Expert Systems With Applications 97 (2018) 163–176

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Document-specific keyphrase candidate search and ranking

Qingren Wang[a,*], Victor S. Sheng[b,*], Xindong Wu[c]

[a] Hefei University of Technology, 193 Tunxi Road, Hefei, Anhui Province 230009 China
[b] University of Central Arkansas, 201 Donaghey Ave., Conway, AR 72035 USA
[c] School of Computing and Informatics, University of Louisiana at Lafayette, 222 James R. Oliver Hall, Lafayette, LA 70504-3694, USA

ARTICLE INFO

Article history:
Received 30 June 2017
Revised 14 December 2017
Accepted 16 December 2017
Available online 16 December 2017

Keywords:
Keyphrase candidate search
Sequential pattern mining
Keyphrase candidate ranking
Entropy

ABSTRACT

This paper proposes an approach KeyRank to extract proper keyphrases from a document in English. It first searches all keyphrase candidates from the document, and then ranks them for selecting top-*N* ones as final keyphrases. Existing studies show that extracting a complete keyphrase candidate set that includes semantic relations in context, and evaluating the effectiveness of each candidate are crucial to extract high quality keyphrases from documents. Based on that words do not repeatedly appear in an effective keyphrase in English, a novel keyphrase candidate search algorithm using sequential pattern mining with gap constraints (called KCSP) is proposed to extract keyphrase candidates for KeyRank. And then an effectiveness evaluation measure pattern frequency with entropy (called PF-H) is proposed for KeyRank to rank these keyphrase candidates. Our experimental results show that KeyRank has better performance. Its first component KCSP is much more efficient than a closely related approach SPMW, and its second component PF-H is an effective evaluation mechanism for ranking keyphrase candidates.[1]

1. Introduction

A keyphrase (Liu, Song, Liu, & Wang, 2012) is an ordered list of words that captures the main points discussed in a natural language document. Keyphrases in a document can help understand the main points of this document. Keyphrases have been successfully used in many text mining tasks, such as automatic indexing, topic extraction, document summarization and text categorization, and so on. Due to the importance of keyphrase, many studies have been conducted to extract high quality keyphrases from documents. This is called keyphrase extraction. Existing keyphrase extraction approaches are based on unsupervised learning and supervised learning (Xie, Wu, & Zhu, 2014). They usually contain two components, keyphrase candidate search and keyphrase selection. Keyphrase candidate search is to extract a keyphrase candidate set from a document. After a keyphrase candidate set is extracted, all these approaches conduct keyphrase selection to select proper keyphrases from the keyphrase candidate set using different technologies. For supervised learning based approaches, keyphrase selection is formulated as a classification task, where each candidate is classified as either a keyphrase or a non-keyphrase. For unsupervised learning based approaches, keyphrase selection is to rank keyphrase candidates in terms of a specific measure, and then the top-*N* (*N* is the number of proper keyphrases, specified by users for a document) keyphrase candidates are selected as keyphrases. Studies (Ercan & Cicekli, 2007; Xu, Yang, & Lau, 2010; Haddoudand & Abdeddaim, 2014) showed that semantic relations in context can help improve the performance for keyphrase extraction. Thus, extracting a complete keyphrase candidate set that includes semantic relations in context, and selecting proper keyphrases from the keyphrase candidate set are crucial to extract high quality keyphrases from documents. In this paper, we focus on the two crucial components of keyphrase extraction, keyphrase candidate search and keyphrase selection, and propose an efficient approach for keyphrase extraction (Wang, Sheng, & Wu, 2017).

The original work on keyphrase extraction simply treats single words with high frequency as keyphrase candidates. However, single words do not capture semantic relations in context. Approaches based on single words with high frequency cannot extract a complete keyphrase candidate set that includes semantic relations in context. Some studies considered contiguous frequently-occurring words as keyphrase candidates, such as Kea (Witten, Paynter, Frank, Gutwin, & Nevill-Manning, 1999). Nevertheless, no matter how many contiguous frequently-occurring words that a keyphrase candidate has, it still ignores some semantic relations in context (Fu, Huang, Sun, Vasilakos, & Yang, 2016; Fu, Ren, Shu, Sun, & Huang, 2016; Li, Li, Yang, & Sun, 2015). Intuitively, single words in a document are the minimum

* Corresponding authors.
E-mail addresses: qingren.wang@mail.hfut.edu.cn (Q. Wang), ssheng@uca.edu (V.S. Sheng), xwu@louisiana.edu (X. Wu).

[1] Our two-page extended abstract is published in AAAI 2017 (Wang, Sheng, & Wu, 2017).

https://doi.org/10.1016/j.eswa.2017.12.031
0957-4174/

The Original Text	The Stemmed Form
Topic aware social influence propagation models The study of influence-driven propagations in social networks and its exploitation for viral marketing purposes has recently received a large deal of attention. However, regardless of the fact that users authoritativeness, expertise, trust and influence are evidently topic-dependent, the research on social influence has surprisingly largely overlooked this aspect. In this article, we study social influence from a topic modeling perspective. We introduce novel topic-aware influence-driven propagation models that, as we show in our experiments, are more accurate in describing real-world cascades than the standard (i.e., topic-blind) propagation models studied in the literature. In particular, we first propose simple topic-aware extensions of the well-known Independent Cascade and Linear Threshold models. However, these propagation models have a very large number of parameters which could lead to overfitting. Therefore, we propose a different approach explicitly modeling authoritativeness, influence and relevance under a topic-aware perspective. Instead of considering user-to-user influence, the proposed model focuses on user authoritativeness and interests in a topic, leading to a drastic reduction in the number of parameters of the model. We devise methods to learn the parameters of the models from a data set of past propagations. Our experimentation confirms the high accuracy of the proposed models and learning schemes.	***topic-awar*** *social influenc* ***propag model*** the studi influenc driven propag social network exploit viral market purpos recent receiv larg deal attent howev fact user authorit expertis trust influenc evid topic depend research social influenc surprisingli larg overlook aspect in articl studi social influenc topic model perspect we introduc novel **topic-awar** influenc driven **propag model** show experi accur describ real world cascad than standard (ie topic blind) propag model studi literatur in propos simpl **topic-awar** extens well known independ cascad linear threshold model howev **propag model** larg number paramet lead overfit therefor propos differ approach explicitli model authorit influenc relev **topic-awar** perspect instead consid user-to-user influenc propos model focus user authorit interest topic lead drastic reduct number paramet model we devis method learn paramet model data set **propag** our experiment confirm high accuraci propos **model** learn scheme
Top Frequent Words	model; topic; propag; influenc; social; …
Top Frequent Patterns with Gap Constraints	topic model: 4; **topic-awar propag model**: 4; social influenc: 3; social influenc model: 3; …
Keywords labeled by authors	social influence; topic modeling; **topic-aware propagation model**; viral marketing;

Fig. 1. Examples of using frequent words vs. wildcard based sequential patterns for keyphrase extraction. (Xie et al., 2014).

meaningful and independent units, and meanwhile a document is an ordered list of words. Therefore, some studies treat the keyphrase candidate search as a task of sequential pattern mining with gap constraints, where single words of documents are viewed as characters of sequences and keyphrase candidates are viewed as patterns. Xie et al. (2014) combined wildcards into sequential pattern mining to search keyphrase candidates from a document (called SPMW), since wildcards can provide gap constraints with a great flexibility for mining patterns to capture semantic relations in the document (Agrawal & Srikant, 1995). Compared with the approaches based on single or contiguous frequently-occurring words, approaches based on sequential pattern mining can discover a richer pattern (keyphrase candidate) set, which helps improve the quality of keyphrase extraction. Note that in this paper a pattern is actually defined as a keyphrase candidate, and keyphrase candidates and patterns are exchangeable respectively since then.

Here we employ and adapt one of the examples provided by Xie et al. (2014) to explain why utilizing sequential pattern mining with gap constraints to search keyphrase candidates is better (see Fig. 1). The example is the title and the abstract of a journal paper published in *Knowledge and Information Systems* (2013). The left part of the second row shows the original title (in italics) and the original abstract, and the right part shows the title and the abstract in a stemmed form. The last row shows four keyphrases labeled by its authors. Among the four keyphrases, the entire string "*topic-aware propagation model*" occurs 0 times in the text (either stemmed or not stemmed). However, sequential pattern mining with gap constraints can extract "*topic-awar propag model*" four times from the text, once in title and three times in the abstract (refer to bold-faced words in the right part of the second row in Fig. 1). Therefore, "*topic-awar propag model*" is extracted as a keyphrase candidate as shown in the fourth row in Fig. 1.

Although sequential pattern mining with gap constraints can extract keyphrase candidates with a higher quality, existing sequential pattern mining based approaches are computational expensive due to two weaknesses. On the one hand, the gap constraints in these approaches play a very important role, but they require users to explicitly specify appropriate gap constraint(s) beforehand. In reality it is often nontrivial and time-consuming for users to provide a proper gap constraint. On the other hand, these approaches need to scan a document multiple times for searching patterns. Repeated document scanning can cause a lot of time overhead, even for a short document. Many studies (Fumarola, Lanotte, Ceci, & Malerba, 2016; Loglisci & Malerba, 2009; Xie et al., 2014) toward closed patterns showed that they preserve information and help keep the computational complexity under control. However, closed patterns do not work well on keyphrase extraction because it neglects to consider the three inherent properties of a pattern to capture a point in a document, especially uncertainty. In this paper, we focus on documents in English and solve the following two issues: 1) reducing the computation time of searching keyphrase candidates, and 2) measuring the probability of a keyphrase candidate to capture a point expressed by its corresponding document.

After having consulted linguists, they confirmed that words do not repeatedly appear in an effective keyphrase in English. Based on their confirmation, we treat keyphrase candidate search as a sequential pattern mining task, and propose a novel keyphrase candidate search algorithm (called KCSP) using sequential pattern mining with gap constraints. KCSP only scans a document once for obtaining every word whose frequency is no less than a given support threshold, and its corresponding appearing positions in the document. Then KCSP generates a corresponding position interval for each word at each appearing position, and treats it as the gap constraint of the word at the current appearing position. Therefore, the gap constraint of a word at an appearing position becomes the inherent property of the word at the current appearing position, rather than an external parameter specified by users.

The gap constraints of words at different appearing positions in a document can be specified automatically and appropriately by words' own appearing positions, which greatly improves the accuracy and the appropriateness of gap constraints of words at different appearing positions and significantly reduces the computation time for searching patterns. Finally, KCSP utilizes a series of calculations between the gap constraints of a front word and the appearing positions of words after to search patterns, which overcomes the weakness of repeatedly scanning a document in the existing sequential pattern mining based approaches. In addition, our KCSP utilizes the order of words appearing in each paragraph to extract keyphrases from each paragraph. It intends to extract neighbor words as keyphrases. In other words, KCSP extracts keyphrases with considering the context of words. This is a way to take advantage of semantics. We notice that KCSP generates duplication patterns during the pattern search process and duplication production and elimination consume time. Therefore, we further completely eradicate duplication production in KCSP.

