Keyword-Extraction: Using TextRank Algorithm

[1]Abhishek Kumar, [2]Parmeet Singh, [3]ParitoshPal Singh,[4]Sharmila Banu K.
[(1,2,3)]Student, B.Tech (CSE), SCOPE, VIT University, Vellore
[4]Assistant Professor(Senior), SCOPE, VIT University, Vellore

ABSTRACT

Keyword extraction is to identify and recommend topical words and phrases from the content of documents. Keyword extraction applies in many areas especially when extracted keywords in the field of information retrieval (IR), is of particular interest because people retrieve helpful information according to keywords. However, a wide variety of keyword extraction methods are proposed and analyzed separately, the research of comparison of different keyword extraction methods is extremely needed.In our paper we have used Text rank algorithm to extract the keywords from a corpus.We also have done the comparison between Text Rank,TF-IDF and RAKE(Rapid Automatic Keyword Extraction) algorithm

Keywords: Text Rank ,Keyword Extraction,RAKE,TF-IDF

INTRODUCTION

A sequence of one or more words as an abstract to the main content of the document is defined as keywords [1]. Therefore, it is possible to consider as a set of phrases semantically covering most of the text. With the explosive growth of information on the Internet, keywords become a crucial method for people to quickly comprehend the content of the document and understand the subject [1]. Hence, keyword extraction is an important technique for document retrieval, web page retrieval, document clustering, summarization, text mining, etc.
We have used to TextRank algorithm which is a variant of PageRank Algorithm proposed by Larry Page.

$$p\left(v_i\right) = \frac{1-d}{n} + d \sum_{v_j \in In\left(V_i\right)} m_j \times p\left(v_j\right)$$

where d is a factor, which normally is set as 0.85. A drawback of this algorithm is that an outlier will draw significant effect to the result, e.g., a dramatic change of one webpage in In(Vi) will bring dramatic value change. Henceforth, several enhancement were made to enhance PageRank

[2]. In light of PageRank, Tarau and Mihalcea proposed TextRank in 2004 [3]. In TextRank, article is divided into basic text units, i.e., words or phrases. As treated as webpage in PageRank, text unit maps to vertex in graph, and edge between vertices refers to the link between text units.

LITERATURE SURVEY

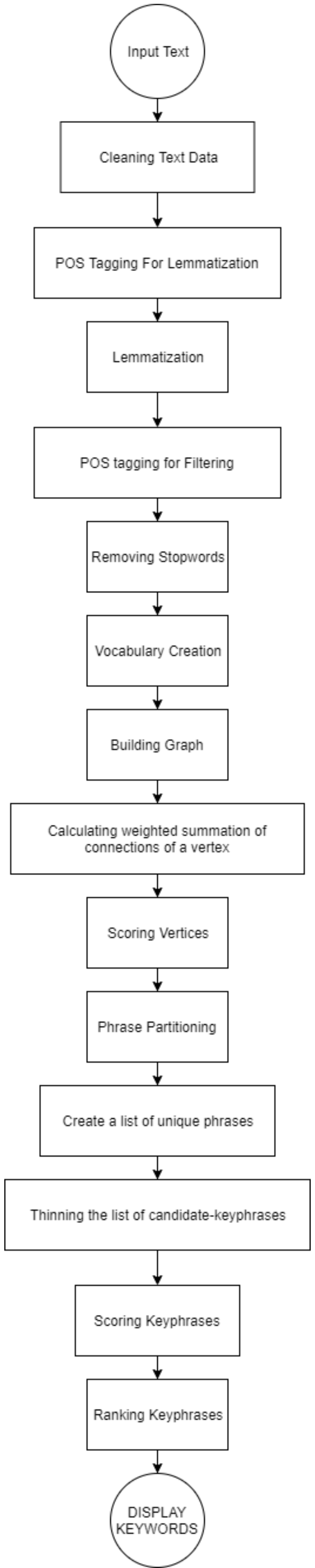
Intuitively, TextRank works well because it does not only rely on the local context of a text unit (vertex), but rather it takes into account information recursively drawn from the entire text (graph). Through the graphs it builds on texts, TextRank identifies connections between various entities in a text, and implements the concept of recommendation. A text unit recommends other related text units, and the strength of the recommendation is recursively computed based on the importance of the units making the recommendation. For instance, in the keyphrase extraction application, co-occurring words recommend each other as important, and it is the common context that enables the identification of connections between words in text. In the process of identifying important sentences in a text, a sentence recommends another sentence that addresses similar concepts as being useful for the overall understanding of the text. The sentences that are highly recommended by other sentences in the text are likely to be more informative for the given text, and will be therefore given a higher score Through its iterative mechanism, TextRank goes beyond simple graph connectivity, and it is able to score text units based also on the “importance” of other text units they link to. The text units selected by TextRank for a given application are the ones most recommended by related text units in the text, with preference given to the recommendations made most influential ones, i.e. the ones that are in turn highly recommended by other related units. The underlying hypothesis is that in a cohesive text fragment, related text units tend to form a “Web” of connections that approximates the model humans build about a given context in the process of discourse understanding.[5]

TF-IDF stands for ‘Term Frequency, Inverse Document Frequency’. It is a method to score the importance of words or terms in a document based on how frequently they appear across multiple documents [16]. TF-IDF commonly used weighting technique for information retrieval and information discovery, it is a statistical method used to assess the importance of a word for one document set or one corpus [16]. The importance of the word is proportional to the number of times when it appears in the document, but it is inversely proportional to the frequency. The weighted forms of TF-IDF are used by search engine applications frequently, as a measure of degree of correlation between a file and a user’s query [16]. The main idea of TF-IDF is that if the frequency of a word or phrase appears in an article is high, TF value is high [16]. In addition, this word or phrase appears in other articles rarely. After that. the word or phrase has great classification ability and suitable to classify. TF (Term Frequency) refers to the number of times that a given word appears in the file [16]. IDF (Inverse Document Frequency) refers that if the less the number of documents that containing the entry, the larger the IDF value, then the entry has a great classification ability [16]. Using TF-IDF can calculate the importance of a keyword in an article.

