Noname manuscript No.
(will be inserted by the editor)

A Self-Verifying Clustering Approach to Unsupervised Matching of Product Titles

Leonidas Akritidis · Athanasios Fevgas · Panayiotis Bozanis · Christos Makris

Received: date / Accepted: date

Abstract The continuous growth of the e-commerce industry has rendered the problem of product retrieval particularly important. As more enterprises move their activities on the Web, the volume and the diversity of the product-related information increase quickly. These factors make it difficult for the users to identify and compare the features of their desired products. Recent studies proved that the standard similarity metrics cannot effectively identify identical products, since similar titles often refer to different products and vice-versa. Other studies employ external data sources to enrich the titles; these solutions are rather impractical, since the process of fetching external data is inefficient. In this paper we introduce UPM, an unsupervised algorithm for matching products by their titles that is independent of any external sources. UPM consists of three stages. During the first stage, the algorithm analyzes the titles and extracts combinations of words out of them. These combinations are evaluated in stage 2 according to several criteria, and the most appropriate of them are selected to form the initial clusters. The third phase is a post-processing verification stage that refines the initial clusters by correcting the erroneous matches. This stage is designed to operate in combination with all clustering approaches, especially when the data possess properties that prevent the co-existence of two data points within the same cluster. The experimental evaluation of UPM with multiple datasets demonstrates its superiority against the state-of-the-art clustering approaches and string similarity metrics, in terms of both efficiency and effectiveness.

Keywords product matching · entity matching · entity resolution · clustering · unsupervised learning · data mining

L. Akritidis, A. Fevgas, P. Bozanis
Data Structuring & Engineering Lab, Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece
E-mail: {leoakr,fevgas,pbozanis}@e-ce.uth.gr
Present address of P. Bozanis: School of Science and Technology, International Hellenic University, Thessaloniki, Greece, E-mail: pbozanis@ihu.gr

C. Makris
Department of Computer Engineering and Informatics, University of Patras, Patras, Greece
E-mail: makri@ceid.upatras.gr

1 Introduction

The online comparison of products is a crucial process, since it is usually the first step in the life cycle of an electronic sale. Before a purchase is completed, the majority of users search, collect, and aggregate the characteristics of both the desired and similar products, if any. For this reason, the role of the product comparison services has been rendered increasingly important. These platforms retrieve data from various sources, including electronic stores, suppliers, and reviews sites, and merge the information that refers to identical products. In the sequel, they present this information to their users, allowing them to compare a variety of parameters such as features and prices. They also facilitate the aggregation of user opinions and reviews.

Since the product related data originates from multiple sources, it presents a high degree of diversity. To implement their comparison tools, the aggregation platforms must develop algorithms that identify identical products. Apparently, the problem of product matching is vital for these platforms, their users, and e-commerce industry in general.

There are two basic types of methods that can be used to achieve the identification of identical products from their titles. The first one includes the utilization of the most significant string similarity metrics, such as the cosine measure and Jaccard index (and/or their variations), and a fixed similarity threshold that determines whether two entities match or not. Nonetheless, in Gopalakrishnan et al. (2012) the authors showed that these metrics are rather ineffective on this particular problem; frequently, identical products are described by very diverse titles, whereas highly similar titles do not necessarily represent identical products.

The second type of methods that can be applied in this case includes the popular clustering algorithms such as k-Means, Agglomerative clustering, DBSCAN, etc. Each of these approaches have their own strong and weak points. For instance, k-Means requires previous knowledge of the number of clusters before it is applied, a requirement that can be hardly satisfied in problems like the one we study here. On the other hand, the family of the Hierarchical clustering methods (divisive, agglomerative, etc.) is accompanied by large time and space complexities that render them prohibitively expensive for handling even medium-sized datasets.

For this reason, the method of Gopalakrishnan et al. (2012) adopted a different strategy; it employs Web search engines with the aim of enriching the product titles with important missing words. A similar approach is also introduced in Londhe et al. (2014), where the titles are modeled as graphs and a clustering algorithm determines whether these graphs form a cohesive community, or are separately clustered. However, the submission of a query to a search engine and the subsequent processing of the returned results are expensive operations. Additionally, the provided APIs do not allow unrestricted usage since there is a limit to the number of the queries that can be submitted on a daily basis. These limitations render these two approaches not applicable to large datasets with millions of products.

In this paper we present *UPM (Unsupervised Product Matcher)*, a three-stage unsupervised algorithm for matching products by their titles. The following list contains a brief description of the core parts of the algorithm and summarizes the contributions of this work:

– UPM is based on the concept of unsupervised entity resolution via clustering. In details, it constructs combinations of the words of the titles and assigns scores

to each one of them, similarly to Akritidis and Bozanis (2018). The highest-scoring combination (called cluster) is the one that best represents the identity of a product. All products within the same cluster are considered to be matching each other.
– UPM performs morphological analysis of the product titles and identifies potentially useful tokens (attributes, models, etc.). Each title is then split into virtual fields, and the tokens are distributed to these fields according to their form and semantics.
– The virtual fields are assigned scores, which, in the sequel, are used by a function to evaluate the combinations. This function also takes into consideration additional properties of a combination, including its position in a title and its frequency.
– A post-processing verification stage is employed, right after the formation of the clusters. In general, the steps of this stage are driven by a blocking condition that prevents two entities from existing together within the same cluster. Under these circumstances, and if such a blocking condition can be formed, the proposed verification stage is applicable to all clustering algorithms as a post-processing cluster refinement step. In the examined problem of products matching, such a blocking condition can be derived from the observation that very rarely a product appears twice within the catalog of a vendor.
– Unlike the aforementioned methods, UPM does not perform pairwise comparisons between the products to determine whether they match or not. Therefore, it avoids the quadratic complexity of this procedure and, also, it does not require the invention of an additional blocking policy.
– The following presentation introduces several hyper-parameters for UPM. However, we show that the dependence of our proposal from these hyper-parameters is small and the choice of specific values for them leads consistently to satisfactory performance.

The rest of the paper is organized as follows: In Section 2 we refer to the most important relevant works about string similarity, clustering, and entity matching. Section 3 presents the contribution of this work and it is organized in four Subsections. In particular, Subsection 3.1 covers some necessary preliminary elements. The proposed method consists of three phases, namely, the analysis of the titles, the initial clusters' formation and a verification stage where the clusters of the previous phase are refined. These phases are described in details in Subsections 3.2, 3.3, and 3.4, respectively. The experimental evaluation of the algorithm is presented in Section 4, and the final conclusions are summarized in Section 5. For research purposes, both the code we developed and the datasets we utilized have been made publicly available on GitHub.

2 Related Work

Due to its importance, the problem of entity matching (also known as entity resolution or record linkage) has been studied systematically and still attracts a significant number of researchers. Earlier studies employed standard string similarity (or distance) methods such as the cosine and edit distance measures (Elmagarmid et al. 2007; Manning et al. 2008; Gomaa and Fahmy 2013). More recent approaches focused on the problem of fuzzy matching and introduced strategies that extended the standard token-based similarity functions (Chaudhuri et al. 2003; Wang et al. 2011a). Another branch of the relevant literature includes works that compute the similarity between two strings by taking into consideration their semantics (Islam and Inkpen 2008; Bär et al. 2012; Lu et al. 2013; Hua et al. 2015).

The cost of computing the value of a similarity measure is considerable and quickly becomes a bottleneck in cases where the volume of the underlying data is large. In Wang et al. (2011b) the authors introduced several optimization techniques with the aim of accelerating this computation by eliminating the redundancy of the existing string similarity metrics. Furthermore, in Shen et al. (2007) a compositional approach to source-aware entity matching is proposed. Another optimization method is the one presented in Xiao et al. (2011), which is a filtering technique for avoiding the computation of the similarity values for all possible pairs of records. This method also exploits the token ordering information. Similarly, Li et al. (2008) proposed efficient filtering algorithms for approximate string searches.

On the other hand, the problem of clustering similar data is quite old with many practical applications such as pattern recognition and classification, data mining and knowledge discovery, data compression and vector quantization, and so on. Two surveys on the most important clustering algorithms and the theories behind them are presented in Jain et al. (1999) and Xu and Wunsch (2005). Entity matching and data clustering are similar problems, therefore the aforementioned string distance and similarity methods can also be applied for clustering.

The hierarchical methods constitute one of the oldest approaches to clustering. The simplest and most popular algorithms include the agglomerative methods with single (Sorensen 1948) and complete linkage (Sneath 1957). Although in some problems the agglomerative methods exhibit satisfactory performance, their big space and time complexities render them unsuitable for processing large-scale datasets. Another popular algorithm is k-Means (MacQueen et al. 1967) that is faster than the hierarchical methods, however, it requires prior knowledge of the number of clusters. It is rather a space partitioning than a clustering method, since it classifies all the input data points into a cluster, regardless of their correlation with that cluster.

DBSCAN is another sophisticated clustering method with two strong points: to begin with, it is capable of detecting outliers (or anomalies) in the datasets, whereas the second concerns the identification of non-isotropic clusters (Ester et al. 1996). On the other hand, its performance depends on the density of the clusters and frequently cannot classify correctly any sparsely distributed data points (Ng et al. 2002). Furthermore, the spectral clustering methods are based on the construction and processing of weighted knowledge graphs (Dhillon et al. 2007; Filippone et al. 2008), where the edges represent the similarity between two vertices. In fuzzy (or soft) clustering, a data point may belong to more than one clusters; a popular algorithm is fuzzy c-Means (Dunn 1973).

Regarding the specific problem of product matching, there are two categories of related works. The first category includes studies that take into consideration only the titles of the products. In particular, Gopalakrishnan et al. (2012) and Londhe et al. (2014) proposed the usage of Web search engines with the aim of enriching the titles with important missing words. In Köpcke et al. (2012) the authors propose a system for matching product titles; however, their solution heavily depends on the existence of key elements in the titles, such as product codes and manufacturer names. In Akritidis and Bozanis (2018) the authors process the product titles by extracting combinations and permutations out of them.

The second category includes methods that take into consideration additional features such as brands, manufacturers, categories, etc. More specifically, FEBRL provides an implementation based on SVMs for learning suitable matcher combinations (Christen 2008), and MARLIN offers a set of several learning methods such as SVMs and decision trees, combined with two similarity measures (Bilenko and Mooney 2003). In de Bakker et al. (2013) product matching is performed by comparing the attributes of the products. Nonetheless, all these methods exhibit one significant problem: since an aggregation service is fed with data from multiple non controlled sources, many of the product attributes that are present in one feed may be absent in another. Even if an attribute is provided by all sources, the data is frequently skewed or incomplete. In such occasions, it is inevitable that the methods of this category will not perform well.

3 UPM: Unsupervised Matching of Product Titles

This section presents our proposed UPM algorithm for matching products by their titles. It is organized in four subsections. Initially, some necessary preliminary elements are provided in Subsection 3.1. Then, the three phases of the algorithm are described in details in Subsections 3.2, 3.3, and 3.4.

3.1 Preliminary Elements and Token Combinations

Let us consider a set of vendors V that includes electronic stores, suppliers, auction platforms etc. Each vendor $v \in V$ distributes a catalog that contains the products s/he provides, accompanied by some additional information. In case this information is organized in a structured or semi-structured form, the catalog is called a *feed* and the products are stored as a collection of successive records. Each record consists of an arbitrary number of attributes, including its title, brand, model, and others.

