PETRARCH 2 : PETRARCHer

Clayton Norris

Petrarch 2 is the fourth generation of a series of Event-Data coders stemming from research by Phillip Schrodt. Each iteration has brought new functionality and usability, and this is no exception. Petrarch 2 takes much of the power of the original Petrarch's dictionaries and redirects it into a faster and smarter core logic. Instead of viewing a sentence as a list of words, Petrarch 2 (henceforth referred to as Petrarch) now views the sentence on the syntactic level. It receives the syntactic parse of a sentence from the Stanford CoreNLP software, and stores this data as a tree structure of linked nodes, where each node is a Phrase object. Prepositional, noun, and verb phrases each have their own version of the Phrase object, which deals with the logic particular to those kinds of phrases. Since this is an event coder, the core of the logic focuses around the verbs: who is acting, who is being acted on, and what is happening. The theory behind this new structure and its logic is founded in Generative Grammar, Information Theory, and Lambda-Calculus Semantics.

1 Tree structure

In an attempt to take advantage of all the syntactic information provided to us by the Stanford Parse, Petrarch implements the sentence coding in such a way that the syntax tree is apparent in the data structure and the logic. It's a pretty simple tree structure, with every phrase or word being its own node, with pointers to the parent node and the set of children nodes[1]. The phrases are stored as objects of the "Phrase" class, which has three subclasses "NounPhrase," "VerbPhrase," and "PrepPhrase." If a phrase doesn't fall under one of these categories, it is just kept as a "Phrase," though if eventually enough reason can be shown to add another type (adjective phrases, for example), it an be done so easily. Each of these phrase types has several methods unique to it and its own version of a *get_meaning*() method, which is what determines the actor coding of a node from the meaning of its children. The simplicity of this approach in comparison with the expensive list-based pattern matching from previous versions of Petrarch yields a significant speed improvement, and the theoretically-grounded tree-based semantic searching yields a more solidified understanding of the relationships between nouns and verbs.

[1] The relationship between nodes is frequently described using terms of familial relation, most frequently, "parent" and "child"

1.1 Syntax trees

Let's start with a short linguistics lesson. Every sentence in English (and most languages) is made up of several "constituents". A constituent can be a single word or a whole phrase (which is a constituent of constituents), but the defining characteristic is that each constituent serves a specific syntactic (i.e. grammatical) role. Constituents of a sentence are associated hierarchically (hence the phrasal constituent-of-constituents), and so the most convenient way of visualizing or storing syntactic structure is in a syntax tree. There is an example of a syntax tree and how it is used in the parse at the end of this document.

The CoreNLP software on which Petrarch relies uses the Penn Treebank II [2] syntax notation, which can differ slightly from canonical generative grammar labeling, but for our purposes they are equally useful. Constituents have specific type, depending on what their head and distribution is. The cases we care most about in this program are Noun, Verb, and Prepositional phrases. Heads, in this case, are the single word that a phrase reduces to, both semantically and syntactically. They can be predictably located by navigating the syntax tree, so Petrarch relies on the idea of phrasal headedness for much of its speed. A head of a phrase can be formalized as the lowest word-level constituent to which there is an unbroken path of phrase-level similarly-typed constituents from the phrase node. Basically, to find the head of an NP, you follow the path of NP's down the tree until you find a Noun. If there's ever a choice of which path to take, in English you will take the rightmost.

2 Flow

The core logic of the semantic parsing is based on the notion that each node in the tree has a meaning, and the meaning of a node is a combination of the meanings of its children. That means that in moving up the tree and going from word-level to sentence-level, words and meanings get combined until you have one noun phrase meaning and one verb phrase meaning. The meaning of the verb phrase is what captures most of the meaning of the sentence, and accounts for all the relevant nouns and verbs below it. When we use the terms "above" and "below," we are referring to ancestral relations in the syntax tree: "above" means "closer to the root," and "below" means "closer to word-level."