We further need to determine that keyphrase selection should be treated as a classification task or a ranking task. Jiang, Hu, and Li (2009) claimed that keyphrase selection by nature is a ranking problem rather than a classification problem. First, it is more natural to consider the likelihood of a keyphrase candidate is a keyphrase in a relative sense, instead of in an absolute sense. Second, the difficulty of making hard decisions on keyphrases or non-keyphrases could be avoided. Finally, information (features) for determining whether a keyphrase candidate is a keyphrase is also relative. Moreover, unsupervised learning based approaches do not require training data. This is one of the most important advantages in the big data era, because there are massive unlabeled data generated in the Internet every day. Thus, we will treat the keyphrase selection as a ranking task in this paper. Many mechanisms based on TF-IDF have been shown to work well in practice despite of its simplicity (Liu, Pennell, Liu, & Liu, 2009; Xia, Wang, Sun, & Wang, 2015), such as pattern frequency-inverse document frequency (Yang & Yu, 2007; Zanmatkesh & Hassanpour, 2011, 2012). However, they are not quite suitable for ranking patterns since they do not consider the three inherent properties of a pattern to capture a point: meaningfulness, uncertainty and uselessness. In this paper, we employ entropy (Shannon, 1948) to measure the three inherent properties of a pattern to capture a point. Intuitively, the greater the entropy of a pattern, the lower probability the pattern captures a point of the corresponding document. Therefore, we propose a new evaluation mechanism pattern frequency with entropy (called PF-H) to rank patterns in the pattern set, and then select the top-N patterns as proper keyphrases. Our ranking measurement PF-H takes sub-patterns and parent patterns (including closed patterns) into consideration. This is another way to take advantage of semantics. Our entire approach is denoted as KeyRank hereafter.

The remainder of the paper is organized as follows. Section 2 introduces related work. Section 3 proposes the pattern search algorithm KCSP. Section 4 introduces the pattern ranking mechanism PF-H. Section 5 reports our experimental results, and then we conclude the paper in Section 6.

2. Related work

Keyphrase extraction approaches usually contain two main components. One performs the keyphrase candidate search. Another performs keyphrase selection from keyphrase candidate set. In the following, we review related work from these two aspects.

2.1. Keyphrase candidate search

Kea (Witten et al., 1999) considered contiguous frequently-occurring words as keyphrase candidates, which usually are limited to three words. However, no matter how many contiguous frequently-occurring words a keyphrase candidate has, it still ignores some semantic relations in context (Fu, Huang et al., 2016; Fu, Ren et al., 2016; Li et al., 2015). Chen, Luesukprasert, and Chou (2007) used the frequency of a term plus its life cycle to determine whether this term is a candidate, which also cannot capture semantic relations in context. Some studies (Ercan & Cicekli, 2007; Xu et al., 2010; Haddoudand & Abdeddaim, 2014) showed that semantic features can improve the performance of keyphrase candidate search and keyphrase extraction. Therefore, many studies used co-occurrence to capture semantic features for keyphrase candidate search. Wan and Xiao (2008) considered that only nouns and adjectives could be used to form a keyphrase candidate, and uses co-occurrence between two words within a window of maximum w words to generate a keyphrase candidate. Cheng, Yan, Lan, and Guo (2014) denoted an unordered word pair co-occurrence as a candidate in short texts with a limited document length, and any two distinct words in a document, no matter contiguous or non-contiguous, can be a candidate. In general, keyphrase candidates extracted by co-occurrence based approaches hold semantic relations in context. However, co-occurrence means the maximum length of a keyphrase candidate is two, which ignores the keyphrase candidates whose lengths are larger than two. Ceci, Appice, and Malerba (2008, 2014) ranked sentences of a document using preference-relations modeled by a probabilistic relational data mining method, and used these sentences to construct the summary of the document.

Sequential pattern mining plays an important role in data mining, and was first introduced by Agrawal and Srikant (1995). It seeks to discover sets of frequent items sharing some temporal relationships, and such patterns have been found to be useful for many applications (Wu, Zhu, He, & Arslan, 2013): stock market (Dorr & Denton, 2009) and sequence classification (Exarchos, Tsipouras, Papaloukas, & Fotiadis, 2008), etc. A number of methods use gap constraints to mine patterns from DNA sequences (Zhang, Kao, Cheung, & Yip, 2007; Zhu & Wu, 2007), since gap constraints (wildcards) can provide a great flexibility for patterns to capture relations. Besides, there exist methods based on Apriori properties (Agrawal & Srikant, 1994; Pei et al., 2001) to improve the efficiency of pattern mining.

Since single words in a document are the minimum meaningful and independent units, and meanwhile a document is an ordered list of words, keyphrase candidate search can be treated as a sequential pattern mining task, where single words of documents are viewed as characters of sequences and keyphrase candidates are viewed as patterns. Feng, Xie, Hu, Li, Cao, and Wu (2011) and Xie et al. (2014) applied sequential pattern mining with wildcards to search keyphrase candidates (called SPAM and SPMW respectively). SPAM utilizes a depth-first traversal of the search space with a vertical bitmap representation. SPMW uses a depth-first pattern growth strategy based on a data structure, a level instance graph, to represent all instances of a pattern with a gap constraint. Although a complete keyphrase candidate set that includes semantic relations in context can be extracted, repeated document scanning and manually setting gap constraints are two inevitable weaknesses. INSGrow (Ding, Lo, & Khoo, 2009) is an efficient algorithm for repetitive gapped subsequence mining. It combines repetitive support into sequential pattern mining to improve the efficiency, since repetitive support captures not only repetitions of a pattern in different sequences but also repetitions within a sequence. PMBC (Wu, Zhu et al., 2013) can automatically and efficiently discover patterns without user-specified gap constraints, and meanwhile recommends the most common gap constraints for users. Although INSGrow and PMBC do not need to set gap constraints manually, they are still not quite suitable for document-specific keyphrase candidate search since most keyphrase candidates they discovered have duplicate words and

many keyphrase candidate sets they extracted are not complete. Besides, they have another common weakness that both of them depend on repeated document scanning. By contrast, our algorithm KCSP can automatically generate appropriate gap constraints, and only scans a document once since it employs interval calculations instead of repeated document scanning.

2.2. Keyphrase selection

For supervised learning based approaches, the keyphrase selection is formulated as a classification task, where each keyphrase candidate is classified as either keyphrase or non-keyphrase. GenEx (Turney, 1999) and Kea (Witten et al., 1999) are two typical supervised approaches for keyphrase selection. In GenEx, a set of feature parameters are tuned by a genetic algorithm to maximize the performance on a training dataset. Kea uses two features, i.e., a TF-IDF value and the first occurrence position of a keyphrase candidate, as input to Naïve Bayes. KeyEx (Xie et al., 2014) also uses these two features. The only difference between Kea and KeyEx is the way of generating the keyphrase candidate set.

For unsupervised learning based approaches, the keyphrase selection is treated as a ranking task (Fu, Sun, Liu, Zhou, & Shu, 2015; Fu, Wu et al., 2016), where keyphrase candidates are ranked by their scores in terms of corresponding measures, and the top-N candidates are selected as keyphrases. TF-IDF is not only widely used in supervised approaches, but also widely used in unsupervised approaches (Grineva, Grinev, & Lizorkin, 2009; Zhang, Zincir-Heywood, & Milios, 2005), and the ranking measurements based on TF-IDF have been shown to work well in practice (Liu et al., 2009; Xia et al., 2015) despite of its simplicity. Yang and Yu (2007) used two or three sequent words to form a key pattern to replace the keyword as the feature, and the distributive strength of key patterns is measured by pattern frequency-inverse document frequency (PF-IDF in short). Hassanpour and Zanmatkesh (2011, 2012) used n-gram, where $n \leq 3$, to form a main phrase with frequency greater than two to replace the keyword, and the importance of a main phrase is also calculated by PF-IDF. Note that the types of patterns in previous works are different from the pattern in our ranking mechanism PF-H. The pattern in the previous works is a contiguous sequence of two or three items extracted from a given sequence of text. However, the pattern in this paper can be a contiguous or a non-contiguous sequence without length limitation.

Some studies (Hasan & Ng, 2014; Liu, Huang, Zheng, & Sun, 2010; Mihalcea & Tarau, 2004) used graph-based ranking methods for automatic keyphrase extraction, in which the importance of a keyphrase candidate is determined by its relatedness to other keyphrase candidates, where "relatedness" may be measured by the frequency of co-occurrence or the semantic relatedness of two keyphrase candidates (DeWilde, 2014). TextRank (Mihalcea & Tarau, 2004) is one of the most famous graph-based approaches. However, TextRank only aggregates the keyphrase candidates of a document, so that the corresponding frequency of co-occurrence or semantic relatedness is not stable and could not accurately reveal the "relatedness" between keyphrase candidates. Our ranking mechanism PF-H is based on an entire corpus.

3. Algorithm for pattern search

3.1. Existing definitions

In this section, we review and adapt some definitions for the problem of sequential pattern mining with gap constraints used in (Wu, Xie, Huang, Hu, & Gao, 2013; Xie et al., 2014).

A paragraph $D = w_1 w_2 \ldots w_n$ is an ordered list of words. Since a document contains a set of paragraphs, which can be viewed as a paragraph database, defined as $SeqDB = \{D_1, D_2, \ldots, D_N\}$. A *wildcard* is a special symbol that can be matched by any word in *SeqDB*. A *gap* is a list of wildcards, and its size refers to the number of wildcards. $g[N_i, M_i]$ $(0 \leq N_i < M_i, i \geq 1)$ is used to represent the ith gap whose size is within the range $[N_i, M_i]$, and is called a gap constraint with the gap flexibility $gf_i = M_i - N_i + 1$.

A *pattern* $P = w_1 g[N_1, M_1] w_2 g[N_2, M_2] \ldots g[N_{m-1}, M_{m-1}] w_m$ is a list of words and gaps, which begins and ends with words, where w_i $(1 \leq i \leq m)$ is a word, and $g[N_i, M_i]$ $(1 \leq i < m)$ is the gap constraint between w_i and w_{i+1}. In addition, these ranges $([N_1, M_1], [N_2, M_2], \ldots, [N_{m-1}, M_{m-1}])$ of gap constraints $(g[N_1, M_1], g[N_2, M_2], \ldots, g[N_{m-1}, M_{m-1}])$ can be the same or different. The number of words in P is called the length of P, denoted by $|P|$, without counting wildcards inside.

Definition 3.1. (Pattern Occurrence and Instance): Given a pattern $P = p_1 p_2 \ldots p_m$ and a paragraph $D = w_1 w_2 \ldots w_n$, if there exists a position sequence $1 \leq i_1 < i_2 < \ldots < i_m \leq n$, such that $w_{i_j} = p_j$ for all $1 \leq j \leq m$, and $N_j \leq i_j - i_{j-1} \leq M_j$ for all $2 \leq j \leq m$, then $(i_1, i_2, \ldots, i_m)$ is called an occurrence of P in D with gap constraints. Given a pattern P and a paragraph database $SeqDB = \{D_1, D_2, \ldots, D_N\}$, if $(i_1, i_2, \ldots, i_m)$ is an occurrence of P in D_j $(D_j \in SeqDB)$, then $(j, <i_1, i_2, \ldots, i_m>)$ is said to be an instance of P in D_j $(D_j \in SeqDB)$. Note that if $(t_1, t_2, \ldots, t_m)$ is another occurrence of P in D_j $(t_m \neq i_m)$, then $(j, <t_1, t_2, \ldots, t_m>)$ is its corresponding instance of P in *SeqDB*, and meanwhile $(j, <i_1, i_2, \ldots, i_m>)$ and $(j, <t_1, t_2, \ldots, t_m>)$ are two different instances of P in *SeqDB*. If $(t_1, t_2, \ldots, t_m)$ is another occurrence of P in D_l $(t_m \neq i_m)$, then $(l, <t_1, t_2, \ldots, t_m>)$ is its corresponding instance of P in *SeqDB*, and meanwhile $(j, <i_1, i_2, \ldots, i_m>)$ and $(l, <t_1, t_2, \ldots, t_m>)$ are also different instances of P in *SeqDB*.

Definition 3.2. (Pattern Support): The support of a pattern P in a paragraph database *SeqDB*, denoted by *sup(P)*, is defined as the number of different instances of P in *SeqDB*. If the support of P is not less than a given support threshold, we say that P is a frequent pattern.

Definition 3.3. (Parent Pattern & Sub-pattern): Given two patterns $P = p_1 p_2 \ldots p_m$ and $Q = q_1 q_2 \ldots q_t$, if $m \leq t$, and there exists a position sequence $1 \leq j_1 < j_2 < \ldots < j_m \leq t$, such that $p_k = q_{j_k}$ for all $1 \leq k \leq m$, then P is called a sub-pattern of Q, and Q is called a parent pattern of P.