Each vendor creates its feed independently of the others; hence, v may provide information about the brand or the category of a product, whereas v' may not. Even if both v and v' include this information in their feeds, there may be discrepancies that inevitably lead to skewed data. Nevertheless, all feeds must contain at least one descriptive title for each included product. Two or more vendors may use diverse titles to describe the same product; the objective of this study is to introduce an unsupervised algorithm that matches identical products by overcoming this diversity.

The string of a product title usually consists of multiple types of substrings, including words, model descriptions, technical specifications, etc. We collectively refer to all these substrings as *tokens*. Let W_t be the set of all tokens of a product title t. Then, a k-combination c_k is defined as any subset of W_t of size k, without repetition and without care for token ordering. For example, if W_t consists of 3 tokens $\{w_1, w_2, w_3\}$, then there are three possible 2-combinations, $\{w_1, w_2\}$, $\{w_1, w_3\}$, $\{w_2, w_3\}$, and one 3-combination, $\{w_1, w_2, w_3\}$. In case t consists of l_t tokens (that is, its length is l_t), then the number of all possible k-combinations is equal to the binomial coefficient:

$$N(l_t, k) = \binom{l_t}{k} = \frac{l_t!}{k!(l_t - k)!} \tag{1}$$

and the construction complexity for every possible value of k is $O(2^{l_t})$.

Notice that k-combinations are different than n-grams: the latter are computed by sliding a window of length n over the examined string, from the left to the right. Therefore, n-grams capture only successive tokens. However, in a product title the important tokens (brand, model, etc.) usually occur in non-adjacent positions within the string. Although the construction of k-combinations is more expensive, they are preferred over n-grams because of their ability to bring non-adjacent tokens together.

For example, there is no common 2-gram or 3-gram in the titles *nVidia GeForce GTX1050 4GB V335-001R* and *GeForce 4GB GDDR5 GTX1050 Graphics Adapter*. Hence, n-grams cannot identify the similarity between these two products. On the other hand, there are two common 2-combinations, namely *GeForce GTX1050* and *GeForce 4GB*, and one 3-combination, *GeForce GTX1050 4GB*. It is apparent that k-combinations outperform n-grams in this particular case. Also notice that despite these two strings refer to the same product, their similarity is small. For instance, their Jaccard index is equal to $3/8 = 0.375$ and, consequently, it does not constitute a reliable solution. Moreover, the first title does not include a product code, whereas the second one does not contain the name of the manufacturer. Therefore, neither the solution of Köpcke et al. (2012) performs well in this case.

Since the construction of all k-combinations is of exponential complexity, it is required to limit their number to a minimum possible value. Fortunately, our experiments showed that titles contain on average 6 to 11 tokens depending on the category of the product, and, also, only a portion of them is important for the identification of a product. For this reason, we limit the computations to the first 2-, 3-, ..., K-combinations of the tokens of the involved titles.

Eventually, for a title that consists of l_t tokens, the total number of combinations to be computed is:

$$N_c(l_t, K) = \sum_{k=2}^{\min(l_t,K)} \binom{l_t}{k} = \sum_{k=2}^{\min(l_t,K)} \frac{l_t!}{k!(l_t-k)!}. \tag{2}$$

For the sake of simplicity, in the presentation that follows we use the term “combination” instead of k-combination and the simplified notation c instead of c_k.

3.2 Stage 1: Title Analysis

The goal of the first stage of UPM is to process the titles and construct all the necessary data structures that will be subsequently used to perform the initial clustering of the products. This section is divided into two subsections that present the morphological analysis of the titles, the identification rules of the token semantics, the basic characteristics of the aforementioned data structures, and the algorithm that is used to build them.

3.2.1 Morphological Analysis & Token Semantics

Each title consists of tokens that are not equally important for the description of a product. Vendors may provide irrelevant information in a title, including payment facilities, special discounts, offers, shipping and delivery data, availabilities and so on. Such kind of information is considered as noise and may degrade the effectiveness of an entity matching algorithm.

Table 1 Identification rules of the token semantics

Type	Semantics	Identification Rule/s
1	Attribute	i) numeric tokens followed by measurement units, or ii) mixed tokens ending in a measurement unit
2	Model	The first mixed token in the title that does not represent an attribute
3	Model	All the rest mixed tokens in the title that do not represent an attribute
4	Model	A numeric token that is not followed by a measurement unit
5	Normal	All the other tokens of the title

The unsupervised extraction of the tokens that reflect the identity of a product (i.e. *hot tokens*) is a particularly challenging task, since vendors use different syntactical rules to express the information of their products. Furthermore, each product type presents its own specificity. Nevertheless, in this paper we perform morphological analysis of the titles with the aim of identifying these hot tokens. In particular, we initially examine the form of the tokens and we categorize each one of them as either:

- *Mixed*, in case it contains both digits and letters, or
- *Numeric*, if it contains only digits (with a decimal or thousands separator), or
- *Alphabetic*, in all other cases.

In the sequel, we identify the following important pieces of information:

Product Attributes: The attributes of a product are important because they differentiate it from other similar products. For example, the *32 GB* version of a cell phone is a different product compared to the *64 GB* version of the same model. This is common in many product types (e.g. hardware, electronics, etc). The process is based on a small lexicon of measurement units (e.g. bytes, hz, bps, etc.) and of their multiples and sub-multiples. By using this lexicon, an attribute is identified either i) when a pair of a numeric token and a measurement unit is encountered (e.g. *32 GB*), or ii) when the ending of a mixed token is a measurement unit and its suffix consists of digits (e.g. *32GB*). In the first case, the tokens are concatenated to eliminate the difference; so, *32 GB* becomes *32GB*.

Models: The model descriptors are the most important part of a product title since they represent its identity. Unfortunately, the models may receive forms that vary significantly among vendors and, also, a model may appear under different forms (e.g. *PS 3* vs. *PS3* vs. *Playstation3*, etc.). Hence, it is rather hard for an unsupervised technique to correctly identify all models with absolute accuracy. Nevertheless, the approach we present here yields significant improvements in the performance of our matching algorithm. We consider that a token is a possible model descriptor if it is either mixed or numeric and it is not followed by a measurement unit. In addition, not all mixed tokens are treated equally. For instance, the first mixed token in a product title is considered to have a greater chance to contain a model compared to the second or the third mixed one.

Normal: In the case a token does not fall into one of the above categories, then it is classified as a *normal* token.

Table 1 contains the five aforementioned semantics, accompanied by their respective identification rules.

The morphological analysis of a title includes several additional steps that are performed with the aim of removing the discrepancies between tokens with the same meaning. More specifically, the product titles of the dataset are parsed sequentially and the following procedures are applied to the extracted tokens:

– *Case folding*: all letters are converted to lower case.
– *Punctuation removal*: all punctuation symbols and marks are removed from a title apart from i) dots and commas that are thousands or decimal separators, and ii) hyphens and slashes that delimit tokens.
– *Duplicate token removal*: the existence of two or more identical tokens in a product title is rare. However, we have found that their removal improves the performance of the algorithm by a significant margin.

3.2.2 Construction of Data Structures

After the tokenization and the morphological analysis has been completed, the extracted tokens and combinations are used to build the appropriate data structures, namely the Token and Combination Lexicons and the Forward Index. In the sequel, we describe the information they maintain as well as their construction algorithm.

Token Lexicon: This is an ordinary lexicon structure L_w that is used to accommodate the tokens extracted from the titles of the products. For each token w, the token lexicon also stores:

i) a unique integer identifier (token ID),
ii) a frequency value f_w that represents the number of products that contain w in their titles, and
iii) a special variable s_w that is set equal to the semantics of w, as indicated by the first column of Table 1.

Combination Lexicon: The combination lexicon L_c stores all k-combinations, with $k \leq K$, of the tokens of the product titles. The representation of the stored combinations is of particular importance, since it must support not only fast lookups, but also searching for combinations with different orderings of their tokens. For instance, consider the case where we have extracted the 3-combination *CPU 3.2GHz 32MB* that does not exist in L_c. Instead, suppose that L_c contains the 3-combination *CPU 32MB 3.2GHz*, which is the same as the one we are searching for but with different ordering of its tokens. In such cases, we desire to identify the equality between the two records to avoid the insertion of the same combination twice.

The proposed algorithm satisfies this requirement by assigning signatures to all combinations. The key concept is that *a combination must have the same signature independently of the ordering of its component tokens.* This means that the two 3-combinations of the previous example, i.e., *CPU 32MB 3.2GHz* and *CPU 3.2GHz 32MB*, must be assigned equal signatures. More specifically, the following procedure is applied. Before a combination c is inserted into L_c, its signature g_c is computed. In the case g_c is not found in L_c, then c does not exist in L_c in any form (i.e., under any ordering of its tokens) and can be safely inserted into it. In the opposite case, c resides in L_c in one form or another.

A simple method for computing the signature of a combination c is via token sorting and hashing. More specifically, this method initially retrieves the IDs of the

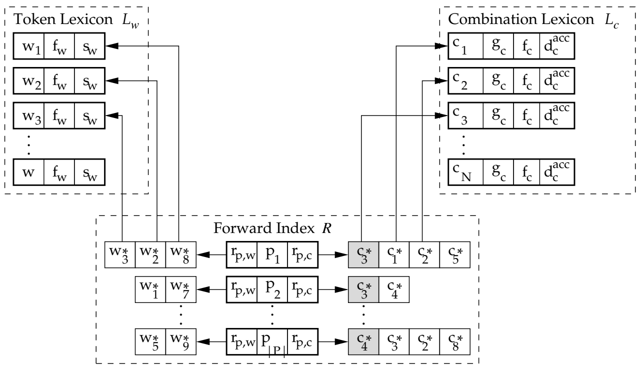

Fig. 1 The connection of the forward index R with the token lexicon L_w (left), and the combination lexicon L_c (right). The grayed boxes indicate the highest-scoring combination (that is, the cluster) of each product $p \in R$. In this example, p_1 and p_2 match each other, because they belong to the same cluster c_3.

component tokens of c and sorts them in increasing order. The sorted values are then concatenated and delimited by a special symbol (e.g. a single space character). The obtained string is subsequently passed through a string hash function h that is common for all combinations. The output of h constitutes the desired signature g_c.

As we demonstrate later in the experimental evaluation, the usage of signatures leads to substantial improvements in the efficiency of the algorithm. Eventually, each combination record $c \in L_c$ is associated with the following attributes:

i) its signature g_c,
ii) a frequency value f_c that stores the number of the titles that contain c, and
iii) a distance accumulator d_c^{acc} that maintains the sum of the distances of c from the beginning of the titles. This value will be used later to assign a score to c.

Forward Index: The forward index R is essentially a list of all product records. Each product p is associated with two pointer lists:

i) the token forward list $r_{p,w}$ that maintains l_t pointers to the tokens of the title t of p; and
ii) the combination forward list $r_{p,c}$, namely, a list of $N_c(l_t, K)$ pointers (given by Eq. 2). Each pointer refers to a combination c of p, where $c \in L_c$.

In Figure 1 we depict the interconnection of the forward index with the token and the combination lexicon data structures. Notice that the existence of pointers in the forward index saves the cost of storing the same data twice.

Construction Algorithm: Algorithm 1 presents the construction methodology of the aforementioned data structures. Initially, each product p enters the forward index R with its token and combination lists empty (step 4). Next, its title t is parsed and its tokens W_t are extracted. Each token $w \in W_t$ passes through a filtration process where the morphological analysis of Subsection 3.2.1 is performed. Moreover, the semantics s_w of w is identified according to the rules of Table 1 (step 7).