Because of the recursive nature of the meaning determination, one call to $get\_meaning()$ from the upper most verb phrase will cause a domino effect that finds the meaning of the rest of the tree. The flow of each specific phrase type is determined by its $get\_meaning()$ method. While the logical flow can't be strictly linearized due to the domino effect of recursion, it can be abstracted to follow these steps:

1. Read Stanford CoreNLP parse into memory using Phrase classes.

[2] https://www.cis.upenn.edu/~treebank/

2. Identify coded actors in noun phrases.

3. Identify the usage of the verbs in the verb phrases based on the dictionary entries.

4. Identify how verbs interact with their constituent verb, prepositional, and noun phrases.

5. Identify how verbs interact with the noun phrases in their subject position.

6. Resolve verb+verb interactions.

7. Return the coding of the uppermost VerbPhrase using CAMEO[3] verb and actor codes, if it satisfies the conditions specified by the user

3 Classes and class-specific flow

3.1 Noun Phrases

The NounPhrase class only has one unique method, *check_date*(), which is what decides which actor code to choose when the code for a specific person or country changes over time. This is taken almost directly from the older version of Petrarch.

The *get_meaning*() method in the noun phrase both matches the patterns for the actors and agents of word-level children, and combines the meanings of constituent PP, NP, and VP children. The priority is given as $WordPatterns > NP > PP > VP$, and only when actors and agents are not coded will the node finding the meaning look at a lower-priority phrase. This means that the noun phrase "American troops in Iraq" would only code as USAMIL but "Troops in Iraq" would code as IRQMIL.

3.1.1 Pronouns

When Petrarch encounters a pronoun, it looks up the tree for an antecedent within the same sentence. If the pronoun is relfexive (ends with -self or -selves), Petrarch looks until it finds a noun, or until it finds a verb phrase with a defined subject, and assigns that as the meaning. However, if the pronoun is not reflexive, Petrarch moves up until it finds an S-level phrase, then begins its search. This is based on the binding rules that pronouns follow in Generative Grammar. Because of the distinction between the two types of pronouns, Petrarch can correctly identify that "itself" in *A said B hurt itself* refers to B, while "it" in *A said B hurt it* refers to A.

Since Petrarch currently has no concept of number or gender, it sometimes makes mistakes in instances where the pronoun reference depends on the characteristics of

[3] http://eventdata.parusanalytics.com/cameo.dir/CAMEO.09b6.pdf

the nouns in the sentence. Such is the case differentiating *Obama told Hillary that he should run for President again* from *Obama told Hillary that she should run for President again*.

3.2 Prepositional Phrases

The *get_meaning*() method of PrepPhrase objects returns the meaning of their non-preposition constituent. This makes it easier for the actor searching to pass through prepositional phrases. The preposition is stored as an attribute of the object and is used in some cases to determine whether or not a certain PrepPhrase should be considered.

3.3 Verb Phrases

Verb phrases drive most of the complex logic of the program. They play the largest role in all three parts of finding "who did what to whom", assigning verb codes and finding the appropriate noun phrases to fit. The *get_meaning*() method of verb phrases relies on three other verb-specific methods:

3.3.1 *get_upper*()

This method is fairly simple. If the VP has an NP specifier [4] with a coded actor, it returns this. Otherwise, this returns nothing.

3.3.2 *get_lower*()

This is slightly more complicated. In most cases, the verb *get_lower*() method behaves very similarly to the NounPhrase *get_meaning*() method. It looks for some coded actor in noun or prepositional phrases, and returns this.

However, if a VP has a VP as a child, it returns the meaning of only that phrase, as well as looking for some sort of negation word. The only VP's with VP children are modals (could, might, will, etc.) or helping verbs (has, is, do, etc.)[5]. These won't have other NP, or PP children that are relevant to this verb, but can have "not" as a child, so this is where negation is flagged.

3.3.3 *get_code*()

This is where the program looks to see if the verb follows a pattern specified in the dictionary. The patterns consist of four parts:

[4] In Syntax, two phrase-level siblings are called specifiers. These occur most frequently between VP's,NP's,and PP's. The NP specifier of a VP is the phrase that contains the grammatical subject of the verb.

