ADVANCED LEARNING FOR TEXT AND GRAPH DATA
MASTER DATA SCIENCE - MVA

Lab session 3: Keyword Extraction

Antoine Tixier, Giannis Nikolentzos and Michalis Vazirgiannis

Thursday, February 2, 2017

1 Introduction

1.1 Motivation and setting

In this lab session, you will get hands-on experience with the **graph of words** representation of text and see how **graph theory** can be used on top of this representation to perform the task of **keyword extraction**.

Keyword extraction is a very important task. Indeed, keywords are used everywhere: from information retrieval (search engines) and ranking, to classification, summarization, text generation (NLG), and data visualization... More specifically, the task of keyword extraction can be:

- *single* or *multidocument*, depending on whether the input is from a single document or multiple ones,
- *extractive* or *abstractive*, depending on whether the extracted content is restricted to the original text or not,
- *generic* or *query-based* or *update*, depending on whether the extracted keywords are generic, impacted by some user request, or based on prior latent information (e.g., browsing history),
- *unsupervised* or *supervised*, depending on whether the extraction process involves labeled data.

Today, we will focus on **unsupervised generic extractive single-document keyword extraction**, which is the most standard of the aforementioned variants.

1.2 Logistics

`Python 2.7.` and the `igraph` and `nltk` libraries are required. Installation instructions can respectively be found here[1] and here[2] .

[1] `http://igraph.org/python/`
[2] `http://www.nltk.org/install.html`

2 Graph-of-Words

Unlike the vector space model that assumes term independence, graph-of-words [1] offer an information-rich way of encoding text, by capturing term dependency, and even term order, if directed edges are used (like in Figure 1).

2.1 Sliding window

A graph $G(V, E)$ is a set of nodes (V) and a set of edges (E). There are different ways of constructing a graph of words. Here, we use the classical approach of [1], based on a sliding window. Each unique term in the document is represented by a node of the graph, and two nodes are linked by an edge if the terms they represent co-occur within the sliding window. Furthermore, edge weights are integers matching co-occurrence counts. This fully statistical approach is based on the distributional hypothesis [2]: *"We shall know a word by the company it keeps"*.

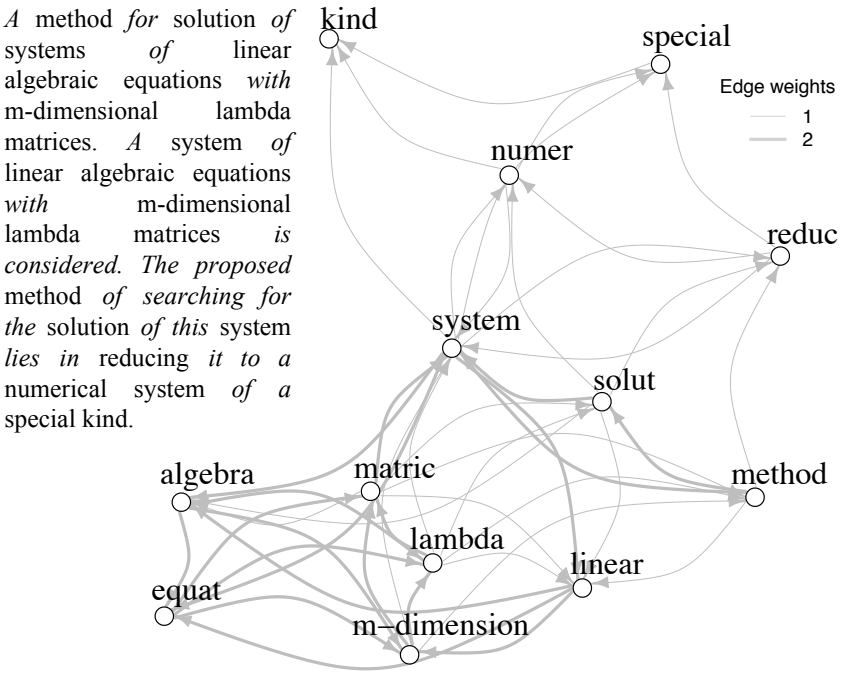

Figure 1: Graph-of-Words representation with POS-based screening, and directed, weighted edges. Non-(nouns and adjectives) in *italic*. Words have been stemmed. Window size $= 4$

2.2 Text pre-processing

Before constructing a graph of words, the document needs to be cleaned. The standard steps include (1) conversion to lower case, (2) punctuation removal, (3) tokenization, and (4) stopword removal. Additionally, for keyword extraction, (5) part-of-speech-based filtering (retaining only nouns and adjectives) and (6) stemming ("winner", "winning", and "win" are all collapsed to "win") can be used as they have empirically been proven useful. These steps are implemented in the `clean_text_simple` function, found within the `code\library.py` file.

