To disable and just
#                   use null codes "---", set to zero; this is the default.
#                   Setting this to a large number will extract anything found in a (NP
#                   noun phrase, though usually true actors contain a small number of words
#                   This must be an integer.
new_actor_length = 0

# write_actor_root: If True, the event record will include the text of the actor root:
#                   The root is the text at the head of the actor synonym set in the
#                   dictionary. Default is False
write_actor_root = False

# write_actor_text: If True, the event record will include include the complete text of
#                   the noun phrase that was used to identify the actor.  Default is False
write_actor_text = False

# require_dyad: Events require a non-null source and target: setting this false is likely
#               to result in a very large number of nonsense events. As happened with the
#               infamous GDELT data set of 2013-2014. And certainly no one wants to see
#               that again.
require_dyad = True

# stop_on_error: If True, parsing errors causing the program to halt; typically used for
#                debugging. With the default [false], the error is written to the error
#                file, record is skipped, and processing continues.
stop_on_error = False


[StanfordNLP]
stanford_dir = ~/stanford-corenlp/
```

Internal Data Structures

The main data format within PETRARCH is a Python dictionary that is structured around unique story IDs as the keys for the dictionary and another dictionary as the value. The value dictionary contains the relevant information for the sentences within the story, and the meta information about the story such as the date and source. The broad format of this internal dictionary is:

```
{story_id: {'sents': {0: {'content': 'String of content', 'parsed': 'StanfordNLP parse tree',
                          'coref': 'Optional list of corefs', 'events': 'List of coded events',
                          'issues': 'Optional list of issues'},
                      1: {'content': 'String of content', 'parsed': 'StanfordNLP parse tree',
                          'coref': 'Optional list of corefs', 'events': 'List of coded events',
                          'issues': 'Optional list of issues'}
                      }
            'meta': {'date': 'YYYYMMDD', 'other': "This is the holding dict for misc info."}
        },
 story_id: {'sents': {0: {'content': 'String of content', 'parsed': 'StanfordNLP parse tree',
                          'coref': 'Optional list of corefs', 'events': 'List of coded events',
                          'issues': 'Optional list of issues'},
                      1: {'content': 'String of content', 'parsed': 'StanfordNLP parse tree',
                          'coref': 'Optional list of corefs', 'events': 'List of coded events',
                          'issues': 'Optional list of issues'}
                      }
            'meta': {'date': 'YYYYMMDD', 'other': "This is the holding dict for misc info."}
```

```
        },
}
```

This consistent internal format allows for the easy extension of the program through external hooks.

PETRARCH Dictionary Formats

There are five separate input dictionaries or lists that PETRACH makes use of: the verb dictionary, the actor dictionary, the agent dictionary, the issues dictionary, and the discard list. The following sections describe these files in greater detail. In addition to this documentation, which is intended for individuals planning to work on dictionaries, the source code contains internal documentation on how the dictionary information is stored by the program.

The PETRARCH dictionaries are generally derived from the earlier TABARI dictionaries, and information on those formats can be found in the TABARI manual:

http://eventdata.parusanalytics.com/tabari.dir/TABARI.0.8.4b2.manual.pdf

General Rules for dictionaries

All of the files are in "flat ASCII" format and should only be edited using a program that produces a a file without embedded control codes; for example Emacs or BBEdit.

Comments in input files:

Comments should be delineated with a hash sign #, as in Python or Unix Everything after this symbol and before the next newline will be ignored by the parser.

```
# this is a Python-like comment, inherited from Unix

something I want # followed by a Python-like comment
```

The program is *not* set up to handle clever variations like nested comments, multiple comments on a line, or non-comment information in multi-line comments: yes, we are perfectly capable of writing code that could handle these contingencies, but it is not a priority at the moment. We trust you can cope within these limits.

Blank lines and lines with only whitespace are also skipped.

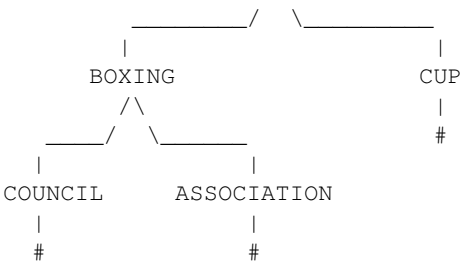
Storage in Memory

When the dictionaries are read by the program, they are read into memories as prefix trees, i.e. Tries. This makes searching the stored dictionaries very efficient during the parse, but adds to the memory overhead and can be somewhat confusing if you don't know what you're working with. This data structure stores each word at a node, and following a path in the tree will lead to a pattern. Let's take a small part of the discard list as an example:

```
WORLD BOXING ASSOCIATION
WORLD BOXING COUNCIL
WORLD CUP
```

These three entries would be stored in the following Trie:

```
      PETRglobals.DiscardList
                  |
                  |
                 WORLD
                  /\
```



Note that all patterns are terminated with a hash sign. This is to signify that there is a pattern that ends here. If no hash sign is present during a matching, then that would be an incomplete match. For the Issue and Actor/Agent dictionaries, the hash sign then links to a storage container with the information associated with the entry.

The Trie is the general principle underlying all of the dictionary storage in Petrarch. The Verb dictionary storage has its own quirks due to the increased complexity of patterns present, but it is still fundamentally a Trie. That will be discussed in the verb section.

Verb Dictionary

The verb dictionary file consists of a set of synsets followed by a series of verb synonyms and patterns.

Synsets:

Synonym sets (synsets) are labelled with a string beginning with & and defined using the label followed by a series of lines beginning with + containing words or phrases. The phrases are interpreted as requiring consecutive words; the words can be separated with underscores (they are converted to spaces). Synset phrases can only contain words, not `$`, `+`, `%` or `^` tokens. Synsets can be used anywhere in a pattern that a word or phrase can be used. A synset must be defined before it is used: a pattern containing an undefined synset will be ignored.

Regular plurals are generated automatically by adding 'S' to the root, adding 'IES' if the root ends in 'Y', and added 'ES' if the root ends in 'SS'. The method for handling irregular plurals is currently different for the verbs and agents dictionaries: these will be reconciled in the future, probably using the agents syntax. Plurals are not created when:

- The phrase ends with _.
- The label ends with _, in which case plurals are not generated for any of the phrases; this is typically used for synonym sets that do not involve nouns

The _ is dropped in both cases. Irregular plurals do not have a special syntax; just enter these as additional synonyms.

Verb Synonym Blocks and Patterns:

A verb synonym block is a set of verbs which are synonymous (or close enough) with respect to the patterns. The program automatically generates the regular forms of the verb if it is regular (and, implicitly, English); otherwise the irregular forms can be specified in `{...}` following the primary verb. An optional code for the isolated verb can follow in `[...]`.

The verb block begins with a comment of the form