[5] If it seems like a verb would have another verb phrase as a child, but it does not fall into one of these categories, it most likely takes a sentence as a child, rather than a verb phrase.

1. Pre-verb noun phrases
2. Pre-verb prepositional phrases [6]
3. Post-verb noun phrases
4. Post-verb prepositional phrases

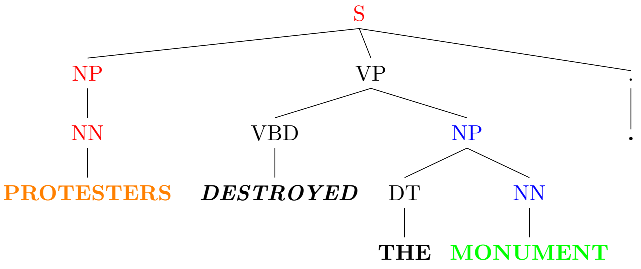
The process from this point differs for active and passive verbs, but only in where each search takes place. Active verbs look for (1) at the closest S-level above the verb, i.e. the nearest point where there will be an NP specifier. It first finds this level via the *get_S*() method, and looks to see if the head of the NP specifier is part of a pattern. If a head is found and there is more to the pattern's noun phrase, then the program begins to look for the rest of the pattern phrase in the noun phrase from which the head came. The verbs find (2) in the same place, but in PP's instead of NP's. Since we almost never see patterns with this format in English, this hasn't been fine tuned. Then the search begins for (3). This involves checking if any of the heads of child NP's are part of a pattern. Then Petrarch follows the same process of looking for longer noun patterns within the phrases of the respective heads. Part (4) looks at child PP's for matches, then matches nouns within the phrases if necessary by the same methods it matched child NP's.

For passive verbs, the process for prepositions is exactly the same. However, it looks for (1) inside the NP's of child PP's with the preposition "by," "from," or "in,". This is simply a specific case to deal with how English deals with the party that is performing the action in a passive sentence. (3) is found in the same place that (1) is found in the active sentences.

As an illustration, consider the active and passive forms of a simple sentence that would match the pattern

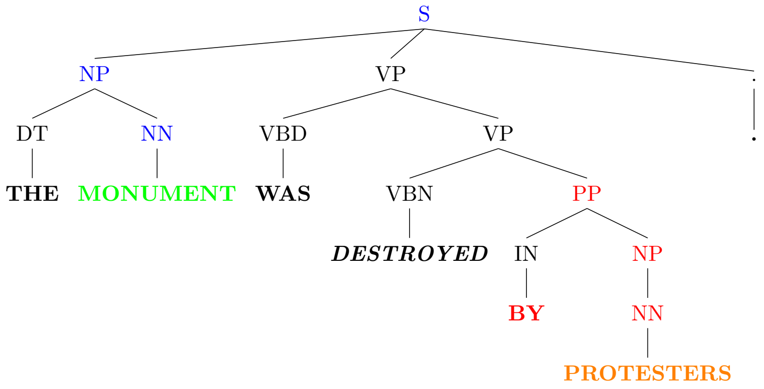
$$protesters * monument \quad [145] \quad \#DESTROY$$

1. The protesters destroyed the monument.



[6] I can't think of a scenario where this would actually be necessary, but the option is there for consistency's sake.

2. The monument was destroyed by protesters.



Key: (1) Location (1) Match (3) Location (3) Match

Note that this is only for matching patterns entered in the dictionary, not Source and Target matching. That happens within the *get_meaning*() method, based on the outcomes of *get_upper*() and *get_lower*().

3.3.4 *get_meaning*()

The *get_meaning*() method of the Verb class first combines the values of the previous three methods in one of a number of ways, depending on what those methods find. In most cases, this method returns a list of events that are described by the subtree of which the verb phrase is the root. Sometimes, however, if there isn't enough verb information available, it will simply return the list of actors described by the subtree. In deciding what to do, the verb has several things to consider:

- Do I have a source actor? (from *get_upper*())
- Do I have a code? (from *get_code*())
- Do I have a S-level or VP child? (from *get_lower*())
- If so, does that child code an event?
- If so, how does the event that I code relate the event that it codes?

