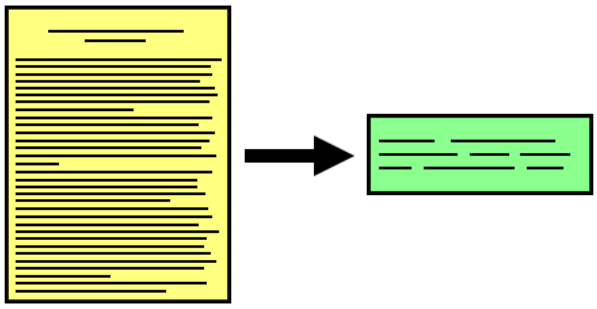
Intro to Automatic Keyphrase Extraction

2014-09-23 feature design frequency statistics keyphrase extraction graph-based ranking NLP task reformulation

I often apply natural language processing for purposes of automatically extracting structured information from unstructured (text) datasets. One such task is the extraction of important topical words and phrases from documents, commonly known as terminology extraction or **automatic keyphrase extraction**. Keyphrases provide a concise description of a document's content; they are useful for document categorization, clustering, indexing, search, and summarization; quantifying semantic similarity with other documents; as well as conceptualizing particular knowledge domains.



Despite wide applicability and much research, keyphrase extraction suffers from poor performance relative to many other core NLP tasks, partly because there's no objectively "correct" set of keyphrases for a given document. While human-labeled keyphrases are generally considered to be the gold standard, humans disagree about what that standard is! As a general rule of thumb, keyphrases should be relevant to one or more of a document's major topics, and the set of keyphrases describing a document should provide good coverage of all major topics. (They should also be understandable and grammatical, of course.) The fundamental difficulty lies in determining which keyphrases are the *most* relevant and provide the *best* coverage. As described in Automatic Keyphrase Extraction: A Survey of the State of the Art, several factors contribute to this difficulty, including document length, structural inconsistency, changes in topic, and (a lack of) correlations between topics.

Methodology

Automatic keyphrase extraction is typically a two-step process: first, a set of words and phrases that could convey the topical content of a document are identified, then these candidates are scored/ranked and the “best” are selected as a document’s keyphrases.

1. Candidate Identification

A brute-force method might consider *all* words and/or phrases in a document as candidate keyphrases. However, given computational costs and the fact that not all words and phrases in a document are equally likely to convey its content, heuristics are typically used to identify a smaller subset of better candidates. Common heuristics include removing stop words and punctuation; filtering for words with certain parts of speech or, for multi-word phrases, certain POS patterns; and using external knowledge bases like WordNet or Wikipedia as a reference source of good/bad keyphrases.

For example, rather than taking all of the *n*-grams (where $1 \leq n \leq 5$) in this post’s first two paragraphs as candidates, we might limit ourselves to only noun phrases matching the POS pattern `{(<JJ>* <NN.*>+ <IN>)? <JJ>* <NN.*>+}` (a regular expression written in a simplified format used by NLTK’s `RegexpParser()`). This matches any number of adjectives followed by at least one noun that may be joined by a preposition to one other adjective(s)+noun(s) sequence, and results in the following candidates:

```
['art', automatic keyphrase extraction', 'changes in topic', 'concise
'content', 'coverage', 'difficulty', 'document', 'document categoriza
'document length', 'extraction of important topical words', 'fundamer
'general rule of thumb', 'gold standard', 'good coverage', 'human-lab
'humans', 'indexing', 'keyphrases', 'major topics', 'many other core
'much research', 'natural language processing for purposes', 'particu
'phrases from documents', 'search', 'semantic similarity with other c
'set of keyphrases', 'several factors', 'state', 'structural inconsis
'summarization', 'survey', 'terminology extraction', 'topics', 'wide
```

Compared to the brute force result, which gives 1100+ candidate *n*-grams, most of which are almost certainly not keyphrases (e.g. “task”, “relative to”, “and the set”, “survey of the state”, …), this seems like a much smaller and more likely set of candidates, right? As document length increases, though, even the number of *likely* candidates can get quite

large. Selecting the best keyphrase candidates is the objective of step 2.