```
--- <GENERAL DESCRIPTION> [<CODE>] ---
```

where the `---` signals the beginning of a new block. The code in `[...]` is the primary code – typically a two-digit+0 cue-category code – for the block, and this will be used for all other verbs unless these have their own code. If no code is present, this defaults to the null code `---` which indicates that the isolated verb does not generate an event. The null code also can be used as a secondary code.

Multiple-word verbs

Multiple-word "verbs" such as "CONDON OFF", "WIRE TAP" and "BEEF UP" are entered by connecting the words with an underscore and putting a '+' in front of the word in the phrase that is going to be identified as a verb. If there is no `{...}`, regular forms are constructed for the word designated by '+'; otherwise all of the irregular forms are given in `{...}`. If you can't figure out which part of the phrase is the verb, the phrase you are looking at is probably a noun, not a verb. Multi-word verbs are treated in patterns just as single-word verbs are treated.

Example:

```
+BEEF_UP
+CORDON_OFF {+CORDONED_OFF +CORDONS_OFF +CORDONING_OFF}
+COME_UPON {+COMES_UPON +CAME_UPON +COMING_UPON}
WIRE_+TAP {WIRE_+TAPS WIRE_+TAPPED  WIRE_+TAPPING }
```

A note on the current state of Petrarch's ability to handle compound verbs: The syntax parser we use (Stanford CoreNLP) often has trouble dealing with pre-compounded verbs, i.e. those where the extra stuff comes before the verb, and because Petrarch relies so heavily on this parser, meanings are sometimes missed. Post-compound verbs don't share this problem, and are more frequently parsed correctly.

Patterns

This is followed by a set of patterns – these begin with – – which are based roughly on the syntax from TABARI patterns, but the patterns in Petrarch's dictionaries also contain some syntactic annotation. Pattern lines begin with a -, and are followed by a five-part pattern:

```
- [Pre-Verb Nouns] [Pre-Verb Prepositoins] * [Post-verb Nouns] [Post-verb prepositions]
```

Any of these can be left empty. Singular nouns are left bare, and should be the "head" of the phrase they are a member of, e.g. the head of "Much-needed financial aid" would be "aid." If multiple nouns or adjectives are needed, then that phrase is put in braces as in `{FINANCIAL AID}`, where the last word is the head. Prepositional phrases are put in parentheses where the first element is the preposition, and the second element is a noun, or a braced noun phrase.

```
* (FOR AID)
* (FOR {FINANCIAL AID})
```

After these comes the CAMEO code in brackets. Make sure there is a space before the open brace. Then, a comment with the intended word to be matched is often included.

Note that these patterns do not contain other verbs. This is different from TABARI, and even earlier versions of Petrarch. This is to simplify the verbs dictionary, and make the pattern matching faster and more effective.

Combinations

Petrarch handles many verb-verb interactions automatically through its reformatting of CAMEO's semantic heirarchy (See utilities.convert_code for more). For instance, if it were parsing the phrase

" A will [help B]"

it would code "to help B" first, then the phrase would become "A will [_ B 0x0040]". And then since `help=0x0040` is a subcategory of `will=0x3000`, then it just adds them together, ending with the code `[A B 0x3040]`. This code is translated back into CAMEO for the final output, yielding `[A B 033]`. This process works for most instances where the idea of the phrase as a whole is a combination of the ideas of its children.

Transformations

Sometimes these verb-vertb interactions aren't represented in the ontology. It is possible to specify what happens when one verb finds that it is acting on another verb. Say you wanted to convert phrases of the form "A said A will attack B" into " A threatens B." You would say

```
~ a (a b WILL_ATTACK) SAY = a b 138
```

This is effectively a postfix notational system, and every line starts with a ~. The first element is the topmost source actor, the last element is the topmost verb (the verbs in the patterns are converted to codes, so synonyms also match).

The inner parenthetical has the same format, with the first element being the lower source, the second the lower target, and the third the lower verb. It is possible to replace letter variables with a period '.' to represent "non-specified actor", or with an underscore _ to specify "non-present actor." Verbs can also be replaced with "Q" to mean "any verb."

These transformations are sometimes necessary, but most cases can be handled by the combination process.

Storage in Memory

The verb dictionary, when stored into memory, has three subdictionaries: words, patterns, and transformations.

The words portion contains the base verbs. They are stored as `VERB--STUFF BEFORE--#--STUFF AFTER--#--INFO`. For most verbs (i.e. those that are not compounds), The entry just goes `VERB -- # -- # -- INFO`.

The transformation contains almost a literal transcription of the pattern, ordered `VERB1--SOURCE1--VERB2--SOURCE2--TARGET2--INFO`.

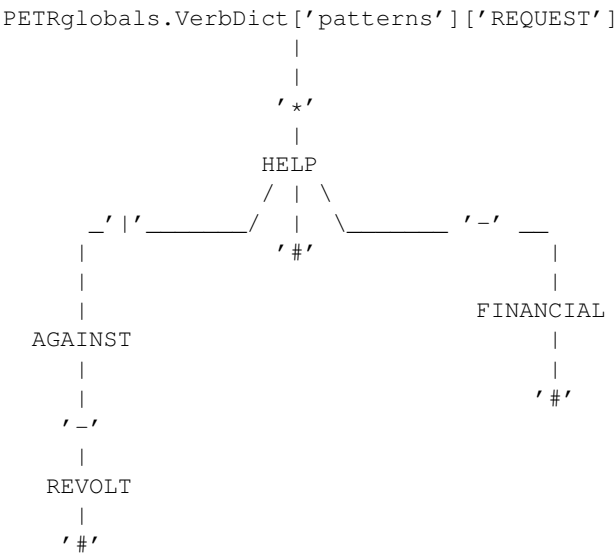
The verb patterns in memory have extra annotative symbols after every word to indicate the type of word that comes next. The very first word encountered is always a noun. Then it follows a series of rules that specify what comes next:

- Comma ',' = The next word is the same type as the previous one
- Asterisk '*' = The first half of the pattern is over, move to the second half
- Hash sign '#' = The pattern is over
- Pipe 'l' = The next word is a preposition
- Dash '-' = The next word is a specifier is a noun or extension of noun phrase

This means that when searching, we only have to check 5 cases, rather than compare to all remaining patterns. As an example, consider these three example entries under 'request':

```
* HELP
* {FINANCIAL HELP}
* HELP (AGAINST REVOLT)
```

They would be stored as



Note that "Financial Help" is stored as "Help Financial," because "Help" is the head of the phrase and is thus much easier to find, and once we've found that we can then look for "Financial."

Actor Dictionary

Actor dictionaries are similar to those used in TABARI (see Chapter 5 of the manual) except that the date restrictions must be on separate lines (in TABARI, this was optional) The general structure of the actors dictionary is a series of records of the form

```
[primary phrase]
[optional synonym phrases beginning with '+']
[optional date restrictions beginning with '\t']
```

A "phrase string" is a set of character strings separated by either blanks or underscores.

A "code" is a character string without blanks

A "date" has the form YYYYMMDD or YYMMDD. These can be mixed, e.g.