Next, a search for w in L_w is performed (step 8). Notice that the search operation returns a pointer $w*$ to the corresponding token record in L_w. In the case the search

Algorithm 1: Data structures construction

```
initialize lexicons $L_w$ and $L_c$ and the forward index $R$;
create the measurement units table $M$;
for each product $p$ with title $t$ do
    $R$.insert($p$);
    $W_t \leftarrow$ tokenize the title $t$ of $p$;
    for each token $w \in W_t$ do
        $s_w \xleftarrow{M}$ rules of Table 1, Subsection 3.2.1;
        $w* \leftarrow L_w$.search($w$);
        if $w* > 0$ then
            $f_w \leftarrow f_w + 1$;
        else
            $f_w \leftarrow 1$;
            $w* \leftarrow L_w$.insert($w$);
        end
        $R$.insert($p, w*$);
    end
    for each $k \in [2, K]$ do
        compute all $k$-combinations $C_k$ of $t$;
        for each $k$-combination $c \in C_k$ do
            $g_c \xleftarrow{h}$ compute signature of $c$;
            $c* \leftarrow L_c$.search($g_c$);
            $d(c,t) \leftarrow$ compute the distance of $c$ from $t$;
            if $c* > 0$ then
                $f_c \leftarrow f_c + 1$;
                $d_c^{acc} \leftarrow d_c^{acc} + d(c,t)$;
            else
                $f_c \leftarrow 1$;
                $d_c^{acc} \leftarrow d(c,t)$;
                $c* \leftarrow L_c$.insert($c$);
            end
            $R$.insert($p, c*$);
        end
    end
end
```

is unsuccessful, w is inserted in L_w with $f_w = 1$ and a pointer to the new record is returned; in the opposite case, its corresponding frequency value f_w increases by 1 (steps 9–14). Finally, the pointer $w*$ is inserted into the token list $r_{p,w}$ of p within the forward index R (step 15).

The procedure continues with the computation of all $2, 3, ..., K$-combinations of t and the generation of their respective signatures (steps 17–20). Then, for each combination c, the lexicon L_c is queried with the signature g_c. If this search is unsuccessful, c is inserted in L_c with $f_c = 1$ and a pointer $c*$ to the new record is returned; otherwise, f_c increases by one (steps 23–30). The algorithm ends with the insertion of the returned pointer $c*$ in the combination list $r_{p,c}$ of p within the forward index R (step 31).

During this process, the distance $d(c,t)$ of c from the beginning of the product title t is calculated, and it is subsequently used to update the distance accumulator d_c^{acc} (steps 25 and 28). This distance value will be employed later by a special function which will assign a score to each generated combination. We provide more details about the usage of the distance accumulator in the next subsection.

From the above discussion, noting that $O(n_c)$ and $O(n_p l_{max})$ are overshooting upper bounds for the number of distinct combinations and tokens, respectively, it can be derived that:

Lemma 1 *The construction algorithm, in the worst case: i) conducts $O(n_p(1 + l_{max}) + n_c)$ insertions to R, $O(n_p l_{max})$ insertions and searches to L_w, $O(n_c)$ insertions and searches to L_c, and $O(n_c)$ distance calculations, ii) generates all k-combinations in $O(n_c K \log K)$ time, while iii) needs $O(n_p(1 + l_{max}) + n_c)$ space, where n_p indicates the number of products, $n_c = O(n_p N_c(l_{max}, K))$ is the number of all k-combinations produced, and l_{max} denotes the maximum number of tokens per product.*

Proof. In the worst case, all titles have l_{max} length, therefore the total number of tokens in the dataset is $n_p l_{max}$. In addition, all $n_p l_{max}$ tokens are unique (i.e., there are no duplicate tokens), hence, all the searches in L_p and L_c will fail. Consequently:

i) Algorithm 1 conducts $O(n_p l_{max})$ (unsuccessful) searches to L_p and $O(n_p l_{max})$ insertions. Similarly, it performs $O(n_c)$ (unsuccessful) searches to L_c and $O(n_c)$ insertions. Regarding the forward index R, the algorithm will insert $O(n_p)$ product records, plus $O(n_p l_{max})$ pointers to $r_{p,w}$, plus $O(n_c)$ pointers to $r_{p,c}$. Then, the total number of insertions to R will be $O(n_p(1 + l_{max}) + n_c)$.

ii) The generation of a combination requires a sorting step during the creation of its signature. Given that each combination consists of K tokens, the sorting cost for one combination is $O(K \log K)$. Consequently, for n_c combinations the total cost is $O(n_c K \log K)$.

iii) $O(n_p l_{max})$, $O(n_c)$, and $O(n_p)$ are the additional space costs required to store all the tokens in L_p, all the combinations in L_c, and all the records in R, respectively. By summing these three costs, the total space bound $O(n_p(1+l_{max})+n_c)$ results. □

3.3 Stage 2: Formation of Initial Clusters

In summary, the purpose of this phase is to compute an importance score I_c for each combination $c \in r_{p,c}$ of each product p of the forward index. The highest-scoring combination will then be declared as the dominating cluster u where p will be mapped to. All the other products that will also be mapped to u will be considered that they match p. Finally, the clusters of all products will be utilized to build the clusters universe U that shall assist us further.

We now elaborate on the form of the combination score function. Initially, we study the properties that a combination must possess to be declared as a dominating cluster, and then we proceed to the quantification of these properties. So, the following parameters can impact the cluster formation:

– *Frequency*: The number of products that contain c is an important parameter, since the more frequent a combination is, the more products will be mapped to it. In contrast, if we select a rare combination, we shall not be able to map any other product to it.
– *Length*: The frequency criterion definitely favors the short combinations because it is more possible to encounter a 2-combination that is common for multiple products, compared to a 3-combination. However, the short combinations are not as descriptive as the longer ones and, also, there is a risk of creating very inhomogeneous clusters that may erroneously contain different products.

– *Position*: A broadly accepted idea in information retrieval dictates that the most important words of a document usually appear early, that is, in a small distance from its beginning.
– *Hot tokens*: A combination that contains multiple highly informational tokens represents the identity of the product more accurately compared to one that does not include such tokens.

Given a title t, a combination c of t, and a token $w \in c$, we consider that $o_w^{(c)}$ is the position (or offset) of w in c and $o_w^{(t)}$ is the position of w in t. By using this notation, the distance $d(c,t)$ between c and t is computed by employing the well-established Euclidean distance for strings:

$$d^2(c,t) = \sum_{w \in c} \left(o_w^{(c)} - o_w^{(t)} \right)^2. \tag{3}$$

Based on this equation, we compute the average distance of c from the beginning of all titles as follows:

$$\overline{d(c)} = \frac{1}{f_c} \sum_{\forall t \,:\, c \subseteq t} d(c,t). \tag{4}$$

The four aforementioned properties of a combination can now be quantified by the following scoring function:

$$I(c) = \frac{k Y_c^2}{\alpha + \overline{d(c)}} \log f_c, \tag{5}$$

where $\alpha > 0$ is a constant that i) prevents $I(c)$ from getting infinite when $\overline{d(c)} = 0$ (namely, when c appears always in the beginning of all titles), and ii) determines the importance of proximity in the overall score of a combination. Recall also that k denotes the length of c in number of tokens.

The Y_c factor constitutes the IR score of c and it is built by adopting the spirit of the BM25F scoring method for structured documents (Lu et al. 2005). This scheme is designed to boost the scores of the words that appear in highly important places of a document (called *fields*), such as its title.

Although a product title is clearly a short unstructured text, here we introduce the idea of splitting a title into *virtual fields*, based on the aforementioned semantics of each token. According to this approach, a title is divided into five virtual fields, from z_1 to z_5. Each field is allowed to contain only tokens that have identical semantics. For example, according to Table 1, z_1 shall accommodate only the tokens that represent the attributes of a product, whereas z_2, z_3 and z_4 enlist the tokens that potentially carry information about the model. Notice that a field may be completely empty, whereas a token can belong to only one field.

Similarly to the spirit of BM25F, the Y_c factor is computed by applying the following equation:

$$Y_c = \sum_{\forall w \in c} idf(w) \frac{Q(z_{s_w})}{1 - b + bk/\overline{l_c}}, \tag{6}$$

where $Q(z_{s_w})$ is the weight of the field that contains a token $w \in c$. Notice here the dependence of this weight from the semantics value s_w; the formula of the Q function (Eq. 9) is discussed in Subsection 4.2. Also, $idf(w) = \log(|P|/f_w)$ is the inverse document frequency (IDF) of w, where $|P|$ is the total number of product

Algorithm 2: Score calculation and cluster selection

```
initialize the cluster universe $U$;
for each product $p \in R$ do
    $r_{p,c}$ ←retrieve the combination forward list of $p$;
    max ← 0;    $u \leftarrow NULL$;
    for each combination $c \in r_{p,c}$ do
        for each token $w \in c$ do
            $|z_{s_w}| \leftarrow |z_{s_w}| + 1$;
        end
        $Y_c \leftarrow 0$;
        for each token $w \in c$ do
            $Q(z_{s_w}) \leftarrow$ Eq. 9 of Subsection 4.2;
            $Y_c \leftarrow Y_c +$ Eq. 6;
        end
        $\overline{d(c)} \leftarrow d_c^{acc} / f_c$;
        $I_c \leftarrow$ Eq. 5;
        if $I_c >$ max then
            max ← $I_c$;
            $u \leftarrow c$;
        end
    end
    $U$.insert$(u, p)$;    // Alg. 3
end
deallocate $L_c - U$;
```

titles. Furthermore, $\overline{l_c}$ is the average length (in number of tokens) of all combinations in the dataset and b is a constant that falls into the range $[0, 1]$.

In conclusion, Eq. 5 indicates that a product should be clustered under a combination that: i) is frequent, ii) is reasonably long, iii) usually occurs near the beginning of the titles, and iv) contains multiple important tokens. In the sequel, we use it to identify the most suitable cluster for each product of the forward index R.

Algorithm 2 describes the details of this procedure. Notice that, since we are only interested in the highest-scoring combination, it is not mandatory to store the scores of all combinations in some dedicated data structure (e.g. heap); a simple computation of the maximum score suffices.

Firstly, an empty set U is initialized. Next, we iterate through the products of R and for each product $p \in R$ we traverse its combination forward list $r_{p,c}$.

Algorithm 3: Insertion of a cluster u and a product p into universe U (step 21 of Algorithm 2)

```
Function $U.insert(u, p)$
    $v \leftarrow$ vendor of $p$;
    if $u \notin U$ then
        $U$.append$(u)$;
    end
    if $v \notin V_u$ then
        $V_u$.insert$(v)$;
    end
    $P_{u,v}$.insert$(p*)$;
end
```

For each combination $c \in r_{p,c}$, the field lengths are maintained according to the semantics z_{s_w} (cf. Table 1) of the tokens w of c (steps 6–8). In steps 9–13 the IR $\underline{\text{score}}$ of Eq. 6 is calculated, whereas the next step computes the average distance $\overline{d(c)}$. Having prepared this data, the score of c is obtained in step 15. In steps 16–19 we conditionally update the maximum score and the highest-scoring combination.

