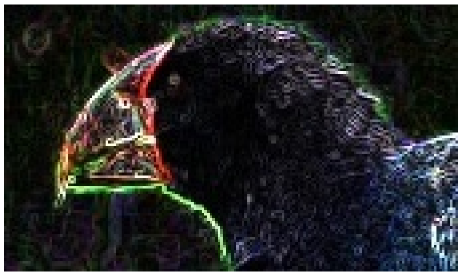

takahe Documentation

Release 0.33

Florian Boudin

February 20, 2013

CONTENTS

Name takahe

Authors Florian Boudin (florian.boudin@univ-nantes.fr)

Version 0.33

Date Feb. 2013

Description takahe is a multi-sentence compression module. Given a set of redundant sentences, a word-graph is constructed by iteratively adding sentences to it. The best compression is obtained by finding the shortest path in the word graph. The original algorithm was published and described in:

> Katja Filippova, Multi-Sentence Compression: Finding Shortest Paths in Word Graphs, *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 322-330, 2010.

History

- 0.33 (Feb. 2013), bug fixes and better code documentation
- 0.32 (Jun. 2012), Punctuation marks are now considered within the graph, compressions are then punctuated
- 0.31 (Nov. 2011), modified context function (uses the left and right contexts), improved docstring documentation, bug fixes
- 0.3 (Oct. 2011), improved K-shortest paths algorithm including verb/size constraints and ordered lists for performance
- 0.2 (Dec. 2010), removed dependencies from nltk (i.e. POS-tagging, tokenization and stop-words removal)
- 0.1 (Nov. 2010), first version

Dependencies

- *networkx* for the graph construction (v1.2+)

Usage A typical usage of this module is:

```
import takahe

# A list of tokenized and POS-tagged sentences
sentences = ['Hillary/NNP Clinton/NNP wanted/VBD to/stop visit/VB ...']

# Create a word graph from the set of sentences with parameters :
# - minimal number of words in the compression : 6
# - language of the input sentences : en (english)
# - POS tag for punctuation marks : PUNCT
compresser = takahe.word_graph(sentences, 6, 'en', "PUNCT")

# Get the 50 best paths
candidates = compresser.get_compression(5)

# Rerank compressions by path length
for cummulative_score, path in candidates:

    # Normalize path score by path length
    normalized_score = cummulative_score / len(path)

    # Print normalized score and compression
    print round(normalized_score, 3), ' '.join([u[0] for u in path])
```

```
# Write the word graph in the dot format
compresser.write_dot('test.dot')
```

Misc The Takahe is a flightless bird indigenous to New Zealand. It was thought to be extinct after the last four known specimens were taken in 1898. However, after a carefully planned search effort the bird was rediscovered by on November 20, 1948. (Wikipedia, http://en.wikipedia.org/wiki/takahe)

CHAPTER

ONE

WORD_GRAPH CLASS

class `takahe.`**`word_graph`**(*sentence_list*, *nb_words=8*, *lang='en'*, *punct_tag='PUNCT'*)

The word_graph class constructs a word graph from the set of sentences given as input. The set of sentences is a list of strings, sentences are tokenized and words are POS-tagged (e.g. `"Saturn/NNP is/VBZ the/DT sixth/JJ planet/NN from/IN the/DT Sun/NNP in/IN the/DT Solar/NNP System/NNP"`). Three optional parameters can be specified. The first parameter is the minimal number of words for the best compression (default value is 8). The second parameter is the language and is used for selecting the correct stopwords list (default is "en" for english, stopword lists are localized in /resources/ directory). The third parameter is the punctuation mark tag used during graph construction (default is PUNCT).

`ambiguous_nodes`(*node*)

Takes a node in parameter and returns the number of possible candidate (ambiguous) nodes in the graph.