3.3.5 *match_transform*()

This method accounts for the fact that ontologies don't always line up exactly with how words work. For example, there are times when you get a sentence like "A says it will attack B," but what you're looking to code is "A threatens B." *Match_transform*() reads transformations from the Verb dictionary and checks to see if any of the events match the transformation format. If that's the case, then the event is converted into a simple (S,T,V) format. The entry in the dictionary for that example would be

$$a \quad (a \quad b \quad WILL\_ATTACK) \quad SAY = a \quad b \quad 138$$

which is basically post-fix notation. This is described in more detail in the dictionary specifications.

3.3.6 *is_valid*()

This method is used to catch a consistent mistake that happens in CoreNLP when a past participle is used as an adjective in front of a noun, but is instead coded as a verb.

3.4 Event extraction

One call to the *get_meaning*() method of the uppermost VP will cause the rest of the tree to be parsed, and return the event coding of that VP, which is the event coding of the whole tree. Since not all events of the sentence at this point might not be complete, the Sentence object which contains the Phrase tree will call *get_meaning*() in its *get_events*() method, and check to see if the event is satisfactory. If the event that is returned by *get_meaning*() is a complete coding (has all three parts), it is assigned to the sentence and the process is complete.

4 Dictionaries

Petrarch uses the same Actor, Agent, Discard, and Issue dictionaries as it always has, but the newest version has brought changes to the format and structure of the Verb Dictionary. The sets of synonymous nouns (synsets) remain the same, as well as how the base verbs are organized and stored. The two biggest differences are the transformations, which $match_transform()$ looks at, and the patterns for matching phrases.

4.1 Patterns

The patterns in the dictionaries should now follow a few simple rules:

1. The intended pattern should contain exactly one verb: the verb being matched

2. The pattern entries should be minimal, i.e. the smallest amount of information necessary to capture the intended phrases.

3. The pattern has up to four parts: Pre-verb nouns, Pre-verb Prepositions, Post-verb nouns, Post-verb prepositions.

The patterns themselves also contain additional annotative symbols to provide the parser with more syntactic information:

- Unmarked words are nouns or particles. These nouns are phrase heads.
- {Bracketed phrases} are for specifying things that can't be covered by a single noun. The last word in the brackets should be the head.
- Prepositional phrases are (in parentheses). The first word is the preposition, the rest is considered as nouns are.
- Note that these prepositional markers can be combined (with {Noun Phrases})

4.2 Verb+Verb interaction

4.2.1 Combinations

Verbs can interact with each other in one of two ways. The first is what we call a combination. This is what happens when the meaning of the two verbs together is literally the meanings of the two verbs individually. These occur mostly frequently to specify the subcode of somewhat vague or high-level CAMEO categories, like *appeal, intend, refuse* or *demand*. This is handled using an internal translation of CAMEO codes into a system that expands the hierarchy of CAMEO beyond the basic top-level/subcode classification system. This allows for more controllable processing of verb combinations that are inherent in CAMEO. So rather than a system where"Intend [030] + Help [070] =Intend to help[033]," we get "Intend [3000] + Help [0040] =Intend to help[3040]." The full conversion schema can be found in the *utilities.py* file under *convert_code*().

Codes are converted and stored as four-digit hex (base 16) codes. The general principle behind it is in the table below. The first three columns encompass the top-level codes, the fourth position is a specifier. For the most part they follow the descriptions here, but some top-level codes have unique subclasses, which don't follow these specifically. Notice that not all combinations refer to CAMEO codes. This is intentional, and means that if we wanted to code things beyond CAMEO we could. The strength of this is predictability and the possibility of semantic addition.