Rapid Automatic Keyword Extraction (RAKE) algorithm is proposed by the paper “Automatic keyword extraction from individual documents” from 2010 [7]. It showed the effect of RAKE algorithm is better than TextRank algorithm [7]. RAKE algorithm is used to extract keywords, which actually extract key phrases and tend to longer phrases. In English keyword extraction, keywords include multiple words generally, rarely include punctuation symbols and stop words [38].
RAKE algorithm uses punctuation marks, such as question mark, exclamation mark, comma, etc.. A document is split into a number of sub sentences [7]. Then, several phrases are separated by using stop words as a separator for each sub sentence. These phrases are used as candidate words for the final extracted keywords. The next essential step is that measure the importance of each phrase. Each phrase can be divided into several words by space [7]. Then, giving a score for each word and accumulating the score [7]. An essential point is to consider the co-occurrence of each word in this phrase. Co-occurrence means the frequency of a few words occurred together in a document. The formula of RAKE algorithm is showed below:
wordScore= wordDegree (w)/ wordFrequency (w)
The score of word ‘w’ is the degree of the word divided by the word frequency. In the study of networks, the degree of a node in a network is the number of connections. The degree is added 1 when a word co-occurring in a phrase. Word frequency means total number of words that occur in the document. Then, the score of each word is summed and sorted for each candidate key phrase, first 1/3 of the total number of candidate phrases is considered as extracted keywords by RAKE algorithm.

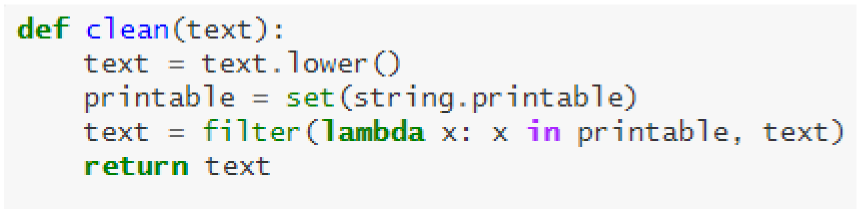
METHODOLOGY

Flowchart


Input Text
Cleaning Text Data
POS Tagging For Lemmatization
Lemmatization
POS tagging for Filtering
Removing Stopwords
Vocabulary Creation
Building Graph
Calculating weighted summation of connections of a vertex
Scoring Vertices
Phrase Partitioning
Create a list of unique phrases
Thinning the list of candidate-keyphrases
Scoring Keyphrases
Ranking Keyphrases
DISPLAY KEYWORDS

Cleaning Text Data

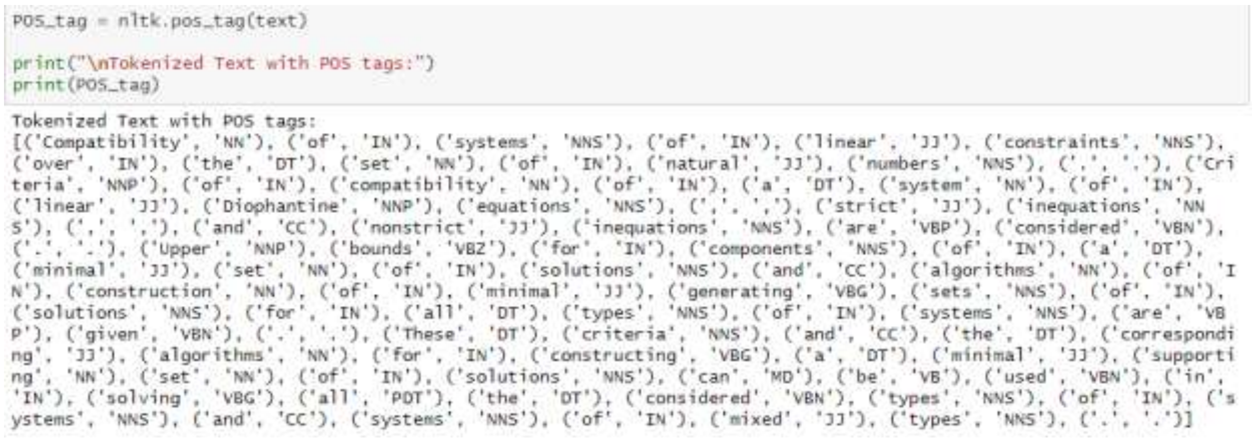
The raw input text is cleaned off non-printable characters (if any) and turned into lower case. The processed input text is then tokenized using NLTK library functions.



```
def clean(text):
    text = text.lower()
    printable = set(string.printable)
    text = filter(lambda x: x in printable, text)
    return text
```

POS Tagging For Lemmatization

NLTK is again used for **POS tagging** the input text so that the words can be lemmatized based on their POS tags.