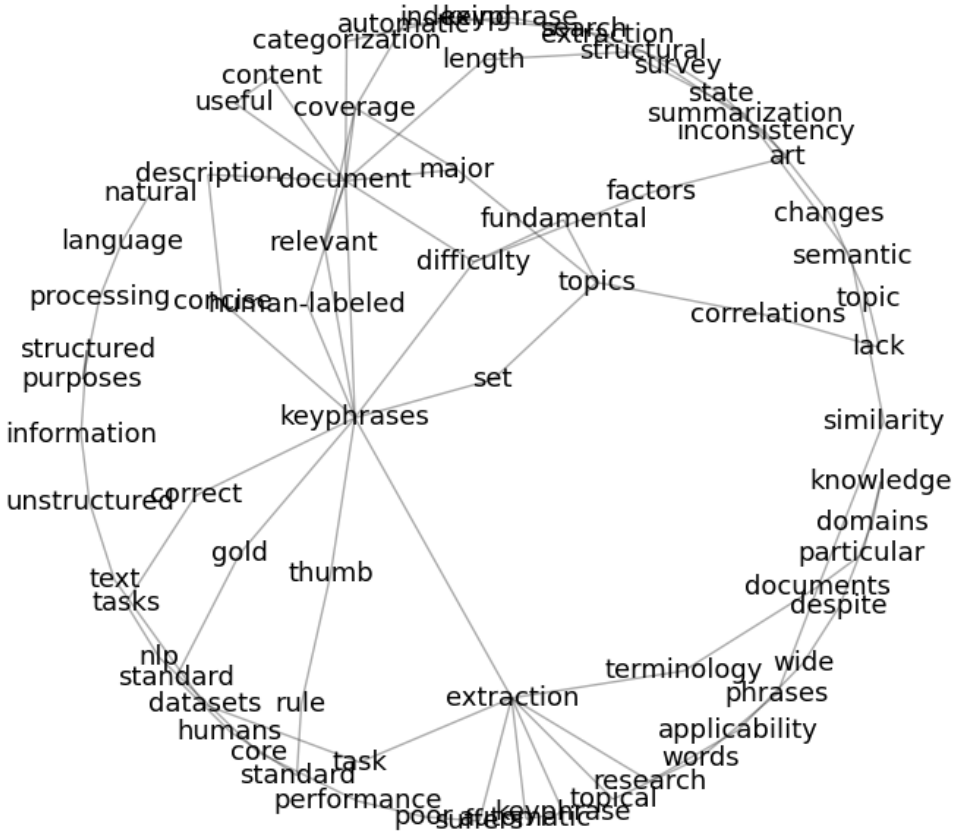
2. Keyphrase Selection

Researchers have devised a plethora of methods for distinguishing between good and bad (or *better* and *worse*) keyphrase candidates. The simplest rely solely on **frequency statistics**, such as TF*IDF or BM25, to score candidates, assuming that a document's keyphrases tend to be relatively frequent within the document as compared to an external reference corpus. Unfortunately, their performance is mediocre; researchers have demonstrated that the best keyphrases aren't necessarily the most frequent within a document. (For a statistical analysis of human-generated keyphrases, check out Descriptive Keyphrases for Text Visualization.) A next attempt might score candidates using multiple statistical features combined in an ad hoc or heuristic manner, but this approach only goes so far. More sophisticated methods apply machine learning to the problem. They fall into two broad categories.

Unsupervised

Unsupervised machine learning methods attempt to discover the underlying structure of a dataset without the assistance of already-labeled examples ("training data"). The canonical unsupervised approach to automatic keyphrase extraction uses a **graph-based ranking** method, in which the importance of a candidate is determined by its relatedness to other candidates, where "relatedness" may be measured by two terms' frequency of co-occurrence or semantic relatedness. This method assumes that more important candidates are related to a greater number of other candidates, and that more of those related candidates are *also* considered important; it does not, however, ensure that selected keyphrases cover all major topics, although multiple variations try to compensate for this weakness.

Essentially, a document is represented as a network whose nodes are candidate keyphrases (typically only key *words*) and whose edges (optionally weighted by the *degree* of relatedness) connect related candidates. Then, a graph-based ranking algorithm, such as Google's famous PageRank, is run over the network, and the highest-scoring terms are taken to be the document's keyphrases.