2.3 Your turn

Fill the gaps in the `clean_text_simple` function. The construction of a graph of words is implemented by the `terms_to_graph` function (found at the same location). Fill the gaps in this function as well and then, use the `gow_toy.py` script to build a graph for the text shown in Figure 1, with a window of size 4. You can validate your results for instance by comparing some edges (source, target, and weight) with Figure 1. Also, experiment with different window sizes and assess the impact on the density of the graph, defined as $|E|/|V|(|V|-1)$ and accessible via the `.density()` method in `igraph`. What do you observe?

3 Graph degeneracy

The concept of graph degeneracy was introduced by [3] with the k-core decomposition technique and was first applied to the study of cohesion in social networks.

3.1 k-core

A core of order k (or **k-core**) of G is a maximal connected subgraph of G in which every vertex v has at least degree k. If the edges are unweighted, the degree of v is simply equal to the count of its incident edges (number of neighbors), while in the weighted case, the degree of v is the sum of the weights of its incident edges. Edge direction can be taken into account in both cases, by considering only the incoming or the outgoing edges rather than all of them. Also, in both cases, node degrees (and thus, k) are integers since edge weights are integers.

3.2 k-core decomposition

As shown in Figure 2, the **k-core decomposition** of G is the set of all its cores from 0 (G itself) to k_{max} (its main core). It forms a hierarchy of nested subgraphs whose cohesiveness and size respectively increase and decrease with k. The **core number** of a node is the highest order of a k-core subgraph that contains this node. The main core of G is a coarse approximation of its densest subgraph. Intuitively, we can assume that the main core of a graph of words will contain good keywords for the associated document (in a following section, we will verify that this assumption holds).

k-core decomposition algorithms can be found in [4] for the unweighted and weighted cases. Both algorithms implement a pruning process that removes the lowest degree node at each step. Algorithm 1 shows the *unweighted* k-core decomposition algorithm.

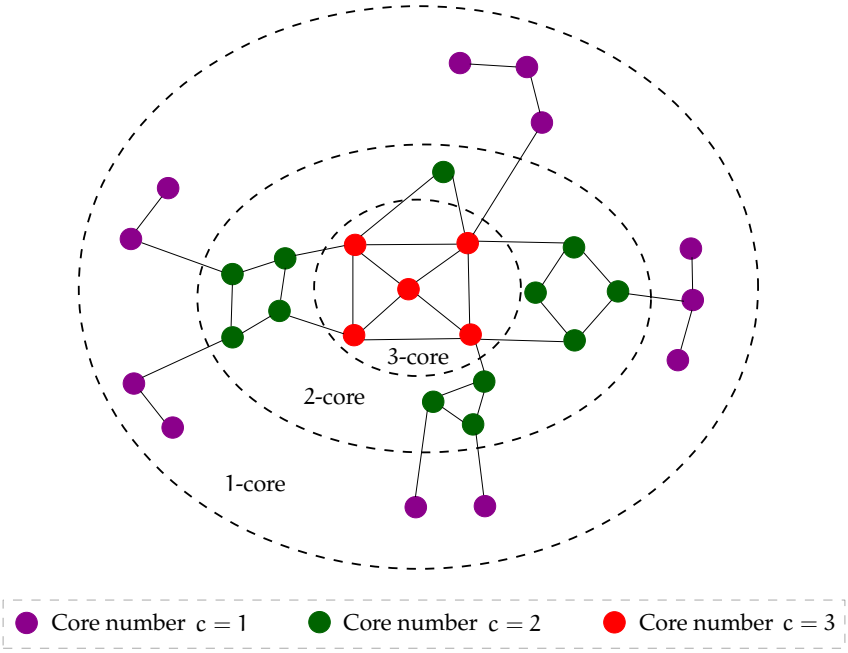

Figure 2: Illustration of the k-core decomposition

Algorithm 1 k-core decomposition

Input: Undirected graph $G = (V, E)$
Output: Core numbers $c(v), \forall v \in V$

