# Universal Petrarch

December 5, 2018

**Abstract**

Universal Petrarch is the fifth-generation of a series of event data coders. The sequential development is KEDS, TABARI,PETRARCH , PETRARCH2 and Universal Petrarch.The goal of Universal Petrarch is to build a robust language agnostic political event data coder. Universal Petrarch is a project which is highly motivated from PETRARCH2 (fourth generation) and aims at developing one structure for many languages, not just English. PETRARCH2 project views the sentence on the syntactic level. It uses constituency (phrase-structure tree) parse of sentences as inputs. But the problem with constituency parse is that it depends heavily on grammatical rules and is thus highly language-dependent. In order to overcome this problem, Universal Petrarch is developed which uses universal dependency parse of sentences. Dependency parse consists of a rooted tree representing the backbone of the syntactic structure with 40 grammatical relations between words. And Universal Dependencies uses a set of universal grammatical relations which can capture any dependency relation between any words in any language. Universal Petrarch now supports English, Spanish and Arabic texts. Universal Petrarch uses the same dictionaries as PETRARCH2 but combines the strength of PETRARCH and PETRARCH2 by supporting two different style of verb dictionaries: one is structural verb dictionary from PETRARCH2 and the other is linear and flexible verb dictionary fromPETRARCH .

## 1 Introduction and Motivation

### 1.1 Dependency tree structure

A constituency parsing breaks a tree into sub-phrases. The non-terminal nodes in the tree are the types of phrases and part of speech tags (POS), and the terminal nodes are the words in a sentence and the edges are unlabeled. Take a simple sentence as an example “Israel said a mortar bomb was launched at it from the Gaza strip on Tuesday”, its constituency parsing tree is shown in figure 1.

On the other hand, a dependency parsing connects two different words according to their relationships. Each node in the tree represents a word. The child nodes are the words which are dependent on the parent nodes, and the edges are labeled by their relationships. For the same sentence as above, its dependency parsing tree is shown in the figure 2. The equivalent CoNLL-U format can be seen in the Appendix 2.