3.2. Algorithm KCSP

In this section, we will explain our algorithm KCSP in detail through a simple example. We will employ a stemmed text snippet with length $L = 69$ from Fig. 1 (see words with underline in Fig. 1) as Example 3.1 in Section 3.3, and the given pattern support threshold *min_sup* is set as 3. Before starting KCSP, some pre-processing techniques (i.e., stop-word removing, stemming and punctuation-mark removing) are utilized to make each paragraph in a paragraph database as an uninterrupted sequence of words. The problem of searching keyphrase candidates can be formalized into four functional parts as follows.

- Part 1: KCSP uses the left-most priority strategy to scan a paragraph database with length L once to obtain every word whose frequency is no less than a given support threshold (*min_sup*) and its corresponding positions and intervals, which are put into an ordered array called *WordArray*. After that, KCSP sorts *WordArray* according to words' first appearing positions from low to high, outputs these words as qualified patterns, and calls Part 2 (with input *WordArray*).
- Part 2: KCSP tries to use the words in *WordArray* one by one as the first word of a pattern. If a word w can be the first word of a pattern P, KCSP calls Part 3 (with input P). When all words in *WordArray* have been tried, KCSP stops.

- Part 3: KCSP chooses a word v from the remaining words in *WordArray*, and calls Part 4 to verify whether P and v could be concatenated as a new pattern P_1. If positive, KCSP outputs P_1 as a qualified pattern and calls Part 3 (with input P_1) recursively; otherwise, KCSP calls Part 2.
- Part 4: KCSP lists all intervals of the last word of pattern P to calculate how many positions of v meet these intervals. If the number of positions of v that meet intervals is no less than *min_sup*, it returns positive.

The notion "*qualified pattern*" mentioned in Part 1 means a frequent pattern with different words, and the notion "*interval*" is defined as follows.

Definition 3.4. (Position Interval of Single Word, Interval in short): Suppose there is a word w with m different appearing positions pos_i in a paragraph database with length L ($1 \leq i \leq m < L$, $1 \leq pos_i < pos_{i+1} < L+1$). Any two adjacent appearing positions pos_i and pos_{i+1} of w can generate an interval, denoted by $Range[pos_i, pos_{i+1}]$.

Since w at its last appearing position pos_m does not have a next adjacent appearing position, the last interval is presented as $Range[pos_m, L+1]$, where $L+1$ is called *virtual end position*. A virtual end position is a common position, which does not exist in reality. It is obvious to determine how many appearing positions w has, how many intervals are generated for w. Like $g[N_i, M_i]$ ($0 \leq N_i < M_i$, $i \geq 1$), the interval $Range[pos_i, pos_{i+1}]$ is used to represent the ith gap whose size is within the range $[pos_i, pos_{i+1}]$, and also can be called a gap constraint with the gap flexibility $gf_i = pos_{i+1} - pos_i + 1$. That is, $Range[pos_i, pos_{i+1}]$ not only presents the gap constraint between w at the appearing position pos_i and w at the appearing position pos_{i+1}, but also becomes the inherent property of w at the appearing position pos_i, rather than an external parameter specified by users. Therefore, the interval $Range[pos_i, pos_{i+1}]$ between w at the appearing position pos_i and w at the appearing position pos_{i+1} can be called the gap constraint of w at pos_i.

The Round 1 in Example 3.1 (discussed in Section 3.3) completes the work of Part 1. During Round 1, KCSP scans the given paragraph once to obtain every word whose frequency is no less than *min_sup*. Meanwhile, KCSP records the appearing positions of these words, and generates intervals for words at different appearing positions according to their appearing positions and the virtual end position. Note that each interval denotes a gap constraint for a word at an appearing position. For example, word *topic-awar* has three appearing positions 4, 27 and 52 (see Fig. 2(a)). The adjacent appearing positions 4 and 27 generate an interval *Range*[4, 27], which presents the gap constraint of *topic-awar* at the appearing position 4. It means that *topic-awar* at the appearing position 4 can be extended with words at appearing positions from 5 to 26. Similarly, the adjacent appearing positions 27 and 52 generates an interval *Range*[27, 52], which presents the gap constraint of *topic-awar* at the appearing position 27. The gap constraint of *topic-awar* at its last appearing position 52 is presented as *Range*[52, 70], since the appearing position 52 does not have the next adjacent appearing position and the position 70 is the virtual end position of the corresponding paragraph. Words and their appearing positions and intervals are put into *WordArray* according to their first appearing positions from low to high (listed in blue dashed rectangles in Fig. 2(a)). After Round 1, KCSP outputs "*topic-awar*", "*propag*", "*model*" and "*propos*" four qualified patterns since their frequencies are no less than *min_sup* (which is set as 3).

Part 2 first uses a *for* loop to try words (except the last one) in *WordArray* one by one as the first word w of a pattern P. Note that every such a for loop corresponds to a round (except Round 1) in Example 3.1. Then Part 2 calls Part 3 to extend P (from line 2 to 8 in Fig. 3). For example, words *topic-awar, propag* and *model* are treated as the first word of a pattern in Round 2 (see figures (a)–(d) in Fig. 2), Round 3 (see figures (e)–(g) in Fig. 2) and Round 4 (see Fig. 2(h)) respectively in Example 3.1 before starting the extension jobs. The pseudo-codes from line 9 to 14 in Fig. 3 are called Round Refreshing Operation (denoted as RRO), which updates *WordArray* to let the *for* loop start over. That is, RRO is called to start a new round after finishing the current round. Specifically, RRO first finds w's next adjacent word v in *WordArray* (line 9 in Fig. 3). Then, it deletes all appearing positions of w that are smaller than the first appearing position of v. After that, it uses an *if* statement to check whether the frequency of w is less than min_*sup*. If it is, w would be deleted from *WordArray* (from line 11 to 13 in Fig. 3). Here what needs to be illustrated is that all deletion operations of RRO are valid during the whole running process of KCSP. For example, after finishing Round 2 in Example 3.1, shown in Fig. 2(a), RRO finds that the appearing position 4 of the first word in *WordArray* (i.e., *topic-awar*) is smaller than the appearing position 6 of the second word in *WordArray* (i.e., the first appearing position of *propag*), so RRO deletes it. After that, because the frequency of *topic-awar* goes down from three to two, which is smaller than *min_sup*, RRO deletes *topic-awar* from *WordArray*. Since all deletion operations of RRO are valid during the whole running process, *topic-awar* does not appear again in Round 3 and Round 4 (see figures (e)–(h) in Fig. 2). Finally, RRO sorts *WordArray* again according to the first appearing positions of words in *WordArray* (line 14 in Fig. 3). The last word in *WordArray* is always viewed as a qualified pattern (line 16 in Fig. 3), since it is no longer necessary to call Step 3 for extension when *WordArray* only has one word left.

Part 3 is called by Part 2 or itself, and starts from the ith ($i > 1$) word w in *WordArray* to search word candidates to extend a pattern P that is initially formed in Part 2 or Part 3 before. If P and w could be concatenated as a new qualified pattern P_1, Part 3 outputs P_1 and calls itself to keep extending P_1 recursively. Note that every such a recursion process corresponds to a step of a round (except Round 1) in Example 3.1. After the recursion process is done, Step Refreshing Operation (denoted as SRO) starts (from line 8 to 14 in Fig. 4). SRO first finds w's next adjacent word v in *WordArray*. Then, it deletes all appearing positions of w that are smaller than the first appearing position of v, and uses an *if* statement to check whether the frequency of w is less than *min_sup*. If it is, w would be deleted from *WordArray*. Here what needs to be said is that all deletion operations of SRO are only valid in the current round. That is, all appearing positions and intervals deleted by SRO during the current round need to be recruited back before starting a new round. For example, in Example 3.1, the appearing positions 7 and 21 of *model* are deleted before starting the third step of Round 2, so that they are not shown in the third and the fourth step of Round 2 (see figures (c) and (d) in Fig. 2). But they appear again in Round 3 (see Fig. 2(e)) and Round 4 (see Fig. 2(h)). Finally, SRO sorts *WordArray* again according to the first appearing position of words in *WordArray*.

Before introducing Part 4, there are some new notions that need to be defined as follows.

Definition 3.5. (Free Interval): Suppose there is a word w with intervals $Range[pos_j, pos_{j+1}]$ and $Range[pos_i, pos_{i+1}]$ in a paragraph database with length L ($1 \leq pos_j < pos_i < L+1$). Note that $Range[pos_i, pos_{i+1}]$ is behind of $Range[pos_j, pos_{j+1}]$. During the keyphrase candidate extension step, when interval $Range[pos_j, pos_{j+1}]$ is not allocated to an appearing position of any other words, but interval $Range[pos_i, pos_{i+1}]$ has been allocated to an appearing position of any other words, then interval $Range[pos_j, pos_{j+1}]$ is called free interval. When an interval $Range[pos_i, pos_{i+1}]$ becomes a free interval, its original upper limit pos_{i+1} is replaced by the virtual end position, which means a free interval can

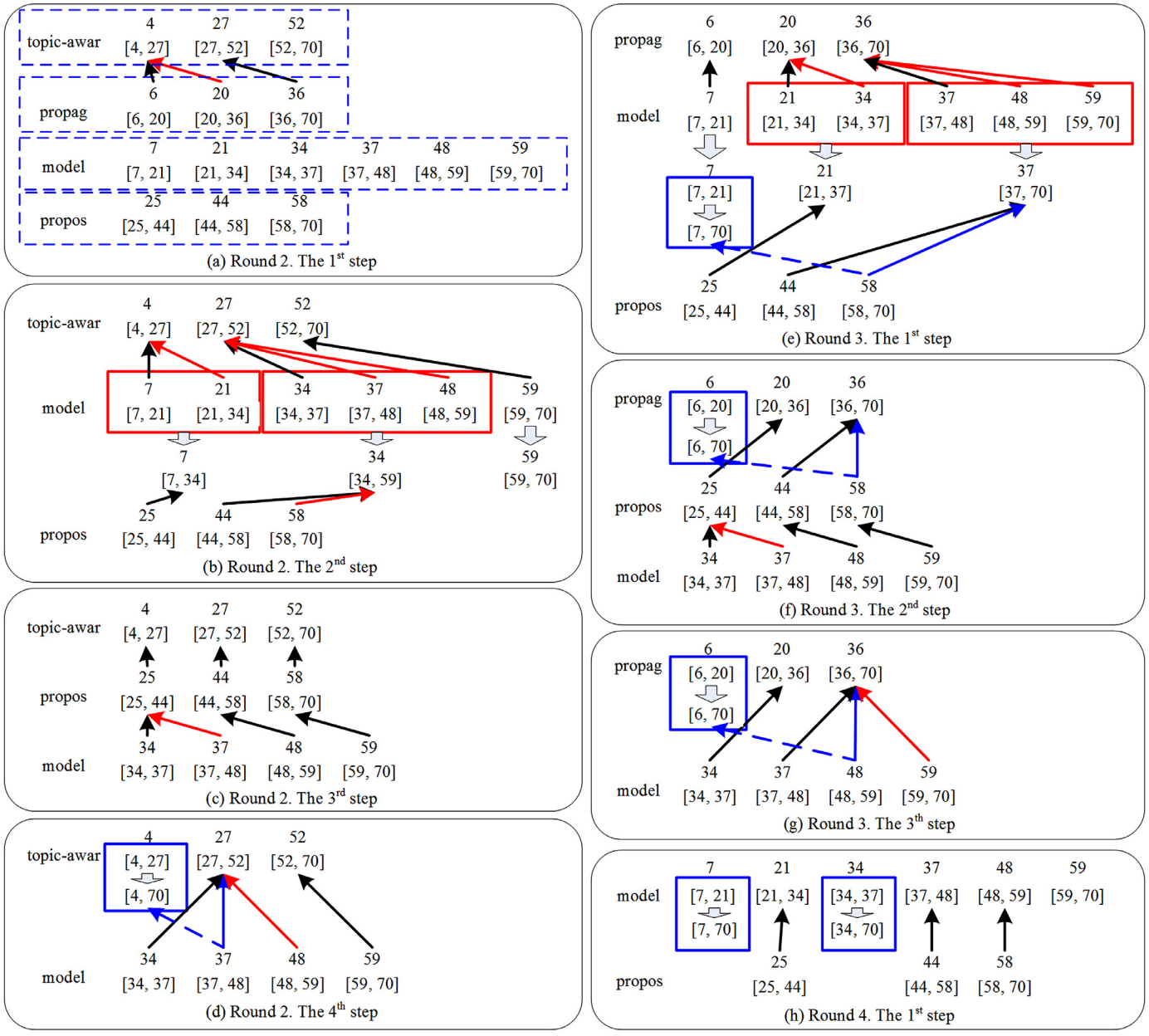

Fig. 2. Running KCSP with an example.

be allocated to appearing position from pos_i to L. If an interval $Range[pos_j, pos_{j+1}]$ has been allocated to an appearing position of any other words, it is no longer a free interval.