```
JAMES_BYRNES_  ; CountryInfo.txt
    [USAELI 18970101-450703]
    [USAGOV 450703-470121]
```

Primary phrase format:

```
phrase_string { optional [code] }
```

If the code is present, it becomes the default code if none of the date restrictions are satisfied. If it is not present and none of the restrictions are satisfied, this is equivalent to a null code

Synonym phrase

```
+phrase_string
```

Date restriction

```
\t[code restriction]
```

where `\t` is the tab character and the restriction [1]_ takes the form

```
<date : applies to times before date
>date : applies to times after date
date-date: applies to times between dates
```

The limits of the date restrictions are interpreted as "or equal to." A date restriction of the form `\t[code]` is the same as a default restriction.

Example:

```
# .actor file produced by translate.countryinfo.pl from CountryInfo.120106.txt
# Generated at: Tue Jan 10 14:09:48 2012
# Version: CountryInfo.120106.txt

AFGHANISTAN_  [AFG]
+AFGHAN_
+AFGANISTAN_
+AFGHANESTAN_
+AFGHANYSTAN_
+KABUL_
+HERAT_

MOHAMMAD_ZAHIR_SHAH_  ; CountryInfo.txt
        [AFGELI 320101-331108]
        [AFGGOV 331108-730717]
        [AFGELI 730717-070723]
```

```
ABDUL_QADIR_  ; CountryInfo.txt
+NUR_MOHAMMAD_TARAKI_  ; CountryInfo.txt
+HAFIZULLAH_AMIN_  ; CountryInfo.txt
        [AFGELI 620101-780427]
        [AFGGOV 780427-780430]
        [AFGELI]

HAMID_KARZAI_  [AFGMIL]; CountryInfo.txt
+BABRAK_KARMAL_  ; CountryInfo.txt
+SIBGHATULLAH_MOJADEDI_  ; CountryInfo.txt
        [AFGGOV 791227-861124]
        [AFGGOV 791227-810611]
```

Detecting actors which are not in the dictionary

Because PETRARCH uses parsed input, it has the option of detecting actors—noun phrases—which are not in the dictionary. This is set using the `new_actor_length` option in the `PETR_config.ini` file: see the description of that file for details.

Agent Dictionary

Basic structure of the agents dictionary is a series of records of the form

```
phrase_string {optional plural}  [agent_code]
```

A "phrase string" is a set of character strings separated by either blanks or underscores. As with the verb patterns, a blank between words means that additional words can occur between the previous word and the next word; a _ (underscore) means that the words must be consecutive.

An "agent_code" is a character string without blanks that is either preceded (typically) or followed by ~. If the ~ precedes the code, the code is added after the actor code; if it follows the code, the code is added before the actor code (usually done for organizations, e.g. `NGO~`)

Plurals:

Regular plurals – those formed by adding 'S' to the root, adding 'IES' if the root ends in 'Y', and added 'ES' if the root ends in 'SS' – are generated automatically

If the plural has some other form, it follows the root inside `{...}` [1]_

If a plural should not be formed – that is, the root is only singular or only plural, or the singular and plural have the same form (e.g. "police"), use a null string inside `{}`.

If there is more than one form of the plural – "attorneys general" and "attorneys generals" are both in use – just make a second entry with one of the plural forms nulled (though in this instance – ain't living English wonderful? – you could null the singular and use an automatic plural on the plural form) Though in a couple test sentences, this phrase confused the CoreNLP parser.

Substitution Markers:

These are used to handle complex equivalents, notably

```
!PERSON! = MAN, MEN, WOMAN, WOMEN, PERSON
!MINST! = MINISTER, MINISTERS, MINISTRY, MINISTRIES
```

and used in the form

```
CONGRESS!PERSON! [~LEG}
!MINIST!_OF_INTERNAL_AFFAIRS
```

The marker for the substitution set is of the form `!...!` and is followed by an = and a comma-delimited list; spaces are stripped from the elements of the list so these can be added for clarity. Every item in the list is substituted for the marker, with no additional plural formation, so the first construction would generate

```
CONGRESSMAN [~LEG}
CONGRESSMEN [~LEG}
CONGRESSWOMAN [~LEG}
CONGRESSWOMEN [~LEG}
CONGRESSPERSON [~LEG}
```

Example:

```
<!-- PETRARCH VALIDATION SUITE AGENTS DICTIONARY -->
<!-- VERSION: 0.1 -->
<!-- Last Update: 27 November 2013 -->

PARLIAMENTARY_OPPOSITION {} [~OPP] #jap 11 Oct 2002
AMBASSADOR [~GOV] # LRP 02 Jun 2004
COPTIC_CHRISTIAN [~CHRCPT] # BNL 10 Jan 2002
FOREIGN_MINISTER [~GOVFRM] # jap 4/14/01
PRESIDENT [~GOVPRS] # ns 6/26/01
AIR_FORCE {} [~MIL] # ab 06 Jul 2005
OFFICIAL_MEDIA {} [~GOVMED] # ab 16 Aug 2005
ATTORNEY_GENERAL {ATTORNEYS_GENERAL} [~GOVATG] # mj 05 Jan 2006
FOREIGN_MINISTRY [~GOV] # mj 17 Apr 2006
HUMAN_RIGHTS_ACTIVISTS  [NGM~] # ns 6/14/01
HUMAN_RIGHTS_BODY  [NGO~] # BNL 07 Dec 2001
TROOP {} [~MIL] # ab 22 Aug 2005
```

Discard List

The discard list is used to identify sentences that should not be coded, for example sports events and historical chronologies. [1] If the string, prefixed with `' '`, is found in the `<Text>...</Text>` sentence, the sentence is not coded. Prefixing the string with a + means the entire story is not coded with the string is found. If the string ends with _, the matched string must also end with a blank or punctuation mark; otherwise it is treated as a stem. The matching is not case sensitive.

Example:

```
+5K RUN #  ELH 06 Oct 2009
+ACADEMY AWARD   # LRP 08 Mar 2004
AFL GRAND FINAL   # MleH 06 Aug 2009
AFRICAN NATIONS CUP   # ab 13 Jun 2005
AMATEUR BOXING TOURNAMENT   # CTA 30 Jul 2009
AMELIA EARHART
ANDRE AGASSI   # LRP 10 Mar 2004
ASIAN CUP   # BNL 01 May 2003
ASIAN FOOTBALL   # ATS 9/27/01
ASIAN MASTERS CUP   # CTA 28 Jul 2009
+ASIAN WINTER GAMES   # sls 14 Mar 2008
ATP HARDCOURT TOURNAMENT   # mj 26 Apr 2006
ATTACK ON PEARL HARBOR   # MleH 10 Aug 2009
AUSTRALIAN OPEN
AVATAR   # CTA 14 Jul 2009
AZEROTH   # CTA 14 Jul 2009  (World of Warcraft)
```

[1] In TABARI, discards were intermixed in the `.actors` dictionary and `.verbs` patterns, using the `[###]` code. They are now a separate dictionary.