The most famous instantiation of this approach is TextRank; a variation that attempts to ensure good topic coverage is DivRank. For a more extensive breakdown, see Conundrums in Unsupervised Keyphrase Extraction, which includes an example of a **topic-based clustering** method, the other main class of unsupervised keyphrase extraction algorithms (which I'm not going to delve into).

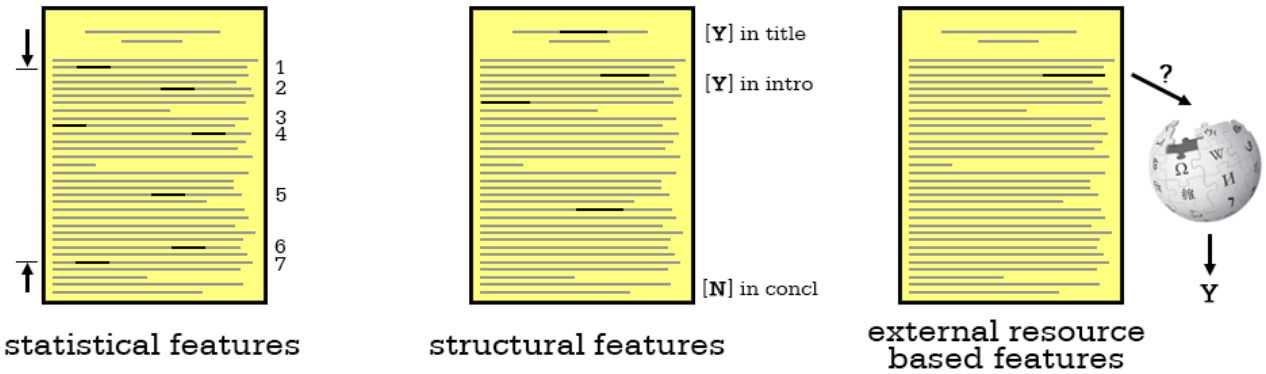
Unsupervised approaches have at least one notable strength: No training data required! In an age of massive but unlabled datasets, this can be a huge advantage over other approaches. As for disadvantages, unsupervised methods make assumptions that don't necessarily hold across different domains, and up until recently, their performance has been inferior to supervised methods. Which brings me to the next section.

Supervised

Supervised machine learning methods use training data to infer a function that maps a set of input variables called features to some desired (and *known*) output value; ideally, this function can correctly predict the (*unknown*) output values of new examples based on their features alone. The two primary developments in supervised approaches to automatic keyphrase extraction deal with **task reformulation** and **feature design**.

Early implementations recast the problem of extracting keyphrases from a document as a **binary classification** problem, in which some fraction of candidates are classified as keyphrases and the rest as *non*-keyphrases. This is a well-understood problem, and there are many methods to solve it: Naive Bayes, decision trees, and support vector machines, among others. However, this reformulation of the task is conceptually problematic; humans don't judge keyphrases independently of one another, instead they judge certain phrases as *more key* than others in a intrinsically relative sense. As such, more recently the problem has been reformulated as a **ranking** problem, in which a function is trained to rank candidates pairwise according to degree of "keyness". The best candidates rise to the top, and the top *N* are taken to be the document's keyphrases.

The second line of research into supervised approaches has explored a wide variety of features used to discriminate between keyphrases and non-keyphrases. The most common are the aforementioned frequency statistics, along with a grab-bag of other **statistical features**: phrase length (number of constituent words), phrase position (normalized position within a document of first and/or last occurrence therein), and "supervised keyphraseness" (number of times a keyphrase appears as such in the training data). Some models take advantage of a document's **structural features** — titles, abstracts, intros and conclusions, metadata, and so on — because a candidate is more likely to be a keyphrase if it appears in notable sections. Others are **external resource-based features**: "Wikipedia-based keyphraseness" assumes that keyphrases are more likely to appear as Wiki article links and/or titles, while phrase commonness compares a candidate's frequency in a document with respect to its frequency in an external corpus. The list of possible features goes on and on.



A well-known implementation of the binary classification method, KEA (as published in Practical Automatic Keyphrase Extraction), used TF*IDF and position of first occurrence (while filtering on phrase length) to identify keyphrases. In A Ranking Approach to Keyphrase Extraction, researchers used a Linear Ranking SVM to rank candidate

keyphrases with much success (but failed to give their algorithm a catchy name).