$i \leftarrow 0$
while $|V| > 0$ **do**
  while $\exists v : degree(v) \leq i$ **do**
    $c(v) \leftarrow i$
    $V \leftarrow V \setminus \{v\}$
    $E \leftarrow E \setminus \{(u, v) | u \in V\}$
  end while
  $i \leftarrow i + 1$
end while

3.3 Your turn

Implement Algorithm 1 by filling the gaps in the `unweighted_k_core` function. You can use the `.strength()` and `.delete_vertices()` `igraph` methods. Then, use your implementation to decompose the graph shown in Figure 1 and compare your results with that obtained via the `.coreness()` method and Figure 3 below:

4 Keyword extraction

4.1 Data set

We will use the test set of the Hulth 2003 dataset [5], that you can find inside the `data\Hulth2003testing` directory. This dataset contains 500 scientific paper abstracts. For each abstract, human annotators

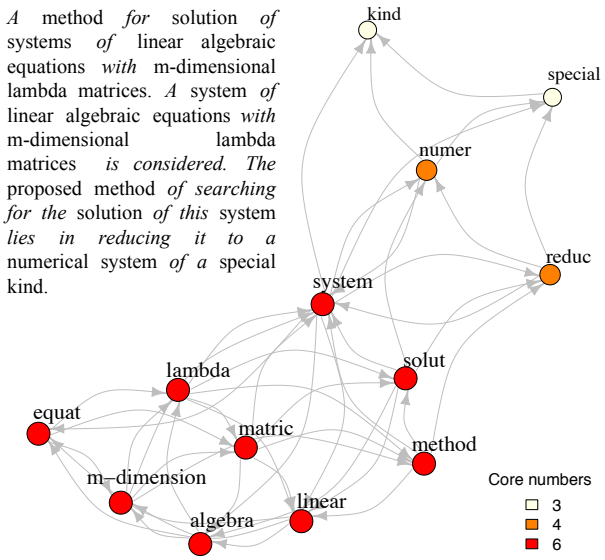

Figure 3: The main core of the graph of words can be used as the keywords for the document

(actually, the authors of the paper) have provided a set of keywords, that we will consider our gold standard (the best possible solution).

4.2 Baselines

We will evaluate the performance of the k-core-based approach against that of Google's PageRank (applied on the same graph-of-words) and tf-idf. In each case, the top p percent nodes will be retained as keywords (e.g., with $p = 33\%$).

4.3 Performance evaluation

We will evaluate the performance of the different techniques in terms of macro-averaged precision, recall and F1 score. Precision can be seen as the *purity* of retrieval while recall measures the *completeness* of retrieval. Finally, the F-1 score is the harmonic mean of precision and recall. More precisely, these metrics are defined as follows:

$$\text{precision} = \frac{tp}{tp + fp}$$

$$\text{recall} = \frac{tp}{tp + fn}$$

$$\text{F1-score} = 2\frac{\text{precision.recall}}{\text{precision} + \text{recall}}$$

Where tp, fp and fn respectively denote the number of true positives (the keywords returned by the system which are also part of the gold standard), false positives (the keywords returned by the system which are *not* part of the gold standard), and false negatives (the keywords from the gold standard that are *not* returned by the system).

Finally, macro-averaging means that we will be computing the metrics for each document and then taking the means at the collection level.

4.4 Your turn

Put everything together by filling the gaps in the `keyword_extraction.py` file and in the `accuracy_metrics` function (inside `library.py`). For the baselines, you can use the `.pagerank()`[3] igraph method, and for tfidf, the `TfidfVectorizer`[4] function from the `scikit-learn` library. Looking at the performance of the different approaches, what can you say? What is the major disadvantage of the unweighted k-core keyword extraction technique? How could it be improved?

[3] `http://igraph.org/python/doc/igraph.Graph-class.html#pagerank`

[4] `http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`

References

[1] Mihalcea, R., Tarau, P. (2004, July). TextRank: Bringing order into texts. Association for Computational Linguistics.

[2] Harris, Z. S. (1954). Distributional structure. Word, 10(2-3), 146-162.

[3] Seidman, S. B. (1983). Network structure and minimum degree. Social networks, 5(3), 269-287.

[4] Batagelj, V., Zavernik, M. (2002). Generalized cores. arXiv preprint cs/0202039.

[5] Hulth, A. (2003, July). Improved automatic keyword extraction given more linguistic knowledge. In Proceedings of the 2003 conference on Empirical methods in natural language processing (pp. 216-223). Association for Computational Linguistics.