Narwhal Spec

Narwhal is an open source Python extension for creating text-aware classes that can read text from a narrow world. A **narrow world** is a collection of related topics with many short text example. For example TripAdvisor hotel reviews are a narrow world with roughly 30 common topics such as noise, wifi, service, nearby attractions, food, transportation, etc. This concept of a narrow world is essentially the same as a "semantic frame" as described by Fillmore [Fillmore]; however the process for extracting *what has been said* is implemented using geometric methods and not statistical ones (see "Best Model Classification" or "The Elements of Narrative" [Waksman]).

When you create a Narwhal class, the generic text processing is taken care of, so you can focus on the details of your narrow world:

- What is the structure of the information to be extracted from expressions in the narrow world?
- What keywords define the topics and how should they be organized into dictionaries?
- What narratives do people use?

Narwhal assumes topic information can be organized as a hierarchy or **tree of keyword dictionaries**, and uses tree nodes as the variables in formulas, called **narrative patterns.** These are templates for how people express themselves in a particular narrow world. They are the building blocks for Narwhal text processing. The basic principle is that rather than "extracting" new meanings from text, pre-existing known meanings are fit to the text, and the best fit is chosen. This is the "geometric" approach.

There is no treatment of grammar or syntax. They are replaced by a reliance on client-defined keywords and narratives, and by implemennting a **moving topic** where a collection of hypothetical narratives is revised as the text is consumed, word by word. This text processing is internal to the Narwhal. Also, there is no treatment of statistics and "machine learning". Interested developers might consider adding "keyword discovery" and "narrative discovery" on top of Narwhal (as discussed in Section 6).

Throughout below I use the example of noise complaints, from the narrow world of hotel reviews.
Sections are as follow:
Section 1 Variables – introduces the **tree of keyword dictionaries.** Its nodes are the variables.
Section 2 Expressions – introduces the notation for **narrative patterns**. These are formulas that use tree node variables. They are the 'models' that will be fit to text.
Section 3 Narwhal classes -
(a) their construction from narrative patterns and a keyword tree;
(b) the **readText()** method.
(c) the goodness of fit metric to judge success or failure of a readText()
Section 4 Collaboration and competition between Narwhal classes
Section 5 Statistical and summary functions, for large collections of examples.
Section 6 future directions

Section 1 Variables – Keyword Dictionaries and DictionaryTrees

Keyword Dictionaries:

A keyword dictionary is a list of strings, called "lexical elements" - typically a list of related keywords. We will find that simple word lists are not enough and that several "parsing" characters need to be included to disambiguate some lexical elements from others with the same spellings within different phrases (eg "*heard* the noise of traffic" or "*heard* about another hotel").

For a keyword dictionary 'keyD', we write the variable 'X' with values in this dictionary as:

```
X( keyD )
```

A word finds a match in X if it finds one in the keyD dictionary.

Compositon of Keyword Dictionaries:

Narwhal uses plus-sign separated lists of keyword dictionaries to create a dictionary that "OR"s together its sub-dictionaries. Thus we write the variable X with values in this combined dictionary as

```
X(  keyD1 + keyD2 + … +  keyDn )
```

A word has a match in X if it has a match *in any* of the sub-dictionaries keyD1, keyD2, …, keyDn.

Example: If we have noise sources such as parties and outdoor construction, these could use separate dictionaires which are "OR"ed together to create a combined dictionary for noise sources.

Narwhal uses pipe separated lists of keyword dictionaries to create a dictionary that "XOR"s together its sub-dictionaries. Thus we write a variable X with values in this combined dictionary as

```
X(  keyD1 | keyD2 | …. |  keyDn )
```

A word has a match in X if it has a match *in exactly one* of the sub-dictionaries keyD1, keyD2, …, keyDn. If a word matches more than one dictionary it is an error (but this is handled internally in Narwhal and doesn't need to be concern a user, unless definitions are incorrect).

Example: If a window is open or closed (and it cannot be both) we might want to put "open" in a 'openD' dictionary and "closed" in another 'closedD'. After that, a variable for the STATE of the window could be written as

```
STATE( openD | closedD )
```

Note: the use of white space is not significant and is for readability.

Category Variables:

You can create a variable with no associated keywords, for example X( ). This is done to create deliberate groupings of children within the tree, as illustrated below. The way a word matches X is for it to match one of the grouped children of X.