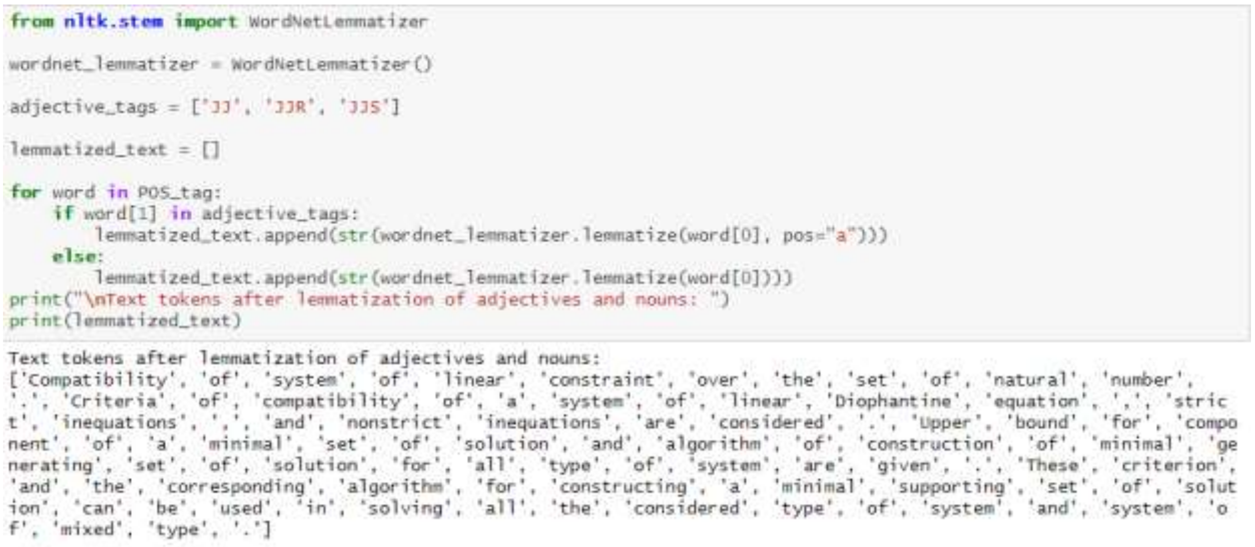


```
POS_tag = nltk.pos_tag(text)

print("\nTokenized Text with POS tags:")
print(POS_tag)
```

```
Tokenized Text with POS tags:
[('Compatibility', 'NN'), ('of', 'IN'), ('systems', 'NNS'), ('of', 'IN'), ('linear', 'JJ'), ('constraints', 'NNS'),
('over', 'IN'), ('the', 'DT'), ('set', 'NN'), ('of', 'IN'), ('natural', 'JJ'), ('numbers', 'NNS'), ('.', '.'), ('Cri
teria', 'NNP'), ('of', 'IN'), ('compatibility', 'NN'), ('of', 'IN'), ('a', 'DT'), ('system', 'NN'), ('of', 'IN'),
('linear', 'JJ'), ('Diophantine', 'NNP'), ('equations', 'NNS'), (',', ','), ('strict', 'JJ'), ('inequations', 'NN
S'), (',', ','), ('and', 'CC'), ('nonstrict', 'JJ'), ('inequations', 'NNS'), ('are', 'VBP'), ('considered', 'VBN'),
('.', '.'), ('Upper', 'NNP'), ('bounds', 'VBZ'), ('for', 'IN'), ('components', 'NNS'), ('of', 'IN'), ('a', 'DT'),
('minimal', 'JJ'), ('set', 'NN'), ('of', 'IN'), ('solutions', 'NNS'), ('and', 'CC'), ('algorithms', 'NN'), ('of', 'I
N'), ('construction', 'NN'), ('of', 'IN'), ('minimal', 'JJ'), ('generating', 'VBG'), ('sets', 'NNS'), ('of', 'IN'),
('solutions', 'NNS'), ('for', 'IN'), ('all', 'DT'), ('types', 'NNS'), ('of', 'IN'), ('systems', 'NNS'), ('are', 'VB
P'), ('given', 'VBN'), ('.', '.'), ('These', 'DT'), ('criteria', 'NNS'), ('and', 'CC'), ('the', 'DT'), ('correspondi
ng', 'JJ'), ('algorithms', 'NN'), ('for', 'IN'), ('constructing', 'VBG'), ('a', 'DT'), ('minimal', 'JJ'), ('supporti
ng', 'NN'), ('set', 'NN'), ('of', 'IN'), ('solutions', 'NNS'), ('can', 'MD'), ('be', 'VB'), ('used', 'VBN'), ('in',
'IN'), ('solving', 'VBG'), ('all', 'PDT'), ('the', 'DT'), ('considered', 'VBN'), ('types', 'NNS'), ('of', 'IN'), ('s
ystems', 'NNS'), ('and', 'CC'), ('systems', 'NNS'), ('of', 'IN'), ('mixed', 'JJ'), ('types', 'NNS'), ('.', '.')]
```

Lemmatization

The tokenized text (mainly the nouns and adjectives) is normalized by **lemmatization**. In lemmatization different grammatical counterparts of a word will be replaced by single basic lemma. For example, 'glasses' may be replaced by 'glass'.

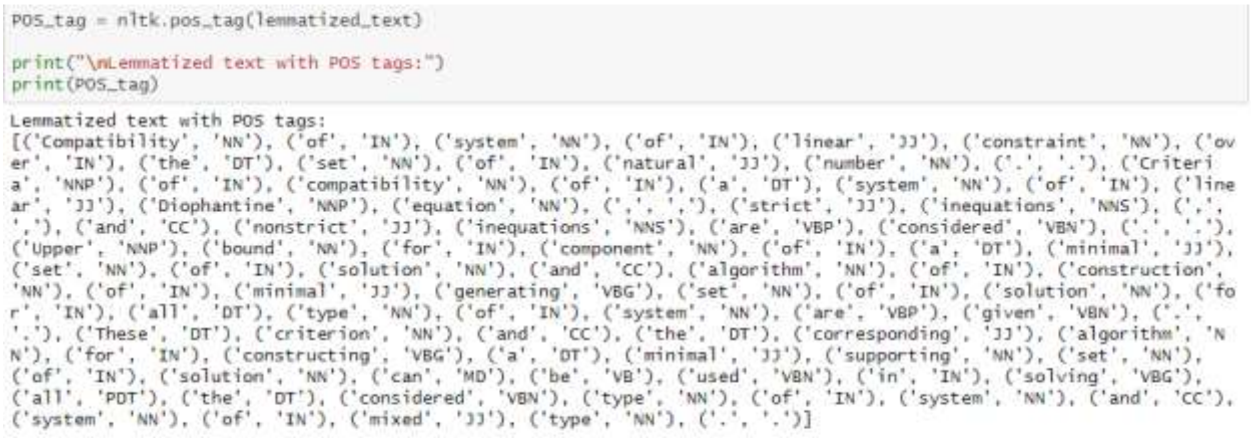


```
from nltk.stem import WordNetLemmatizer

wordnet_lemmatizer = WordNetLemmatizer()

adjective_tags = ['JJ', 'JJR', 'JJS']

lemmatized_text = []

for word in POS_tag:
    if word[1] in adjective_tags:
        lemmatized_text.append(str(wordnet_lemmatizer.lemmatize(word[0], pos="a")))
    else:
        lemmatized_text.append(str(wordnet_lemmatizer.lemmatize(word[0])))
print("\nText tokens after lemmatization of adjectives and nouns: ")
print(lemmatized_text)
```

```
Text tokens after lemmatization of adjectives and nouns:
['Compatibility', 'of', 'system', 'of', 'linear', 'constraint', 'over', 'the', 'set', 'of', 'natural', 'number',
'.', 'Criteria', 'of', 'compatibility', 'of', 'a', 'system', 'of', 'linear', 'Diophantine', 'equation', ',', 'stric
t', 'inequations', ',', 'and', 'nonstrict', 'inequations', 'are', 'considered', '.', 'Upper', 'bound', 'for', 'compo
nent', 'of', 'a', 'minimal', 'set', 'of', 'solution', 'and', 'algorithm', 'of', 'construction', 'of', 'minimal', 'ge
nerating', 'set', 'of', 'solution', 'for', 'all', 'type', 'of', 'system', 'are', 'given', '.', 'These', 'criterion',
'and', 'the', 'corresponding', 'algorithm', 'for', 'constructing', 'a', 'minimal', 'supporting', 'set', 'of', 'solut
ion', 'can', 'be', 'used', 'in', 'solving', 'all', 'the', 'considered', 'type', 'of', 'system', 'and', 'system', 'o
f', 'mixed', 'type', '.']
```