Consider Example 3.1 as an example, shown in Fig. 2(h), *Range*[7, 21] and *Range*[21, 34] are two of intervals of word *model* and *Range*[21, 34] is behind of *Range*[7, 21]. During the pattern extension step, *Range*[21, 34] has been allocated to the appearing position 25 of word *propos*, but *Range*[7, 21] is not allocated to any appearing position of *propos*. Therefore, *Range*[7, 21] is a free interval during this pattern extension step and its upper limit 21 is replaced by the virtual end position 70 (see left blue rectangle in Fig. 2(h)). Similarly, *Range*[34, 37] is also a free interval and its upper limit 37 is replaced by the virtual end position 70 when *Range*[37, 48] has been allocated to appearing position 44 of *propos* (see right blue rectangle in Fig. 2(h)). However, *Range*[59, 70] is not a free interval because there is no allocated interval behind.

Definition 3.6. (Qualified Position & Qualified Interval and Redundancy Position & Redundancy Interval): Suppose there is a word w_1 with intervals $Range[pos_{1i}, pos_{1(i+1)}]$ ($1 \le pos_{1i} < L+1$), and a word w_2 with n different appearing positions pos_{2j} ($1 \le j \le n < L$, $1 \le pos_j < L+1$) in a paragraph database with length L. w_2 has intervals like $Range[pos_{2j}, pos_{2(j+1)}]$. During the keyphrase candidate extension step, 1) if an appearing position pos_{2j} of w_2 has an allocated interval $Range[pos_{1i}, pos_{1(i+1)}]$ of w_1, then the appearing position pos_{2j} of w_2 is called a qualified position of interval $Range[pos_{1i}, pos_{1(i+1)}]$ and its corresponding interval $Range[pos_{2j}, pos_{2(j+1)}]$ is called a qualified interval; and 2) if any two (or more) adjacent appearing positions pos_{2j} and $pos_{2(j+1)}$ of w_2 have the same allocated interval $Range[pos_{1i}, pos_{1(i+1)}]$ of w_1, meanwhile there is no free interval in front of interval $Range[pos_{1i}, pos_{1(i+1)}]$, then the appearing position pos_{2j} of w_2 is also called an qualified position of interval $Range[pos_{1i}, pos_{1(i+1)}]$ and its corresponding interval $Range[pos_{2j}, pos_{2(j+1)}]$ is called a qualified interval too. However, the appearing

Part 2 FirstWordDetermination(*WordArray*)
Input: an ordered array *WordArray* assigned by Part 1.
Output: The set of frequent sequential patterns *FrequencyList*.
1: **for** *i*←0 to *WordArray.length*-1 **do**
2: a node *W*←*WordArray*[*i*];
3: *WordUsedGap.gapMaxList*←*W.gapMaxList*;
4: *WordUsedGap.gapMinList*←*W.gapMinList*;
5: *P*←*W*
6: *sup*(*P*)←*W. frequence*
7: *FrequencyList.add*(*P*);
8: patternRecursion(*WordArray, P,i+1, sup*(*P*), *WordUsedGap*);
9: a node *V*=*WordArray*[*i*+1];
10: delete all positions of *W* that is smaller than first position of *V*
11: **if** *W.frequence*< *min_sup* **then**
12: delete *W*;
13: **end if**
14: *WordArray.sort*;
15: **end for**
16: *FrequencyList.add*(*WordArray.lastword*);

Fig. 3. Part 2 of KCSP.

Part 3 PatternExtension(*WordArray*, *P*, *i*, *sup(P)*, *WordUsedGap*)
Input: ordered array *WordArray* assigned by Part 1;
pattern *P*;
the i^{th} word in *WordArray* that need to be calculated *i*;
pattern's frequency *sup*(*P*);
the array *WordUsedGap* includes
allocated positions of the $(i\text{-}1)^{th}$ word.
Output: qualified patterns.
1: **for** *j* ←*i* to *WordArray.length* **do**
2: a node *W*←*WordArray*[*j*];
3: *W*←IntervalCalculation(*W*, *sup*(*P*), *WordUsedGap*);
4: **if** *W* is not null **then**
5: *P*←*P* + *W*;
6: *FrequencyList.add*(*P*);
7: **patternRecursion**(*WordArray, P, j*+1, *sup*(*P*), *WordUsedGap*);
8: a node *V*←*WordArray*[*j*+1];
9: delete all positions of *W* that is smaller than first position of *V*
10: **if** *W.frequence*< *min_sup* **th**
11: delete *W*;
12: **end if**
13: **end if**
14: *WordArray.sort*;
15: **end for**

Fig. 4. Part 3 of KCSP.

position $pos_{2(j+1)}$ of w_2 is called a redundancy position of interval *Range*[pos_{1i}, $pos_{1(i+1)}$] and its corresponding interval *Range*[$pos_{2(j+1)}$, $pos_{2(j+2)}$] is called a redundancy interval. The redundancy position needs to be deleted, and intervals *Range*[pos_{2j}, $pos_{2(j+1)}$] and *Range*[$pos_{2(j+1)}$, $pos_{2(j+2)}$] need to be merged into a new interval, i.e., *Range*[pos_{2j}, $pos_{2(j+2)}$]. The new interval is still a qualified interval.

For example, during the pattern extension step in Example 3.1 (see Fig. 2(e)), the appearing position 7 of word *model* has an allocated interval *Range*[6, 20] of *propag*, so it is a qualified position, and its corresponding interval *Range*[7, 21] is a qualified interval. The appearing positions 21 and 34 have the same allocated interval *Range*[20, 36] of *propag*, since there is no free interval before *Range*[20, 36], the appearing position 21 is a qualified position

Part 4 IntervalCalculation(*W*, *sup*(*P*), *WordUsedGap*)
Input: the word candidate that need to be calculated *W*;
pattern's frequency *sup*(*P*);
the array includes allocated positions of the $(i\text{-}1)^{th}$ node.
Output: *null* or *W*.
1: *i_Tail*←0, *i_ Frequency*←0;
2: **for** *i*←0 to *W.frequence* **do**
3: a position *pos*←*W.gapMinList*[*i*];
4: **for** *j*←*i_Tail* to *WordUsedGap.length* **do**
5: *i_Min*←*WordUsedGap.gapMinList*[*j*];
6: *i_Max*←*WordUsedGap.gapMaxList*[*j*];
7: **if** *pos*>*i_Min* && *pos*<*i_Max* **then**
8: **if** this interval has not be allocated **then**
9: *i_ Frequency*++;
10: records the *pos* of *W*;
11: **else if** there is free interval before this interval **then**
12: *i_ Frequency*++;
13: records the *pos* of *W*;
14: **end If**
15: **break**;
16: **end if**
17: **end for**
18: **end for**
19: **if** *i_ Frequency* > *min_sup* **then**
20: *WordUsedGap* is revised as intervals of allocated positions of *W*;
21: **if** *i_ Frequency* < *sup(P)* **then** *Sup(P)*←*i_ Frequency*;
22: return *W*;
23: **else**
24: return *null*;

Fig. 5. Part 4 of KCSP.

and its corresponding interval *Range*[21, 34] is a qualified interval. However, the appearing position 34 is a redundancy position and its corresponding interval *Range*[34, 37] is a redundancy interval. The redundancy position 34 needs to be deleted and the intervals *Range*[21, 34] and *Range*[34, 37] need to be merged into a new qualified interval *Range*[21, 37]. Analogously, the appearing position 37 of word *model* is a qualified position, and its corresponding interval *Range*[37, 48] is a qualified interval. But the appearing positions 48 and 59 are redundancy positions that need to be deleted, and their corresponding intervals *Range*[48, 59] and *Range*[59, 70] are redundancy intervals that need to be merged with *Range*[37, 48] to generate a new qualified *Range*[37, 70].

Part 4 has a two-layer embedded *for* loop, which plays a key role in KCSP. The input parameter *WordUsedGap* stores all qualified intervals (gap constraints) of the last word *p* in a pattern *P*, which are used to verify the input word candidate *w* if *P* and *w* could be concatenated as a new qualified pattern. The outside *for* loop (line 2 in Fig. 5) is used to choose *w*'s appearing positions one by one (line 3 in Fig. 5). When a position *pos* of *w* is chosen, the inner *for* loop (line 4 in Fig. 5) tries to find an interval *Range*[*i_Min, i_Max*] in *WordUsedGap* to calculates whether *pos* could be an qualified position of *Range*[*i_Min, i_Max*]. If it could be, Part 4 records *pos* (from line 5 to 16 in Fig. 5). There are four situations between the *pos* and *Range*[*i_Min, i_Max*]: 1) *pos* is not a qualified position of *Range*[*i_Min, i_Max*]; 2) *pos* is a qualified position of *Range*[*i_Min, i_Max*], and this interval has not been allocated to another position; 3) *pos* is a qualified position of *Range*[*i_Min, i_Max*], although this interval has been allocated to another position, there is a free interval before it; and 4) *pos* is a qualified position of *Range*[*i_Min, i_Max*], but this interval has been allocated to another position and meanwhile there is no free interval before it. The situations 2) and 3) are what we expect. In other words, Part 4 is used to calculate how many appearing positions of a word candidate

w could meet intervals of the last word *p* in *P*. For example, during the calculation process in Example 3.1 (see Fig. 2(b)), Part 4 calculates how many appearing positions of *model* could meet intervals of *topic-awar*. The appearing positions 7 and 21 of *model* meet *Range*[4, 27] of *topic-awar* respectively. However, there is no free interval of *topic-awar* in front of *Range*[4, 27]. According to Definition 3.6, the appearing position 21 needs to be deleted and the intervals *Range*[7, 21] and *Range*[21, 34] of *model* need to be merged into a new interval, i.e., *Range*[7, 34] (see the left red rectangle in Fig. 2(b)). Similarly, the appearing positions 34, 37 and 48 of *model* meet *Range*[27, 52] of *topic-awar* respectively. Part 4 performs the same operations for the appearing positions 34, 37 and 48 of *model* (see the right red rectangle in Fig. 2(b)). The appearing position 59 of *model* meets *Range*[52, 70] of *topic-awar* (presented as a black solid line in Fig. 2(b)). After that, *model* has three qualified positions 7, 34 and 59 and three qualified intervals *Range*[7, 34], *Range*[34, 59] and *Range*[59, 70]. At the end, Part 4 revises *WordUsedGap* using *w*'s qualified intervals, and returns *w* if its frequency is no less than *min_sup*. Besides, *sup(P)* needs to be revised as *w*'s frequency if its frequency is less than *P*'s frequency (from line 19 to 22 in Fig. 5). Otherwise, Part 4 returns *null*.