The Keyword Dictionary Tree:

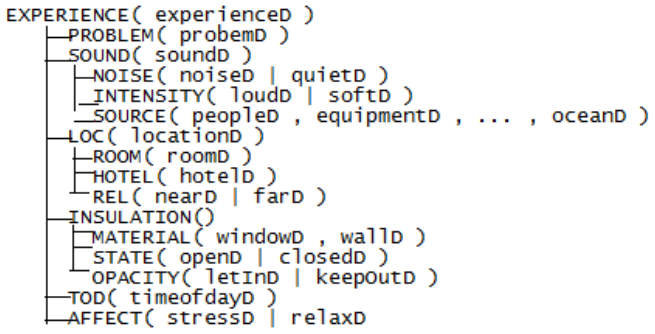

Figure 1 A Tree of Dictionaries

All the variables in a Narwhal class are given together, in an indented paragraph format. There is one line per variable. Indentation is used to indicate dependency. Thus if X and Y are variables, we write Y as a child of X indented on the line below X, like this:

```
X
  Y
```

Narwhal considers a word match in a child variable Y to also count as a match in parent X. This convention extends the matching of X beyond its own dictionaries to include the dictionaries of its children.

Indentation follows the usual Python rules – use any amount of white space, as long as it is consistent.

Example: The tree illustrated above can be notated as follows, where the "D" ending indicates a keyword dictionary. This is a general organization for the sound experience at a hotel:

```
EXPERIENCE( experienceD )
  PROBLEM( problemD )
  SOUND( soundD )
    NOISE( noiseD | quietD )
    INTENSITY( loudD | softD )
    SOURCE( peopleD + equipmentD + ... + oceanD )
  LOC( locationD )
    ROOM( roomD )
    HOTEL( hotelD )
    REL( nearD | farD )
  INSULATION()
    MATERIAL( windowD , wallD )
    STATE( openD | closedD )
    OPACITY( letInD | keepOutD )
  TOD( timeofdayD )
  AFFECT( stressD | relaxD
```

To illustrate word matching for this structure: the word "sound" is in a list called 'soundD' which defines the SOUND variable. But INTENSITY is a child variable of SOUND and so the word "loud", which is in a list 'loudD' defining INTENSITY, is considered a match for the parent SOUND variable.

Note: it is good practice to use a copy of a tree in dynamic code. That way different copies of the same tree can be in different states without interfering with each other.

Tree structure is both arbitrary and significant:
The typical expressions people use are not contained in this tree. Ultimately, the tree serves only as a repository for words, used by the narrative patterns. Although it's structure might be organized differently, it defines what variables are available in the system and it limits what formulas can be written using narrative patterns. In theory, the tree is analogous to a "number system" used to "measure" text.

Section 2 Expressions – Narrative Patterns

Below, **nouns, verbs**, and **adjectives** are assumed to be associated to Narwhal variables or to other narratives constructed recursively from the following relations:

The adjective relation*:*

```
X_rel/A
```

Here 'X' is a noun, 'rel' is the type of relationship and 'A' is an attribute for that relationship.
Example: "our room was near the elevators"; where 'X' is a room-related keyword variable, 'rel' is a location-related keyword variable, and 'A' is a noise source keyword variable.

We also write

```
X_/A
```

where the 'rel' variable, defining the type of relationship, is absent. This is either because there are no keywords associated to it, or because we wish to ignore them, or because we a don't bother to name them in an informal discussion.
Example: "the red ball" where X is "ball" and A is "red".

The verb relation:

```
X-act->Y
```

Here 'X', and 'Y' are nouns, and 'act' is an action verb.
Example: "the open windows let in the noise from the street below"; where 'X' is a keyword variable with words like "wall", "window", "balcony"; and the 'act' is a keyword variable with words like "let in"; and 'Y' is a keyword variable of noise sources.

Note: You should use the adjective relationship for expressing something that is persistent in time, and the verb relationship for expressing an event occurring at one moment in time.

The transformation relation:

```
X::Y
```

Here 'X' and 'Y' are nouns. This relation expresses causality or connection between 'X' and 'Y'.
Example: "we could not sleep because of the street noise". Here 'X' is the street noise, and

```
Y = I _sleep/poor
```

where 'I' is a self-related keyword variable; 'sleep' is a sleep-related keyword variable and 'poor' is a sleep-quality keyword variable.

The sequence relation:

`X,Y`

Here 'X' and 'Y' are nouns. This expresses sequence without causality. The order of the sequence may or may not be significant.

Example: "the hotel was quiet and the food was good', where 'X' is 'hotel_noise/quiet' and 'Y' is the 'food_quality/good'.

The contrast operator:

X*

Here 'X' is a noun. This expresses a negation or contrasts with 'X'.