POS tagging for Filtering

The **lemmatized text** is **POS tagged** here. The tags will be used for filtering later on.



```
POS_tag = nltk.pos_tag(lemmatized_text)

print("\nLemmatized text with POS tags:")
print(POS_tag)
```

```
Lemmatized text with POS tags:
[('Compatibility', 'NN'), ('of', 'IN'), ('system', 'NN'), ('of', 'IN'), ('linear', 'JJ'), ('constraint', 'NN'), ('ov
er', 'IN'), ('the', 'DT'), ('set', 'NN'), ('of', 'IN'), ('natural', 'JJ'), ('number', 'NN'), ('.', '.'), ('Criteri
a', 'NNP'), ('of', 'IN'), ('compatibility', 'NN'), ('of', 'IN'), ('a', 'DT'), ('system', 'NN'), ('of', 'IN'), ('line
ar', 'JJ'), ('Diophantine', 'NNP'), ('equation', 'NN'), (',', ','), ('strict', 'JJ'), ('inequations', 'NNS'), (',',
','), ('and', 'CC'), ('nonstrict', 'JJ'), ('inequations', 'NNS'), ('are', 'VBP'), ('considered', 'VBN'), ('.', '.'),
('Upper', 'NNP'), ('bound', 'NN'), ('for', 'IN'), ('component', 'NN'), ('of', 'IN'), ('a', 'DT'), ('minimal', 'JJ'),
('set', 'NN'), ('of', 'IN'), ('solution', 'NN'), ('and', 'CC'), ('algorithm', 'NN'), ('of', 'IN'), ('construction',
'NN'), ('of', 'IN'), ('minimal', 'JJ'), ('generating', 'VBG'), ('set', 'NN'), ('of', 'IN'), ('solution', 'NN'), ('fo
r', 'IN'), ('all', 'DT'), ('type', 'NN'), ('of', 'IN'), ('system', 'NN'), ('are', 'VBP'), ('given', 'VBN'), ('.',
'.'), ('These', 'DT'), ('criterion', 'NN'), ('and', 'CC'), ('the', 'DT'), ('corresponding', 'JJ'), ('algorithm', 'N
N'), ('for', 'IN'), ('constructing', 'VBG'), ('a', 'DT'), ('minimal', 'JJ'), ('supporting', 'NN'), ('set', 'NN'),
('of', 'IN'), ('solution', 'NN'), ('can', 'MD'), ('be', 'VB'), ('used', 'VBN'), ('in', 'IN'), ('solving', 'VBG'),
('all', 'PDT'), ('the', 'DT'), ('considered', 'VBN'), ('type', 'NN'), ('of', 'IN'), ('system', 'NN'), ('and', 'CC'),
('system', 'NN'), ('of', 'IN'), ('mixed', 'JJ'), ('type', 'NN'), ('.', '.')]
```

POS Based Filtering

Any word from the lemmatized text, which isn't a noun, adjective, or gerund (or a 'foreign word'), is here considered as a **stopword**(non-content). This is based on the assumption that usually keywords are noun, adjectives or gerunds.

Punctuations are added to the stopword list too.

Removing Stopwords

Removing stopwords from lemmatized_text. Processeced_text contains the result.

```
stopwords = []

wanted_POS = ['NN','NNS','NNP','NNPS','JJ','JJR','JJS','VBG','FW']

for word in POS_tag:
    if word[1] not in wanted_POS:
        stopwords.append(word[0])

punctuations = list(str(string.punctuation))

stopwords = stopwords + punctuations
```

```
processed_text = []
for word in lemmatized_text:
    if word not in stopwords:
        processed_text.append(word)
print("\n",processed_text)
```

```
 ['Compatibility', 'system', 'linear', 'constraint', 'set', 'natural', 'number', 'Criteria', 'compatibility', 'syste
m', 'linear', 'Diophantine', 'equation', 'strict', 'inequations', 'nonstrict', 'inequations', 'Upper', 'bound', 'com
ponent', 'minimal', 'set', 'solution', 'algorithm', 'construction', 'minimal', 'generating', 'set', 'solution', 'typ
e', 'system', 'criterion', 'corresponding', 'algorithm', 'constructing', 'minimal', 'supporting', 'set', 'solution',
'solving', 'type', 'system', 'system', 'mixed', 'type']
```

Vocabulary Creation

Vocabulary will only contain unique words from processed_text.

```
vocabulary = list(set(processed_text))
print("\n",vocabulary)
```

```
 ['minimal', 'corresponding', 'solving', 'equation', 'set', 'system', 'supporting', 'mixed', 'strict', 'criterion',
'Diophantine', 'Upper', 'type', 'component', 'compatibility', 'number', 'bound', 'constructing', 'linear', 'nonstric
t', 'solution', 'constraint', 'construction', 'generating', 'inequations', 'natural', 'Compatibility', 'Criteria',
'algorithm']
```

Building Graph

TextRank is a graph based model, and thus it requires us to build a graph. Each words in the vocabulary will serve as a vertex for graph. The words will be represented in the vertices by their index in vocabulary list.

The weighted_edge matrix contains the information of edge connections among all vertices. I am building wieghted undirected edges.

weighted_edge[i][j] contains the weight of the connecting edge between the word vertex represented by vocabulary index i and the word vertex represented by vocabulary j.

If weighted_edge[i][j] is zero, it means no edge connection is present between the words represented by index i and j.