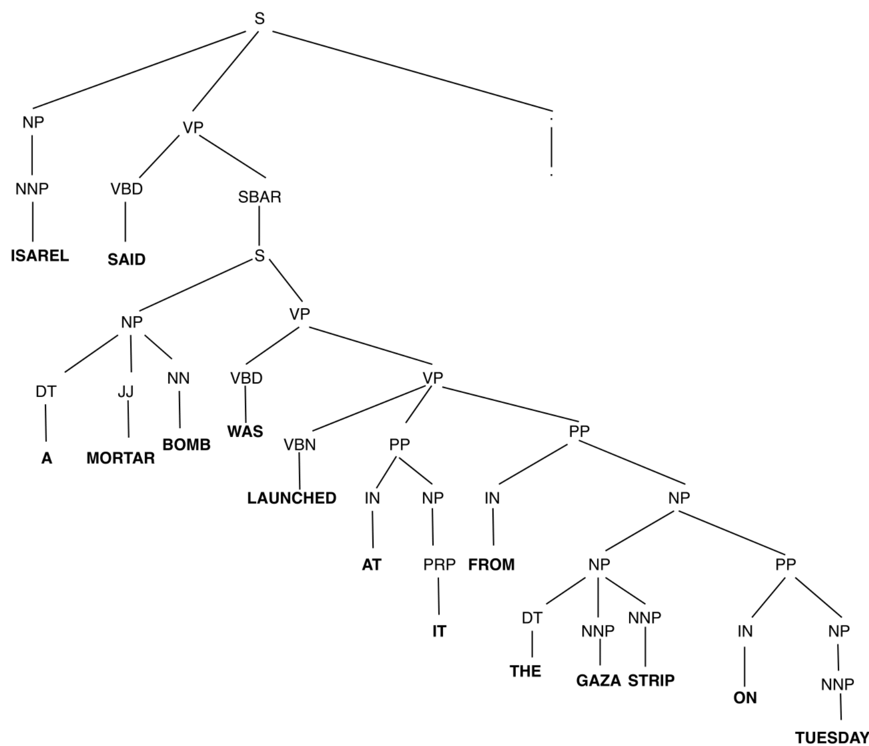


Figure 1: Constituency Parsing of the sentence

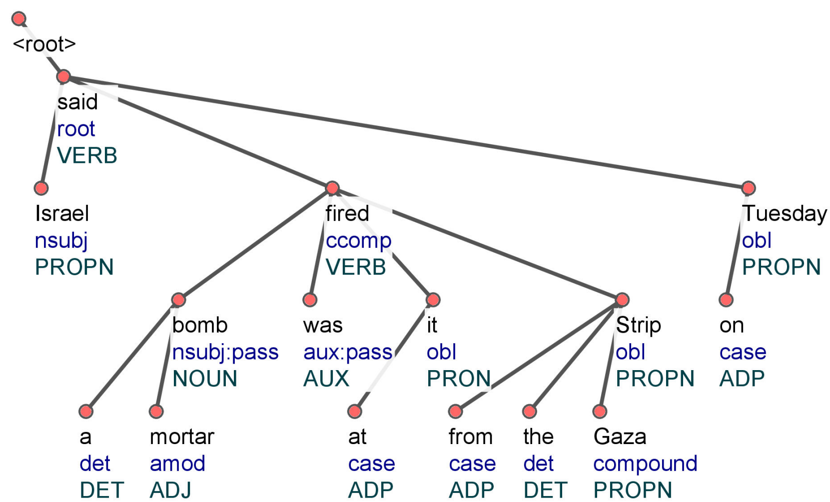


Figure 2: Dependency Parsing of the sentence

The dependency parse trees of dependency grammars see all nodes as terminal (e.g. words), which means they do not acknowledge the distinction between terminal and non-terminal categories. They are simpler on average than

constituency parse trees because they contain fewer nodes. Also, the phrase structure information (e.g., Noun Phrase, Verb Phrase) can be recovered from dependency structures.

### 1.2 Motivation behind usage of both PETRARCH and PETRARCH2 verb dictionaries

PETRARCH refers to Python Engine for Text Resolution And Related Coding Hierarchy. PETRARCH is designed to process fully-parsed news summaries in Penn Treebank format, from which 'whom-did-what-to-whom' relations are extracted.

PETRARCH2 takes much of the power of the originalPETRARCH's dictionaries and redirects it into a faster and smarter core logic. PETRARCH handled sentences largely as a list of words, incorporating some syntactic information. But, PETRARCH2 views the sentence entirely on the syntactic level. The logic for PETRARCH2 more strongly follows the tree structure provided by the TreeBank parse.

Despite of the advancements of PETRARCH2, for Universal Petrarch, both PETRARCH and PETRARCH2 verb dictionaries are used. PETRARCH2 aims to make the pattern matching faster by simplifying the verb dictionaries which involves keeping only one verb in a pattern, removing all the negation modifiers and annotating noun phrases and prepositional phrases. However, in some complex cases, for example, in Spanish double negation can be meaningful. PETRARCH dictionary provides the flexibility to contain patterns to handle those cases.

## 2 Generating Event Data with Universal Petrarch

In this section, we describe how event data is generated with Universal Petrarch. The work flow of Universal Petrarch can be abstracted to the following steps. More detail on each step is provided after the algorithm is described.

Given a sentence as the input:

- Preprocess the sentence by tokenization, tagging, lemmatization and dependency parsing (see Section 2.1)
- Identify all the noun phrases and verb phrases in the sentence;
- Identify root verbs in the sentence;
- if PETRARCH2 verb dictionary is used: (see Section 2.2)
  - Combine noun phrases and verb phrases into triplets (source, target, verb, other related nouns) based on dependency relations;
  - For each triplet, we do the following:
    - Match source and target in dictionary and identify their codes;
    - Match the verb in dictionary and identify its codes;
    - Match the verb phrase for patterns:
      - Match the pre-verb noun and pre-verb prepositional phrases (through *match_noun*() function);

        - Match the post-verb noun and post-verb prepositional phrases (through $match_lower()$ function);
        - Update the source and target
  - For each triplet which contains non-root verb, we do the following:
    - Resolve $root\,verb + verb$ interactions through transformations and combinations.
    - Return the event coding
  - If no events are returned in the last step, return the event coding of triplet which contains root verb

- if PETRARCH verb dictionary is used: (see Section 2.3)
  - For each verb phrase, we do the following:
    - Match the verb in dictionary and identifying their codes;
    - Match verb phrase for patterns and identify source and target;
      - Match the word sequence before verb (through $upper_match()$ function);
      - Match the word sequence after verb (through $petrarch1_verb_pattern_match()$ function);
    - From all matched patterns, identify the best matched pattern and return the event coding

A flow diagram is shown in the figure 3.

