Text Summarization

Extractive summary of input article.

Author: Nishit Jain
Email: nishitjain1997@gmail.com
Source: Github

Research

Overview

Extractive text summarization is a type of Automated Text Summarization (summarization using computers) in which the summary is made up of complete sentences picked from the original text. To achieve this, each sentence is given a relevance ranking and the top most relevant sentences are picked for the summary.

Approach

This is an unsupervised summary algorithm which is an amalgamation of the TextRank algorithm given by Mihalcea et al. and the Feature Term based method given by Suneetha Manne et al. The scores given by TextRank algorithm are enhanced by extracting additional word-level and sentence-level features to incorporate both semantic and syntactic meaning in the scores.

1. TextRank algorithm
   - Each sentence is converted into a **Sentence Vector** by taking the average of the Glove embeddings of each word in that sentence.
   - Pairwise Cosine Similarity of Sentence Vectors is used to create a similarity matrix of sentences.

   The similarity matrix is used to build a directed graph, which is then fed into Google's PageRank algorithm to give a score to each sentence. The sentence that is most similar to all sentences in the text supposedly contains ideas from them all and should be considered in the summary.

2. Feature Term based method. The features extracted include:
    - Resultant Term Weight: This denotes the amount of information conveyed by each word in the sentence. It is a product of the Term Weight (normalized frequency of word) and Inverse Sentence Frequency (log of ratio of number of sentences and number of appearances of the term).
    - Sentence Weight: Parts of Speech tagging to ensure include syntactic meaning of words. Ratio of number of noun and verb terms in a sentence to the total number of terms in all sentences.
    - Sentence Position: Sentences coming at the beginning contain more generalized ideas while those coming in the middle contain more specific ideas.
    - Sentence Length: A normalization term to prevent longer sentences from dominating shorter sentences. Gives score per unit length of sentence

$$\text{Feature Rank} = \frac{(\sigma(\text{Resultant Term Weight}) + \text{Sentence Weight} + \text{Sentence Position})}{\text{Sentence Length}}$$

$$\text{Final Score} = 0.8 * \text{Text Rank} + 0.2 * \text{Feature Rank}$$

Assumptions

1. The extractive summary has been limited to 3 lines.
2. This is a single-document summary algorithm.

Bibliography

1. Mihalcea R., &, Tarau P. (2004). TextRank: Bringing Order into Texts. (W04-3252). Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing
2. Manne S., &, Fatima S. (2012). A Feature Terms based Method for Improving Text Summarization with Supervised POS Tagging. (10.5120/7494-0541). International Journal of Computer Applications (0975 – 8887).
3. Jagadeesh J., &, Pingali P., &, Varma V. (2005). Sentence Extraction Based Single Document Summarization. (IIIT/TR/2008/97). Workshop on Document Summarization, 19th and 20th March, 2005, IIIT Allahabad.

Output

The summary for the article will be stored in 'output.txt' file in root directory.

Python

This script has been developed in and tested for Python3. It has not been tested for Python2.

Example

Sample Input

S.No.	News Article

S.No.	News Article
1	The U.N. Security Council approved a resolution Monday to send 4,200 peacekeepers to Abyei, Sudan, as part of a recent agreement between Sudan and Southern Sudan.
2	The resolution will establish, for six months, the United Nations Interim Security Force for Abyei (UNISFA), comprising "a maximum of 4,200 military personnel, 50 police personnel, and appropriate civilian support," the resolution states.
3	It passed the council unanimously, 15-0.
4	In a statement released by the State Department, Secretary Hiliary Clinton commended the swift passage of the resolution.
5	"Abyei has been a source of regional tension for many years," the statement said.
6	"We urge the parties to reach an immediate cease-fire and to provide aid workers with the unfettered access required to deliver humanitarian assistance to innocent civilians affected by the conflict."
7	A week ago, the Sudanese government and the Sudan People's Liberation Movement signed an agreement to allow peacekeepers in Abyei, aimed at ending strife that has ravaged much of the country.
8	The two sides agreed in principle on the need for a third party to monitor the ill-defined border between north and south before the scheduled July 9 independence for the south.
9	The U.N. peacekeepers will "monitor and verify the redeployment of any Sudan Armed Forces, Sudan People's Liberation Army or its successor" from the Abyei area, among other tasks, the Security Council resolution states.