```
BADMINTON  # MleH 28 Jul 2009
BALLCLUB   # MleH 10 Aug 2009
BASEBALL
BASKETBALL
BATSMAN  # MleH 14 Jul 2009
BATSMEN  # MleH 12 Jul 2009
```

Issues List

The optional `Issues` dictionary is used to do simple string matching and return a comma-delimited list of codes. The standard format is simply a set of lines of the form

```
<string> [<code>]
```

For purposes of matching, a `' '` is added to the beginning and end of the string: at present there are no wild cards, though that is easily added.

The following expansions can be used (these apply to the string that follows up to the next blank):

```
n: Create the singular and plural of the noun
v: Create the regular verb forms ('S','ED','ING')
+: Create versions with ' ' and '-'
```

The file format allows # to be used as a in-line comment delimiter.

Issues are written to the event record as a comma-delimited list to a tab-delimited field, e.g.

```
20080801    ABC     EDF     0001    POSTSECONDARY_EDUCATION 2, LITERACY 1   AFP0808-01-M008-02
20080801    ABC     EDF     0004        AFP0808-01-M007-01
20080801    ABC     EDF     0001    NUCLEAR_WEAPONS 1       AFP0808-01-M008-01
```

where `XXXX NN`, corresponds to the issue code and the number of matched phrases in the sentence that generated the event.

This feature is optional and triggered by a file name in the `PETR_config.ini` file at `issuefile_name = Phoenix.issues.140225.txt`.

In the current code, the occurrence of an ignore phrase of either type cancels all coding of issues from the sentence.

Example:

```
<ISSUE CATEGORY="ID_ATROCITY">
n:atrocity [ID_ATROCITY]
n:genocide [ID_ATROCITY]
ethnic cleansing [ID_ATROCITY]
ethnic v:purge [ID_ATROCITY]
ethnic n:purge [ID_ATROCITY]
war n:crime [ID_ATROCITY]
n:crime against humanity [ID_ATROCITY]
n:massacre [ID_ATROCITY]
v:massacre [ID_ATROCITY]
al+zarqawi network [NAMED_TERROR_GROUP]
~Saturday Night massacre
~St. Valentine's Day massacre
~~Armenian genocide  # not coding historical cases
</ISSUE>
```

Input Formats

There are two input formats for PETRARCH: the processing pipeline and the XML input. The following sections describe the details of these input types and formats.

Pipeline

The pipeline input is made for integration with the processing pipeline. The processing pipeline has tight integration with a MongoDB instance. Thus, the relevant PETRARCH functions are designed to work with this input format. Specifically, the input is a list of dictionaries, with each dictionary holding an single entry in the MongoDB instance. The PETRARCH function to interface with the pipeline is `run_pipeline()`. This function is designed to be dropped into the main processing pipeline script with a call such as:

```
output = petrarch.run_pipeline(holding, write_output=False)
```

where `holding` is the list of dictionaries described above. For more information about `run_pipeline()` and its output formats, please view the relevant documentation.

XML Input

The main input format for PETRARCH is an XML document with each entry in the document a sentence or story to be parsed. The inputs can be either individual sentences or entire stories. Additionally, the input can contain pre-parsed information from StanfordNLP or just the plain text with the Stanford parse left up to TABARI. Whether the input is parsed or not is indicated using the `-P` flag in the PETRARCH command-line arguments.

In general, the XML format is:

```
<Sentences>

<Sentence date = "YYYYMMDD" id = "storyString_sent#" source = "AFP" sentence = "Boolean">
<Text>
</Text>
<Parse>
</Parse>
</Sentence>

</Sentences>
```

Again, the `<Parse></Parse>` blocks are optional. Each attribute of the entries has a fairly obvious role. The `date` attribute is the date of the entry in a `YYYYMMDD` format. The `id` attribute is a unique ID for the entry. If the entry is a single sentence, the format of the ID should be `storyString_sentNumber` or `ABCDEFGHIJKLM_1` which would indicate story `ABCDEFGHIJKLM` and sentence 1. The `sentence` attribute indicates whether the text in the entry is from a single sentence or a block of sentences, such as from the lead paragraph of a news story. Finally, the `source` attribute indicates what source the material came from, such as Agence-France Presse.

General record fields:

All of these tags should occur on their own lines.

`<Sentence>...</Sentence>`:

Delimits the record. The <Sentence...> tag can have the following fields: date: date of the text in YYYYMMDD format. This is required; if it is not present the record will be skipped

id: identification string in any format [optional] category:

category in any format; this is used by the <Include> and
<Exclude> options [optional]

place: code to be used for anonymous actors [optional]

`</Text>...</Text>`:

Delimits the source text. This is used only for the display. The tags should occur on their own lines

`<Parse>...</Parse>`:

Delimits the TreeBank parse tree text: this used only for the actual coding.

Contributing to PETRARCH

One of the primary goals of PETRARCH and PETRARCH2 is to be software useable by a broad community of researchers and end users. Towards this end, we welcome contributions from anyone and everyone.

The project and issue tracker are hosted on Github: https://github.com/openeventdata/petrarch2. Please feel free to open issues regarding the software on the Github page. Additionally, individuals may use our gitter channel for more real-time communication.

In general, we find the community values from scikit-learn to echo our own:

Our community, our values

We are a community based on openness and friendly, didactic, discussions.
We aspire to treat everybody equally, and value their contributions.
Decisions are made based on technical merit and consensus.
Code is not the only way to help the project. Reviewing pull requests, answering questions to help others on mailing lists or issues, organizing and teaching tutorials, working on the website, improving the documentation, are all priceless contributions.
We abide by the principles of openness, respect, and consideration of others of the Python Software Foundation: https://www.python.org/psf/codeofconduct/

Tests

Petrarch has a testing suite using pytest and TravisCI. This is run upon a pull request to GitHub, and notfies us if your version passes. If you want to test them yourself, just go into the main directory of Petrarch and run

$ py.test

and the tests will be run. If it fails any tests, the PR will probably not be accepted unless you provide a compelling reason.

Contributing Process

You can check out the latest version of the Phoenix Pipeline by cloning this repository using git.