Supervised approaches have generally achieved better performance than unsupervised approaches; however, good training data is hard to find (although here's a decent place to start), and the danger of training a model that doesn't generalize to unseen examples is something to always guard against (e.g. through cross-validation).

Results

Okay, now that I've scared/bored away all but the truly interested, let's dig into some code and results! As an example document, I'll use all of the text in this post *up to* this results section; as a reference corpus, I'll use all other posts on this blog. In principle, a reference corpus isn't necessary for single-document keyphrase extraction (case in point: TextRank), but it's often helpful to compare a document's candidates against other documents' in order to characterize its particular content. Consider that *tf*idf* reduces to just *tf* (term frequency) in the case of a single document, since *idf* (inverse document frequency) is the same value for every candidate.

As mentioned, there are many ways to extract candidate keyphrases from a document; here's a simplified and compact implementation of the "noun phrases only" heuristic method:

```
def extract_candidate_chunks(text, grammar=r'KT: {(<JJ>* <NN.*>+ <IN>
    import itertools, nltk, string

    # exclude candidates that are stop words or entirely punctuation
    punct = set(string.punctuation)
    stop_words = set(nltk.corpus.stopwords.words('english'))
    # tokenize, POS-tag, and chunk using regular expressions
    chunker = nltk.chunk.regexp.RegexpParser(grammar)
    tagged_sents = nltk.pos_tag_sents(nltk.word_tokenize(sent) for se
    all_chunks = list(itertools.chain.from_iterable(nltk.chunk.tree2c
                                                    for tagged_sent i
    # join constituent chunk words into a single chunked phrase
    candidates = [' '.join(word for word, pos, chunk in group).lower(
                  for key, group in itertools.groupby(all_chunks, lam

    return [cand for cand in candidates
            if cand not in stop_words and not all(char in punct for c
```

When `text` is assigned to the first two paragraphs of this post, `set(extract_candidate_chunks(text))` returns more or less the same set of candidate keyphrases as listed in 1. Candidate Identification. (Additional cleaning and filtering code improves the list a bit and helps to makes up for tokenizing/tagging/chunking errors.) For comparison, the original TextRank algorithm performs best when extracting all (unigram) nouns and adjectives, like so:

```
def extract_candidate_words(text, good_tags=set(['JJ','JJR','JJS','NN
    import itertools, nltk, string

    # exclude candidates that are stop words or entirely punctuation
    punct = set(string.punctuation)
    stop_words = set(nltk.corpus.stopwords.words('english'))
    # tokenize and POS-tag words
    tagged_words = itertools.chain.from_iterable(nltk.pos_tag_sents(r
                                                                    t
    # filter on certain POS tags and lowercase all words
    candidates = [word.lower() for word, tag in tagged_words
                  if tag in good_tags and word.lower() not in stop_wo
                  and not all(char in punct for char in word)]

    return candidates
```

In this case, `set(extract_candidate_words(text))` gives basically the same set of words visualized as a network in the sub-section on unsupervised methods.

Code for keyphrase selection depends entirely on the approach taken, of course. It's relatively straightforward to implement the simplest, frequency statistic-based approach using scikit-learn or gensim:

```
def score_keyphrases_by_tfidf(texts, candidates='chunks'):
    import gensim, nltk

    # extract candidates from each text in texts, either chunks or wo
    if candidates == 'chunks':
        boc_texts = [extract_candidate_chunks(text) for text in texts
    elif candidates == 'words':
```

```
        boc_texts = [extract_candidate_words(text) for text in texts]
    # make gensim dictionary and corpus
    dictionary = gensim.corpora.Dictionary(boc_texts)
    corpus = [dictionary.doc2bow(boc_text) for boc_text in boc_texts]
    # transform corpus with tf*idf model
    tfidf = gensim.models.TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]

    return corpus_tfidf, dictionary
```