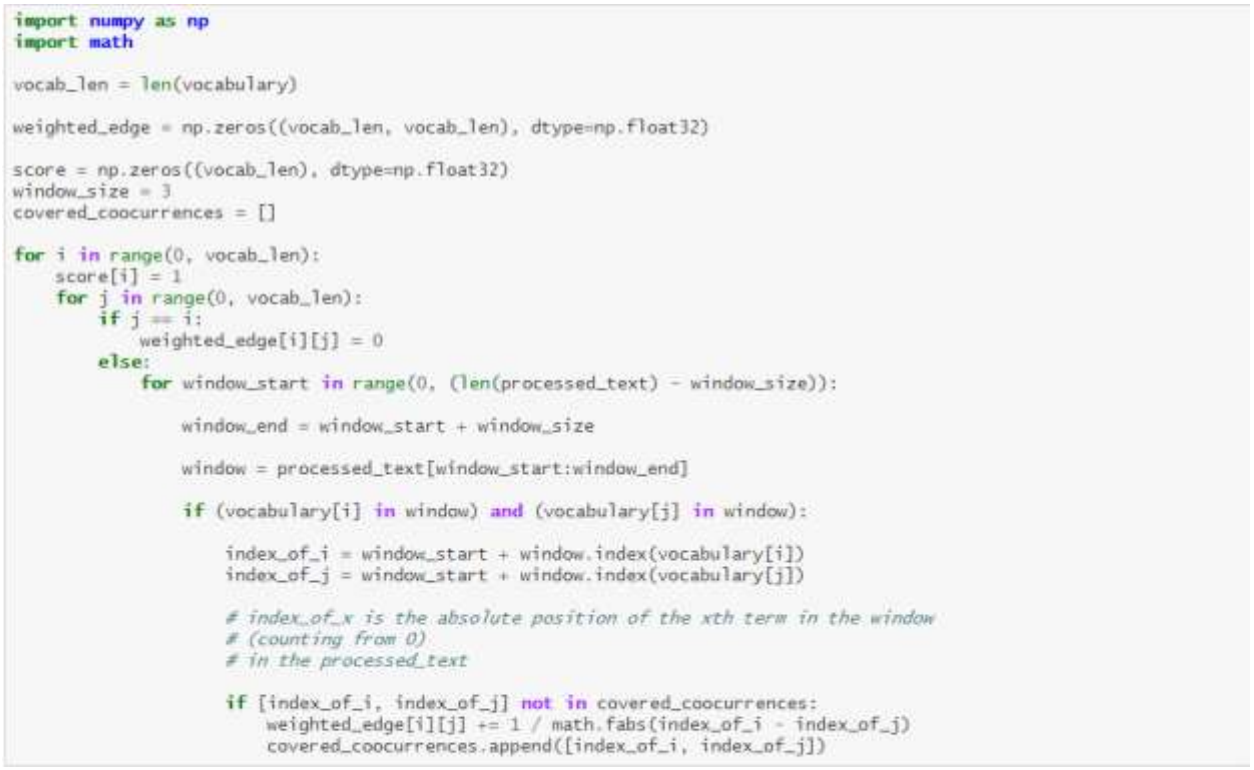
There is a connection between the words (and thus between i and j which represents them) if the words co-occur within a window of a specified 'window_size' in the processed_text.

The value of the weighted_edge[i][j] is increased by (1/(distance between positions of words currently represented by i and j)) for every connection discovered between the same words in different locations of the text.

The covered_coocurrences list (which is contain the list of pairs of absolute positions in processed_text of the words whose co occurrence at that location is already checked) is managed so that the same two words located in the same positions in processed_text are not repetitively counted while sliding the window one text unit at a time.

The score of all vertices are intialized to one.

Self-connections are not considered, so weighted_edge[i][i] will be zero.



```python
import numpy as np
import math

vocab_len = len(vocabulary)

weighted_edge = np.zeros((vocab_len, vocab_len), dtype=np.float32)

score = np.zeros((vocab_len), dtype=np.float32)
window_size = 3
covered_coocurrences = []

for i in range(0, vocab_len):
    score[i] = 1
    for j in range(0, vocab_len):
        if j == i:
            weighted_edge[i][j] = 0
        else:
            for window_start in range(0, (len(processed_text) - window_size)):

                window_end = window_start + window_size

                window = processed_text[window_start:window_end]

                if (vocabulary[i] in window) and (vocabulary[j] in window):

                    index_of_i = window_start + window.index(vocabulary[i])
                    index_of_j = window_start + window.index(vocabulary[j])

                    # index_of_x is the absolute position of the xth term in the window
                    # (counting from 0)
                    # in the processed_text

                    if [index_of_i, index_of_j] not in covered_coocurrences:
                        weighted_edge[i][j] += 1 / math.fabs(index_of_i - index_of_j)
                        covered_coocurrences.append([index_of_i, index_of_j])
```

Calculating weighted summation of connections of a vertex

inout[i] will contain the sum of all the undirected connections\edges associated with the vertex represented by i.

```python
inout = np.zeros((vocab_len),dtype=np.float32)

for i in xrange(0,vocab_len):
    for j in xrange(0,vocab_len):
        inout[i]+=weighted_edge[i][j]
```

Scoring Vertices

The formula used for scoring a vertex represented by i is:

score[i] = (1-d) + d x [Summation(j) ((weighted_edge[i][j]/inout[j]) x score[j])] where j belongs to the list of vertices that has a connection with i.

d is the damping factor.

The score is iteratively updated until convergence.

```
MAX_ITERATIONS = 50
d=0.85
threshold = 0.0001 #convergence threshold

for iter in xrange(0,MAX_ITERATIONS):
    prev_score = np.copy(score)

    for i in xrange(0,vocab_len):

        summation = 0
        for j in xrange(0,vocab_len):
            if weighted_edge[i][j] != 0:
                summation += (weighted_edge[i][j]/inout[j])*score[j]

        score[i] = (1-d) + d*(summation)

    if np.sum(np.fabs(prev_score-score)) <= threshold: #convergence condition
        print "Converging at iteration "+str(iter)+"...."
        break
```

```
Converging at iteration 29....
```

```
for i in xrange(0,vocab_len):
    print "Score of "+vocabulary[i]+": "+str(score[i])
```

```
Score of upper: 0.816792
Score of set: 2.27184
Score of constructing: 0.667288
Score of number: 0.688316
Score of solving: 0.642318
Score of system: 2.12032
Score of compatibility: 0.944584
Score of strict: 0.823772
Score of criterion: 1.22559
Score of type: 1.08101
Score of minimal: 1.78693
Score of supporting: 0.653705
Score of generating: 0.652645
Score of linear: 1.2717
Score of diophantine: 0.759295
Score of component: 0.737641
Score of bound: 0.786006
Score of nonstrict: 0.827216
Score of inequations: 1.30824
Score of natural: 0.688299
Score of algorithm: 1.19365
Score of constraint: 0.674411
Score of equation: 0.799815
Score of solution: 1.6832
Score of construction: 0.659809
Score of mixed: 0.235822
```