```
git clone https://github.com/openeventdata/petrarch2.git
```

To contribute to the phoenix pipeline you should fork the repository, create a branch, add to or edit code, push your new branch to your fork of the phoenix pipeline on GitHub, and then issue a pull request. See the example below:

```
git clone https://github.com/YOUR_USERNAME/petrarch2.git
git checkout -b my_feature
git add... # stage the files you modified or added
git commit... # commit the modified or added files
git push origin my_feature
```

Commit messages should first be a line, no longer than 80 characters, that summarizes what the commit does. Then there should be a space, followed by a longer description of the changes contained in the commit. Since these comments are tied specifically to the code they refer to (and cannot be out of date) please be detailed.

Note that `origin` (if you are cloning the forked the phoenix pipeline repository to your local machine) refers to that fork on GitHub, *not* the original (upstream) repository `https://github.com/openeventdata/petrarch2.git`. If the upstream repository has changed since you forked and cloned it you can set an upstream remote:

```
git remote add upstream https://github.com/eventdata/petrarch2.git
```

You can then pull changes from the upstream repository and rebasing against the desired branch (in this example, development). You should always issue pull requests against the development branch.

```
git fetch upstream
git rebase upstream/development
```

More detailed information on the use of git can be found in the git documentation.

Coding Guidelines

The following are some guidelines on how new code should be written. Of course, there are special cases and there will be exceptions to these rules. However, following these rules when submitting new code makes the review easier so new code can be integrated in less time.

Uniformly formatted code makes it easier to share code ownership. The petrarch project tries to closely follow the official Python guidelines detailed in PEP8 that detail how code should be formatted and indented. Please read it and follow it.

In addition, we add the following guidelines:

- Use underscores to separate words in non-class names: n_samples rather than nsamples.
- Avoid multiple statements on one line. Prefer a line return after a control flow statement (if/for).
- Use relative imports for references inside petrarch.
- Please don't use `import *`. It is considered harmful by the official Python recommendations. It makes the code harder to read as the origin of symbols is no longer explicitly referenced, but most important, it prevents using a static analysis tool like pyflakes to automatically find bugs in petrarch. Use the numpy docstring standard in all your docstrings.

These docs draw heavily on the contributing guidelines for scikit-learn.

PETRARCH Package

petrarch2 Module

petrarch2.**check_discards**(*SentenceText*)
Checks whether any of the discard phrases are in SentenceText, giving priority to the + matches. Returns [indic, match] where indic

0 : no matches 1 : simple match 2 : story match [+ prefix]

petrarch2.**close_tex**(*fname*)

petrarch2.**do_coding**(*event_dict*)
Main coding loop Note that entering any character other than 'Enter' at the prompt will stop the program: this is deliberate. <14.02.28>: Bug: PETRglobals.PauseByStory actually pauses after the first

sentence of the *next* story

petrarch2.**get_issues**(*SentenceText*)
Finds the issues in SentenceText, returns as a list of [code,count]

<14.02.28> stops coding and sets the issues to zero if it finds *any* ignore phrase

petrarch2.**get_version**()

petrarch2.**main**()

petrarch2.**open_tex**(*filename*)

petrarch2.**parse_cli_args**()
Function to parse the command-line arguments for PETRARCH2.

petrarch2.**read_dictionaries**(*validation=False*)

petrarch2.**run**(*filepaths*, *out_file*, *s_parsed*)

petrarch2.**run_pipeline**(*data*, *out_file=None*, *config=None*, *write_output=True*, *parsed=False*)

PETRglobals Module

PETRreader Module

exception PETRreader.**DateError**
Bases: exceptions.Exception

PETRreader.**check_attribute**(*targattr*)
Looks for targetattr in AttributeList; returns value if found, null string otherwise.

PETRreader.**close_FIN**()

PETRreader.**dstr_to_ordate**(*datestring*)
Computes an ordinal date from a Gregorian calendar date string YYYYMMDD or YYMMDD.

PETRreader.**extract_attributes**(*theline*)
Structure of attributes extracted to AttributeList At present, these always require a quoted field which follows an '=', though it probably makes sense to make that optional and allow attributes without content

PETRreader.**find_tag**(*tagstr*)

PETRreader.**get_attribute**(*targattr*)
Similar to check_attribute() except it raises a MissingAttr error when the attribute is missing.

PETRreader.**make_noun_list**(*nounst*)

PETRreader.**make_plural_noun**(*noun*)
Create the plural of a synonym noun st

PETRreader.**open_FIN**(*filename*, *descrstr*)

PETRreader.**parse_Config**(*config_path*)
Parse PETRglobals.ConfigFileName. The file should be ; the default is PETR_config.ini in the working directory but this can be changed using the -c option in the command line. Most of the entries are obvious (but will eventually be documented) with the exception of

1. **actorfile_list and textfile_list are comma-delimited lists. Per the usual rules** for Python config files, these can be continued on the next line provided the the first char is a space or tab.
2. If both textfile_list and textfile_name are present, textfile_list takes priority. textfile_list should be the name of a file containing text file names; # is allowed as a comment delimiter at the beginning of individual lines and following the file name.
3. **For additional info on config files, see** http://docs.python.org/3.4/library/configparser.html

   or try Google, but basically, it is fairly simple, and you can probably just follow the examples.

PETRreader.**read_FIN_line**()
def read_FIN_line(): Reads a line from the input stream fin, deleting xml comments and lines beginning with # returns next non-empty line or EOF tracks the current line number (FINnline) and content (FINline) calling function needs to handle EOF (len(line) == 0)

PETRreader.**read_actor_dictionary**(*actorfile*)
This is a simple dictionary of dictionaries indexed on the words in the actor string. The final node has the key '#' and contains codes with their date restrictions and, optionally, the root phrase in the case of synonyms.

Example:

UFFE_ELLEMANN_JENSEN_ [IGOEUREEC 820701-821231][IGOEUREEC 870701-871231] # president of the CoEU from DENMARK# IGOrulers.txt

the actor above is stored as:

{u'UFFE': {u'ELLEMANN': {u'JENSEN': {u'#': [(u'IGOEUREEC', [u'820701', u'821231']), (u'IGOEUREEC', [u'870701', u'871231'])]}}}}

PETRreader.**read_agent_dictionary**(*agent_path*)
Reads an agent dictionary Agents are stored in a simpler version of the Actors dictionary: a list of phrases keyed on the first word of the phrase. The individual phrase lists begin with the code, the connector from the key, and then a series of 2-tuples containing the remaining words and connectors. A 2-tuple of the form ('', ' ') signals the end of the list.