## 2.1 Preprocessing

When the input of Universal Petrarch are raw texts, some preprocessing needs to be done. Stanford CoreNLP[1] is used for sentence splitting and tokenization. For Arabic texts, an extra step before tokenization is required. Stanford Arabic Word Segmenter[2] is used to segment clitics from words. In Arabic, clitics are attached to a stem or to each other. When concatenated, clitics can generate a chain of up to four clitics before the stem (proclitics) and three clitics after the stem (enclitics). Clitics serve syntactic functions (such as negation, definition, conjunction or preposition). Segmenting clitics attached to words reduces lexical sparsity and simplifies syntactic analysis. Then, UDPipe [3] is used to extract morphological information (e.g., Parts Of Speech (POS) tags and lemmas of words) and generate dependency parse trees.

Note that the above steps are performed in a pipeline: sentence splitting, segmentation (only for Arabic), tokenization, tagging, lemmatization and dependency parsing, which means that each step depends on the output of previous step.

## 2.2 Getting events from petrarch2 patterns

This section explains the each step mentioned in flow of matching PETRARCH2 verb dictionary depicted in Section 2 in details.

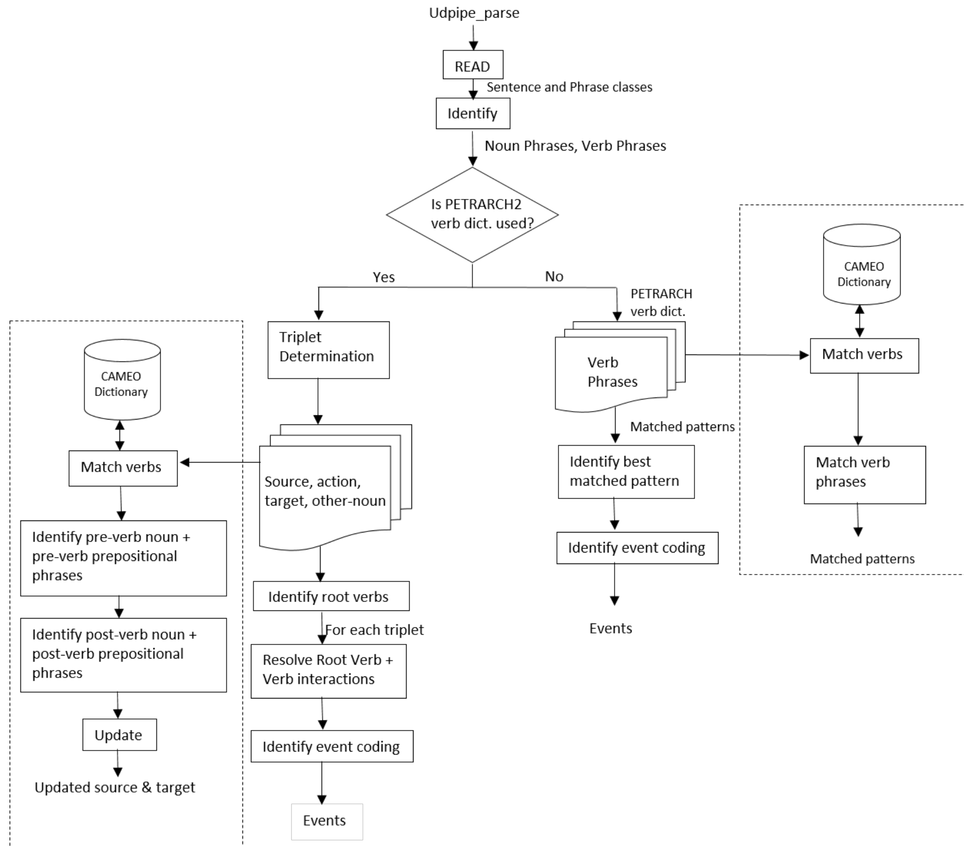


Figure 3: Work Flow of Event Coding in Universal Petrarch

### 2.2.1 Getting triplets from sentences

The first step of the process is to identify and extract phrases of the sentence from dependency relations and convert them into triplets, which are usually composed by source, target, the verb and other-nouns.