Sample Output

The U.N. Security Council approved a resolution Monday to send 4,200 peacekeepers to Abyei, Sudan, as part of a recent agreement between Sudan and Southern Sudan. A week ago, the Sudanese government and the Sudan People's Liberation Movement signed an agreement to allow peacekeepers in Abyei, aimed at ending strife that has ravaged much of the country. The U.N. peacekeepers will "monitor and verify the redeployment of any Sudan Armed Forces, Sudan People's Liberation Army or its successor" from the Abyei area, among other tasks, the Security Council resolution states.

```
import nltk
import os
import re
import spacy
import sys
import unicodedata

import networkx as nx
import numpy as np
import pandas as pd

nltk.download('punkt')
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize
from sklearn.metrics.pairwise import cosine_similarity

# Root directory
root = '.'
# Loading Spacy for Parts-of-Speech tagging.
nlp_spacy = spacy.load('en', parse=True, tag=True)
# Loading list of stopwords
stop_words = stopwords.words('english')




"""
  For TextRank Algorithm.
"""
def remove_extraneous_text(sentence:str)->str:
  """
    Input: String
    Output: String
    Takes a news article as input and removes extra spaces and reporting location from it.
  """
  # Remove multiple spaces
  sentence = re.sub(" +", " ", sentence)

  # Remove reporting location
  if ") --" in sentence:
    sentence = sentence.split(") --")[-1]

  # Remove media name from article
  if "(CNN)" in sentence:
    sentence = sentence.split("(CNN)")[-1]

  return sentence

def remove_stopwords(sentence:str)->str:
  """
    Input: String
    Output: String
    Takes a sentence as input and returns the sentence after removing all stopwords.
  """
```

```
  sentence = " ".join([word for word in sentence.split() if word not in stop_words])

  return sentence

def lemmatize_text(sentence:str)->str:
  """
    Input: String
    Output: String
    Takes a sentence as input and uses Spacy to convert each word into it's lemma.
  """
  sentence = nlp_spacy(sentence)
  sentence = ' '.join([word.lemma_ if word.lemma_ != "-PRON-" else word.text for word in sentence])
  return sentence

def clean_text(sentence:str)->str:
  """
    Input: String
    Output: String
    Takes a sentence and cleans by:
     - Converting to lowercase
     - Remove non alphabetic characters
     - Removing extraneous characters
     - Removing stopwords
     - Lemmatizing words
  """
  sentence = sentence.lower()
  sentence = re.sub("[^a-zA-Z]", " ", sentence)
  sentence = remove_extraneous_text(sentence)
  sentence = remove_stopwords(sentence)
  sentence = lemmatize_text(sentence)
  return sentence




"""
  For Feature Term enhancements.
"""
def get_total_terms(cleaned_sentences:list)->int:
  """
    Input: List
    Output: Int
    Takes in a list of sentences and returns total number of tokens in those sentences.
  """
  total_terms = 0

  for sentence in cleaned_sentences:
    total_terms += len(sentence.split())

  return total_terms

def get_term_frequencies(cleaned_sentences:list)->dict:
  """
    Input: List
    Output: Dict
```

```
    Takes in a list of sentences and returns a dictionary containing Tokens as keys and their frequencies as values.
  """
  freq_dict = {}

  for sentence in cleaned_sentences:
    for word in sentence.split():
      freq_dict[word] = freq_dict.get(word, 0) + 1

  return freq_dict

def get_term_weights(cleaned_sentences:list)->dict:
  """
    Input: List
    Output: Dict
    Takes in a list of sentences and returns a dictionary containing Tokens as keys and their weightage as values.
    The weight is calculated using formula:
          TW(ti) = (TF(ti) * 1000) / (Nt)
    where ti is each token, TW is term weight, TF is term frequency and Nt is total number of terms
  """
  total_terms = get_total_terms(cleaned_sentences)
  term_freq_dict = get_term_frequencies(cleaned_sentences)
  term_weights = dict()

  for key, value in term_freq_dict.items():
    term_weights[key] = (value * 1000) / total_terms

  return term_weights

def inverse_sentence_frequency(cleaned_sentences:list)->dict:
  """
    Input: List
    Output: Dict
    Takes in a list of sentences and returns a dictionary containing Tokens as keys and their inverse sentence frequency as values.
    The inverse sentence frequency is calculated as:
          ISF(ti) = log((Ns) / Nti)
    where ti is each token, ISF is inverse sentence frequency, Ns is total number of sentences in paragraph and Nti are the total number of
    sentences in which ti appeared in that paragraph.
  """
  vocabulary = set()

  for sentence in cleaned_sentences:
    vocabulary = vocabulary.union(set(sentence.split()))

  isf = dict()
  number_of_sentences = len(cleaned_sentences)

  for word in vocabulary:
    number_of_appearances = 0

    for sentence in cleaned_sentences:
      if word in sentence:
        number_of_appearances += 1

    isf[word] = np.log(number_of_sentences / number_of_appearances)