The combination with the maximum score is subsequently selected as the *dominating cluster*, or simply the *cluster* u of p. Next, u is inserted into the global set U, along with the corresponding product p, according to the steps 1–10 of Algorithm 3. Recall that, technically, a cluster is merely a k-combination object and, as such, it possesses the same properties. To support the verification stage of the next subsection, a cluster u must be extended to include the following elements:

– A list V_u that contains the vendors of the products of u.
– One list $P_{u,v}$ per vendor $v \in V_u$ that stores the products that belong to u and are also provided by v.

These elements are computed immediately during the insertion of u and p into U (step 21 of Algorithm 2): Initially, the vendor v of p is inserted into the list V_u, provided that $v \notin V_u$. Then, p is inserted into the corresponding list of products $P_{u,v}$.

Finally, Algorithm 2 deallocates the resources occupied by data that are not useful for the next stage, including the combinations that have not been declared clusters, that is, $L_c - U$. In overall,

Lemma 2 *The initial formation of clusters involves $O(n_c l_{max})$ computation steps and $O(|U| + n_p) = O(n_p)$ lexicon insertions and searches, in the worst case.*

Proof. The number of computation steps $O(n_c l_{max})$ derives directly from the three loops of Algorithm 2 (initiated at steps 2, 5, and 6). This concerns i) the calculation of the IR scores and the average distances for each of the n_c produced k-combinations, and ii) the determination of the field lengths for each token of each k-combination.

Moreover, for each of the n_p products, there is exactly one search in U for the dominating cluster (step 21 of Algorithm 2), thus the total number of searches is $O(n_p)$. In the worst case, all these searches will be unsuccessful; consequently, there will also be $O(n_p)$ insertions of dominating clusters into the global set U. □

3.4 Stage 3: Verification & Cluster Refinement

The procedures of the previous sections achieve their goal, that is, unsupervised matching of products by using only their titles. However, there is still room for improvement.

3.4.1 Blocking Conditions

Here we present a post-processing verification step that attempts to recognize the false matches of the previous stages. In the absence of training data, the process is based on a *blocking condition* $\mathcal{C}$, namely, a boolean expression that prevents two or more entities from existing together within the same cluster. The most significant advantage of the verification step is that if such a blocking condition exists (or can be formulated), then the algorithm is applicable to all existing clustering algorithms.

For example, in social network analysis applications, clustering algorithms are frequently used to group users with similar interests. However, a blocking expression could prevent two or more users with certain attributes from being grouped together within the same cluster (e.g. a user who likes hamburgers cannot belong to a cluster of vegetarians).

In the problem of product matching, a simple blocking condition may derive from the following observation: in the vast majority of cases, *each product appears only once in the feed of the same vendor*, or equivalently, *a vendor does not include identical products in her/his catalog.* Using this observation, $\mathcal{C}$ can be formed like so:

$$\mathcal{C} = \text{A cluster } u \text{ cannot contain multiple products from the same vendor } v. \quad (7)$$

The argument for proving the validity of $\mathcal{C}$ is as follows: Suppose that u contains two products p_1 and p_2 from the same vendor v. Since u contains only products that match each other, p_1 is identical to p_2. But then, v included the same product multiple times in her/his catalog, a statement that contradicts our initial observation.

Notice that our starting observation is not bullet-proof. At first, there exist some rare cases where, by mistake, the same product exists multiple times within a catalog of a vendor. Moreover, which products are called identical? How do we treat two products with the same brand and model but with different colors? In this article we do not deal with such complex questions. On the contrary, our intention here is to demonstrate that even under simple (and perhaps questionable) blocking conditions, such as the one of Eq. 7, the application of this verification stage leads to substantial improvements in the accuracy of a clustering algorithm.

3.4.2 Cluster Refinement Algorithm

The blocking condition of Eq. 7 drives the entire verification stage. Based on it, we say that v is a *violating vendor* of a cluster u, if u contains two or more of her/his products. In this case, u is an *invalid cluster* and it requires a special validation process to be applied to it. In short, this process allows only one product from a vendor v in the cluster u and evicts the rest of its products from u. The evicted products can either migrate to another existing cluster according to some criteria, or be transferred to a new cluster.

To support this procedure, additionally to lists $V_u, P_{u,v}$, the *clustroid product* π_u is also associated with each cluster; namely, π_u is the product in u that has the greatest similarity with all the other products (i.e., it maximizes the sum of similarities with all the other products). Clustroid π_u will be used as the representative of u and its title shall constitute the label of u. Consequently, π_u cannot leave u.

After all the required data have been prepared (Algs. 1–3), the verification stage of Algorithm 4 is executed. The procedure begins with the initialization of a special cluster that is called the *Cluster of Deletions, CoD.* This special cluster will be used to store all the deleted products from the clusters of U. We shall explain its usefulness shortly. For the time being, notice that, similarly to the regular clusters, all products in CoD are also stored in $P_{u,v}$ lists, that is, they are grouped by vendor.

In steps 2–5 of Algorithm 4 we visit each cluster $u \in U$ and we sort its list of vendors V_u by increasing vendor ID. This action will allow us to compute the intersection of V_u with V_{CoD} in a linear $O(|V_u| + |V_{CoD}|)$ cost. In addition, the clustroid π_u is calculated by finding the product that is the most similar to all other products

Algorithm 4: Verification and cluster refinement

```
// Initialization
initialize CoD (Cluster of Deletions);
for each cluster $u \in U$ do
    sort $V_u$;
    compute $\pi_u$;
end
// Products deletion from clusters and CoD construction
for each cluster $u \in U$ do
    for each vendor $v \in V_u$ do
        if $|P_{u,v}| > 1$ then
            for each product $p \in P_{u,v}$ do
                compute $S_p \leftarrow sim(p, \pi_u)$;
            end
            sort $P_{u,v}$ in decreasing $S_p$ order;
            for each product $p \in (P_{u,v} - P_{u,v}[0])$ do
                $P_{CoD,v}$.insert($p*$); // move $p$ to CoD
                $P_{u,v}$.remove($p*$); // and delete it from $u$
            end
            $V_{CoD}$.insert($v$);
        end
    end
end
// For each existing cluster, identify and move the most similar products
Set similarity threshold $\tau$;
for each cluster $u \in U$ do
    for each vendor $v \in V_{CoD}$ do
        if $v \notin V_u$ then
            $candidate_p \leftarrow NULL$; // the candidate product is initially null
            $\max S \leftarrow 0$;
            for each product $p \in P_{CoD,v}$ do
                if $sim(p, \pi_u) > \max S$ then
                    $\max S \leftarrow sim(p, \pi_u)$;
                    if $\max S > \tau$ then
                        $candidate_p \leftarrow p$; // set the candidate product
                    end
                end
            end
            if $candidate_p \neq NULL$ then
                $U$.insert($u, candidate_p*$); // if the candidate is set, transfer
                $P_{CoD,v}$.remove($candidate_p*$); // $candidate_p$ from CoD to $u$
            end
        end
    end
end
// If CoD still contains products, then create new clusters
$U' \leftarrow NULL$;
for each vendor $v \in V_{CoD}$ do
    for each product $p \in P_{CoD,v}$ do
        $candidate_c \leftarrow NULL$; // the candidate cluster is initially null
        for each cluster $u \in U'$ do
            $candidate_c \leftarrow$ Perform steps 27–34; // compute $candidate_c$
        end
        if $candidate_c \neq NULL$ then
            $U$.insert($candidate_c, p*$); // if the candidate is set, transfer
        else
            $U$.insert(new Cluster,$p*$); // otherwise, create a new cluster
        end
        $P_{CoD,v}$.remove($p*$); // remove the product from CoD
    end
end
```

of the cluster. From now on, the clustroid is treated as the representative of the cluster and its title will be used to compute the similarities with other products. These steps need $O(\sum_{u\in U}(|V_u|\log(|V_u|)+|u|^2))$ time, where $\sum_{u\in U}|V_u| \leq \sum_{u\in U}|u| = n_p$. In the best case, when there are only singleton clusters or clusters with just few members, this cost is linear $O(n_p)$, whereas, in the worst case, is upper bounded by $O(n_p^2)$ (when there are few clusters and all but one consist of a single product.)

In the sequel, for each cluster $u \in U$ we traverse its list of vendors V_u and in case a violator v is found (i.e., $|P_{u,v}| > 1$), we identify which product of v will stay in u. This is achieved by calculating the similarity score of each product $p \in P_{u,v}$ with the clustroid π_u, and by sorting $P_{u,v}$ in decreasing similarity score order (steps 9–12). This is accomplished in $O(\sum_{u\in U,v\in V_u}(1+|P_{u,v}|\log|P_{u,v}|) = O(n_p\log n_p)$ worst case time. The first record of the list – namely, the most similar product to π_u – is selected to remain in u; the rest $(P_{u,v} - P_{u,v}[0])$ products will eventually abandon u. The total number of lexicon operations needed is, in the worst case, $O(\sum_{u\in U,v\in V_u}|P_{u,v}|+1) = O(n_p)$ insertions and deletions.

As stated earlier, there exist two options to handle the evicted products. The first one is to search for another similar cluster (provided that it does not contain products of the same violating vendor) and move the products there. If no such cluster exists, the second option is applied; a new cluster must be created to accommodate these products. Notice that this is a greedy approach; according to it, as soon as a product p from a violating vendor v is encountered, then we immediately search for another highly similar cluster u. The problem with this approach is that there could be another product p' of v that is even more similar to u than p is (i.e., $sim(p',\pi_u) > sim(p,\pi_u)$). Hence, if p is moved immediately (greedily) to u, then p' will not be able to enter u because p and p' are provided by the same vendor.

To overcome this problem, we adopt an inverted logic. Instead of searching for the most similar cluster for a product, we search for the best products to be inserted into a cluster. This strategy guarantees that each cluster will receive the most similar products. For this reason, we avoid the greedy transfers each time a violating vendor is encountered, but instead we move her/his evicted products to CoD for further processing. This explains the usefulness of CoD.

The steps 21–41 of Algorithm 4 describe this process. Initially, a similarity threshold τ is set. Then, for each cluster $u \in U$, we examine if there exist products in CoD that are highly similar to u (that is, the clustroid of u). Notice that this examination is performed only for the products of the vendors who are not common between u and CoD; otherwise u would become invalid. Consequently, for a vendor v who belongs in V_{CoD} but not in V_u (step 24), we move the most similar product of v to u, provided that this similarity is greater than the threshold τ. The $candidate_p$ product in Algorithm 4 plays this role; initially it is set blank, and gets updated each time its similarity with the clustroid of u exceeds both the maximum value $\max S$ and τ (steps 27–34). In case there is no product of v that is highly similar to u, then the candidate remains blank and no transfer takes place (steps 35–38). The aforementioned steps take $O(|U|n_p^{CoD})$ time and $O(n_p^{CoD})$ lexicon insertions and deletions, where U is the set of clusters formed at the end of step 19, and n_p^{CoD} is the number of products that should be moved due to the blocking condition.

At the end of this process, each cluster $u \in U$ has received the products that are most similar to it. However, CoD may still contain some products that were not similar to any of the existing clusters. The steps 42–56 illustrate the method for handling these remaining products. At first, we define U' as the subset of U that contains the recently created clusters. Initially it holds that $U' = \emptyset$. Now, for each vendor v in CoD and for each product p in $P_{CoD,u}$ we search in U' for a cluster that is the most similar to p (steps 45–48). If such a cluster exists (step 50), p is transferred there; otherwise, a new cluster is created in U' and p is moved to it. These final steps need $O(|U'|n_p^{CoD})$ time and employ $O(n_p^{CoD})$ lexicon insertions and deletions, U' the set of extra clusters introduced.