3.3. Full rounds and steps of Example 3.1

In this section, we will introduce the full rounds and steps occurred in Example 3.1. KCSP altogether takes four rounds to finish extracting and outputting qualified patterns as follows (see Fig. 2).

Round 1. KCSP scans the given paragraph once to obtain every word whose frequency is no less than *min_sup*. Meanwhile, KCSP records the appearing positions of these words, and generates intervals for words at different appearing positions according to their appearing positions and the virtual end position. Words and their appearing positions and intervals are put into *WordArray* according to their first appearing positions from low to high (listed in blue dashed rectangles in Fig. 2(a)). Each interval denotes a gap constraint for a word at an appearing position. For example, the interval *Range*[4, 27] of *topic-awar* is the gap constraint of *topic-awar* at the appearing position 4, which means *topic-awar* at the appearing position 4 can be extended with a word at appearing positions from 5 to 26. After Round 1, KCSP outputs "*topic-awar*", "*propag*", "*model*" and "*propos*" four qualified patterns since their frequencies are no less than *min_sup* (which is set as 3).

Round 2. KCSP treats *topic-awar* (the first word in *WordArray*) as the first word of a pattern. KCSP first calculates how many appearing positions of *propag* (the next word of *topic-awar*) could meet intervals of *topic-awar*. Shown in Fig. 2(a), the appearing positions 6 and 36 of *propag* meet intervals *Range*[4, 27] and *Range*[27, 52] of *topic-awar* respectively (represented as black solid lines). Although the appearing position 20 of *propag* also meets the interval *Range*[4, 27] (represented as a red solid line), unfortunately this interval has been allocated to the appearing position 6 of *propag*, and meanwhile there is no free interval of *topic-awar* in front of *Range*[4, 27]. Therefore, the appearing position 20 of *propag* is a redundancy position and its corresponding interval *Range*[20, 36] is a redundancy interval. However, *propag* only has two qualified positions 6 and 36, which means the pattern "*topic-awar propag*" is not a qualified pattern since its frequency is smaller than *min_sup* (which is set as 3). Therefore, KCSP stops extending "*topic-awar propag*", and calls SRO to start the second step of Round 2. Because of this, it is no longer necessary to delete and merge the redundancy position and the redundancy interval of *propag* respectively.

SRO first finds all appearing positions of the second word in *WordArray* (i.e., *propag*) that are smaller than the first appearing position of the third word in *WordArray* (i.e., *model*) and deletes them. Then SRO checks whether the frequency of *propag* is smaller than *min_sup* (which is set as 3). If it is, *propag* would be deleted from *WordArray*. Shown in Fig. 2(a), the appearing position 6 of *propag* is smaller than the first appearing position 7 of *model*, so SRO deletes it. Since SRO finds that the frequency of *propag* goes down from three to two, which is smaller than *min_sup*, SRO deletes *propag* from *WordArray*. That is why *propag* is not shown again during Round 2 (see figures (b)–(d) in Fig. 2). Finally, SRO sorts words in *WordArray* again according to their first appearing positions from low to high.

The second step of Round 2 starts after finishing SRO (see Fig. 2(b)). Similar to the first step, KCSP first calculates how many appearing positions of *model* (the next word of *topic-awar*) could meet intervals of *topic-awar*. Shown in Fig. 2(b), the appearing positions 7 and 21 of *model* meet *Range*[4, 27] of *topic-awar* respectively. However, there is no free interval of *topic-awar* in front of *Range*[4, 27]. According to Definition 3.6, the appearing position 21 needs to be deleted and the intervals *Range*[7, 21] and *Range*[21, 34] of *model* need to be merged into a new interval, i.e., *Range*[7, 34] (see the left red rectangle in Fig. 2(b)). Similarly, the appearing positions 34, 37 and 48 of *model* meet *Range*[27, 52] of *topic-awar* respectively. KCSP performs the same operations for the appearing positions 34, 37 and 48 of *model* (see the right red rectangle in Fig. 2(b)). The appearing position 59 of *model* meets *Range*[52, 70] of *topic-awar* (represented as a black solid line). After that, *model* has three qualified positions 7, 34 and 59 and three qualified intervals *Range*[7, 34], *Range*[34, 59] and *Range*[59, 70]. KCSP outputs pattern "*topic-awar model*" and continues extending it with word *propos* (the next word of *model* in *WordArray*). However, *propos* only has two qualified positions 25 and 44 (two black solid lines) after finishing the pattern extension, which means pattern "*topic-awar model propos*" is not a qualified pattern since its frequency is smaller than *min_sup* (which is set as 3). Therefore, KCSP calls SRO to start the third step of Round 2. Since the deletion and merging operations for redundant positions and intervals are only valid in the current step, all redundant positions deleted and redundant intervals merged during the current step need to be recruited back before KCSP calls SRO. That is why the appearing positions 37 and 48 of *model* are shown again during the third and fourth step of Round 2 (see Fig. 2(c) and (d)). However, the appearing positions 7 and 21 of *model* are not shown again because of SRO.

Similar to the second step, the third step starts after finishing SRO (see Fig. 2(c)). After this step, KCSP outputs patterns "*topic-awar propos*" and "*topic-awar propos model*" since their frequencies equal to *min_sup*.

Similar to the second step, the fourth step starts after finishing SRO (see Fig. 2(d)). The appearing position 34 of *model* meets the interval *Range*[27, 52] of *topic-awar* (represented as a black solid line). Note that the interval *Range*[4, 27] of *topic-awar* is not allocated to an appearing position of *model* while the interval *Range*[27, 52] of *topic-awar* has been allocated to the appearing position 34 of *model*. Therefore, the interval *Range*[4, 27] becomes a free interval and its upper limit 27 is replaced by the virtual end position 70 (represented as a blue solid rectangle). This operation is used to ensure that there is no missing occurrence of the pattern by increasing the number of wildcards between two words (i.e., *topic-awar* and *model*). Note that the replacements of the upper limit for free intervals are only valid in the current step. The appearing position 37 of *model* also meets *Range*[27, 52] (represented as a blue solid line). Although *Range*[27, 52] has been allocated to the appearing position 34, KCSP finds a free interval *Range*[4, 70] of *topic-awar* in front of *Range*[27, 52] and allocates it to the appearing position 37 of *model* (represented as a blue dashed line). After this reallocation, *Range*[4, 70] is no longer a free interval according to Definition 3.5. The appearing position 48 of *model* meets *Range*[27, 52] too (represented as a red solid line), but there is no free interval of *topic-awar* in front of *Range*[27, 52], so that KCSP treats the appearing position 48 of *model* as a redundancy

position (represented as a red solid line). Finally, KCSP outputs pattern "*topic-awar model*" since its frequency is three (two black solid lines and one blue dashed line), which equals *min_sup*.

After the above four steps, KCSP finishes Round 2 since *model* is the last word in *WordArray*, and calls RRO to start Round 3. Note that all appearing positions and intervals deleted by SRO during the current round need to be recruited back before KCSP calls RRO, since deletion operations of SRO are only valid in the current round. RRO first finds all the appearing positions of the first word in *WordArray* (i.e., *topic-awar*) that are smaller than the first appearing position of the second word in *WordArray* (i.e., *propag*) and deletes them. Then RRO checks whether the frequency of *topic-awar* is smaller than *min_sup*. If it is, *topic-awar* would be deleted from *WordArray*. Shown in Fig. 2(a), the appearing position 4 of *topic-awar* is smaller than the appearing position 6 of *propag*, so RRO deletes it. Furthermore, RRO finds that the frequency of *topic-awar* goes down from three to two, which is smaller than *min_sup* (which is set as 3), RRO deletes *topic-awar* from *WordArray*. Since deletion operations of RRO are valid during the whole running process of KCSP, *topic-awar* is not shown again in Round 3 and Round 4 (see figures (e), (f), (g) and (h) in Fig. 2). Finally, RRO sorts words in *WordArray* again according to their first appearing positions from low to high.

Round 3. Similar to Round 2, KCSP starts the first step after finishing RRO (see Fig. 2(e)), and then starts the second and third steps after finishing SRO respectively (see Fig. 2(f) and (g)). KCSP outputs qualified patterns "*propag model*", "*propag model propos*" during the first step, qualified patterns "*propag propos*", "*propag propos model*" during the second step, and a qualified pattern "*propag model*" during the third step.

Round 4. Similar to Round 2, KCSP starts the first step after finishing RRO (see Fig. 2(h)), and outputs qualified pattern "*model propos*" at last.

After finishing running the above four rounds, we find that KCSP produces duplication patterns during the pattern search process. Although duplication patterns can be filtered out, it is obvious that duplication production and elimination waste computation time and memory space. In the next subsection, we will analyze the reason of duplication production and look for a suitable solution to completely eradicate duplication production.

3.4. Eradicating duplication production

Note that both SRO and RRO are run several times in the above explanation in Section 3.3 because they are used to ensure that the array *WordArray* is valid and ordered. However, if two words in *WordArray* have lots of appearing positions respectively, which let them be able to withstand after more than two SROs or RROs, duplication patterns may be produced. For example, v and *w* are two words with many appearing positions in *WordArray*, and *w* is in front of *v*. A pattern *wv* may be generated before the first SRO/RRO. After finishing the first SRO/RRO, *v* and *w* are still in *WordArray* and *v* has been moved to the front of *w*. Then, a pattern *vw* may be generated. After finishing the second SRO/RRO, if both *v* and *w* are still in *WordArray*, it is obvious that *w* will be moved to the front of *v* again and the pattern *wv* would be generated again. Therefore, duplication appears. If both *w* and *v* have lots of appearing positions that let them be able to withstand with SROs/RROs many times, more duplication patterns are produced. Here we completely eradicate duplication production by updating the Part 2 and the Part 3 of KCSP.

First, a boolean variable *usedflag* is added for each word *w* in *WordArray* to record whether *w* has been treated as a word of a pattern *P* during the *P*'s extension step, which is initialized as 0. When *w* is treated as a word of a pattern *P*, its *usedflag* is set to 1.

Updated Part 2 FirstWordDetermination(*WordArray*)

Input: ordered array *WordArray* assigned by Part 1.
Output: The set of frequent sequential patterns *FrequencyList*.

```
1:  for i←0 to WordArray.length-1 do
2:      a node W=WordArray[i];
3:      WordUsedGap.gapMaxList←W.gapMaxList;
4:      WordUsedGap.gapMinList←W.gapMinList;
5:      P←W
6:      sup(P)←W. frequence
7:      FrequencyList.add(P);
8:      WordArray[i].usedflag=1;
9:      patternRecursion(WordArray, P,i+1, sup(P), WordUsedGap);
10:     for j←i+1 to WordArray.length-1do
11:       if WordArray[j].Usedflag==0 then
12:           a node V=WordArray[j];
13:           break;
14:       end if
15:     end for
16:     if V is null then break;
17:     delete positions of words in front of V
        that are smaller than first position of V
18:     if (any word in front of V).frequence< min_sup then
19:       delete this word;
20:     end if
21:     WordArray.sort;
22: end for
23: FrequencyList.add(WordArray.lastword);
```

Fig. 6. Updated Part 2 of KCSP.

This modification is shown in the updated Parts 2 and 3 of KCSP, illustrated in Fig. 6 and Fig. 7 respectively.

Second, the RRO in Part 2 and the SRO in Part 3 of KCSP are modified. The new SRO and RRO no longer simply find the *w*'s next adjacent word from *WordArray*. Instead, it uses a *for* loop to find the first word *v* with *usedflag* = 0 after *w* in *WordArray* (from line 10 to 15 in Fig. 6 and from line 11 to 16 in Fig. 7). When the new SRO and RRO finds the first word *v* with *usedflag* = 0 after *w*, it deletes all appearing positions of words in front of *v* that are smaller than the first appearing position of *v*. After that, the new SRO/RRO uses an *if* statement to check whether the words in front of *v* whose frequencies are smaller than *min_sup*. If there exists such a word, it would be deleted from *WordArray* (from line 18 to 20 in Fig. 6 and from line 19 to 21 in Fig. 7).