First we assign `texts` to a list of normalized text content (stripped of various YAML, HTML, and Markdown formatting) from all previous blog posts *plus* the first two sections of this post, then we call `score_keyphrases_by_tfidf(texts)` to get all posts back in a sparse, *tf*idf*-weighted representation. It's now trivial to print out the 20 candidate keyphrases with the highest *tf*idf* values for this blog post:

```
keyphrase                           tfidf
----------------------------------------
keyphrases.......................... 0.573
document............................ 0.375
candidates.......................... 0.306
approaches.......................... 0.191
approach............................ 0.115
candidate........................... 0.115
major topics........................ 0.115
methods............................. 0.115
automatic keyphrase extraction..... 0.076
frequency statistics................ 0.076
keyphrase........................... 0.076
keyphrase candidates................ 0.076
network............................. 0.076
relatedness......................... 0.076
researchers......................... 0.076
set of keyphrases................... 0.076
state............................... 0.076
survey.............................. 0.076
function............................ 0.075
performance......................... 0.075
```

Not too shabby! Although you can clearly see how stemming or lemmatizing candidates would improve results (*candidate* / *candidates*, *approach* / *approaches*, and *keyphrase* / *keyphrases* would normalize together). You can also see that this approach seems to favor unigram keyphrases, likely owing to their much higher frequencies of occurrence in natural language texts. Considering that human-selected keyphrases are most often bigrams (according to the analysis in Descriptive Keyphrases for Text Visualization), this seems to be another limitation of such simplistic methods.

Now, let's try a bare-bones implementation of the TextRank algorithm. To keep it simple, only unigram candidates (not chunks or *n*-grams) are added to the network as nodes, the co-occurrence window size is fixed at 2 (so only adjacent words are said to "co-occur"), and the edges between nodes are unweighted (rather than weighted by the number of co-occurrences). The *N* top-scoring candidates are taken to be its keywords; sequences of adjacent keywords are merged to form key *phrases* and their individual PageRank scores are averaged, so as not to bias for longer keyphrases.

```
def score_keyphrases_by_textrank(text, n_keywords=0.05):
    from itertools import takewhile, tee, izip
    import networkx, nltk

    # tokenize for all words, and extract *candidate* words
    words = [word.lower()
             for sent in nltk.sent_tokenize(text)
             for word in nltk.word_tokenize(sent)]
    candidates = extract_candidate_words(text)
    # build graph, each node is a unique candidate
    graph = networkx.Graph()
    graph.add_nodes_from(set(candidates))
    # iterate over word-pairs, add unweighted edges into graph
    def pairwise(iterable):
        """s -> (s0,s1), (s1,s2), (s2, s3), ..."""
        a, b = tee(iterable)
        next(b, None)
        return izip(a, b)
    for w1, w2 in pairwise(candidates):
        if w2:
            graph.add_edge(*sorted([w1, w2]))
    # score nodes using default pagerank algorithm, sort by score, ke
    ranks = networkx.pagerank(graph)
    if 0 < n_keywords < 1:
```

```
        n_keywords = int(round(len(candidates) * n_keywords))
    word_ranks = {word_rank[0]: word_rank[1]
                  for word_rank in sorted(ranks.iteritems(), key=lamb
    keywords = set(word_ranks.keys())
    # merge keywords into keyphrases
    keyphrases = {}
    j = 0
    for i, word in enumerate(words):
        if i < j:
            continue
        if word in keywords:
            kp_words = list(takewhile(lambda x: x in keywords, words[
            avg_pagerank = sum(word_ranks[w] for w in kp_words) / flo
            keyphrases[' '.join(kp_words)] = avg_pagerank
            # counter as hackish way to ensure merged keyphrases are
            j = i + len(kp_words)

    return sorted(keyphrases.iteritems(), key=lambda x: x[1], reverse
```

With `text` as the first two sections of this post, calling `score_keyphrases_by_textrank(text)` returns the following top 20 keyphrases:

```
keyphrase                                textrank
-----------------------------------------------
keyphrases.......................          0.028
candidates.......................          0.022
document.........................          0.022
candidate keyphrases.............          0.019
best keyphrases..................          0.018
keyphrase candidates.............          0.017
likely candidates................          0.015
best candidates..................          0.015
best keyphrase candidates........          0.014
features.........................          0.013
keyphrase........................          0.012
keyphrase extraction.............          0.012
extraction.......................          0.012
methods..........................          0.011
candidate........................           0.01
words............................           0.01
```

```
automatic keyphrase extraction.....     0.01
approaches.........................    0.009
problem............................    0.009
set................................    0.008
```

Again, not too shabby, but obviously there's room for improvement. You can see that this algorithm occasionally produces novel and high-quality keyphrases, but there's a fair amount of noise, too. Normalization of candidates (*keyphrase* / *keyphrases*, …) could help, as could better cleaning and filtering. Furthermore, experimenting with different aspects of the algorithm — like DivRank, SingleRank, ExpandRank, CollabRank, and others — including co-occurrence window size, weighted graphs, and the manner in which keywords are merged into keyphrases, has been shown to produce better results.