Summarizing the above discussion, we have that:

Lemma 3 *The verification and cluster refinement stage, in the worst case, needs $O(n_p^2)$ computation steps and employs $O(n_p)$ lexicon operations.*

Proof. According to the previous analysis, in the worst case, the time costs for the four parts of the cluster refinement algorithm are the following:

– $O(n_p^2)$ for the initialization process (steps 2–5),
– $O(n_p \log n_p)$ for the construction of CoD (steps 6–20),
– $O(|U|n_p^{CoD})$ for transferring the products from CoD to the most appropriate clusters (steps 21–41), and
– $O(|U|n_p^{CoD})$ for handling the remaining products of CoD (steps 42–56).

Consequently, we get $O(n_p^2 + n_p \log n_p + |U|n_p^{CoD} + |U|n_p^{CoD}) = O(n_p^2)$ computation steps. Regarding the lexicon operations, in the worst case we have $O(n_p)$, $O(n_p^{CoD})$, and $O(n_p^{CoD})$ lexicon insertions and deletions for steps 13–16, 35–38, and 49–54, respectively. Consequently, the cluster refinement algorithm employs, in total, $O(n_p + n_p^{CoD} + n_p^{CoD}) = O(n_p)$ lexicon operations. □

Here we must note that the above complexities are quite pessimistic; in practice, Algorithm 4 requires considerably fewer steps. More specifically, its cost mainly depends on how well the previous two stages performed, namely, how many product matches violate the aforementioned blocking condition. The combination of Lemmata 1–3, leads to the following theorem for the overall complexity of UPM:

Theorem 1 *UPM, in the worst case, executes $O(n_p^2 + n_c(K \log K + l_{max}))$ computation steps and $O(n_p(1+l_{max})+n_c)$ lexicon operations, using $O(n_p(1+l_{max})+n_c)$ space, where n_p denotes the number of products, $n_c = O(n_p N_c(l_{max}, K))$ is the number of all k-combinations produced, and l_{max} indicates the maximum number of tokens per product.*

Proof. The bounds are derived by combining the computation steps, the lexicon operations, and the space cost of each stage of UPM, as proved in the respective lemmata. Table 2 summarizes these complexities for the three stages of UPM; its last row proves the Theorem.

Table 2 Time and space complexities of each stage of UPM

Stage	Computation Steps	Lexicon Operations	Space Cost
1	$O(n_c K \log K)$	$O(n_p l_{max} + n_c)$	$O(n_p(1 + l_{max}) + n_c)$
2	$O(n_c l_{max})$	$O(n_p)$	–
3	$O(n_p^2)$	$O(n_p)$	–
Total	$O(n_p^2 + n_c(K \log K + l_{max}))$	$O(n_p(1 + l_{max}) + n_c)$	$O(n_p(1 + l_{max}) + n_c)$

□

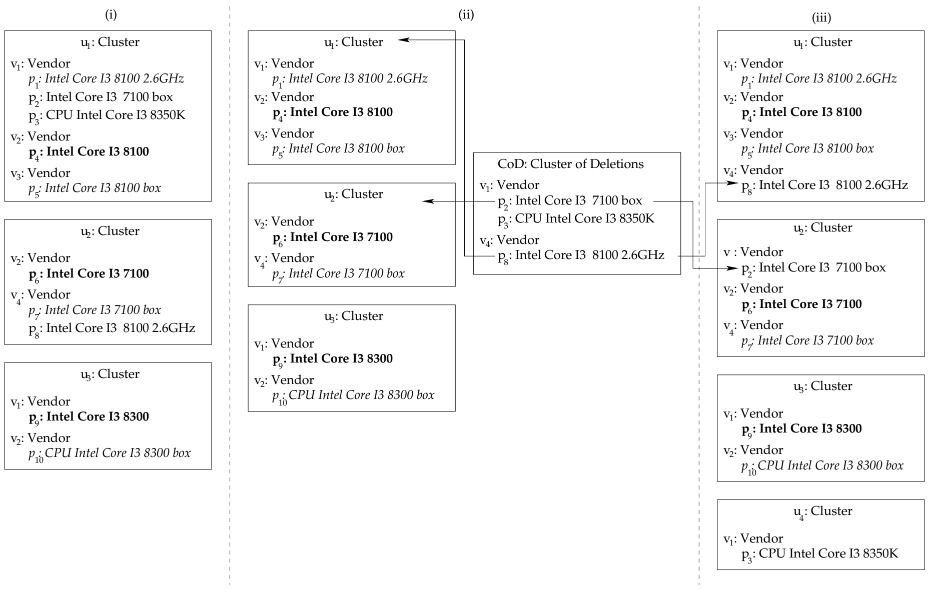
(i)
u_1: Cluster
v_1: Vendor
p_1: Intel Core I3 8100 2.6GHz
p_2: Intel Core I3 7100 box
p_3: CPU Intel Core I3 8350K
v_2: Vendor
p_4: Intel Core I3 8100
v_3: Vendor
p_5: Intel Core I3 8100 box
u_2: Cluster
v_2: Vendor
p_6: Intel Core I3 7100
v_4: Vendor
p_7: Intel Core I3 7100 box
p_8: Intel Core I3 8100 2.6GHz
u_3: Cluster
v_1: Vendor
p_9: Intel Core I3 8300
v_2: Vendor
p_{10}: CPU Intel Core I3 8300 box
(ii)
CoD: Cluster of Deletions
v_1: Vendor
p_2: Intel Core I3 7100 box
p_3: CPU Intel Core I3 8350K
v_4: Vendor
p_8: Intel Core I3 8100 2.6GHz
(iii)
u_4: Cluster
v_1: Vendor
p_3: CPU Intel Core I3 8350K

Fig. 2 Verification stage example: Left part (i): The initial clusters generated by a clustering algorithm. The titles in bold, italic and normal fonts represent the clustroids, the staying, and the evicted products respectively. Middle part (ii): The clusters after the deletion of the violating products accompanied by the Cluster of Deletions (CoD). Right part (iii): The clusters after the application of Algorithm 4.

3.4.3 Example

In this subsection we provide a running example of the verification stage. Throughout this example, all similarities between two strings t and t' are calculated by using the well-known Jaccard index $\mathcal{J} = |t \cap t'|/|t \cup t'|$. We also set the value of the similarity threshold equal to $\tau = 0.7$. Figure 2 illustrates our test scenario.

Let us suppose that the clusters that have been generated by UPM (or any other clustering method) are the ones depicted on the left part of the figure. The products are stored in groups of vendors. According to the blocking condition of Eq. 7, cluster u_1 is invalid because it includes three products from the same vendor v_1. That is, the algorithm erroneously decided that p_1, p_2 and p_3 match each other; these are clearly false matches, or else, true negatives. Similarly, the second cluster is also invalid because it contains two products from vendor v_4. On the contrary, the third cluster u_3 is a valid one.

We begin the application of Algorithm 4 by computing the clustroids of the clusters (denoted by boldface fonts). In cluster u_1 the product that has the maximum similarity with the other products is p_4, as its sum of similarities $sim(p_1, p_4) + sim(p_2, p_4) + sim(p_3, p_4) + sim(p_5, p_4) = 4/5 + 3/6 + 3/6 + 4/5 = 2.6$ is the greatest in the cluster. Hence, $\pi_{u_1} = p_4$ and, in a similar spirit, $\pi_{u_2} = p_6$, and $\pi_{u_3} = p_9$.

We continue with the construction of the Cluster of Deletions (CoD) by identifying the products that will be evicted from the clusters (steps 6–20 of Algorithm 4). In cluster u_1, vendor v_1 is a violating one and it is required that 2 out of her/his 3 products to be evicted. We compute the similarities of p_1, p_2 and p_3 with clustroid

$\pi_{u_1} = p_4$ and we get $sim(p_1, \pi_{u_1}) = 4/5$, $sim(p_2, \pi_{u_1}) = 3/6$, and $sim(p_3, \pi_{u_1}) = 3/6$. Since p_1 is the most similar product to the clustroid, it remains in u_1, whereas p_2 and p_3 get deleted and are inserted into CoD. Similarly, in u_2 the product to be evicted is p_8. The middle part of Figure 2 depicts CoD and the three clusters after the deletions of the violating products. The titles in italic and normal fonts denote the remaining and the leaving products, respectively.

Next, we identify that are the most suitable products of CoD to be transferred to each cluster (steps 21–41 of Algorithm 4). Cluster u_1 cannot receive any of the products p_2 and p_3 because it contains products of v_1; u_1 already has one product from v_1, i.e., p_1. In contrary, u_1 may receive the product p_8 of v_4, provided that the similarity of p_8 with the clustroid π_{u_1} is greater than threshold $\tau = 0.7$. Indeed, $sim(p_8, \pi_{u_1}) = 4/5 = 0.8 > \tau$ and, thus, p_8 moves from CoD to u_1.

Then, we move to the next cluster u_2 that does not contain a product from v_1. CoD includes 2 products from v_1 and it is required that we determine which one of them will be transferred to u_2, by computing their similarities with clustroid π_{u_2}. The similarities are found to be $sim(p_2, \pi_{u_2}) = 4/5$ and $sim(p_3, \pi_{u_2}) = 3/6$. Therefore, p_2 is the candidate product and, since $sim(p_2, \pi_{u_2}) > \tau$, p_2 enters u_2. Now CoD is only left with one product p_3 of v_1. That product cannot enter u_3, because u_3 already includes one product of v_1, namely, p_9.

At this point, all three clusters have been examined and all products from CoD that were similar to them have been moved. However, there is still one product in CoD (i.e., p_3) that must be arranged. The steps 42–56 of Algorithm 4 dictate that a new cluster u_4 must be created and inserted in U' (recall that U' is a subset of the clusters universe U). Then, p_3 is transferred into u_4. The right part (iii) of Fig. 2 depicts the final form of the four clusters.

The reader may compare the initial (left part of Fig. 2) with the final form of the clusters (right part) to realize how the verification stage has corrected the erroneous matches of the first case.

4 Experiments

This section analyzes the results of the experimental evaluation of the proposed algorithm. The presentation is organized as follows: Initially, to satisfy the requirements of the evaluation, we utilized 18 real-world datasets from different product categories that were acquired from two online product comparison platforms. Their basic characteristics are described in details in Subsection 4.1. In the sequel, in Subsection 4.2 we refer to the setting of the hyper-parameters of UPM. Notice that in all our tests we used the same values for these hyper-parameters. Finally, Subsections 4.3 and 4.4 contain the effectiveness and efficiency measurements of UPM against several state-of-the-art clustering methods and string similarity metrics, respectively.

The experiments were conducted on a single machine equipped with an Intel CoreI7 7700@3.6GHz processor and 32GB of RAM, running Ubuntu Linux 16.04 LTS. All methods were implemented in C++ and compiled by gcc with the -O3 speed optimization flag. We have made both this code and the datasets publicly available on GitHub[1] to allow the interested researchers verify our results and work further on our findings.