The method invoked to extract these triplets is called *get_phrase*(), and it goes through all the words in the sentence, identifying the verbs (including multi-word verbs and single word verbs) based on the *universal PoS tags*, which mark the core part-of-speech categories through the classes expressed in Table 1 (see fourth column of Table 2 in Appendices as an example). The negation modifiers are also identified based on dependency relations in this step.

Then, for each identified verb, method *get_source_target*() identifies all the noun and prepositional phrases related to it, and categorizes them as source, target and other-noun. Specifically, the head of source, target and other-nouns are decided by the following dependencies relations:

**source**: nsubj

**target**: obj, dobj, iobj, nsubjpass, nsubj:pass

**other-noun**: nmod, obl, advmod

Then the complete noun phrase including all the modifiers are extracted by traversing the subtree of the head of noun phrase. Note that the source, target extracted here are purely based on grammatical rules. They are potentially the source actor and target actor of the output events. Whether they should belong to the part of the action or not is decided in the pattern matching phase (see section 2.2.3). Other-noun is used as a place to store all the other noun phrases that are related to the verb. For instance, in the sentence "President Obama expressed sorrow for the horrific shootings in South Carolina", the output of this step is: source "President Obama", target "sorrow" and other-noun "the horrific shootings in South Carolina".

There are three important tasks which are also performed while getting phrases. First, the voice of the verb (*nsubjpass*) is identified. Second, the *conjunct* relations between verbs are solved, by identifying the source and target for both verbs.

Specifically in universal dependency, a *conjunct* is the relation between two elements connected by a coordinating conjunction, such as "and", "or", etc. Conjunctions are treated asymmetrically (the head of the relation is the first conjunct and all the other conjuncts depend on it via the *conj* relation.

For instance, consider the following sentence: "Ukraine ratified a sweeping agreement with the European Union and savored a historic triumph."

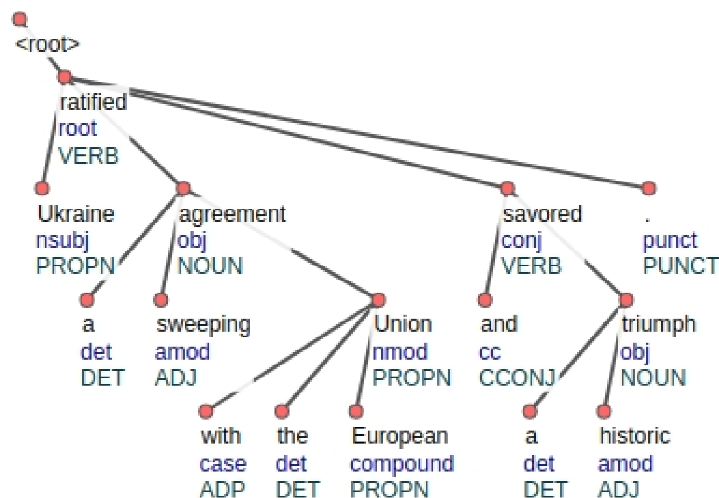


Figure 4: Dependency Parsing Tree for conjunct example.

Note in the universal dependency tree depicted in Figure 4 that there is a conjunct relation between verbs *ratified* and *savored*. In this case, Universal Petrarch successfully identifies both triplets $\langle Ukraine, savored, triumph\rangle$ and $\langle Ukraine, ratified, agreement\rangle$.

Third, the compound noun phrases are handled. For instance, in the sentence "Brazil and the United States are seeking...", two sources are identified: "Brazil" and "the United States" and two triplets are formed accordingly: $\langle Brazil, seeking, ...\rangle$ and $\langle the\,United\,States, seeking, ...\rangle$

### 2.2.2 Identifying noun phrase codes

For each triplet identified above, the coder then identifies the codes of source and target.

First, it extracts the smallest core sub-phrase of the noun phrases by removing all the prepositional phrases, noun modifiers and adjective modifiers from it. This step is necessary because sometimes the non-core modifiers may be matched in the dictionary, especially noun modifiers in complex phrases. For example, for the noun phrase "The Al Qaeda-linked Somali militant group al-Shabab", if we match the entire phrase, "Al Qaeda" will be matched in the actor dictionary with code "IMGMUSALQ". However, by identifying the smallest core sub-phrase as "Somali militant group al-Shabab" and distinguishing it from non-core noun modifier "Al Qaeda-linked", we are able to match the expected code "SOMUAF". This is an improvement compared to PETRARCH2 , since PETRARCH2 cannot break down the above phrase into smaller constituents. Using dependency parse makes us able to identify non-core modifiers in a noun phrase easily since the non-core modifiers are usually connected to the core sub-phrase via some specific relations, such as "amod" and "nmod". If both actor code and agent code are matched in this step, the coder stops at this step and return the code of the noun phrase.