Lastly, let's try a supervised algorithm. I prefer a ranking approach over binary classification, for conceptual as well as result quality reasons. Conveniently, someone has already implemented a pairwise Ranking SVM in Python — and blogged about it! Feature design is something of an art; drawing on multiple sources for inspiration, I extracted a diverse grab-bag of features:

- **frequency-based:** term frequency, g^2 , corpus and web "commonness" (as defined here)
- **statistical:** term length, spread, lexical cohesion, max word length
- **grammatical:** "is acronym", "is named entity"
- **positional:** normalized positions of first and last occurrence, "is in title", "is in key excerpt" (such as an abstract or introductory paragraph)

Feature extraction can get very complicated and convoluted. In the interest of brevity and simplicity, then, here's a partial example:

```
def extract_candidate_features(candidates, doc_text, doc_excerpt, doc
    import collections, math, nltk, re

    candidate_scores = collections.OrderedDict()

    # get word counts for document
    doc_word_counts = collections.Counter(word.lower()
                                          for sent in nltk.sent_toker
                                          for word in nltk.word_toker
```

```
    for candidate in candidates:

        pattern = re.compile(r'\b'+re.escape(candidate)+r'(\b|[,;.!?]

        # frequency-based
        # number of times candidate appears in document
        cand_doc_count = len(pattern.findall(doc_text))
        # count could be 0 for multiple reasons; shit happens in a si
        if not cand_doc_count:
            print '**WARNING:', candidate, 'not found!'
            continue

        # statistical
        candidate_words = candidate.split()
        max_word_length = max(len(w) for w in candidate_words)
        term_length = len(candidate_words)
        # get frequencies for term and constituent words
        sum_doc_word_counts = float(sum(doc_word_counts[w] for w in c
        try:
            # lexical cohesion doesn't make sense for 1-word terms
            if term_length == 1:
                lexical_cohesion = 0.0
            else:
                lexical_cohesion = term_length * (1 + math.log(cand_c
        except (ValueError, ZeroDivisionError) as e:
            lexical_cohesion = 0.0

        # positional
        # found in title, key excerpt
        in_title = 1 if pattern.search(doc_title) else 0
        in_excerpt = 1 if pattern.search(doc_excerpt) else 0
        # first/last position, difference between them (spread)
        doc_text_length = float(len(doc_text))
        first_match = pattern.search(doc_text)
        abs_first_occurrence = first_match.start() / doc_text_length
        if cand_doc_count == 1:
            spread = 0.0
            abs_last_occurrence = abs_first_occurrence
        else:
            for last_match in pattern.finditer(doc_text):
                pass
```

```
            abs_last_occurrence = last_match.start() / doc_text_lengt
            spread = abs_last_occurrence - abs_first_occurrence

        candidate_scores[candidate] = {'term_count': cand_doc_count,
                                       'term_length': term_length, 'm
                                       'spread': spread, 'lexical_coh
                                       'in_excerpt': in_excerpt, 'in_
                                       'abs_first_occurrence': abs_fi
                                       'abs_last_occurrence': abs_las

    return candidate_scores
```

As an example, `candidate_scores["automatic keyphrase extraction"]` returns the following features:

```
{'abs_first_occurrence': 0.029178287921046986,
 'abs_last_occurrence': 0.9301652006007295,
 'in_excerpt': 1,
 'in_title': 1,
 'lexical_cohesion': 0.9699006820274416,
 'max_word_length': 10,
 'spread': 0.9009869126796826,
 'term_count': 6,
 'term_length': 3}
```