[1] https://github.com/lakritidis/UPM-full

Table 3 The experimental datasets accompanied by their characteristics

Dataset	$\|V\|$	$\|P\|$	**Titles**	$\overline{l_t}$
PriceRunner				
CPUs	37	1901	3862	11.285
Digital Cameras	103	836	2697	9.605
Dishwashers	94	1678	3424	6.819
Microwaves	114	1039	2342	7.591
Mobile Phones	84	1837	4081	8.416
Refrigerators	118	5172	11291	7.847
TVs	129	1678	3564	10.263
Washing Machines	87	1703	4044	7.931
Aggregate	306	15844	35305	8.560
Skroutz				
Air Conditioners	216	1442	13595	10.497
Car Batteries	66	2097	5864	8.073
Cookers & Ovens	163	1355	10858	6.455
CPUs	92	356	1906	9.115
Digital Cameras	152	973	4111	8.802
Refrigerators	161	1697	16177	5.955
TVs	205	1246	7002	7.382
Watches	212	60559	178657	6.517
Aggregate	652	68512	238170	6.827

4.1 Datasets

To ensure the robustness of our evaluation and to avoid results that were accidentally obtained, we conducted experiments by using multiple real-world datasets. In particular, we crawled two popular product comparison platforms, namely, PriceRunner[2] and Skroutz[3], and we constructed 8 datasets out of each one. Each of these 16 datasets represents a specific product category. The categories were selected with two criteria, in order to: i) study the performance difference of the same methods on similar products that were provided by different vendors, and ii) examine the effectiveness of the algorithms on products from diverse categories. For this reason, we included products from both identical and different categories in our experiments. Moreover, we created one aggregate dataset per platform that contains all the products from all 8 categories combined. These datasets enable the examination of the performance on heterogeneous datasets.

It is stressed that despite these 18 datasets were obtained from only two sources, they are totally unrelated to each other. This is due to the fact that each dataset contains products from different vendors and from multiple categories. Since each vendor describes a product on its own way and each category has its own specificities, there is no obvious relation among the employed datasets. For example, the dataset that contains the *washing machines* of PriceRunner is different than the *CPUs* or the *digital cameras* of this platform. Similarly, the *CPUs* dataset of Skroutz contains different vendors, descriptions and attributes compared to the *CPUs* of PriceRunner.

Another issue that needs to be clarified is that the comparison platforms collect many thousands of product feeds by various suppliers and vendors. This collection

[2] https://www.pricerunner.com/
[3] https://www.skroutz.gr/

procedure is performed multiple times per day with the aim of capturing new offers and changes in availabilities, prices, and other attributes of the products. Consequently, the platforms usually cannot manually clean or correct the incoming feeds and the information they present is the same as it is provided by the vendors themselves. This, in turn, means that here we do not deal with purged and well-formatted data, but with raw data that has not been processed, filtered, or purged in advance.

To facilitate prices and features comparison, the platforms group the same products into clusters. These clusters were utilized to establish the ground-truth for the evaluation of the various methods. More specifically, both platforms consider that all the titles within a cluster represent the same product. Hence, each dataset is accompanied by a special "matches" file that stores all the pairs of matching titles of all clusters. This file is subsequently used to verify the effectiveness of each method.

Table 3 presents the 18 experimental datasets accompanied by several useful characteristics. The first 9 rows concern the datasets that were crawled from PriceRunner, whereas the rest 9 are about the ones that were acquired from Skroutz. Columns 2, 3, and 4 display the distinct number of vendors, products, and product titles of each dataset, respectively. Moreover, the fifth column shows the average length of the titles. This last column is particularly important for UPM, since it shall determine the upper bound of K according to the discussion that follows in the next subsection.

4.2 Hyper-parameters Setting and Sensitivity Study

During the presentation of UPM we introduced several hyper-parameters that affect the construction and the scoring of k-combinations. Here we conduct an analysis of the performance of UPM against the the values of these hyper-parameters. The purpose of this analysis is to: i) study in depth the sensitivity of the algorithm against the fluctuation of these hyper-parameters, and ii) to discover indicative values that consistently lead UPM to satisfactory performance. In all cases, the results presented in the following subsections were obtained by using these indicative values.

For the requirements of this study, we employed 6 datasets from Table 3. Particularly, we selected the two aggregate datasets from PriceRunner and Skroutz, plus 4 random datasets (2 from each platform). Our methodology was to measure the matching quality of UPM by modifying the value of each hyper-parameter (one at a time), while using stable values for the rest of them. The performance was measured by using the popular $F1$ metric, defined by

$$F1 = 2\mathcal{P}\mathcal{R}/(\mathcal{P}+\mathcal{R}),$$

where $\mathcal{P}$ and $\mathcal{R}$ represent Precision and Recall, respectively. More details about the the effectiveness evaluation are provided in Subsection 4.3.

We begin with K, the modifier which determines the maximum number of tokens within a single k-combination. The left diagram of Fig. 3 illustrates the performance of UPM in the 6 aforementioned datasets, by setting $\alpha = 1$, $b = 1$, and $\tau = 0.5$ and by modifying the value of K in the range $[2, 6]$. Regarding the first three hyper-parameters α, b, and τ, we shall shortly show that the utilized values led to optimal or near-optimal performance. Several conclusions derive from this diagram:

- the value K^* which maximized the effectiveness of UPM was equal to the half of the average title length $\overline{l_t}$ minus 1, i.e., $K^* = \lfloor \overline{l_t}/2 \rfloor - 1$. For example, the *CPUs*

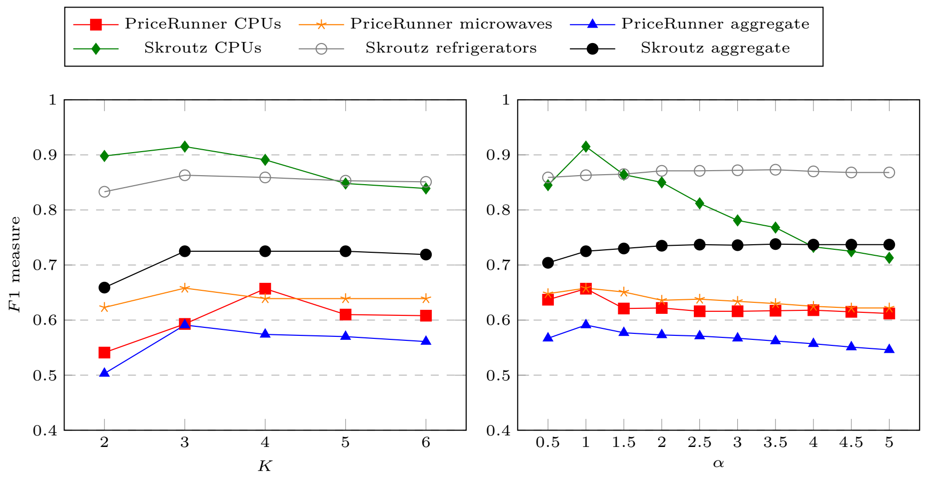

Fig. 3 The performance of UPM against its hyper-parameters i) K (left), and ii) α (right).

of PriceRunner are described on average by 11.285 tokens (see right column of Table 3), therefore, the highest performance was achieved by setting $K^* = \lfloor(11.285/2)\rfloor - 1 = 4$. On the other hand, for the *CPUs* of Skroutz where $\overline{l_t} = 9.115$, the setting which maximized performance was $K^* = 3$. This observation leads to the conclusion that only a portion of the tokens of a title are actually important for the identification of a product, and in particular, only half of them.

– UPM is robust against the fluctuations of the value of K, since larger or smaller values degrade the matching performance by only a small margin. For example, in the aggregate dataset of PriceRunner, the $F1$ values for $K = 3, 4, 5$ and 6 were 0.59, 0.57 ,0.57, and 0.56, respectively. The largest performance gap was recorded on the PriceRunner dataset of *CPUs*, where the $F1$ for $K = 3$ and $K = 4$ was measured at 0.593 and 0.657, respectively. These are differences which do not exceed 10–11% and indicate that although the increase of K leads to the production of more k-combinations, the scoring strategy of Subsection 3.3 assigns the highest scores to most suitable among them.
– The special case of $K = 2$ (i.e., combinations of two tokens) is an exception to the previous conclusion. The left diagram of Fig. 3 illustrates that, in this case, the matching performance of UPM was degraded for all datasets. For example, in the *aggregate* dataset of Skroutz the product titles contain on average 6.827 tokens; hence, according to our previous discussion, it holds that $K = \lfloor 6.827/2 \rfloor - 1 = 2$. With this setting, UPM achieved $F1 = 0.659$, a value that is considerably lower than the $F1 = 0.725$ which corresponds to $K = 3$.

These results are not surprising. To sufficiently describe the vast majority of products, at least three tokens are required on the title: one or more for the brand name, one or more for the model, and one or more for additional important information such as the attributes. Consequently, although $K = 2$ is a perfectly valid setting, we restrict the lower bound of K to 3, and we transform the previous setting of $K^* = \lfloor \overline{l_t}/2 \rfloor - 1$ into the following equation for the optimal value of K:

$$K^* = \max(3, \left\lfloor \overline{l_t}/2 \right\rfloor - 1). \tag{8}$$

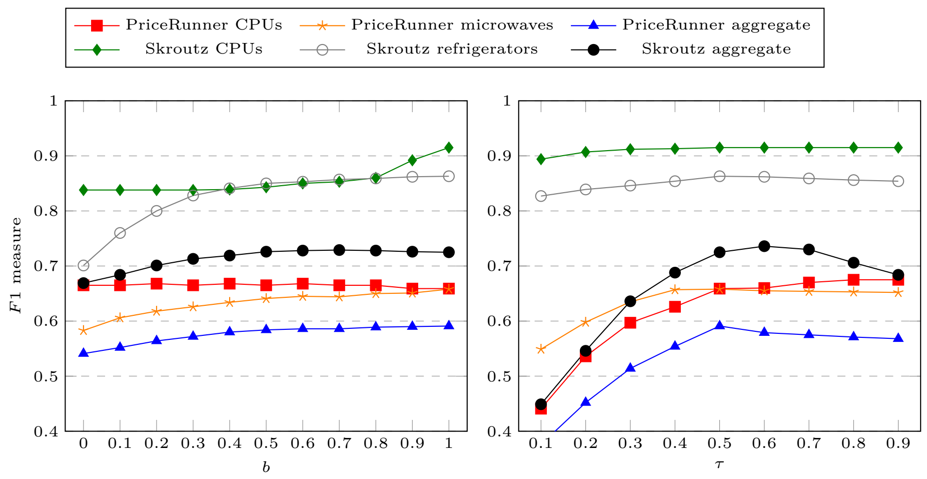

Fig. 4 The performance of UPM against its hyper-parameters i) b (left), and ii) the similarity threshold of the verification stage τ (right).

The second parameter is α, which appears in Eq. 5 and regulates the importance of proximity in the score of a combination. The right diagram of Fig. 3 shows that, in most cases, the matching quality of UPM is maximized for $\alpha = 1$. The *aggregate* dataset of Skroutz is an exception, since the best results are obtained for $\alpha = 3.5$. However, even in this case, the difference between the $F1$ values for $\alpha = 1$ and $\alpha = 3.5$ is small and does not exceed 1.5% ($F1 = 0.727$ vs. $F1 = 0.737$). Consequently, we consider that $\alpha = 1$ is an ideal setting and we use this value in all our experiments.

The third parameter of the algorithm is b, introduced in Eq. 6. The left diagram of Fig. 4 is clear and indicates that the highest the value of this hyper-parameter is, the best performance UPM achieves. Since b falls into the range $[0, 1]$, we set $b = 1$.

The next hyper-parameter to determine is the similarity threshold τ of Algorithm 4, which controls the similarity between an evicted product and a candidate cluster. The right diagram of Fig. 4 illustrates the $F1$ fluctuations against the variations of τ in the range $[0.1, 0.9]$. It is observed that, in the majority of the cases, the best performance is obtained by setting $0.5 \leq \tau \leq 0.7$. In the experiments of the following subsections we tuned τ at 0.5. However, the interested researcher may find that in some datasets, better results can be obtained by employing slightly higher values.