```

```
  return isf

def word_weights(cleaned_sentences:str)->dict:
  """
    Input: List
    Output: Dict
    Takes in a list of sentences and returns a dictionary containing Tokens as keys and their resultant weightage as values.
    The weightage is calculated as:
          RW(ti) = ISF(ti) * TW(ti)
    where ti is each token, RW is resultant weightage, ISF is inverse sentence frequency and TW is term weightage.
  """

  term_weights = get_term_weights(cleaned_sentences)
  inverse_sentence_freq = inverse_sentence_frequency(cleaned_sentences)

  resultant_weights = dict()

  for word in term_weights.keys():
    resultant_weights[word] = term_weights[word] * inverse_sentence_freq[word]

  return resultant_weights

def pos_tagging(cleaned_sentences:list)->list:
  """
    Input: List
    Output: List
    Takes in a list of sentences and returns a list of lists, where each Token is represented as a tuple of the form (Token, POS tag).
  """
  tagged_sentences = []

  for sentence in cleaned_sentences:
    sentence_nlp = nlp_spacy(sentence)

    tagged_sentence = []

    for word in sentence_nlp:
      tagged_sentence.append((word, word.pos_))

    tagged_sentences.append(tagged_sentence)

  return tagged_sentences

def sentence_weights(tagged_sentences:list, total_terms:int)->list:
  """
    Input: List, Int
    Output: List
    Takes in a list of POS tagged sentences and total number of terms. Returns a list containing the sentence weight of each sentence.
    The sentence weight is calculated as:
          SW(si) = Number of nouns and verbs in sentence / total number of terms in paragraph.
  """
  sent_weights = []

  for sentence in tagged_sentences:
    relevance_count = 0
```

```
221
222     for word, tag in sentence:
223       if tag == 'NOUN' or tag == 'VERB':
224         relevance_count += 1
225
226     sent_weights.append(relevance_count / total_terms)
227
228   return sent_weights
229
230 def sentence_position(cleaned_sentences:list)->list:
231   """
232     Input: List
233     Output: List
234     Takes in a list of sentences and returns weight for each sentence based
    on it's position.
235   """
236   sent_position = []
237   number_of_sentences = len(cleaned_sentences)
238
239   weights = [0, 0.25, 0.23, 0.14, 0.08, 0.05, 0.04, 0.06, 0.04, 0.04, 0.15]
240
241   for i in range(1, len(cleaned_sentences)+1):
242     sent_position.append(weights[int(np.ceil(10 * (i /
    number_of_sentences)))])
243   return sent_position
244
245 def sentence_length(cleaned_sentences:list)->list:
246   """
247     Input: List
248     Output: List
249     Takes in a list of sentences and returns a list containing length of each
    sentence.
250   """
251   sent_len = []
252
253   for sentence in cleaned_sentences:
254     sent_len.append(len(sentence.split()))
255
256   return sent_len
257
258
259
260
261 """
262   Functions to rank sentences.
263 """
264 def text_rank(sentences:list, word_embeddings:dict)->dict:
265   """
266     Input: List, Dict
267     Output: Dict
268     Takes a list of sentences and Glove word embeddings as input and returns
    a dictionary containing sentences index as key and rank as value.
269     The ranking is done based on the PageRank algorithm
270   """
271   # Clean sentences for PageRank algorithm.
272   clean_sentences = pd.Series(sentences).str.replace("[^a-zA-Z]", " ")
273   clean_sentences = [s.lower() for s in clean_sentences]
274   clean_sentences = [remove_stopwords(r) for r in clean_sentences]
275
```

```
  # Replace each word with Glove embeddings. The Sentence vector is the average of the sum of embeddings of all words in that
  # sentence.
  sentence_vectors = []
  for i in clean_sentences:
    if len(i) != 0:
      v = sum([word_embeddings.get(w, np.zeros((100, ))) for w in i.split()]) / (len(i.split()) + 0.001)
    else:
      v = np.zeros((100, ))
    sentence_vectors.append(v)

  # Initialize a similarity matrix for pair of sentences
  sim_mat = np.zeros([len(sentences), len(sentences)])