Secondly, if no match is observed in actor or agent dictionaries, then another attempt is performed by adding adjective modifiers and noun modifiers. For instance, in the noun phrase "Gondor's main opposition group in Arnor", the attempt for matching in the first step is "opposition group". But only agent code is matched, and actor code is not found. So we perform the second attempt by matching "Gondor's main opposition group". Here both agent code " OPP" and actor code "GON" are matched. The coder returns the code "GONOPP" for the noun phrase.

In the cases where no code is found in any of these attempts, *textMatching*() method looks for matches in the entire noun phrase, always prioritizing the longest matched text. Note that, the code of a noun phrase is the combination of the matched actor and agent codes. In the cases where the actor code overrides the agent code, duplicates are removed. For example, for the noun phrase "President Michel Suleiman", the coder matches actor code "LBNGOV" from "Michel Suleiman" and agent code " GOV" from "President". In this case, the coder will not return "LBNGOVGOV" but returns the code "LBNGOV" by removing the duplicated "GOV".

Similarly to PETRARCH2, *check_date*() method resolves data restrictions on actor codes, deciding which code the matched actor should receive given the date (same actor may change over time).

### 2.2.3 Identifying verb phrase codes

This step consists of looking for verb codes and patterns in the verb dictionary for each triplet extracted in previous step. The main method invoked for this task is *get_verb_code*().

First, it looks for a block meaning in the dictionary corresponding to the verbs stored in the triplets. Then, it checks source and pre-verb prepositional phrases to look for patterns in dictionary (through method *get_noun*()).

Following, the next step is to analyze the post-verb phrases in order to iden-

tify corresponding patterns in dictionary. The priority here is to find patterns that match the entire post-verb structure. If no pattern is identified, then it further looks for patterns considering the combination of words in post-verb part. In the cases where multiple patterns are matched, we keep the longest matched pattern containing most information.

The last step is to check if source or target is part of matched pattern. If it is, the source or target is merged as part of the action and new source or target is identified. Specifically, it looks for the noun phrase in other-noun that is closest to verb but not in the matched pattern as the new source or new target.

Consider, for instance, the following sentence mentioned above: "President Obama expressed sorrow for the horrific shootings in South Carolina". When matching the post-verb phrases, it first match the entire post-verb structure "sorrow for the horrific shootings in South Carolina". Pattern "- * SORROW" is matched. "sorrow" is part of the matched pattern but is identified as target in the phrase extracting step. So"for the horrific shootings in South Carolina" is identified as the new target.

### 2.2.4 Resolving interactions between verbs

#### 2.2.4.1 Identifying root verbs

This is a plain step which identifies the root verbs of a sentence (by invoking $get_rootNode()$ method). It begins by checking the children of the sentinel node $\langle root \rangle$ which points to the root of the sentence in the dependency parse tree (as we can see in Figure 2, $\langle root \rangle$ is pointing to root verb "said"). If its child is a verb, then it marks this verb as well as other parallel verbs if exists (e.g. verbs with "parataxis" and "conj" relation) as root verbs. In the case the child is not a verb, which usually happens when a sentence has a copula, then it checks all the grandchildren of node $\langle root \rangle$ and mark grandchildren verbs as root verbs.

#### 2.2.4.2 Resolving root verb + verb interactions

The main goal of this task is to resolve the relation between root verbs and related non-root verbs in the sentences. Specifically, a non-root verb is considered as related to a root verb if the non-root verb depends on the root verb via the *advcl*, *ccomp*, or *xcomp* relation. For each triplet contains non-root verb, $match_transform()$ check to see if the triplet and triplet with root verb follows any of the verb transformation patterns specified in the Verb Dictionary file. If the transformation is present, it adjusts the event accordingly.