3.5. Discussions on KCSP

3.5.1. The time and space complexity of KCSP

We use *L* to denote the length of a paragraph, *F* to denote the number of frequent words, i.e., the length of *WordArray* ($F \leq L$), *m* to denote the average length of patterns, *h* to denote the average number of intervals that *WordUsedGap* stores, and *l* to denote the average number of appearing positions that a word candidate *W* has. Since Part 1 needs to scan the whole document once to extract frequent words, its time complexity is $O(L)$. Part 2 uses a *for* loop to try the frequent words one by one as the first word of a pattern, and uses another *for* loop for execution of round refreshing operation, so the maximal time complexity of Part 2 is $O(F^2)$. Part 3 starts from the *i*th frequent word of *WordArray* to get a word candidate *w* one by one to extend a pattern *P*. If *P* and *w* could be formed as a new qualified pattern P_1, Part 3 is called recursively to keep extending P_1. In addition, Part 3 uses another *for* loop for execution of step refreshing operation. Thus,

Updated Part 3 PatternExtension(*WordArray*, *P*, *i*, *sup(P)*, *WordUsedGap*)

Input: ordered array *WordArray* assigned by Part 1;
pattern *P*;
the i^{th} word in *WordArray* that need to be calculated *i*;
pattern's frequency *sup*(*P*);
the array including allocated positions of the $(i\text{-}1)^{th}$ word.
Output: qualified patterns.

```
1:  for i ← 0 to WordArray.length do
2:    a node W←WordArray[i];
3:    W←IntervalCalculation(W, sup(P), WordUsedGap);
4:    if W is not null then
5:      if W.usedflag==0 then
6:        P←P + W;
7:        FrequencyList.add(P);
8:        WordArray[i].usedflag=1;
9:        patternRecursion(WordArray, P, i, sup(P), WordUsedGap);
10:     end if
11:     for j←i+1 to WordArray.length-1do
12:       if WordArray[j].usedflag==0 then
13:         a node V=WordArray[j];
14:         break;
15:       end if
16:     end for
17:     if V is null then break;
18:     delete positions of words in front of V
        that are smaller than first position of V
19:     if (any word in front of V).frequence< min_sup then
20:       delete this word;
21:     end if
22:   end if
23:   WordArray.sort;
24: end for
```

Fig. 7. Updated Part 3 of KCSP.

the maximal time complexity of Part 3 is $O(mF^2)$. Since Part 4 has a two layer embedded *for* loop for calculation, its time complexity is $O(lh)$. Hence, the time complexity of KCSP is $O(L+F^2+mlhF^2)$. During the keyphrase candidate search process, KCSP needs $O(F)$ space to store frequent words, and $O(m)$ space to maintain the stack for each pattern. Supposed there are n patterns extracted, the space complexity of KCSP is $O(F+mn)$.

3.5.2. The advantages of KCSP

Existing sequential pattern mining based approaches treat gap constraints as external parameters that require users to explicitly specify them beforehand. However, manually specifying gap constraints is a time-consuming job and meanwhile the specified gap constraints usually are generalized and inappropriate. KCSP turns gap constraints into the inherent property of words at different appearing positions by utilizing a conversion operation. That is, KCSP treats the interval $Range[pos_i, pos_{i+1}]$ of a word w at the appearing position pos_i, which is generated by pos_i and the next adjacent appearing position (i.e., pos_{i+1}) of w, as the gap constraint between w at the appearing position pos_i and w at the appearing position pos_{i+1}. Such a conversion operation leads to four advantages. 1) It overcomes the weakness of requiring users to explicitly specify appropriate gap constraints beforehand, since the gap constraints of words at different appearing positions in a document can be specified automatically and appropriately by words' own appearing positions. 2) Automatically specifying gap constraint obviously reduces the setup time and human labor. 3) For each word at each appearing position, there is a gap constraint specified, which greatly improves the accuracy and appropriateness and significantly reduces the computation time for searching keyphrase candidates. And 4) KCSP can replace repeatedly scanning a document (which is another weakness of existing sequential pattern mining) with a series of calculation based on words' appearing positions and intervals to search keyphrase candidates. Intuitively, calculation significantly reduces computation time.

3.5.3. The characteristics of the extracted patterns

The pattern set (i.e., the keyphrase candidate set) of a document extracted by KCSP includes many sub-patterns which have the same supports with their parent-patterns. Many studies (Fumarola et al., 2016; Loglisci & Malerba, 2009; Xie et al., 2014) pinpointed the idea that mining only the set of closed sequential patterns may help avoid the generation of unnecessary subsequences, leading to more compact results and saving computational time and space costs. However, it does not work well on keyphrase extraction because they ignored the inherent properties of a pattern to capture a point, especially uncertainty denoting the expression depth of the point expressed by the pattern (which will be introduced in Section 4 later). That is, we cannot absolutely determine that the points expressed by the closed patterns are more useful than the points expressed by the other patterns for the document. Consider the example in Fig. 1, although "*social influenc*" and "*social influenc model*" are two keyphrase candidates having the same support, they express different points in the document. Therefore, we extract all available patterns and employ entropy for calculating and ranking them.

4. Mechanism for keyphrase candidate ranking

In this section, we will introduce the second component of KeyRank—the pattern ranking evaluation PF-H. TF-IDF is recognized as the most important invention in information retrieval, and its functionalities in information retrieval were deeply discussed by Salton and McGil (1986). Many mechanisms based on TF-IDF have been shown to work well in practice despite of its simplicity (Liu et al., 2009; Xia et al., 2015). Since in this paper keyphrase candidates in a document are treated as independent patterns, we use patterns instead of terms in TF-IDF, and obtain an intermediate mechanism pattern frequency–inverse document frequency (called PF-IDF) as follows:

$$PF-IDF(p) = (n_p/N_p) \times log(D/D_p) \quad (1)$$

where p denotes a pattern, n_p denotes the number of times p occurs in a document d, N_p denotes the number of patterns d has, D is the number of document in the corpus, and D_p denotes how many documents in the corpus contain the pattern p. Note that the pattern p in the above mechanism PF-IDF is different from the ones discussed by Yang and Yu (2007) and Zanmatkesh and Hassanpour (2011, 2012), since it can be either a contiguous or a non-contiguous sequence without length limitation. As we know, the purpose of IDF is to measure the probability that a given document d contains a specific pattern (term) as the relative document frequency. IDF is not quite suitable for ranking patterns since it ignores the three inherent properties of a pattern to capture a point: meaningfulness, uncertainty and uselessness (which will be introduced in the next paragraph). In this paper, we use entropy (Shannon, 1948) to measure the three inherent properties of a pattern to capture a point, and propose a new evaluation mechanism called pattern frequency with entropy (PF-H in short) as follows:

$$PF-H(p) = (n_p/N_p) \times \log(H(p)) \quad (2)$$

where $H(p)$ denotes the entropy of the pattern p. Intuitively, the greater the entropy of a pattern, the lower probability the pattern captures a point of the corresponding document. The entropy of a pattern depends on its three probabilities in the corpus: 1) the probability as an independent form; 2) the probability as a subform; and 3) the probability of other situations (i.e., neither 1) nor 2)). The three probabilities of a pattern are statistically obtained

from the corpus, instead of a single document, for stability and accuracy (Cheng et al., 2014).

The meaningfulness of a pattern to capture a point denotes the usefulness and the accuracy of the point expressed by the pattern. It is measured by the probability of a pattern p as an independent form (denoted by P_i), which is calculated by Eq. (3) below. The more times p occurs in the corpus, the more useful and accurate the point expressed by p. Consider the example in Fig. 1 as an example, the maximum occurrence time of pattern "*topic model*" means the point expressed by "*topic model*" is more useful and accurate to the document than other patterns. A pattern with an independent form has four kinds of situations: 1) it only has parent pattern(s); 2) it only has sub-pattern(s); 3) it has sub-pattern(s) and parent pattern(s); and 4) it does not have sub-pattern(s) and parent pattern(s).

$$P_i = \begin{cases} \#NIC/\#TN, & 0 < \#NIC < \#TN \\ 0, & \#NIC = 0 \end{cases} \tag{3}$$

where #NIC denotes the number times of a pattern p as an independent form appears in the corpus, and #TN denotes the total number of patterns the corpus has. $P_i = 0$ for the case that p does not exist in the corpus. This only exists theoretically since it is meaningless and unrealistic to calculate P_i for p that does not exist in the corpus.

The uncertainty of a pattern to capture a point denotes the expression depth of the point expressed by the pattern. It is measured by the probability of a pattern p as a sub-form (denoted by P_s), which is calculated by Eq. (4) below. A pattern with sub-form means it can be extended into another pattern(s) with other word(s). The more times p as a sub-form occurs in the corpus, the more unsteady the point expressed by p, because 1) different patterns express a same point with a different expression depth; and 2) different patterns express totally different points. Looking at the example in Fig. 1 again, pattern "*social influenc*" is a sub-pattern of pattern "*social influenc model*", but they express different points in the document.

$$P_s = \begin{cases} \#NSC/\#TN, & 0 < \#NSC < \#TN \\ 0, & \#NSC = 0 \end{cases} \tag{4}$$

where *#NSC* denotes the number times of the pattern p as a sub-form appears in the corpus. In contrast to the situation where $P_i = 0$ in Eq. (3), the situation $P_s = 0$ in Eq. (4) does exist not only in theory but also in reality. There are two kinds of forms where $P_s = 0$ occurs: 1) p is a super-pattern (a pattern that does not have parent patterns); and 2) p does not have sub-pattern(s).

The probability of other situations (denoted by P_o) is defined as the uselessness of p capturing a point, which is calculated by Eq. (5) below. For example, since the patterns appearing in the corpus are extracted from many documents, a point expressed by p being useful to document D_a but not to document D_b needs to be taken into account.

$$P_o = 1 - P_i - P_s \tag{5}$$

In summary, the entropy of a pattern completely evaluates the meaningfulness, the uncertainty, and the uselessness of the pattern in the corpus, which is defined by Eq. (6), when the situation $P_s = 0$ occurs, $H(p)$ can be calculated by Eq. (7).

$$H(p) = P_i \log(1/P_i) + P_s \log(1/P_s) + P_o \log(1/P_o) \tag{6}$$

$$H(p) = P_i \log(1/P_i) + P_o \log(1/P_o) \tag{7}$$

5. Experiments

5.1. Experiments for pattern search

In order to investigate the computational efficiency of KCSP, we conduct experiments on a dataset INSPEC (Hulth, 2013), which contains 2000 abstracts (1000 for training, 500 for development and 500 for testing). In order to investigate the effectiveness of eradicating duplication production (i.e., the modified Part 2 and Part3 in Figs. 6 and 7 respectively), we compare KCSP with its previous version KCSP0, which does not eradicate duplication production and uses the previous Part2 and Part3 shown in Figs. 3 and 4 instead. As we mentioned before, SPMW (Xie et al., 2014) is an effective keyphrase candidate search algorithm using sequential pattern mining with gap constraints. In the experiments, we compare KCSP and KCSP0 with SPMW only in terms of the computation time of patterns extracted, because these three algorithms always extract the same pattern set for each document. The performance of these three algorithms is evaluated on the entire 2000 abstracts of the dataset INSPEC.