Phrase Partitioning

Partitioning lemmatized_text into phrases using the stopwords in it as delimiters. The phrases are also candidates for key phrases to be extracted.

```
phrases = []

phrase = " "
for word in lemmatized_text:

    if word in stopwords_plus:
        if phrase!= " ":
            phrases.append(str(phrase).strip().split())
        phrase = " "
    elif word not in stopwords_plus:
        phrase+=str(word)
        phrase+=" "

print "Partitioned Phrases (Candidate Keyphrases): \n"
print phrases
```

```
Partitioned Phrases (Candidate Keyphrases):

[['compatibility'], ['system'], ['linear', 'constraint'], ['set'], ['natural', 'number'], ['crit
erion'], ['compatibility'], ['system'], ['linear', 'diophantine', 'equation'], ['strict', 'inequ
ations'], ['nonstrict', 'inequations'], ['upper', 'bound'], ['component'], ['minimal', 'set'],
['solution'], ['algorithm'], ['construction'], ['minimal', 'generating', 'set'], ['solution'],
['type'], ['system'], ['criterion'], ['algorithm'], ['constructing'], ['minimal', 'supporting',
'set'], ['solution'], ['solving'], ['type'], ['system'], ['system'], ['mixed', 'type']]
```

Create a list of unique phrases.

Repeating phrases\keyphrase candidates has no purpose here, anymore.

```
unique_phrases = []

for phrase in phrases:
    if phrase not in unique_phrases:
        unique_phrases.append(phrase)

print "Unique Phrases (Candidate Keyphrases): \n"
print unique_phrases
```

```
Unique Phrases (Candidate Keyphrases):

[['compatibility'], ['system'], ['linear', 'constraint'], ['set'], ['natural', 'number'], ['crit
erion'], ['linear', 'diophantine', 'equation'], ['strict', 'inequations'], ['nonstrict', 'inequa
tions'], ['upper', 'bound'], ['component'], ['minimal', 'set'], ['solution'], ['algorithm'], ['c
onstruction'], ['minimal', 'generating', 'set'], ['type'], ['constructing'], ['minimal', 'suppor
ting', 'set'], ['solving'], ['mixed', 'type']]
```

Thinning the list of candidate-keyphrases.

Removing single word keyphrases-candidates that are present multi-word alternatives.

```
for word in vocabulary:
    #print word
    for phrase in unique_phrases:
        if (word in phrase) and ([word] in unique_phrases) and (len(phrase)>1):
            #if len(phrase)>1 then the current phrase is multi-worded.
            #if the word in vocabulary is present in unique_phrases as a single-word-phrase
            # and at the same time present as a word within a multi-worded phrase,
            # then I will remove the single-word-phrase from the list.
            unique_phrases.remove([word])

print "Thinned Unique Phrases (Candidate Keyphrases): \n"
print unique_phrases
```

```
Thinned Unique Phrases (Candidate Keyphrases):

[['compatibility'], ['system'], ['linear', 'constraint'], ['natural', 'number'], ['criterion'],
['linear', 'diophantine', 'equation'], ['strict', 'inequations'], ['nonstrict', 'inequations'],
['upper', 'bound'], ['component'], ['minimal', 'set'], ['solution'], ['algorithm'], ['constructi
on'], ['minimal', 'generating', 'set'], ['constructing'], ['minimal', 'supporting', 'set'], ['so
lving'], ['mixed', 'type']]
```

Scoring Keyphrases

Scoring the phrases (candidate keyphrases) and building up a list of key phrases\keywords by listing untokenized versions of tokenized phrases\candidate-keyphrases. Phrases are scored by adding the score of their members (words\text-units that were ranked by the graph algorithm)

```
phrase_scores = []
keywords = []
for phrase in unique_phrases:
    phrase_score=0
    keyword = ''
    for word in phrase:
        keyword += str(word)
        keyword += " "
        phrase_score+=score[vocabulary.index(word)]
    phrase_scores.append(phrase_score)
    keywords.append(keyword.strip())

i=0
for keyword in keywords:
    print "Keyword: '"+str(keyword)+"', Score: "+str(phrase_scores[i])
    i+=1
```

```
Keyword: 'compatibility', Score: 0.944583714008
Keyword: 'system', Score: 2.12031626701
Keyword: 'linear constraint', Score: 1.94610738754
Keyword: 'natural number', Score: 1.37661552429
Keyword: 'criterion', Score: 1.2255872488
Keyword: 'linear diophantine equation', Score: 2.83080631495
Keyword: 'strict inequations', Score: 2.13201224804
Keyword: 'nonstrict inequations', Score: 2.135455966
Keyword: 'upper bound', Score: 1.60279768705
Keyword: 'component', Score: 0.737640619278
Keyword: 'minimal set', Score: 4.05876886845
Keyword: 'solution', Score: 1.68319940567
Keyword: 'algorithm', Score: 1.19365406036
Keyword: 'construction', Score: 0.659808635712
Keyword: 'minimal generating set', Score: 4.71141409874
Keyword: 'constructing', Score: 0.66728836298
Keyword: 'minimal supporting set', Score: 4.71247345209
Keyword: 'solving', Score: 0.642318367958
Keyword: 'mixed type', Score: 1.31682945788
```

Ranking Keyphrases

Ranking keyphrases based on their calculated scores. Displaying top keywords_num no. of keyphrases.

```
sorted_index = np.flip(np.argsort(phrase_scores),0)

keywords_num = 10

print "Keywords:\n"

for i in xrange(0,keywords_num):
    print str(keywords[sorted_index[i]])+", ",
```

```
Keywords:

minimal supporting set,  minimal generating set,  minimal set,  linear diophantine equation,  no
nstrict inequations,  strict inequations,  system,  linear constraint,  solution,  upper bound,
```

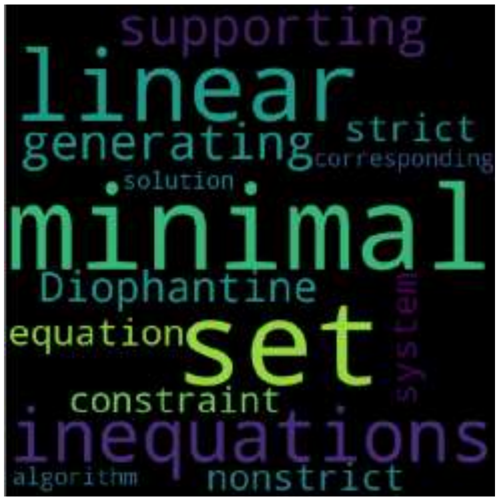
RESULTS

Input:

Compatibility of systems of linear constraints over the set of natural numbers. Criteria of compatibility of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered. Upper bounds for components of a minimal set of solutions and algorithms of construction of minimal generating sets of solutions for all types of systems are given. These criteria and the corresponding algorithms for constructing a minimal supporting set of solutions can be used in solving all the considered types of systems and systems of mixed types.