Consider the sentence "Russia said it attacks three Ukrainian naval vessels." as an example. The triplet with root verb is $\langle Russia, said, [] \rangle$ with event code $\langle RUS, [], 010 \rangle$ and the triplet with non-root verb is $\langle Russia, attacks, three\, Ukrainian\, naval\, vessels \rangle$ with code $\langle RUS, UKR, 190 \rangle$. The transformation pattern " a (a b ATTACK) SAY = a b 015" is matched. In the end, one event with code $\langle RUS, UKR, 015 \rangle$ is returned.

If no transformation is present, it checks if the event is of the following forms, and then convert this to (a b P+Q):

a (b . Q) P<br>
a (a b Q) P

where P is root verb, Q is non-root verb, . means any actor and [] means no actor. In PETRARCH2, the first form "a (b . Q) P" is used. Universal Petrarch

add the second form as an generalization of the transformation pattern " a (a b Q) SAY = a b Q".

### 2.3 Getting events from petrarch patterns

This is the main function that covers all the steps mentioned in flow of matching PETRARCH verb dictionary depicted in Section 2. Currently, it follows the logic of PETRARCH . It goes through each verb in the sentence, check the word sequence before the verb and word sequence after the verb for matched patterns. In verb patterns, if blanks are used between words, the coder skips over intermediate words; if a "_" (underscore) is used between words, the coder matches consecutive words. If there are actor indicators in the pattern, such as "$" for source,"+" for target, assign the noun phrase indicated by the signs as source and target accordingly. If there is no indicator in the pattern, it finds the source, target based on the grammatical rules and resolve pattern and actor overlap in the similar way as section 2.2. If multiple patterns are matched, it returns the longest matched pattern, which is the pattern contains the most tokens, as output.

## 3 Dictionaries

There are six input dictionaries or lists that Universal Petrarch makes use of:

- verb dictionary: a dictionary of the verb synonyms and verb patterns used to code events.
- actor dictionary: a dictionary of the proper nouns used to identify the sources and targets of events.
- agent dictionary: a dictionary of the common nouns that can be associated with actor codes.
- issues dictionary: a file contains character strings used to identify the context of the text.
- discard list: a file contains a list of strings used to identify sentences that should not be coded.
- PICO(PETRARCH Internal Coding Ontology) dictionary: a dictionary of code mappings between CAMEO codes and internal codes.

Universal Petrarch uses the same Actor, Discard, Agent, and Issues dictionaries as PETRARCH2 . It uses two different verb dictionaries: one is PETRARCH2 verb dictionary and the other is PETRARCH verb dictionary.

The structure of the PETRARCH2 Verb Dictionary is as same as in PETRARCH2 , with some updated verb codes and added patterns. Like PETRARCH2 , synonymous verbs are listed under a block, along with the patterns associated with that block. For each block or each verb, there is a default code. This way of storing and organizing the verbs is benecial as for pattern matching, the patterns that should be checked are the patterns associated only with that specic block.

Universal Petrarch also uses the PETRARCH verb dictionary. The structures of the verb dictionaries in both PETRARCH and PETRARCH2 are also the same, the only differences are the pattern simplification and the use of the parenthesis and curly braces in PETRARCH2 . Though PETRARCH verb dictionary is not as structured as PETRARCH2 verb dictionary, but the flexibility of the PETRARCH verb dictionary makes it more suitable for Spanish data.

One difference between PETRARCH2 and Universal Petrarch is that there is no need to list all irregular forms of verbs in the verb dictionaries any more. Universal Petrarch only store the primary verb form and doesn't store all the forms of verbs internally as PETRARCH2 . During the verb code matching phrase, it uses the lemmas of verbs to match in the dictionary.

Another difference between PETRARCH2 and Universal Petrarch is the way they handle Paired Codes [1]. There is an assortment of circumstances where the event coding schemes generates symmetric events of the form:

<Actor_1> Event_1 <Actor_2>
<Actor_2> Event_2 <Actor_1>

For example, a meeting between Israel and Egypt would generate the pair:

ISR EGY 031 (meet with)
EGY ISR 031 (meet with)

A visit by a Jordanian official to Syria would generate the pair:

JOR SYR 032 (visit; go to)
SYR JOR 033 (receive visit; host)

In the dictionary, these combinations are coded automatically by using a pair of codes separated by a colon (:). for example:

FLEW
- $ * TO + [032 : 033]

would do the visit-and-receive pair, while MEET [031 : 031] generates two events for Universal Petrarch with the same code but with the source and target reversed. While in PETRARCH2 , the first code is used in the active voice case, whereas the code after (:) is used for the passive voice case.