  # Calculate cosine similarity for each pair of sentences
  for i in range(len(sentences)):
    for j in range(len(sentences)):
      if i != j:
        sim_mat[i][j] = cosine_similarity(sentence_vectors[i].reshape(1, 100), sentence_vectors[j].reshape(1, 100))[0, 0]

  # Create a PageRank graph using similarity matrix
  nx_graph = nx.from_numpy_array(sim_mat)
  scores = nx.pagerank(nx_graph)

  return scores

def feature_rank(sentences:list)->dict:
  """
    Input: List
    Output: Dict
    Takes a list of sentences as input and returns a dict containig ranking of each sentence.
    The ranking is calculated using word and sentence level features.
  """
  cleaned_sentences = [clean_text(sentence) for sentence in sentences]

  term_weights = word_weights(cleaned_sentences)
  tagged_sentences = pos_tagging(cleaned_sentences)
  total_terms = get_total_terms(cleaned_sentences)
  sent_weights = sentence_weights(tagged_sentences, total_terms)
  sent_position = sentence_position(cleaned_sentences)
  sent_len = sentence_length(cleaned_sentences)

  sentence_scores = []

  for index, sentence in enumerate(cleaned_sentences):
    score = 0

    for word in sentence.split():
      score += term_weights[word]

    score *= sent_weights[index]
    score += sent_position[index]

    if sent_len[index] != 0:
      score /= sent_len[index]
    else:
      score = 0
```

```
332
333     sentence_scores.append(score)
334
335   sentence_scores = sentence_scores / np.sum(sentence_scores)
336
337   final_scores = dict()
338
339   for i in range(len(sentence_scores)):
340     final_scores[i] = sentence_scores[i]
341
342   return final_scores
343
344
345
346
347 def main()->None:
348   """
349     The driver function.
350   """
351
352   # Path to input file
353   input_filepath = os.path.join(root, sys.argv[-1])
354
355   if not os.path.exists(input_filepath):
356     # Check if input file does not exist.
357     print("Could not find input file at location '%s'" % (input_filepath))
358     return
359
360   input_text = ""
361
362   with open(input_filepath, 'r') as f:
363     input_text = f.read()
364
365
366
367   # Location of Glove word embeddings.
368   glove_location = os.path.join(root, 'embeddings', 'glove.6B.100d.txt')
369
370   if not os.path.exists(glove_location):
371     # Check if word embeddings do not exist.
372     print("Could not find Glove Word Embeddings. Kindly download from
'https://drive.google.com/open?id=1cQBYwoLHZzHk4w8zdgcSPFmOP5Xq-x0z' \
373       and save in './embeddings' location.")
374     return
375
376   print("Loading Glove Word embeddings.")
377
378   # Dictionary to store embeddings
379   word_embeddings = {}
380
381   # Open file and load embeddings in memory
382   f = open(glove_location, encoding='utf-8')
383   for line in f:
384     values = line.split()
385     word = values[0]
386     coefs = np.asarray(values[1:], dtype='float32')
387     word_embeddings[word] = coefs
388   f.close()
389
390   print("Embeddings loaded.")
```

```
391
392   print("Creating summary.")
393
394   sentences = sent_tokenize(input_text)
395   text_rank_scores = text_rank(sentences, word_embeddings)
396   feature_rank_scores = feature_rank(sentences)
397
398   final_scores = dict()
399   for i in range(len(text_rank_scores.keys())):
400     final_scores[i] = 0.8 * text_rank_scores[i] + 0.2 *
feature_rank_scores[i]
401
402   ranked_sentences = sorted(((final_scores[i], s, i) for i, s in
enumerate(sentences)), reverse=True)[:3]
403   ranked_sentences = sorted(ranked_sentences, key=lambda x: x[2])
404
405   output_text = ""
406   for i in range(len(ranked_sentences)):
407     output_text += ranked_sentences[i][1] + ' '
408
409   with open('output.txt', 'w') as f:
410     f.write(output_text.strip())
411   print("Summary stored in 'output.txt'.")
412
413
414 if __name__ == '__main__':
415   if len(sys.argv) != 2:
416     print("The syntax to run this program is: 'python run.py file_name.txt'")
417   else:
418     main()
```