Note that SPMW needs to set up gap constraint before running. To make a fair and comprehensive comparison, we follow the gap constraint and the support threshold settings that SPMW used. That is, the minimal gap size is 0, the maximal gap size is the length of a paragraph, and the minimal support threshold is set to 3. Besides, during the pattern search process we run the three algorithms (SPMW, KCSP, and KCSP0) respectively on each document nine times, and the average computation time of each algorithm on each document is reported. In order to present the computation time comparisons clearly, we divided 2000 documents into different categories in terms of the computation time of SPMW, i.e., category 0–2, category 2–99, category 100–999, category 1000–9999, category 10,000–99,999, and category 100,000-. For example, the category 0–2 contains all documents on which SPMW uses 0 to 2 milliseconds to extract their pattern sets. Our experimental results are shown in Fig. 8. The computation time of the three algorithms (SPWM, KCSP, and KCSP0) in category 0–2 is very close, and the computation time of the three algorithms in category 100,000- varies widely, so the time comparisons of these two categories are not shown in Fig. 8. In order to show the computation time comparisons well for other four categories, we first sort the computation time of SPMW on documents within each category in an incremental order. That is why we can see that the computation time of SPMW increases smoothly in Fig. 8. Based on the sorting of SPMW, we show the corresponding computation time of KCSP and KCSP0 on each document in each category.

Shown in Fig. 8, both KCSP0 and KCSP are much more efficient than SPMW. The reasons are: 1) SPMW scans a document multiple times for searching patterns, which causes a lot of time overhead; and 2) SPMW requires users to explicitly specify appropriate gap constraint(s) beforehand. In reality it is time-consuming, which leads to users usually to provide a general gap constraint instead. However, this kind of operation further increases the time for scanning the document. By contrast, both KCSP0 and KCSP only scan a document once, and employ a series of calculations instead of repeatedly scanning a document to reduce the computation time for searching patterns. In addition, both KCSP0 and KCSP automatically generate appropriate gap constraints for words at different appearing positions, which further reduces the computation time. There are a few documents where KCSP0 uses a little more time than SPMW does (see Fig. 8(a) and (b)). This is because KCSP0 produces duplication patterns during pattern search, which we have analyzed in Section 3. KCSP is more efficient than KCSP0 because KCSP0 produces duplication patterns during the pattern search process. There are also a few documents where KCSP uses a little more time than KCSP0 does (see the left corner in Fig. 8(a)).

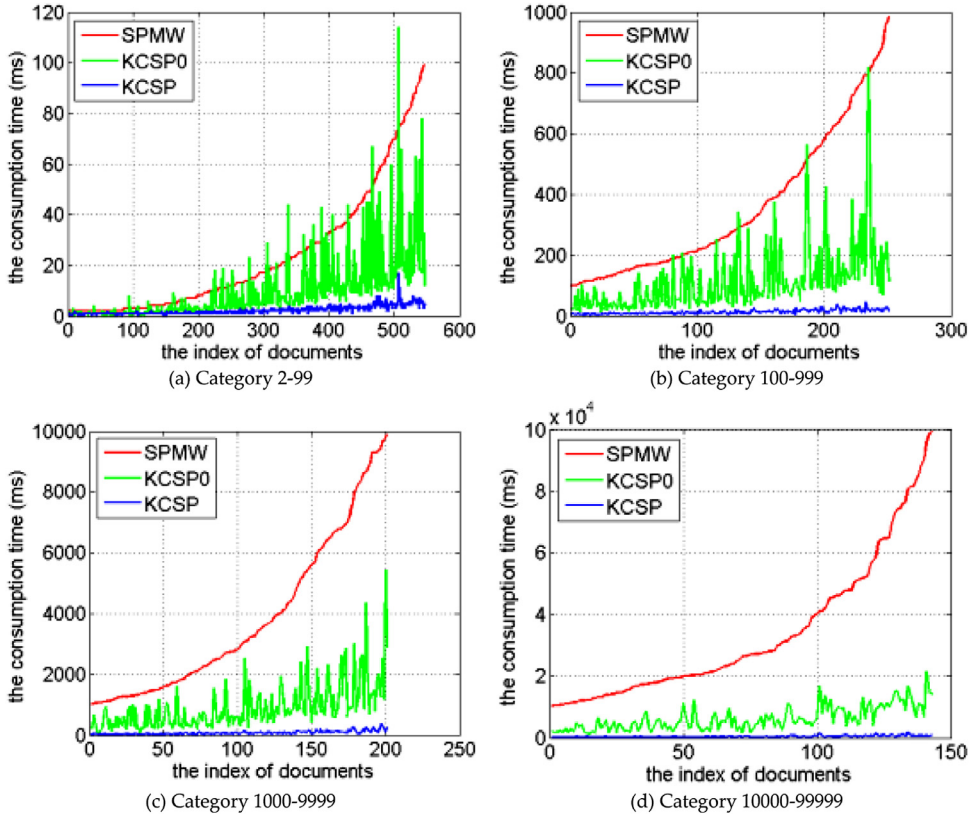

Fig. 8. The time comparisons between SPMW, KCSP, and KCSP0 in (a) category 2–99, (b) category 100–999, (c) category 1000–9999, and (d) category 10,000–99,999.

This is because KCSP uses additional *if* statement and *for* loop, which needs additional consumption time to eradicate duplication production. This phenomenon only appears when 1) there is no duplication produced; or 2) too little duplication leads to KCSP consuming more additional time than duplication eradication does.

5.2. Experiments for pattern ranking

To investigate the performance of KeyRank, we conduct experiments on two datasets SemEval-2010 (Kumar & Srinathan, 2008) and INSPEC (Hulth, 2013). The dataset SemEval-2010 contains 244 articles (144 for training and 100 for testing), and the dataset INSPEC contains 2000 abstracts (1000 for training, 500 for development and 500 for testing).

In order to evaluate the performance of our approach KeyRank, we compare it with a popular approach TextRank (Mihalcea & Tarau, 2004), since they both are unsupervised approaches and treat the keyphrase selection as a ranking task. In addition, to investigate the performance of our evaluation mechanism PF-H, we compare KeyRank with a supervised approach KeyEx (Xie et al., 2014) which formulates the keyphrase selection as a classification task, since they both treat keyphrase candidate search as a sequential pattern mining task, and always extract the same pattern set for each document. Furthermore, we replace the ranking mechanism of TextRank with our PF-H to investigate whether PF-H can improve the performance of TextRank, and we will say TextRank-A as a way to refer to TextRank. Note that TextRank-A always extracts the same pattern set as TextRank does for each document. Besides, precision (P in short), recall (R in short), and the F_1 score are used as the performance metrics of keyphrase extraction, which are defined as follows.

$$P = \#correct/\#extracted \tag{8}$$

$$R = \#correct/\#labeled \tag{9}$$

$$F_1 = 2 \times P \times R/(P+R) \tag{10}$$

where *#correct* denotes the number of correctly extracted keyphrases, *#extracted* denotes the number of extracted keyphrases, and #labeled denotes the number of labeled keyphrases.

The experimental results on the dataset SemEval-2010 in terms of P, R, and F_1 scores are shown in Fig. 9(a)–(c). Note that KeyEx is a supervised learning based approach. Its performance is evaluated on the 100 test documents. However, KeyRank, TextRank and TextRank-A are unsupervised learning based approaches. To make fair comparisons, the performance of KeyRank, TextRank and TextRank-A is also evaluated on the 100 test documents. The three figures show that TextRank-A always performs better than TextRank, which shows the effectiveness of PF-H. KeyRank performs significantly better than TextRank-A and KeyEx. When the number of extracted keyphrases is small (from 3 to 15), TextRank-A performs significantly better than KeyEx. However, when the number of extracted keyphrases increases, the advantage of TextRank-A diminishes.

In addition, we also notice that the precisions of the four approaches decrease when the number of extracted keyphrases varies from 3 to 25 (see Fig. 9(a)), and the recalls of the four approaches increase as the number of extracted keyphrases varies from 3 to 25 (see Fig. 9(b)). This is because with the increment of the number of extracted keyphrases, more correct keyphrases are extracted, since the number of labeled keyphrases is fixed, recall increases with the increment of the number of extracted keyphrases. However, the growth rate of the number of correctly extracted keyphrases is less than the growth rate of the number of extracted keyphrases, so that precision drops. Because of the opposite performance in terms

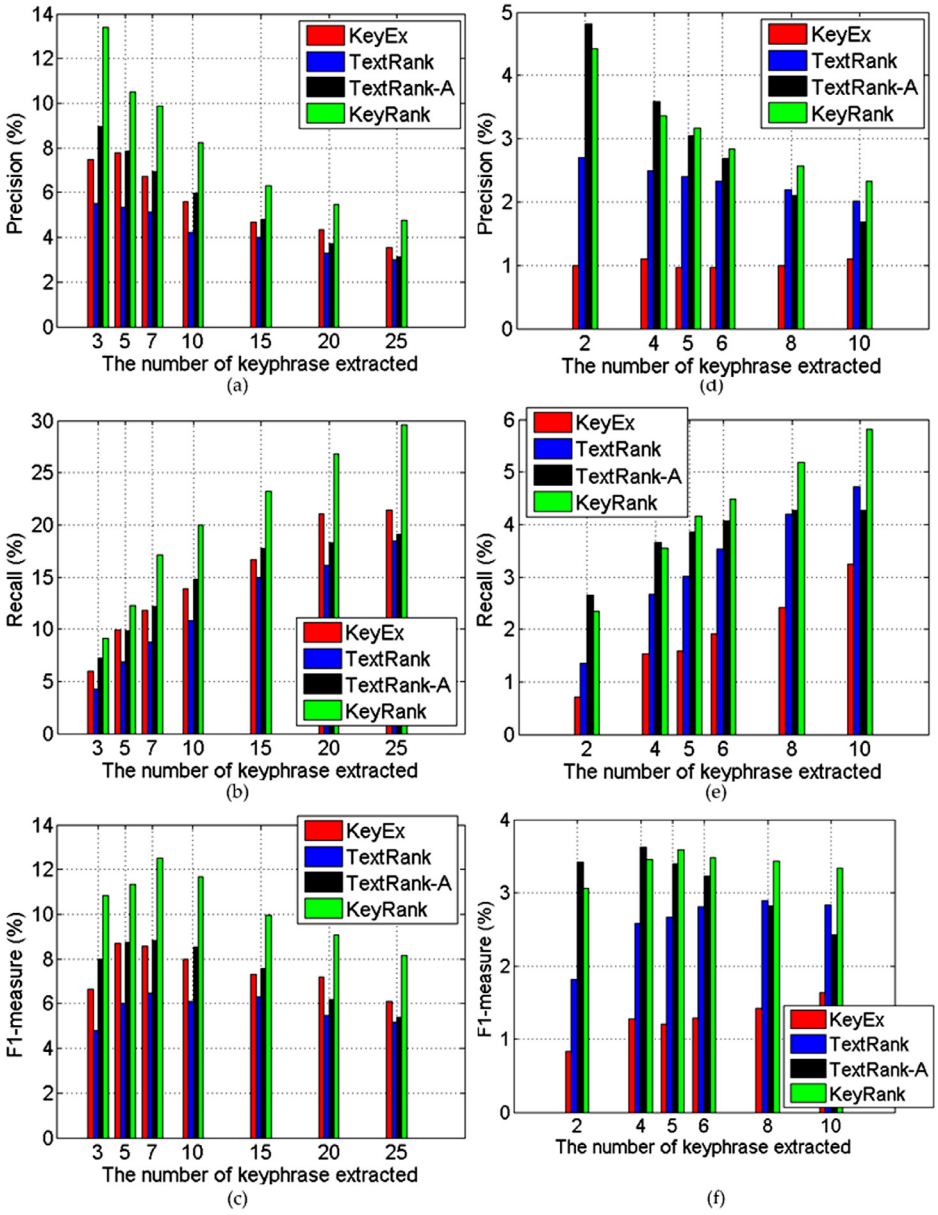

Fig. 9. The (a) precisions, (b) recalls, and (c) F_1 scores of KeyRank, KeyEx, TextRank, and TextRank-A on SemEval-2010; The (d) precisions, (e) recalls, and (f) F_1 scores of KeyRank, KeyEx, TextRank, and TextRank-A on INSPEC.

of precisions and recalls, the F_1 scores of the four approaches increase when the number of extracted keyphrases is small (see Fig. 9(c)). After the number of extracted keyphrases increases to a certain number, their F_1 scores start to drop gradually.