## 3.1 petrarch verb patterns

Patterns in PETRARCH verb dictionary follows a couple of syntactic rules:

1. Patterns have a verb synonym block that consists of a set of synonymous verbs with respect to the patterns.

2. The pattern set is terminated with a blank line.

3. Alternatives of multiple-word "verbs" must be specified: they are not constructed automatically.

4. 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.

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

5. 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 – but those definitions can occur anywhere in the file.

## 3.2 petrarch2 verb patterns

The patterns in PETRARCH2 verb dictionaries follow the same rules as PETRARCH2 :

1. Intended pattern should contain exactly one verb.

2. Ensuring small and effective dictionary by keeping the pattern entries minimal. i.e. the smallest amount of information necessary to capture the intended phrases.

3. Patterns can consist of four parts: Pre-verb nouns, Pre-verb Prepositions, Post-verb nouns, Post-verb prepositions. Nouns are indicated by {} and prepositions are indicated by ().

The additional annotative symbols for these patterns are similar to PETRARCH2 .

## 3.3 Verb + Verb Interaction

For *root verb* + *verb* interaction, the Combinations and Transformations steps are as same as PETRARCH2 . But it differs in the way how the system deals with active voice and passive voice, which was absent in PETRARCH2 . Also, another major difference is that, in Universal Petrarch, the HEX code mapping is only used for transformation step (Section 2.2.4.2). While in PETRARCH2, the HEX code mapping is used in the entire verb code matching phase. Also working with another ontology is easier in Universal Petrarch. The main requirement is to provide the mapping between the verbs as an external dictionary. For example, we can provide the rules of verb combination like "VERB_CODE1+VERB_CODE2=VERB_CODE3" or rules of negation like "negation of VERB_CODE1 = VERB_CODE4". It does not necessarily need the CAMEO code to HEX code mapping. And only small amount of effort is needed to modify the coder to use those rules.

### 3.3.1 Combinations:

Combinations deals with the scenario when the meaning of the two verbs together is literally the meanings of the two verbs individually. The only difference is that unlike PETRARCH2 , the code mappings of verb combinations is stored as an external dictionary called Petrarch Internal Coding Ontology (PICO) instead of storing inside the coder. It deals with the mapping between the CAMEO code to a HEX code. Every CAMEO code has a HEX code mapping. It is used in two situations – Handling negative modifiers and summation of two verbs.

i. Handling negative modifiers: If a negative modifier is found for a verb, based on the design of PETRARCH2 dictionaries, the CAMEO code is first mapped to a HEX code. Then hex code 0xFFFF is subtracted from the code. After that, the HEX code is converted back to the CAMEO code. For example, "provide help" has HEX code "0040" and "not provide help" has HEX code "0040-FFFF= FFBF". However, not all the HEX codes

of the negative modifiers have the mappings back to the CAMEO code. In that case, Universal Petrarch just returns the original CAMEO code for the verb (with FFFF being subtracted from the HEX code), rather than the negated version. The subtraction of FFFF indicates that here a negative case has occurred, however no mapping was found for that negative modifier in the dictionary.

ii. Summation of two verbs: This is used when the meaning of the two verbs together is literally the meanings of the two verbs individually [2]. For example, the verbs "Want" and "Help" have their own CAMEO codes in the dictionary (Want [100] and Help [070] ). These codes are converted to HEX codes at first. Then we sum the HEX codes and get the new HEX code. In this case, Want [100] + Help [070] = Want to Help [0170]. Then this new HEX code is mapped back to the CAMEO code. There still exists the issue that no mapping is available for the new HEX code. To tackle this problem, there is a hierarchy defined in PICO. If some code is dominant, which means its corresponding HEX code is larger, even though there are other verbs related to that code, it still returns the code of the dominant verb.

If no mapping is found in PICO, Universal Petrarch can't handle those cases. Right now, PICO only supports CAMEO and Plover code. But as mentioned above, if one wants to work with another ontology and the mapping of verbs is provided as an external dictionary to replace PICO, the coder can be modified easily with small amount of efforts.

#### 3.3.2 Transformations:

Transformations deals with the scenario when the meaning of the verb interaction depends also on the relationships between the nouns that are acting and being acted upon. The difference between "A says B attacked C" and "A says A attacked B" is such a case. The first is equivalent to "A blames B for an attack," and the second "A takes credit for an attack on B." Since this depends on the nouns involved, we must consider them in the transformation category and not the combination category.