Connector: blank: words can occur between the previous word and the next word _ (underscore): words must be consecutive: no intervening words

FORMATTING OF THE AGENT DICTIONARY [With some additional coding, this can be relaxed, but anything following these rules should read correctly] Basic structure is a series of records of the form

phrase_string {optional plural} [agent_code]

Material that is ignored 1. Anything following '#' 2. Any line beginning with '#' or '<!' 3. Any null line (that is, line consisting of only

A "phrase string" is a set of character strings separated by either blanks or underscores.

A "agent_code" is a character string without blanks that is either preceded (typically) or followed by '~'. If the '~' precedes the code, the code is added after the actor code; if it follows the code, the code is added before the actor code (usually done for organizations, e.g. NGO~)

Plurals: Regular plurals – those formed by adding 'S' to the root, adding 'IES' if the root ends in 'Y', and added 'ES' if the root ends in 'SS' – are generated automatically

If the plural has some other form, it follows the root inside {...}

If a plural should not be formed – that is, the root is only singular or only plural, or the singular and plural have the same form (e.g. "police"), use a null string inside {}.

If there is more than one form of the plural – "attorneys general" and "attorneys generals" are both in use – just make a second entry with one of the plural forms nulled (though in this instance – ain't living English wonderful? – you could null the singular and use an automatic plural on the plural form) Though in a couple test sentences, this phrase confused SCNLP.

Substitution Markers: These are used to handle complex equivalents, notably

!PERSON! = MAN, MEN, WOMAN, WOMEN, PERSON !MINST! = MINISTER, MINISTERS, MINISTRY, MINISTRIES

and used in the form

CONGRESS!PERSON! [~LEG] !MINIST!_OF_INTERNAL_AFFAIRS

The marker for the substitution set is of the form !...! and is followed by an = and a comma-delimited list; spaces are stripped from the elements of the list so these can be added for clarity. Every time in the list is substituted for the marker, with no additional plural formation, so the first construction would generate

CONGRESSMAN [~LEG] CONGRESSMEN [~LEG] CONGRESSWOMAN [~LEG] CONGRESSWOMEN [~LEG] CONGRESSPERSON [~LEG]

== Example === <!– PETRARCH VALIDATION SUITE AGENTS DICTIONARY –> <!– VERSION: 0.1 –> <!– Last Update: 27 November 2013 –>

PARLIAMENTARY_OPPOSITION {} [~OPP] #jap 11 Oct 2002 AMBASSADOR [~GOV] # LRP 02 Jun 2004 COPTIC_CHRISTIAN [~CHRCPT] # BNL 10 Jan 2002 FOREIGN_MINISTER [~GOVFRM] # jap 4/14/01 PRESIDENT [~GOVPRS] # ns 6/26/01 AIR_FORCE {} [~MIL] # ab 06 Jul 2005 OFFICIAL_MEDIA {} [~GOVMED] # ab 16 Aug 2005 ATTORNEY_GENERAL {ATTORNEYS_GENERAL} [~GOVATG] # mj 05 Jan 2006 FOREIGN_MINISTRY [~GOV] # mj 17 Apr 2006 HUMAN_RIGHTS_ACTIVISTS [NGM~] # ns 6/14/01 HUMAN_RIGHTS_BODY [NGO~] # BNL 07 Dec 2001 TROOP [~MIL] # ab 22 Aug 2005

PETRreader.**read_discard_list**(*discard_path*)

Reads file containing the discard list: these are simply lines containing strings. If the string, prefixed with ' ', is found in the <Text>...</Text> sentence, the sentence is not coded. Prefixing the string with a '+' means the entire story is not coded with the string is found [see read_record() for details on story/sentence identification]. If the string ends with '_', the matched string must also end with a blank or punctuation mark; otherwise it is treated as a stem. The matching is not case sensitive.

The file format allows # to be used as a in-line comment delimiter.

File is stored as a simple list and the interpretation of the strings is done in check_discards()

===== EXAMPLE ===== +5K RUN # ELH 06 Oct 2009 +ACADEMY AWARD # LRP 08 Mar 2004 AFL GRAND FINAL # MleH 06 Aug 2009 AFRICAN NATIONS CUP # ab 13 Jun 2005 AMATEUR BOXING TOURNAMENT # CTA 30 Jul 2009 AMELIA EARHART ANDRE AGASSI # LRP 10 Mar 2004 ASIAN CUP # BNL 01 May 2003 ASIAN FOOTBALL # ATS 9/27/01 ASIAN MASTERS CUP # CTA 28 Jul 2009 +ASIAN WINTER GAMES # sls 14 Mar 2008 ATP HARDCOURT TOURNAMENT # mj 26 Apr 2006 ATTACK ON PEARL HARBOR # MleH 10 Aug 2009 AUSTRALIAN OPEN AVATAR # CTA 14 Jul 2009 AZEROTH # CTA 14 Jul 2009 (World of Warcraft) BADMINTON # MleH 28 Jul 2009 BALLCLUB # MleH 10 Aug 2009 BASEBALL BASKETBALL BATSMAN # MleH 14 Jul 2009 BATSMEN # MleH 12 Jul 2009

PETRreader.**read_issue_list**(*issue_path*)

"Issues" do simple string matching and return a comma-delimited list of codes. The standard format is simply

<string> [<code>]

For purposes of matching, a ' ' is added to the beginning and end of the string: at present there are not wild cards, though that is easily added.

The following expansions can be used (these apply to the string that follows up to the next blank)

n: Create the singular and plural of the noun v: Create the regular verb forms ('S','ED','ING') +: Create versions with ' ' and '-'

The file format allows # to be used as a in-line comment delimiter.

File is stored in PETRglobals.IssueList as a list of tuples (string, index) where index refers to the location of the code in PETRglobals.IssueCodes. The coding is done in check_issues()

Issues are written to the event record as a comma-delimited list to a tab-delimited field, e.g.

20080801 ABC EDF 0001 POSTSECONDARY_EDUCATION 2, LITERACY 1 AFP0808-01-M008-02 20080801 ABC EDF 0004 AFP0808-01-M007-01 20080801 ABC EDF 0001 NUCLEAR_WEAPONS 1 AFP0808-01-M008-01

where XXXX NN, corresponds to the issue code and the number of matched phrases in the sentence that generated the event.

This feature is optional and triggered by a file name in the PETR_config.ini file at

issuefile_name = Phoenix.issues.140225.txt

<14.02.28> NOT YET FULLY IMPLEMENTED The prefixes '~' and '~~' indicate exclusion phrases:

~ [if the string is found in the current sentence, do not code any of the issues] in section – delimited by <ISSUE CATEGORY="...">...</ISSUE> – containing the string

~~ [if the string is found in the current *story*, do not code any of the issues] in section

In the current code, the occurrence of an ignore phrase of either type cancels all coding of issues from the sentence

===== EXAMPLE =====

<ISSUE CATEGORY="ID_ATROCITY"> n:atrocity [ID_ATROCITY] n:genocide [ID_ATROCITY] ethnic cleansing [ID_ATROCITY] ethnic v:purge [ID_ATROCITY] ethnic n:purge [ID_ATROCITY] war n:crime [ID_ATROCITY] n:crime against humanity [ID_ATROCITY] n:massacre [ID_ATROCITY] v:massacre [ID_ATROCITY] al+zarqawi network [NAMED_TERROR_GROUP] ~Saturday Night massacre ~St. Valentine's Day massacre ~~Armenian genocide # not coding historical cases </ISSUE>

PETRreader.**read_pipeline_input**(*pipeline_list*)

Reads input from the processing pipeline and MongoDB and creates the global holding dictionary. Please consult the documentation for more information on the format of the global holding dictionary. The function iteratively parses each file so is capable of processing large inputs without failing.