`build_graph`()

Constructs a directed word graph from the list of input sentences. Each sentence is iteratively added to the directed graph according to the following algorithm:

- Word mapping/creation is done in four steps:
  1. non-stopwords for which no candidate exists in the graph or for which an unambiguous mapping is possible or which occur more than once in the sentence
  2. non-stopwords for which there are either several possible candidates in the graph
  3. stopwords
  4. punctuation marks

For the last three groups of words where mapping is ambiguous we check the immediate context (the preceding and following words in the sentence and the neighboring nodes in the graph) and select the candidate which has larger overlap in the context, or the one with a greater frequency (i.e. the one which has more words mapped onto it). Stopwords are mapped only if there is some overlap in non-stopwords neighbors, otherwise a new node is created. Punctuation marks are mapped only if the preceding and following words in the sentence and the neighboring nodes are the same.

- Edges are then computed and added between mapped words.

Each node in the graph is represented as a tuple ('word/POS', id) and possesses an info list containing (sentence_id, position_in_sentence) tuples.

`compute_statistics`()

This function iterates over the cluster's sentences and computes the following statistics about each word:

- term frequency (self.term_freq)

`get_compression` (*nb_candidates=50*)
Searches all possible paths from **start** to **end** in the word graph, removes paths containing no verb or shorter than *n* words. Returns an ordered list (smaller first) of nb (default value is 50) (cummulative score, path) tuples. The score is not normalized with the sentence length.

`get_directed_context` (*node*, *k*, *dir='all'*, *non_pos=False*)
Returns the directed context of a given node, i.e. a list of word/POS of the left or right neighboring nodes in the graph. The function takes four parameters :

- node is the word/POS tuple
- k is the node identifier used when multiple nodes refer to the same word/POS (e.g. k=0 for (the/DET, 0), k=1 for (the/DET, 1), etc.)
- dir is the parameter that controls the directed context calculation, it can be set to left, right or all (default)
- non_pos is a boolean allowing to remove stopwords from the context (default is false)

`get_edge_weight` (*node1*, *node2*)
Compute the weight of an edge *e* between nodes *node1* and *node2*. It is computed as e_ij = (A / B) / C with:

- A = freq(i) + freq(j),
- B = Sum (s in S) 1 / diff(s, i, j)
- C = freq(i) * freq(j)

A node is a tuple of ('word/POS', unique_id).

`graph` = **None**
The directed graph used for fusion.

`k_shortest_paths` (*start*, *end*, *k=10*)
Simple implementation of a k-shortest paths algorithms. Takes three parameters: the starting node, the ending node and the number of shortest paths desired. Returns a list of k tuples (path, weight).

`length` = **None**
The number of sentences given for fusion.

`load_stopwords` (*path*)
This function loads a stopword list from the *path* file and returns a set of words. Lines begining by '#' are ignored.

`max_index` (*l*)
Returns the index of the maximum value of a given list.

`nb_words` = **None**
The minimal number of words in the compression.

`pre_process_sentences` ()
Pre-process the list of sentences given as input. Split sentences using whitespaces and convert each sentence to a list of (word, POS) tuples.

`punct_tag` = **None**
The stopword tag used in the graph.

`resources` = **None**
The path of the resources folder.

`sentence` = **None**
A list of sentences provided by the user.

`sep` = **None**
: The separator used between a word and its POS in the graph.

`start` = **None**
: The start token in the graph.

`stop` = **None**
: The end token in the graph.

`stopword_path` = **None**
: The path of the stopword list, e.g. stopwords.[lang].dat.

`stopwords` = **None**
: The set of stopwords loaded from stopwords.[lang].dat.

`term_freq` = **None**
: The frequency of a given term.

`verbs` = **None**
: The list of verb POS tags required in the compression. At least *one* verb must occur in the candidate compressions.

`write_dot`(*dotfile*)
: Outputs the word graph in dot format in the specified file.

CHAPTER

TWO

INDICES AND TABLES

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

PYTHON MODULE INDEX

INDEX