#### 3.3.3 Separating Passive voice:

In Universal Petrarch, the passive voices are determined separately than active voices. Passive voices are identified at first from the dependency relation *nsubjpass*. If the verb is in passive voices, its source actors will be switched to target actors and vice versa.

# Appendices

**List of dependency relations mentioned in this file:**

[2]https://github.com/openeventdata/PETRARCH2 /blob/master/PETRARCH2 .pdf

- advcl: adverbial clause modifier
- advmod: adverbial modifier
- amod: adjectival modifier
- ccomp: clausal complement
- conj: conjunct
- iobj: indirect object
- nmod: nominal modifier
- nsubj: nominal subject
- nsubjpass: passive nominal subject
- obj: object
- obl: oblique nominal
- parataxis: parataxis
- root: root
- xcomp: open clausal complement

**List of Universal POS tags:**

| Open class words | Closed class words | Other |
|---|---|---|
| *ADJ: adjective* | ADP: adposition | PUNCT: punctuation |
| *ADV: adverb* | AUX: auxiliary | SYM: symbol |
| *INTJ: interjection* | CCONJ: coordinating conjunction | X: other |
| *NOUN: noun* | DET: determiner | |
| PROPN: proper noun | NUM: numeral | |
| VERB: verb | PART: particle | |
| | PRON: pronoun | |
| | SCONJ: subordinating conjunction | |

Table 1: Universal POS tags.

**Output of Universal Dependency parsing (from UDPipe) in CoNLL-U format:**

```
# text = Israel said a mortar bomb was launched at it from the Gaza strip on Tuesday.
```

Please note that CoNLL-U format expressed in Table 2 still includes columns DEPREL (list of secondary dependencies) and MISC (any other annotation), which were suppressed since they are entirely blank for the sentence in this example.

**Other topics**

| Id | Form | Lemma | UPosTag | XPosTag | Feats | Head | DepRel |
|---|---|---|---|---|---|---|---|
| *1* | Israel | Israel | PROPN | NNP | Number=Sing | 2 | nsubj |
| *2* | said | say | VERB | VBD | Mood=Ind Tense=Past VerbForm=Fin | 0 | root |
| *3* | a | a | DET | DT | Definite=Ind PronType=Art | 5 | det |
| *4* | mortar | mortar | ADJ | JJ | Degree=Pos | 5 | amod |
| 5 | bomb | bomb | NOUN | NN | Number=Sing | 7 | nsubj:pass |
| 6 | was | be | AUX | VBD | Mood=Ind Number=Sing Person=3 Tense=Past VerbForm=Fin | 7 | aux:pass |
| 7 | launched | launch | VERB | VBN | Tense=Past VerbForm=Part Voice=Pass | 2 | ccomp |
| 8 | at | at | ADP | IN | | 9 | case |
| 9 | it | it | PRON | PRP | Case=Acc Gender=Neut Number=Sing Person=3 PronType=Prs | 7 | obl |
| 10 | from | from | ADP | IN | | 13 | case |
| 11 | the | the | DET | DT | Definite=Def PronType=Art | 13 | det |
| 12 | Gaza | Gaza | PROPN | NNP | Number=Sing | 13 | compound |
| 13 | strip | strip | NOUN | NN | Number=Sing | 7 | obl |
| 14 | on | on | ADP | IN | | 15 | case |
| 15 | Tuesday | Tuesday | PROPN | NNP | Number=Sing | 2 | obl |
| 16 | . | . | PUNCT | . | | 2 | punct |

Table 2: Example of CoNLL-U format.

- Some additional methods of Class Sentence and another Classes may be available here, depending on the degree of details we intend to work with.
- Besides, we can add some examples or code excerpts on this space.
- How to run universal Petrarch. Maybe provide examples about how to run it from command line or from python code (using *petrarch_ud.py*)

# References


[1] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[2] Will Monroe, Spence Green, and Christopher D Manning. Word segmentation of informal arabic with domain adaptation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, volume 2, pages 206–211, 2014.

[3] Milan Straka and Jana Straková. Tokenizing, pos tagging, lemmatizing and parsing ud 2.0 with udpipe. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 88–99, Vancouver, Canada, August 2017. Association for Computational Linguistics.