Finally, we present a strategy for setting the field weights of Eq. 6. One approach would be to simply a fixed weight to each field; for instance, one may consider that the model fields are twice as important as the field that contains the normal tokens. Although this approach delivers good results in some cases, it has two flaws: i) the weights are set arbitrarily in an ad-hoc manner, and ii) a set of predefined field weights that works well in one case may lead to poor performance in another. For these reasons, we dropped the idea of assigning fixed values to field weights, and we propose the following function:

$$Q(z_{s_w}) = 1/|z_{s_w}|, \tag{9}$$

where $|z_{s_w}|$ is the number of tokens of field z_{s_w}. Eq. 9 implements the intuition that the more tokens a field contains, the less important its tokens are, and vice versa.

The analysis we presented here demonstrated the robustness (and in some cases, the immunity) of UPM against the fluctuations of its hyper-parameters. Certainly, this is a highly desirable property for any unsupervised learning algorithm and for UPM, it has two explanations. First, notice that a change in the value of a hyper-parameter affects the scoring of *all* k-combinations on a similar manner, because all combinations are assigned scores by the same score functions. This means that, although the scores of the combinations change, the highest scoring combination (which constitutes the cluster where the product will be placed into) remains the same in the majority of the cases. Obviously, this is a strong indication of the effectiveness of the adopted scoring strategy. The second reason is that, even if an inappropriate combination receives the highest score (i.e., it is incorrectly declared the cluster of a product), the application of the verification stage will correct most of these errors.

4.3 Effectiveness Evaluation

We now proceed to the presentation of the effectiveness measurements of UPM in terms of product matching quality. This part of the experiments is organized into two subsections, where UPM is compared against several state-of-the-art clustering methods and string similarity measures, respectively.

The proposed algorithm achieves product matching by constructing clusters of similar products. To evaluate its output we applied the following methodology: Initially, we iterate through each cluster and for each product in the cluster, we create one pairwise match record with each of the rest of the products in the same cluster. In other words, we create a database with all the distinct product pairs within a cluster. Next, we compare the records of this database with the ones of the aforementioned matches' file and we count the number of true positives and negatives.

4.3.1 Comparison with Other Clustering Algorithms

In this subsection we aim to:

1. Demonstrate the superiority of UPM over three state-of-the-art clustering methods, namely, Leader Clustering (*LC*), Agglomerative (Hierarchical) Clustering Analysis (*HCA*), and *DBSCAN*.
2. Prove that the verification stage of Subsection 3.4 can be applied to all these approaches as a post-processing procedure. Hence, apart from the aforementioned baseline methods, we also compare UPM against their *refined* versions that are denoted by the *R-* prefix. For example, the notation *R-LC* refers to the Refined Leader Clustering algorithm, that is, Leader Clustering combined with the post-processing verification stage. In contrast, we also examine an "unrefined" version of UPM (abbreviated as *U-UPM*), with the aim of testing the effectiveness of only the first two stages of the algorithm, that is, without the verification stage.
3. Show that the application of the verification stage improves the matching quality of all clustering algorithms.

Regarding the popular k-Means algorithm, its requirement for prior knowledge of the number of clusters was a prohibitive factor for its inclusion in our experiments. Moreover, the application of prediction techniques (such as the elbow and silhouette methods) requires the repetitive execution of k-Means before the centroids are stabilized and the number of the clusters is estimated. Consequently, they are particularly expensive especially in the case of large datasets. On the other hand, fuzzy clustering algorithms (such as fuzzy c-Means) allow each data point to belong to more than one cluster. Therefore, they are not appropriate for this particular problem.

Furthermore, we did not include results from the method of Gopalakrishnan et al. (2012). This algorithm submits queries to Web search engines in order to enrich the product titles with important missing words (one query per title) and assign importance scores to the words of the enriched titles (one query per pair of words, per title). If we had applied this method, for example, on the *aggregate* dataset of Skroutz (about $24 \cdot 10^4$ titles and 7 words per title), the required number of queries would be 5.3 million. Clearly, this cost renders the method unsustainable. Notice also that the method of Gopalakrishnan et al. (2012) is compared against only one similarity metric by employing 2 small datasets. In contrast, here we evaluate UPM against 3+3 clustering algorithms and 7 string similarity metrics, using 18 datasets.

Figures 5 and 6 illustrate the performance of UPM against the three aforementioned clustering methods for the 9 datasets of PriceRunner. Each diagram depicts the fluctuation of the $F1$ scores for various similarity thresholds, ranging from 0.1 to 0.9. Please recall that the similarity threshold τ determines whether two entities e_1 and e_2 match or not. That is, e_1 matches e_2 only if their similarity value exceeds τ. Nonetheless, since UPM does not perform pairwise comparisons between products, its $F1$ values are represented by straight horizontal lines.

On the other hand, the adversary clustering methods operate by computing distances (similarities) between products, or between products and clusters. We selected the weighted version of the Cosine measure to calculate these similarities; in the next subsection we show that, in this particular problem, this metric achieved the highest matching quality. In LC, a cluster is represented by its *leader product*, that is, the first product that is inserted into the cluster. Therefore, all product-cluster similarities are converted to product-leader similarities. In HCA, we applied the agglomerative approach with simple linkage, and stopped merging the clusters when the similarity between any pair of clusters did not exceed τ. In DBSCAN, the *minPoints* parameter that determines whether a data point is an outlier (or noise) or not, was set equal to 2. This means that two similar products are allowed to form a cluster.

There are three qualitative conclusions that derive from these diagrams: i) UPM outperformed all the baseline clustering methods for all datasets by a large margin, ii) UPM outperformed even the refined versions of these algorithms albeit by a smaller margin, and iii) the application of the verification stage to a clustering algorithm leads to considerable performance benefits.

In 5 out of 9 cases, the $F1$ score of UPM was measured between 0.6 and 0.7: 0.67 for *refrigerators* (Fig. 5f) and *dishwashers* (Fig. 5c), 0.66 for *CPUs* (Fig. 5a), and 0.65 for *digital cameras* (Fig. 5b) and *microwaves* (Fig. 5d). In all nine datasets, the strongest opponent was R-LC. The greatest performance difference was observed in the case of *mobile phones* (Fig. 5e), where UPM was about 31% more accurate ($F1 = 0.573$ vs $F1 = 0.437$), whereas the smallest gap was approximately 6% in the dataset of *TVs* (Fig. 6a). The third most accurate algorithm was R-HCA, which, according to the diagrams, achieved $F1$ values lower by about 0.01–0.1 compared to those of R-LC. Regarding the *aggregate* dataset of Fig. 6c, UPM scored $F1 = 0.59$ and outperformed R-LC and R-HCA by about 15% ($F1 = 0.51$) and 23% ($F1 = 0.48$), respectively. Notice that R-LC achieved its highest $F1$ scores for $\tau = 0.3$ and $\tau = 0.4$,

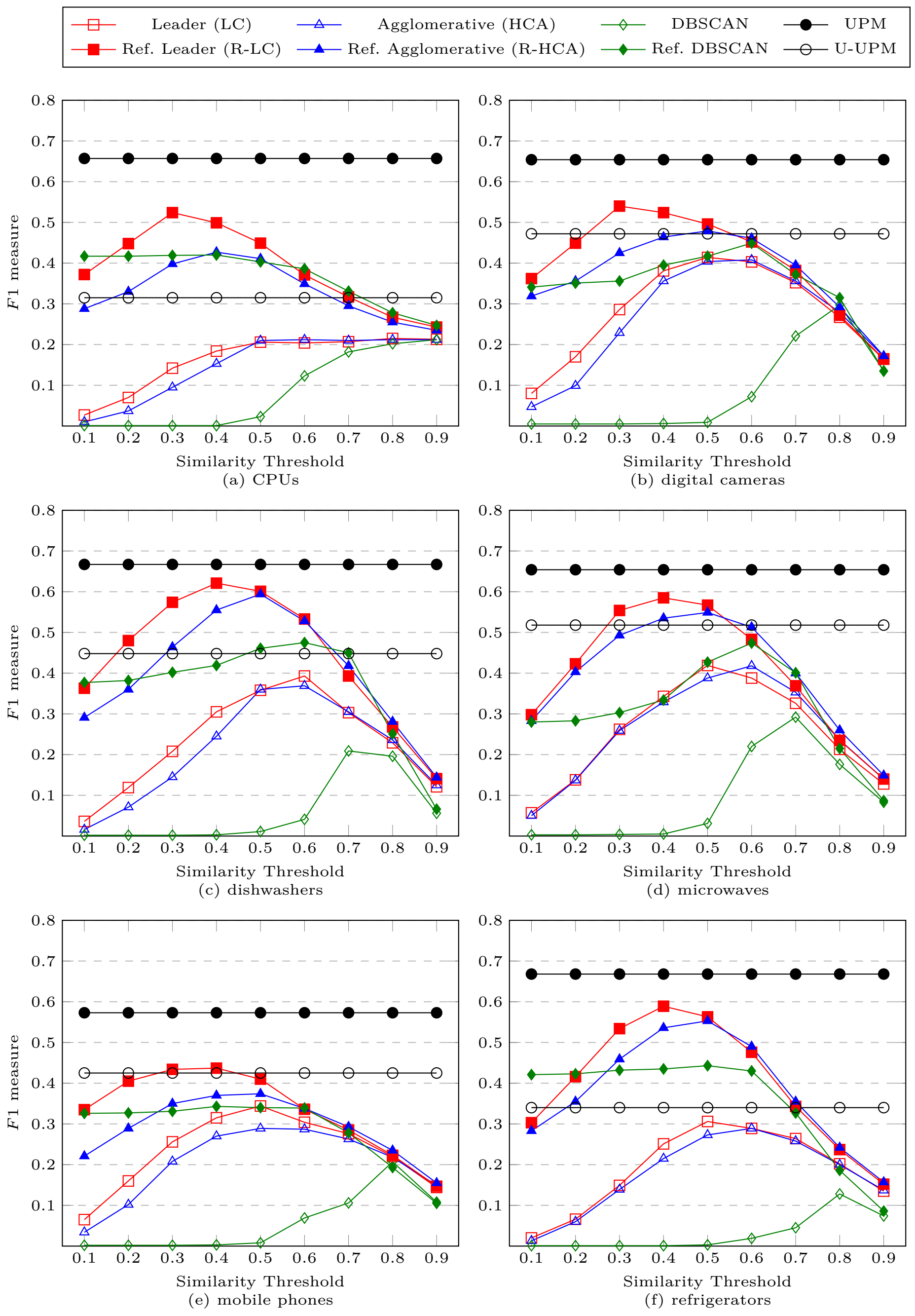

Fig. 5 Performance comparison of UPM and U-UPM against 3 baseline clustering algorithms and their refined versions for the first 6 PriceRunner datasets of Table 3.

whereas HCA was rather more stable, since its matching accuracy in all cases peaked at $\tau = 0.5$.