Extracted Keywords:

- minimal supporting set,
- minimal generating set,
- minimal set,
- linear diophantine equation,
- nonstrict inequations,
- strict inequations,
- system,
- linear constraint,
- solution,
- upper bound,

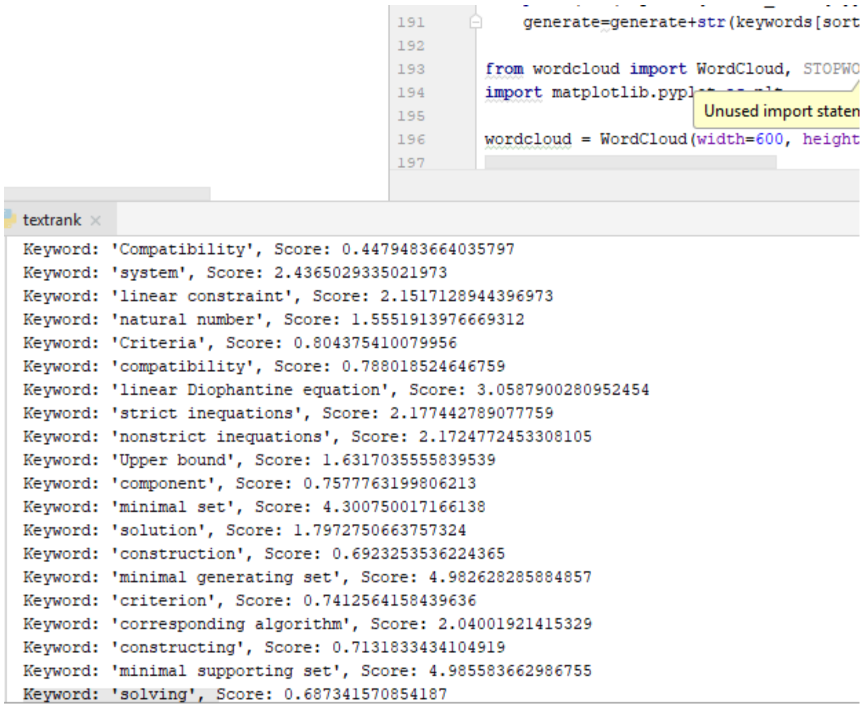


COMPARISON

Input - **Compatibility of systems of linear constraints over the set of natural numbers. Criteria of compatibility of a system of linear Diophantine equations, strict inequations, and nonstrict inequations are considered. Upper bounds for components of a minimal set of solutions and algorithms of construction of minimal generating sets of solutions for all types of systems are given. These criteria and the corresponding algorithms for constructing a minimal supporting set of solutions can be used in solving all the considered types of systems and systems of mixed types.**

<u>TEXTRANK ALGORITHM</u>

Output -



```
    generate=generate+str(keywords[sort

from wordcloud import WordCloud, STOPWO
import matplotlib.pypl
                         Unused import staten
wordcloud = WordCloud(width=600, height
```

textrank

```
Keyword: 'Compatibility', Score: 0.4479483664035797
Keyword: 'system', Score: 2.4365029335021973
Keyword: 'linear constraint', Score: 2.1517128944396973
Keyword: 'natural number', Score: 1.5551913976669312
Keyword: 'Criteria', Score: 0.804375410079956
Keyword: 'compatibility', Score: 0.788018524646759
Keyword: 'linear Diophantine equation', Score: 3.0587900280952454
Keyword: 'strict inequations', Score: 2.177442789077759
Keyword: 'nonstrict inequations', Score: 2.1724772453308105
Keyword: 'Upper bound', Score: 1.6317035555839539
Keyword: 'component', Score: 0.7577763199806213
Keyword: 'minimal set', Score: 4.300750017166138
Keyword: 'solution', Score: 1.7972750663757324
Keyword: 'construction', Score: 0.6923253536224365
Keyword: 'minimal generating set', Score: 4.982628285884857
Keyword: 'criterion', Score: 0.7412564158439636
Keyword: 'corresponding algorithm', Score: 2.04001921415329
Keyword: 'constructing', Score: 0.7131833434104919
Keyword: 'minimal supporting set', Score: 4.985583662986755
Keyword: 'solving', Score: 0.687341570854187
```

Advantages:

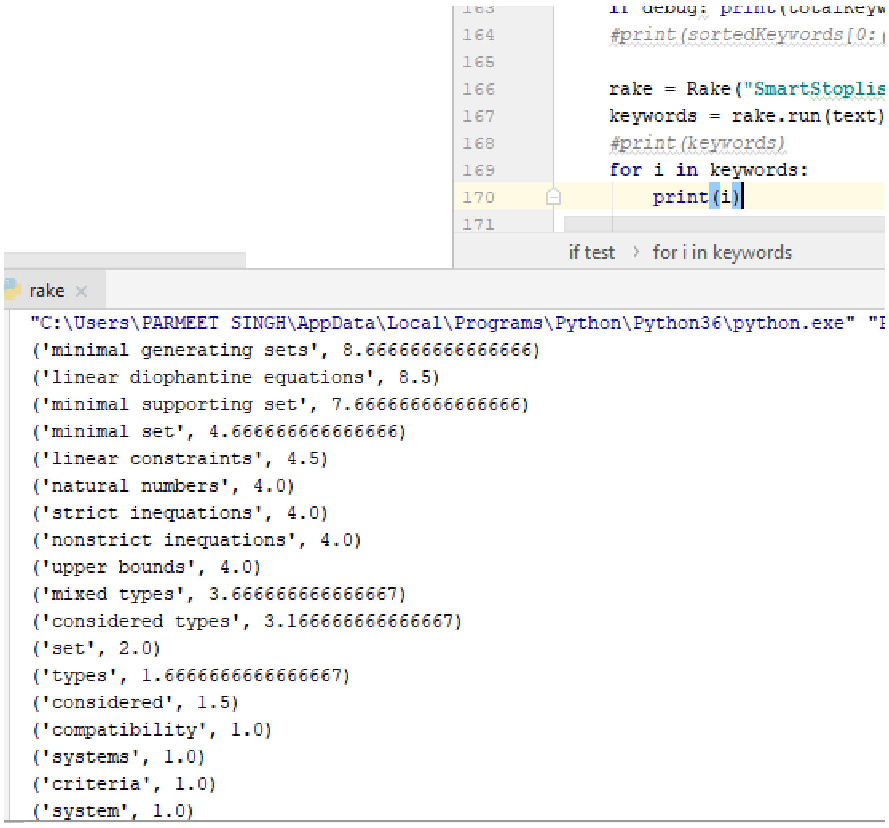
- Compared with supervised methods, TextRank does not need to learn and train a number of documents in advance.
- TextRank algorithm is completely unsupervised. It exclusively relies on information acquired from the text itself, which makes it easily portable to other text collections.