The last thing to do is train a Ranking SVM model on an already-labeled dataset; I used the SemEval 2010 keyphrase extraction dataset, plus a couple extra bits and pieces, which can be found in this GitHub repo. When applied to the first two sections of this blog post, the 20 top-scoring candidates are as follows:

```
keyphrase                                ranksvm
-----------------------------------------------
keyphrase extraction.............   1.736
document categorization..........   1.151
particular knowledge domains.....   1.031
phrases from documents...........   1.014
keyphrase........................    0.97
terminology extraction...........   0.951
keyphrases.......................   0.909
```

```
set of keyphrases.................  0.895
concise description...............  0.873
document..........................  0.691
human-labeled keyphrases..........  0.643
candidate identification..........  0.642
frequency of co-occurrence........  0.636
candidate keyphrases..............  0.624
wide applicability................  0.604
rest as non-keyphrases............  0.578
binary classification problem.....  0.567
canonical unsupervised approach....  0.566
structural inconsistency..........  0.556
paragraphs as candidates..........  0.548
```

Now *that* is a nice set of keyphrases! There's some bias for longer keyphrases (and longer words within keyphrases), perhaps because the training dataset was about 90% scientific articles, but it's not inappropriate for this science-ish blog's content.

All of the code shown here has been pared down and simplified for demonstration purposes. Adding extensive candidate cleaning, filtering, case/syntactic normalization, and de-duplication can dramatically reduce noise and improve results, as can incorporating additional features and external resources into the keyphrase selection algorithm. Furthermore, although all of these methods were presented in the context of single-document keyphrase extraction, there are ways to extract keyphrases from *multiple* documents and thus categorize/cluster/summarize/index/conceptualize entire corpora. This really is just an introduction to an ongoing challenge in natural language processing research.

On a final note, just for kicks, here are the top 50 keyphrases from my long-neglected Thomas Friedman corpus:

```
keyphrase                                    score
-----------------------------------------------
United States........................... 393.736
Bush Administration..................... 390.941
Administration.......................... 310.831
Israel.................................. 256.609
Palestine Liberation Organization....... 256.148
Middle East............................. 182.171
```

```
President Bush.......................... 170.812
Clinton................................. 166.669
Administration officials................ 164.812
Clinton Administration.................. 158.695
Lebanon................................. 150.466
Baker................................... 141.051
President Clinton....................... 138.680
Secretary of State...................... 135.036
Soviet Union............................ 133.976
West Bank............................... 128.490
Palestinian............................. 121.275
State Department........................ 107.860
Washington.............................. 102.507
Prime Minister.......................... 101.282
Saudi Arabia............................ 100.062
White House.............................  83.649
Beirut..................................  81.046
Reagan Administration...................  80.338
Israeli officials.......................  80.119
Yasir Arafat............................  79.334
Israeli.................................  74.289
Israeli Army............................  70.909
China...................................  69.484
Saddam Hussein..........................  68.645
United Nations..........................  63.641
President...............................  62.621
America.................................  59.833
foreign policy..........................  59.444
Bush....................................  56.545
Lebanese Army...........................  55.878
Arafat..................................  53.848
American officials......................  52.991
President Obama.........................  51.924
Iraq....................................  51.896
peace conference........................  51.549
Bill Clinton............................  50.438
west Beirut.............................  49.418
Jerusalem...............................  48.914
Israeli Government......................  47.910
Gorbachev...............................  44.902
Syria...................................  44.791
```

```
Administration official.................  41.743
Palestinian guerrillas..................  39.849
Lebanese................................  39.299
```

Looks like a who’s who of U.S. politics and international relations over the past 30 years. Not too shabby, Friedman!

← previous ↑

Soheil Dal • 5 months ago

Very nice post! One of the best summaries on keyphrase extraction I've seen. There is a new approach called SGRank that's beating the unsupervised state of the art on the semeval 2010 dataset: http://www.aclweb.org/antholog...

thanks

Shashank Sonkar Soheil Dal • 4 months ago

Please share the code if possible. It will be very helpful. **@Soheil Dal**

bjdewilde Mod Soheil Dal • 5 months ago

Thank you very much! Believe it or not, I started implementing SGRank just a couple days ago. :) It's less a new approach than a smart combination of multiple past approaches, but the improvement in performance is undeniable, and great to see. Thanks again, Soheil — especially for the reminder that I really ought to update this blog. I'm still here!

Shashank Sonkar bjdewilde • 4 months ago

Please share the code if possible. It will be very helpful. **@Soheil Dal** **@bjdewilde**