Parameters **pipeline_list: List.** :

List of dictionaries as stored in the MongoDB instance. These records are originally generated by the web scraper.

Returns **holding: Dictionary.** :

Global holding dictionary with StoryIDs as keys and various sentence- and story-level attributes as the inner dictionaries. Please refer to the documentation for greater information on the format of this dictionary.

PETRreader.**read_verb_dictionary**(*verb_path*)

Verb storage:

Storage sequence:

Upper Noun phrases

Upper prepositional phrases

•

Lower noun phrases

Lower prepositional phrases #

•symbol acts as extender, indicating the noun phrase is longer

, symbol acts as delimiter between several selected options

PETRreader.**read_xml_input**(*filepaths*, *parsed=False*)

Reads input in the PETRARCH XML-input format and creates the global holding dictionary. Please consult the documentation for more information on the format of the global holding dictionary. The function iteratively parses each file so is capable of processing large inputs without failing.

Parameters **filepaths: List.** :

List of XML files to process.

parsed: Boolean. :

Whether the input files contain parse trees as generated by StanfordNLP.

Returns **holding: Dictionary.** :

Global holding dictionary with StoryIDs as keys and various sentence- and story-level attributes as the inner dictionaries. Please refer to the documentation for greater information on the format of this dictionary.

PETRreader.**show_verb_dictionary**(*filename=u''*)

PETRwriter Module

PETRwriter.**get_actor_text**(*meta_strg*)

Extracts the source and target strings from the meta string.

PETRwriter.**pipe_output**(*event_dict*)

Format the coded event data for use in the processing pipeline.

Parameters **event_dict: Dictionary.** :

The main event-holding dictionary within PETRARCH.

Returns **final_out: Dictionary.** :

StoryIDs as the keys and a list of coded event tuples as the values, i.e., {StoryID: [(full_record), (full_record)]}. The `full_record` portion is structured as (story_date, source, target, code, joined_issues, ids, StorySource) with the `joined_issues` field being optional. The issues are joined in the format of ISSUE,COUNT;ISSUE,COUNT. The IDs are joined as ID;ID;ID.

PETRwriter.**write_events**(*event_dict*, *output_file*)

Formats and writes the coded event data to a file in a standard event-data format.

Parameters **event_dict: Dictionary.** :

The main event-holding dictionary within PETRARCH.

output_file: String. :

Filepath to which events should be written.

PETRwriter.**write_nullactors**(*event_dict*, *output_file*)

Formats and writes the null actor data to a file as a set of lines in a JSON format.

Parameters **event_dict: Dictionary.** :

The main event-holding dictionary within PETRARCH.

output_file: String. :

Filepath to which events should be written.

PETRwriter.**write_nullverbs**(*event_dict*, *output_file*)

Formats and writes the null verb data to a file as a set of lines in a JSON format.

Parameters **event_dict: Dictionary.** :

The main event-holding dictionary within PETRARCH.

output_file: String. :

Filepath to which events should be written.

utilities Module

utilities.**code_to_string**(*events*)

Converts an event into a string, replacing the integer codes with strings representing their value in hex

utilities.**combine_code**(*selfcode*, *to_add*)

Combines two verb codes, part of the verb interaction framework

Parameters **selfcode,to_add: ints** :

Upper and lower verb codes, respectively

Returns **combined value** :

utilities.**convert_code**(*code*, *forward=1*)

Convert a verb code between CAMEO and the Petrarch internal coding ontology.

New coding scheme:

0 0 0 0 2 Appeal 1 Reduce 1 Meet 1 Leadership 3 Intend 2 Yield 2 Settle 2 Policy 4 Demand 3 Mediate 3 Rights 5 Protest 4 Aid 4 Regime 6 Threaten 5 Expel 5 Econ 1 Say 6 Pol. Change 6 Military 7 Disapprove 7 Mat. Coop 7 Humanitarian 8 Posture 8 Dip. Coop 8 Judicial 9 Coerce 9 Assault 9 Peacekeeping A Investigate A Fight A Intelligence B Consult B Mass violence B Admin. Sanctions

C Dissent D Release

E Int'l Involvement F D-escalation

In the first column, higher numbers take priority. i.e. "Say + Intend" is just "Intend" or "Intend + Consult" is just Consult

Parameters **code: string or int, depending on forward** :

Code to be converted

forward: boolean :

Direction of conversion, True = CAMEO -> PICO

Returns Forward mode: :

active, passive [int] The two parts of the code [XXX:XXX], converted to the new system. The first is an inherent active meaning, the second is an inherent passive meaning. Both are not always present, most codes just have the active.

utilities.**extract_phrases**(*sent_dict*, *sent_id*)

Text extraction for PETRglobals.WriteActorText and PETRglobals.WriteEventText

Parameters story_dict: Dictionary. :

Story-level dictionary as stored in the main event-holding dictionary within PETRARCH.

story_id: String. :

Unique StoryID in standard PETRARCH format.

Returns text_dict: Dictionary indexed by event 3-tuple. :

List of texts in the order [source_actor, target_actor, event]

utilities.**init_logger**(*logger_filename*)

utilities.**nulllist = []**

<16.06.27 pas> This might be better placed in PETRtree but I'm leaving it here so that it is clear it is a global. Someone who can better grok recursion than I might also be able to eliminate the need for it.

utilities.**parse_to_text**(*parse*)

utilities.**story_filter**(*story_dict*, *story_id*)

One-a-story filter for the events. There can only be only one unique (DATE, SRC, TGT, EVENT) tuple per story.

Parameters story_dict: Dictionary. :

Story-level dictionary as stored in the main event-holding dictionary within PETRARCH.

story_id: String. :

Unique StoryID in standard PETRARCH format.

Returns filtered: Dictionary. :

Holder for filtered events with the format {(EVENT TUPLE): {'issues': [], 'ids': []}} where the 'issues' list is optional.

PETRtree Module

class PETRtree.**NounPhrase**(*label*, *date*, *sentence*)

Bases: PETRtree.Phrase

Class specific to noun phrases.

Methods: get_meaning() - specific version of the super's method check_date() - find the date-specific version of an actor

Methods

check_date(*match*)
Method for resolving date restrictions on actor codes.

Parameters **match: list** :

Dates and codes from the dictionary

Returns **code: string** :

The code corresponding to how the actor should be coded given the date

convert_existential()

get_meaning()

get_text()
Noun-specific get text method

return_meaning()

class PETRtree.**Phrase**(*label*, *date*, *sentence*)
This is a general class for all Phrase instances, which make up the nodes in the syntactic tree. The three subtypes are below.

Methods

get_head()
Method for finding the head of a phrase. The head of a phrase is the rightmost word-level constituent such that the path from root to head consists only of similarly-labeled phrases.