Example: "Our room was not ready when we arrived", where 'X' is 'room_/ready', is expressed as

(room_/ready)*

The order of precedence:

Use the following priority, from highest to lowest:

'_/', '->' , '::' , ','

Example:

```
(windows_/open)-letin->(noise_/loud_/fromstreet)::(I_sleep_/poor)
```

This could be written without parentheses:

```
windows_/open-letin->noise_/loud_/fromstreet::I_sleep_/poor
```

Or **you can insert white space** for readability. White space is ignored by the Narwhal (except in the indentation format):

```
windows_/open -letin-> noise_/loud_/fromstreet :: I_sleep_/poor
```

Example: A noise complaint typically uses these narratives patterns:

```
SOUND ("there was a sound")
SOUND_/TOD ("there was a sound at a time of day")
PROBLEM_/SOUND ("a problem with noise")
(PROBLEM_/SOUND)* ("noise was not an issue")
LOC_RELATION_/SOURCE ("the room was near the elevators")
MATERIAL -OPACITY-> SOUND ("open windows let in traffic noise")
(SOURCE,LOCATION)_/INTENSITY :: NOISE ("loud traffic from the
highway")
```

Good/Bad and Value for narratives:

One finds that opinion statements are communicating a value or "sentiment" in the form of 'good' or 'bad'. To properly analyze narratives of greater complexity it is helpful to assign such a 'value' to each of the smaller parts of the narrative and consider how it is preserved, or not, as the parts are assembled, along with implied negations, into larger wholes. This is much the same as how logicians treat the value of 'truth' or 'falsity' of expressions, except Narwhal has a broader spectrum of responses to an incoming

expression. Aside from preserving 'value', Narwhal allows meaningful expressions to have a 'value' that is irrelevant and ignored - for example in technical descriptions, questions, or commands. Narwhal also allows meaningful expressions to be considered "off topic" and ignored entirely.

The contrast between Narwhal "value" and logical "truth" is clear from their different treatments of the word "but". In logic "A and B" and "A but B" are considered equivalent. In Narwhal the "but" reverses the value implied by A while asserting the value implied by B; so "A and B" becomes (A,B) and "A but B" becomes (A*,B) and "A although B" becomes (A, B*)

We apply these conventions to assembling the value for a narrative pattern, in terms of the values of its parts.

- An exclusive variable returns with polarity reversed whenever a non-default (first) dictionary is used.
- If an "attribute narrative" is tested against text and the 'relation' has negative polarity then it reverses the polarity of the whole. If the 'value' has negative polarity then it sets the polarity of the whole to be negative, ignoring the polarity of the 'relation'.
- If an "event narrative" is tested against text, then the polarity of the whole is the product of the polarity of the 'action' and the 'value' (using the multiplicative group on {-1,1}).
- If a "transformation narrative" of the form A::B is tested against text, then the polarity of the whole is the polarity of B.
- If a "sequence narrative" of the form A,B is tested against text, then the polarity of the whole is the polarity of B (???).

Section 3 Narwhal classes

Construction:

You should define all dictionaries and the tree in Python files that are imported to your Python code, where you can write:

```
P = nwClass(tree, narratives[] )
```

The readText() method:

```
P.readText( stringIn )
```

The goodness of fit metric (the more word matches the better):

Section 4 Competition between Narwhal Classes

Secton 5 Summary and Archival Functions

Section 6 The Future of Narwhal

It is hoped that other developers will contribute to Narwhal. The general goal is: easier development of text-aware classes that can work in larger (less "narrow") worlds with longer examples of text. It would make sense to add more pre-defined language structures as resources for the developer.

For example:

- Truism (I know of 8, listed on Sphinxmoth and detailed in "The Elements of Narrative") appear to be general story mechanisms that are almost always used to communicate detailed information – which is provided inside a "cause and effect" format.
- Verb tables. A number of basic verbs can be listed in a table and narrative patterns that use them made available as resources.
- Ability to understand sequences of quantities.
- Integration with FrameNet [Berkeley], an archive of existing semantic frames. In principle any semantic frame could become a dedicated Narwhal class for reading text from the "corpus" that is part of the frame's definition.
- Automatic or semi-automatic discovery of new keywords to add to existing lists. One of the maintenance chores of Narwhal is the addition of keywords to the existing lists. Currently this is done manually by testing examples that are 'detected' in the narrow world but which are incompletely 'read'. Developers interested in machine learning may create a Narwhal client with such capabilities.
- The automatic discovery of narrative patterns would be interesting; applying "machine learning" concepts to the models rather than the underlying data.
- Narwhal will need maturation of the moving topic implementation.
- It will be possible, for example, to build a 'question' chatbot for a consumer that becomes temporarily enhanced when it comes in contact with an 'answer' chatbot from a vendor. This means such chatbot classes will need a way to announce their capabilities to each other in an interactive community.
- Narwhal does context - so a first sentence is most critical in a paragraph.
- Narwhal does community: basic support for interactions and announcing capability: Expand Section 4 to cover capability negotiation between Narwhal classes
- Surely you see the possibility of bridging between FrameNet and AIML, by creating libraries of narrative patterns?

Summary

Narwhal implements a system of formal symbolic reasoning that is different from traditional logic and yet encompasses and describes some of the same aspects of natural language. With the "moving topic", it is able to translate narrow world text into symbolic form, building symbolic expressions and distill the 'good' or 'bad' sentiment polarity. After which, Narwhal provides a concept of goodness-of-fit to determine if something has been said, according to the narratives you have defined. It is hoped that Narwhal will enable the creation of text interfaces for data objects by non-experts.