Disadvantages:

- Compared with TF-IDF, the relation between the effect of keyword extraction of TextRank and the effect of segmentation is extremely large.
- TextRank performance is not better than TF-IDF based on 50 documents.
- Due to TextRank needs to achieve words construction and repeated iterations, the extraction speed is slower than TF-IDF.

RAKE ALGORITHM

Output -



```
if debug: print(totalKeyw
#print(sortedKeywords[0:
rake = Rake("SmartStoplis
keywords = rake.run(text)
#print(keywords)
for i in keywords:
    print(i)
```

```
"C:\Users\PARMEET SINGH\AppData\Local\Programs\Python\Python36\python.exe" "I
('minimal generating sets', 8.666666666666666)
('linear diophantine equations', 8.5)
('minimal supporting set', 7.666666666666666)
('minimal set', 4.666666666666666)
('linear constraints', 4.5)
('natural numbers', 4.0)
('strict inequations', 4.0)
('nonstrict inequations', 4.0)
('upper bounds', 4.0)
('mixed types', 3.666666666666667)
('considered types', 3.166666666666667)
('set', 2.0)
('types', 1.6666666666666667)
('considered', 1.5)
('compatibility', 1.0)
('systems', 1.0)
('criteria', 1.0)
('system', 1.0)
```

Advantages:
Compared with TextRank, RAKE is more computationally efficient than TextRank while achieving higher precision and recall.
Due to RAKE has simplicity and efficiency, it can be used in many applications.
Compared with TextRank, the running speed of RAKE is faster.
Disadvantages:
RAKE makes key phrases to acquire higher weight, so the accuracy is inadequate.

RAKE is a simple keyword extraction library, which mainly solves the phrases that contain frequent words. The strength of RAKE algorithm is easy to use, the drawback is limited in accuracy and necessary parameter configuration, it also abandoned a lot of effective phrases. One of the differences between RAKE and TextRank is that RAKE did not normalized candidate words.

TF-IDF

Advantages:

- It is suitable for keyword extraction.
- People can easily compute the similarity between two documents by using TF-IDF

Disadvantages:

- Compared with topic models (LDA, PLSA, etc.), TF-IDF cannot capture semantics.
- TF-IDF mainly focuses on “word frequency” measuring the importance of a word. It is not accurate because some important words may not occur many times.
- TF-IDF can not reflect the location of words. It is not accurate that different locations of the same word have same importance.

CONCLUSION

In this paper, we introduced TextRank – a graph based ranking model for text processing, and show how it can be successfully used for natural language application.Particularly we talked about how TextRank can be used to extract the keywords from a text.We have compared the TextRank algorithm with other algorithms used for keyword extraction by seeing their pros and cons.. An important aspect of TextRank is that it does not require deep linguistic knowledge, nor domain or language specific annotated corpora, which makes it highly portable to other domains, genres, or language.

REFERENCES

1. Weinan Fan, “Comparison of Keyword Extraction with Unsupervised Methods” (2017) School of Computer Science and Informatics, Cardiff University.
2. Xiaolei Zuo,”The enhancement of TextRank algorithm by using word2vec and its application on topic extraction(2017)”J. Phys.: Conf. Ser. 887 012028
3. Page L, Brin S, Motwani R et al. The PageRank citation ranking: Bringing order to the web[R]. Stanford: Stanford InfoLab, 1999
4. Mihalcea R, Tarau P. TextRank: Bringing Order into Texts [J]. UNT, 2004, 90:404-411
5. Rada Mihalcea and Paul Tarau. TextRank: Bringing Order into Texts Department of Computer Science University of North TexasS. Siddiqi and A. Sharan, "Keyword and Keyphrase Extraction Techniques: A Literature Review", International Journal of Computer Applications, vol. 109, no. 2, pp. 18-23, 2015.
6. R. Mihalcea and P. Tarau, "TextRank: Bringing Order into Texts", Department of Computer Science, University of North Texas.

7. S. Rose, D. Engel, N. Cramer and W. Cowley, "Automatic Keyword Extraction from Individual Documents", Text Mining, pp. 1-20, 2010
8. R. Kibble, "Introduction to natural language processing", Undergraduate, University of
9. Y. Kerner, "Automatic Extraction of Keywords from Abstracts", pp. 843-849, 2003.
10. R. S. Wills, "Google's PageRank: The Math Behind the Search Engine", Department of Mathematics North Carolina State University, 2016
11. S. Beliga, "Keyword extraction: a review of methods and approaches", 2014
12. D. Pawar, M. Bewoor and S. Patil, "Text Rank: A Novel Concept for Extraction Based Text Summarization", International Journal of Computer Science and Information Technologies, pp. 3301 - 3304, 2014.
13. G. Ercan and I. Cicekli, "Using lexical chains for keyword extraction", Information Processing Management, pp. 1705–1714, 2007
14. E. Frank, G. W. Paynter, I. H. Witten, C. Gutwin, and C. G. Nevill-Manning. 1999. Domain-specific keyphrase extraction. In Proceedings of the 16th International Joint Conference on Artificial Intelligence.
15. S. Siddiqi and A. Sharan, "Keyword and Keyphrase Extraction Techniques: A Literature Review", International Journal of Computer Applications, vol. 109, no. 2, pp. 18-23, 2015
16. J. Ramos, "Using TF-IDF to Determine Word Relevance in Document Queries", Department of Computer Science, Rutgers University, 23515 BPO Way, Piscataway, NJ, 08855.