0	0	0	0
1 Say	1 Reduce	1 Meet	1 Leadership
2 Appeal	2 Yield	2 Settle	2 Policy
3 Intend		3 Mediate	3 Rights
4 Demand		4 Aid	4 Regime
5 Protest		5 Expel	5 Econ
6 Threaten		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 De-escalation

The one class not present here is 120, which classifies rejections and refusal to cooperate. Because the action of "refusing to do X" is so often the same as "not doing X," these are simply categorized as the value of their cooperative version minus 0xFFFF. So, since "provide aid" is 0040, "refuse to provide aid" is 0040-FFFF = -FFBF. This is functionally equivalent to the negative, since there is no positive FFFF code, the subtraction always yields a negative value. This allows us to convert negations such as "WILL NOT HELP" = $0 - FFFF + 0040 = -FFBF$ = "REFUSE TO HELP."

4.2.2 Transformations

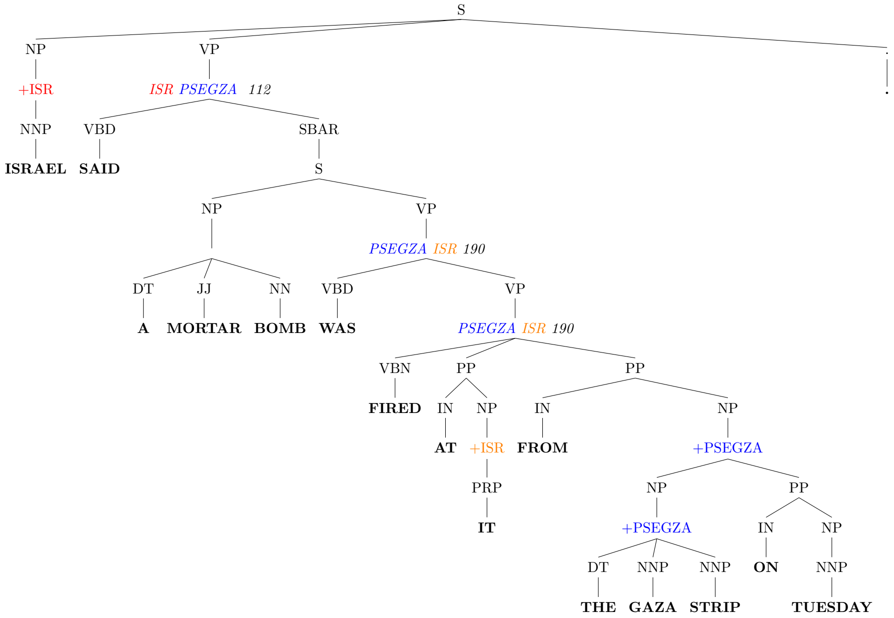
Sometimes this is insufficient, like when the meaning of the verb interaction depends also on the relationships between the nouns that are acting and being acted upon. The difference between "A says B attacked C" and "A says A attacked B" is such a case. The first is equivalent to "A blames B for an 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. The specification on how these are formatted is in the documentation.

5 Example

Consider the sentence

- "Israel said a mortar bomb was launched at it from the Gaza strip on Tuesday"

Petrarch would code this sentence as *ISR PSEGZA 112* with the following tree: [7]



The color coding shows where the actor codes come from. The significant steps taken in this parse involve the verbs "said" and "fired," and the pronoun "it." The pronoun coreference follows the non-reflexive matching process described above. When Petrarch is analyzing "fired," it

1. Identifies the verb as passive
2. Finds the target under the prepositional phrase with "it"
3. Identifies the antecedent of "it" to be "ISR"
4. Finds the source under the prepositional phrase with "from"

Then, the analysis of "said" follows the process:

1. Finds the lower event (PSEGZA ISR 190)
2. Identifies the subject of "said" as ISR
3. Matches this with the dictionary-specified verb transformation
   a (b . ATTACK) SAY = a b 112
4. Transforms this into *ISR PSEGZA 112*.

[7] For those unfamiliar with CAMEO verb codes, 190 is an organized attack, while 112 is an accusation of aggression