Parameters **self: Phrase object that called the method** :

Returns **possibilities[-1]: tuple (string,NounPhrase)** :

(The text of the head of the phrase, the NounPhrase object whose rightmost child is the head).

get_meaning()
Method for returning the meaning of the subtree rooted by this phrase, is overwritten by all subclasses, so this works primarily for S and S-Bar phrases.

Parameters **self: Phrase object that called the method** :

Returns **events: list** :

Combined meanings of the phrases children

get_parse_string()
recursive rendering of labelled phrase element and children as a string: when called from ROOT it returns the original input string

get_parse_text()
This is a fairly specific debugging function: to recover the original parse, use indented_parse_print(self, level=0) or get_parse_string(self)

get_text()

indented_parse_print(*level=0*)
recursive print of labeled phrase elements and children with line feeds and indentation

mix_codes(*agents*, *actors*)
Combine the actor codes and agent codes addressing duplicates and removing the general "~PPL" if there's a better option.

Parameters **agents, actors** : Lists of their respective codes

Returns **codes: list** :

[Agent codes] x [Actor codes]

print_to_stdout(*indent*)

resolve_codes(*codes*)
Method that divides a list of mixed codes into actor and agent codes

Parameters **codes: list** :

Mixed list of codes

Returns **actorcodes: list** :

List of actor codes

agentcodes: list :

List of actor codes

return_head()

class PETRtree.**PrepPhrase**(*label*, *date*, *sentence*)
Bases: PETRtree.Phrase

Methods

get_meaning()
Return the meaning of the non-preposition constituent, and store the preposition.

get_prep()

class PETRtree.**Sentence**(*parse*, *text*, *date*)
Holds the information of a sentence and its tree.

Methods

class PETRtree.**VerbPhrase**(*label*, *date*, *sentence*)
Bases: PETRtree.Phrase

Subclass specific to Verb Phrases

Methods

__init__: Initialization and Instatiation is_valid: Corrects a known stanford error regarding miscoded noun phrases get_theme: Returns the coded target of the VP get_meaning: Returns event coding described by the verb phrase get_lower: Finds meanings of children get_upper: Finds grammatical subject get_code: Finds base verb code and calls match_pattern match_pattern: Matches the tree to a pattern in the Verb Dictionary get_S: Finds the closest S-level phrase above the verb match_transform: Matches an event code against transformation patterns in the dictionary	

check_passive()

Check if the verb is passive under these conditions:

1. Verb is -ed form, which is notated by stanford as VBD or VBN
2. Verb has a form of "be" as its next highest verb

Parameters **self: VerbPhrase object calling the method** :

Returns **self.passive: boolean** :

Whether or not it is passive

get_S()

Navigate up the tree following a VP path to find the closest s-level phrase. There is the extra condition that if the S-level phrase is a "TO"-phrase without a second subject specified, just so that "A wants to help B" will navigate all the way up to "A wants" rather than stopping at "to"

Parameters **self: VerbPhrase object that called the method** :

Returns **level: VerbPhrase object** :

Lowest non-TO S-level phrase object above the verb

get_code()

Match the codes from the Verb Dictionary.

Step 1. Check for compound verb matches

Step 2. Check for pattern matches via match_pattern() method

Parameters **self: VerbPhrase object that called the method** :

Returns **code: int** :

Code described by this verb, best read in hex

get_lower()

Find the meaning of the children of the VP, and whether or not there is a "not" in the VP.

If the VP has VP children, look only at these.

Otherwise, this function pretty much is identical to the NounPhrase.get_meaning() method, except that it doesn't look at word-level children, because it shouldn't have any.

Parameters **self: VerbPhrase object that called the method** :

Returns **self.lower: list** :

Actor codes or Event codes, depending on situation

negated: boolean :

Whether a "not" is present

get_meaning()

This determines the event coding of the subtree rooted in this verb phrase.

Four methods are key in this process: get_upper(), get_lower(), get_code() and match_transform().

First, get_meaning() gets the verb code from get_code()

Then, it checks passivity. If the verb is passive, then it looks within verb phrases headed by [by, from, in] for the source, and for an explicit target in verb phrases headed by [at,against,into,towards]. If no target is found, this space is filled with 'passive', a flag used later to assign a target if it is in the grammatical subject position.

If the verb is not passive, then the process goes:

1) call get_upper() and get_lower() to check for a grammatical subject and find the coding of the subtree and children, respectively.

2) If get_lower returned a list of events, combine those events with the upper and code, add to event list.

3.Otherwise, combine upper, lower, and code and add to event list

4) Check to see if there are S-level children, if so, combine with upper and code, add to list.

5.call match_transform() on all events in the list

Parameters self: VerbPhrase object that called the method :

Returns events: list :

List of events coded by the subtree rooted in this phrase.

get_theme()

This is used by the NounPhrase.get_meaning() method to determine relevant information in the VerbPhrase.

get_upper()

Finds the meaning of the specifier (NP sibling) of the VP.

Parameters self: VerbPhrase object that called the method :

Returns self.upper: List :

Actor codes of spec-VP

is_valid()

This method is largely to overcome frequently made Stanford errors, where phrases like "exiled dissidents" were marked as verb phrases, and treating them as such would yield weird parses.

Once such a phrase is identified because of its weird distribution, it is converted to a NounPhrase object

match_pattern()

Match the tree against patterns specified in the dictionary. For a more illustrated explanation of how this process works, see the Petrarch2.pdf file in the documentation.

Parameters self: VerbPhrase object that called the method :

Returns False if no match, dict of match if present. :

match_transform(*e*)

Check to see if the event e follows one of the verb transformation patterns specified at the bottom of the Verb Dictionary file.

If the transformation is present, adjust the event accordingly. If no transformation is present, check if the event is of the form:

a (b . Q) P , where Q is not a top-level verb.

and then convert this to (a b P+Q)

Otherwise, return the event as-is.

Parameters **e: tuple** :

Event to be transformed

Returns **t: list of tuples** :

List of modified events, since multiple events can come from one single event

return_S()

return_code()

return_lower()

return_meaning()

return_passive()

return_upper()

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

p

u

p

u

C

D

E

F

G

I

M

N

O

P

R

S

U

V

W