The experimental results on the dataset INSPEC in terms of precision, recall and F_1 score are shown in Fig. 9(d)–(f). Since KeyEx is a supervised learning based approach, its performance is evaluated on the 500 test documents. To make fair comparisons, the performance of KeyRank, TextRank and TextRank-A is also evaluated on the same 500 test documents. Besides, the number of extracted keyphrases in this experiment varies from 2 to 10 since the average length of these documents is short. Shown in these three figures, KeyRank always performs significantly better than KeyEx and TextRank, and TextRank-A performs better than TextRank when the number of extracted keyphrases is smaller than 8, which also shows the effectiveness of PF-H, although the advantage of TextRank-A diminishes when the number of extracted keyphrases is greater than 8.

6. Conclusion

In this paper, we proposed a new approach KeyRank for keyphrases extraction. It contains two main components: KCSP and PF-H. KCSP is a document-specific keyphrase candidate search algorithm using sequential pattern mining with gap constraints. PF-H is an evaluation mechanism using entropy to rank keyphrase candidates. The experimental results show that KeyRank performs best. Its first component KCSP is much more efficient than a closely related approach SPMW, and its second component PF-H is an effective evaluation mechanism for ranking patterns.

After having consulted linguists again, they also confirmed that words do not repeatedly appear in an effective keyphrase in Chinese. In the future work we will conduct corresponding experiments to evaluate the performance of our approach KeyRank for extracting proper keyphrases from documents in Chinese.

Acknowledgments

This research has been supported by the National Key Research and Development Program of China 2016YFB1000901, the Program for Changjiang Scholars and Innovative Research Team in University (PCSIRT) of the Ministry of Education, China IRT17R32, the National Natural Science Foundation of China 61728205, 91746209, 61673152 and 61503116, and the US National Science Foundation IIS-1115417 and IIS-1613950.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th international conference on very large data bases* (pp. 487–499).

Agrawal, R., & Srikant, R. (1995). Mining sequential patterns. In *Proceedings of the eleventh international conference on data engineering* (pp. 3–14).

Ceci, M., Appice, A., & Malerba, D. (2008). Emerging pattern based classification in relational data mining. In *Proceedings of the19th international conference on database and expert systems applications*. doi:10.1007/978-3-540-85654-2_28.

Ceci, M., Loglisci, C., & Macchia, L. (2014). Ranking sentences for keyphrase extraction: A relational data mining approach. In *Proceedings of the 10th Italian research conference on digital libraries* (pp. 52–59). doi:10.1016/j.procs.2014.10.011.

Chen, K., Luesukprasert, L., & Chou, S. T. (2007). Hot topic extraction based on timeline analysis and multi-dimensional sentence modeling. *IEEE Transactions on Knowledge and Data Engineering, 19*(8), 1016–1025. http://dx.doi.org/10.1109/TKDE.2007.1040.

Cheng, X., Yan, X., Lan, Y., & Guo, J. (2014). BTM: Topic modeling over short texts. *IEEE Transactions on Knowledge and Data Engineering, 26*(12), 2928–2941. http://dx.doi.org/10.1109/TKDE.2014.2313872.

DeWilde, B. (2014). *Intro to automatic keyphrase extraction* http://bdewilde.github.io/blog/2014/09/23/intro-to-automatic-keyphrase-extraction.

Ding, B., Lo, D., & Khoo, S. (2009). Efficient mining of closed repetitive gapped subsequences from a sequence database. In *Proceedings of IEEE 25th international conference on data engineering* http://dx.doi.org/10.1109/ICDE.2009.104.

Dorr, D. H., & Denton, A. M. (2009). Establishing relationships among patterns in stock market data. *Data Knowledge Engineering, 68*(3), 318–337. http://dx.doi.org/10.1016/j.datak.2008.10.001.

Ercan, G., & Cicekli, I. (2007). Using lexical chains for keyword extraction. *Information Processing and Management, 43*(6), 1705–1714. doi:10.1016/j.ipm.2007.01.015.

Exarchos, T., Tsipouras, M., Papaloukas, C., & Fotiadis, D. (2008). A two-stage methodology for sequence classification based on sequential pattern mining and optimization. *Data Knowledge Engineering, 66*(3), 467–487. http://dx.doi.org/10.1016/j.datak.2008.05.007.

Feng, J., Xie, F., Hu, X., Li, P., Cao, J., & Wu, X. (2011). Keyword extraction based on sequential pattern mining. In *Proceedings of the 3rd international conference on internet multimedia computing and service* (pp. 34–38).

Fu, Z., Huang, F., Sun, X., Vasilakos, A. V., & Yang, C. (2016). Enabling semantic search based on conceptual graphs over encrypted outsourced data. *IEEE Transactions on Services Computing, pp*(99) 1-1. doi:10.1109/TSC.2016.2622697.

Fu, Z., Ren, K., Shu, J., Sun, X., & Huang, F. (2016). Enabling personalized search over encrypted outsourced data with efficiency improvement. *IEEE Transactions on Parallel and Distributed Systems, 27*(9), 2546–2559.

Fu, Z., Sun, X., Liu, Q., Zhou, L., & Shu, J. (2015). Achieving efficient cloud search services: Multi-keyword ranked search over encrypted cloud data supporting parallel computing. *IEICE Transactions on Communications, E98*(B(1)), 190–200.

Fu, Z., Wu, X., Guan, C., Sun, X., & Ren, K. (2016). Toward efficient multi-keyword fuzzy search over encrypted outsourced data with accuracy improvement. *IEEE Transactions on Information Forensics and Security, 11*(12), 270602716.

Fumarola, F., Lanotte, P. F., Ceci, M., & Malerba, D. (2016). CloFAST: Closed sequential pattern mining using sparse and vertical id-lists. *Knowledge & Information Systems, 48*(2), 429–463.

Grineva, M., Grinev, M., & Lizorkin, D. (2009). Extracting key terms from noisy and multi-theme documents. In *Proceedings of the 18th international conference on World Wide Web* (pp. 661–670).

Haddoudand, M., & Abdeddaim, S. (2014). Accurate keyphrase extraction by discriminating overlapping phrases. *Journal of Information Science, 40*(4), 1–13. http://dx.doi.org/10.1177/0165551514530210.

Hasan, K. S., & Ng, V. (2014). Automatic keyphrase extraction: A survey of the state of the art. In *Proceedings of the 52nd annual meeting of the association for computational linguistics* (pp. 1262–1273).

Hassanpour, H., & Zahmatkesh, F. (2012). An adaptive meta-search engine considering the user's field of interest. *Journal of King Saud University-Computer and Information Sciences, 24*(1), 71–81.

Hulth, A. https://github.com/snkim/AutomaticKeyphraseExtraction.

Jiang, X., Hu, Y., & Li, H. (2009). A ranking approach to keyphrase extraction. In *Proceedings of the 32nd international ACM SIGIR conference on research and development in information retrieval* (pp. 756–757).

Kumar, N., & Srinathan, K. (2008). Automatic keyphrase extraction from scientific documents using *N*-gram filtration technique. In *Proceedings of the 8th ACM symposium on document engineering* (pp. 199–208).

Li, J., Li, X., Yang, B., & Sun, X. (2015). Segmentation-based image copy-move forgery detection scheme. *IEEE Transactions on Information Forensics and Security, 10*(3), 507–518.

Liu, F., Pennell, D., Liu, F., & Liu, Y. (2009). Unsupervised approaches for automatic keyword extraction using meeting transcripts. In *Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics* (pp. 620–628).

Liu, X., Song, Y., Liu, S., & Wang, H. (2012). Automatic taxonomy construction from keywords. In *Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 1433–1441).

Liu, Z., Huang, W., Zheng, Y., & Sun, M. (2010). Automatic keyphrase extraction via topic decomposition. In *Proceedings of the 2010 conference on empirical methods in natural language processing* (pp. 366–376).

Loglisci, C., & Malerba, D. (2009). Mining multiple level non-redundant association rules through two-fold pruning of redundancies. In *Proceedings of the 2009 international conference on machine learning and data mining in pattern recognition* (pp. 251–265). Springer-Verlag.

Mihalcea, R., & Tarau, P. (2004). TextRank: Bringing order into texts. In *Proceedings of the 2004 EMNLP* (pp. 404–411).

Pei, J., Han, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., & Hsu, M. C. (2001). PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th international conference on data engineering*.

Salton, G., & McGil, M. J. (1986). *Introduction to modern information retrieval*. New York: McGraw-Hill.

Shannon, C. (1948). A mathematical theory of communication. *The Bell System Technical Journal, 27*(379-423), 623–656.

Turney, P. D. (1999). Learning to extract keyphrases from text. In *NRC TR ERB-1057* (pp. 1–43). National Research Council, Institute for Information Technology.

Wan, X., & Xiao, J. (2008). Single document keyphrase extraction using neighborhood knowledge. In *Proceedings of the 23rd national conference on artificial intelligence* (pp. 855–860). AAAI.

Wang, Q., Sheng, V. S., & Wu, X. (2017, February). Keyphrase extraction with sequential pattern mining. In *Proceedings of the 31st national conference on artificial intelligence (AAAI) (extended abstract and poster), February 4-9 San Francisco, California* (pp. 5003–5004).

Witten, I. H., Paynter, G. W., Frank, E., Gutwin, C., & Nevill-Manning, C. G. (1999). KEA: Practical automatic keyphrase extraction. In *Proceedings of the 4th ACM conference on digital libraries* (pp. 1–23).

Wu, X., Xie, F., Huang, Y., Hu, X., & Gao, J. (2013). Mining sequential patterns with wildcards and the One-Off condition. *Journal of Software, 24*(8), 1804–1815.

Wu, X., Zhu, X., He, Y., & Arslan, N. A. (2013). PMBC: Pattern mining from biological sequences with wildcard constraints. *Computers in Biology and Medicine, 43*(5), 481–492.

Xia, Z., Wang, X., Sun, X., & Wang, Q. (2015). A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems, 27*(2), 340–352.

Xie, F., Wu, X., & Zhu, X. (2014). Document-specific keyphrase extraction using sequential patterns with wildcards. In *Proceedings of the 2014 IEEE international conference on data mining* (pp. 1055–1060).

Xu, S., Yang, S., & Lau, F. (2010). Keyword extraction and headline generation using novel word features. In *Proceedings of the 24th AAAI conference on artificial intelligence* (pp. 1461–1466).

Yang, Y., & Yu, S. (2007). Chinese text clustering for topic detection based on word pattern relation. In *Research and development in intelligent systems XXIII* (pp. 408–412). Springer.

Zanmatkesh, F., & Hassanpour, H. (2011). Designing a meta-search engine considering the user's field of interest. In *Proceedings of the5th symposium on advances in science and technology* (pp. 1–8).

Zhang, M., Kao, B., Cheung, D. W., & Yip, K. Y. (2007). Mining periodic patterns with gap requirement from sequences. *ACM Transactions on Knowledge Discovery from Data, 1*(2), 1–39. http://dx.doi.org/10.1145/1267066.1267068.

Zhang, Y., Zincir-Heywood, N., & Milios, E. (2005). Narrative text classification for automatic key phrase extraction in web document corpora. In *Proceedings of the 7th annual ACM international workshop on web information and data management* (pp. 51–58).

Zhu, X., & Wu, X. (2007). Mining complex patterns across sequences with gap requirements. In *Proceedings of the 20th international joint conference on artificial intelligence* (pp. 2934–2940).