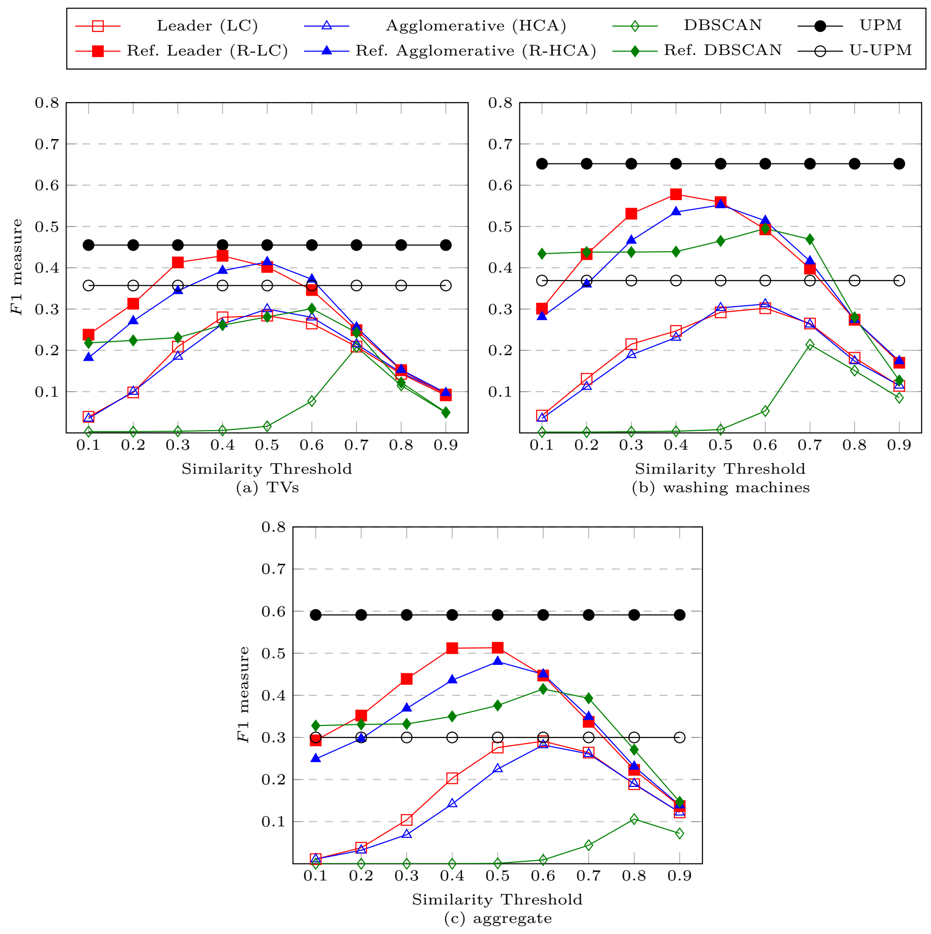

Fig. 6 Performance comparison of UPM and U-UPM against 3 baseline clustering algorithms and their refined versions for the last 3 PriceRunner datasets of Table 3. The solid colored markers indicate the execution of the post-processing verification stage.

R-DBSCAN was considerably less effective, as its $F1$ scores were never greater than 0.5. Consequently, it seems that density-based clustering does not work well in this specific problem. Moreover, the discovery of the neighbors of a data point (and then the neighbors of the neighbors, and so on) erroneously matches data points that are very distant from the starting one.

As stated earlier, the refined versions outperformed the baseline clustering methods by a large margin that in some datasets reached 250%. More specifically, the application of the post-processing verification stage in the *CPUs* dataset improved the output of Leader Clustering by 2.44 times ($F1 = 0.524$ vs. $F1 = 0.215$). In the same dataset, R-HCA and R-DBSCAN outperformed their baseline implementations by almost two times. Similarly, in the case of the *aggregate* dataset, R-LC, R-HCA, and R-DBSCAN were much more accurate than their original versions, since they achieved $F1$ scores that were higher by roughly 1.7, 1.7, and 3.9 times, respectively. Differences of such magnitudes are also observed for the rest of the utilized datasets.

Regarding the “unrefined” version of UPM, Figures 5 and 6 show that it outperformed all the baseline clustering methods algorithms in all datasets. For example, in the dataset which contained the *CPUs*, U-UPM achieved $F1 = 0.315$, a value that was larger than that of LC, HCA and DBSCAN by 45-49%. On the other hand, the refined UPM method was more than two times more effective, since its $F1$ in this dataset was measured at 0.657. Similar differences were recorded for other datasets as well (e.g. for *TVs* and *refrigerators*) and in general, the performance gap between UPM and U-UPM in these 9 datasets was substantial. Nevertheless, these measurements also prove that even without the application of the verification stage, the first two stages of Subsections 3.2 and 3.3 are sufficient to surpass the accuracy of all the baseline clustering algorithms.

These conclusions were verified in the measurements on the Skroutz datasets, illustrated in Figures 7 and 8. In general, in these datasets all methods performed better compared to the PriceRunner datasets. The highest matching accuracy was obtained in the dataset that contained *CPUs* (Fig. 7d), where UPM scored a remarkable $F1 = 0.915$. R-LC, R-HCA and R-DBSCAN achieved $F1 = 0.79$ (-16%), $F1 = 0.62$ (-48%), and $F1 = 0.43$ (-114%), respectively. In addition, in two cases, *refrigerators* (Fig. 7f) and *cookers & ovens* (Fig. 7c), the $F1$ scores of UPM exceeded or approached 0.8 (0.863 and 0.795 respectively), and outperformed the strongest opponent, R-LC, by approximately 10% in both cases.

In the *aggregate* dataset of the Skroutz platform with the $24 \cdot 10^4$ diverse titles (Fig. 8c), UPM was also the best performing method, since it achieved $F1 = 0.725$ and prevailed over R-LC, R-HCA, and R-DBSCAN by approximately 8%, 20%, and 26%, respectively. Similarly to the datasets of PriceRunner, the effectiveness of the baseline clustering algorithms was considerably lower. In the *aggregate* dataset, R-LC, R-HCA, and R-DBSCAN outperformed their baseline counterparts by roughly 31%, 22%, and 32%, respectively. These results attest the usefulness of the proposed verification stage, since the matching quality of all algorithms is improved in all cases by remarkable margins.

Similarly to the datasets originating from PriceRunner, the clustering accuracy of U-UPM was higher compared to those of the baseline clustering approaches. However, there are two elements which differentiate the effectiveness of U-UPM in the Skroutz datasets: The first one is that here, the performance gap between UPM and U-UPM was on average considerably lower than the gap which was measured in the datasets of PriceRunner. Indicatively, the percentage differences in the $F1$ values of UPM and U-UPM in the datasets which contained *air conditioners*, *CPUs*, *refrigerators*, and *TVs* were 16.2%, 11%, 4.2%, and 2.4% respectively. The second element is that in some cases, U-UPM not only outperformed the baseline clustering methods, but their refined variants as well. These observations strengthen the importance of the first two stages of the algorithm and demonstrate that the operations performed by the verification stage depend on the performance of these stages.

Another important conclusion that derives from the visual comparison of these 18 diagrams is that the adversary clustering algorithms achieve their highest performance for different values of the similarity threshold τ. For example, the performance of R-LC peaked at 4 different values of τ, ranging from 0.2 to 0.5. R-HCA was considerably more stable, since in most cases, its effectiveness was maximized for $\tau = 0.5$. Nevertheless, there are some individual cases where the highest $F1$ scores were obtained at different similarity thresholds – e.g., $\tau = 0.3$ for the *air conditioners* dataset of Skroutz, and $\tau = 0.4$ for *CPUs*. Similarly to R-LC, the performance

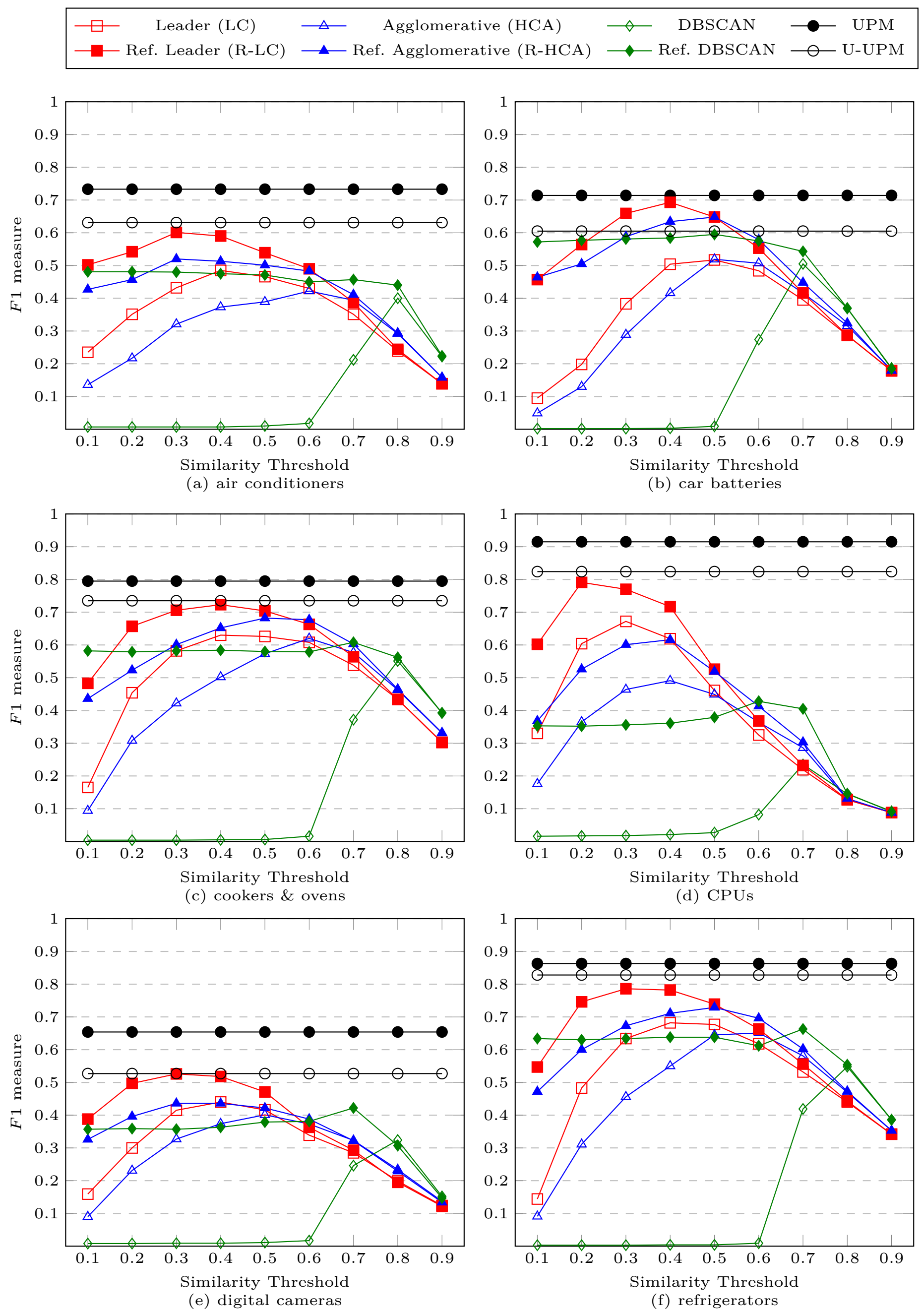

Fig. 7 Performance comparison of UPM and U-UPM against 3 baseline clustering algorithms and their refined versions for the first 6 Skroutz datasets of Table 3.

of R-DBSCAN was maximized for 4 different values of τ, ranging from 0.4 to 0.7. These measurements, in combination with the “immunity” of UPM against the fluc-

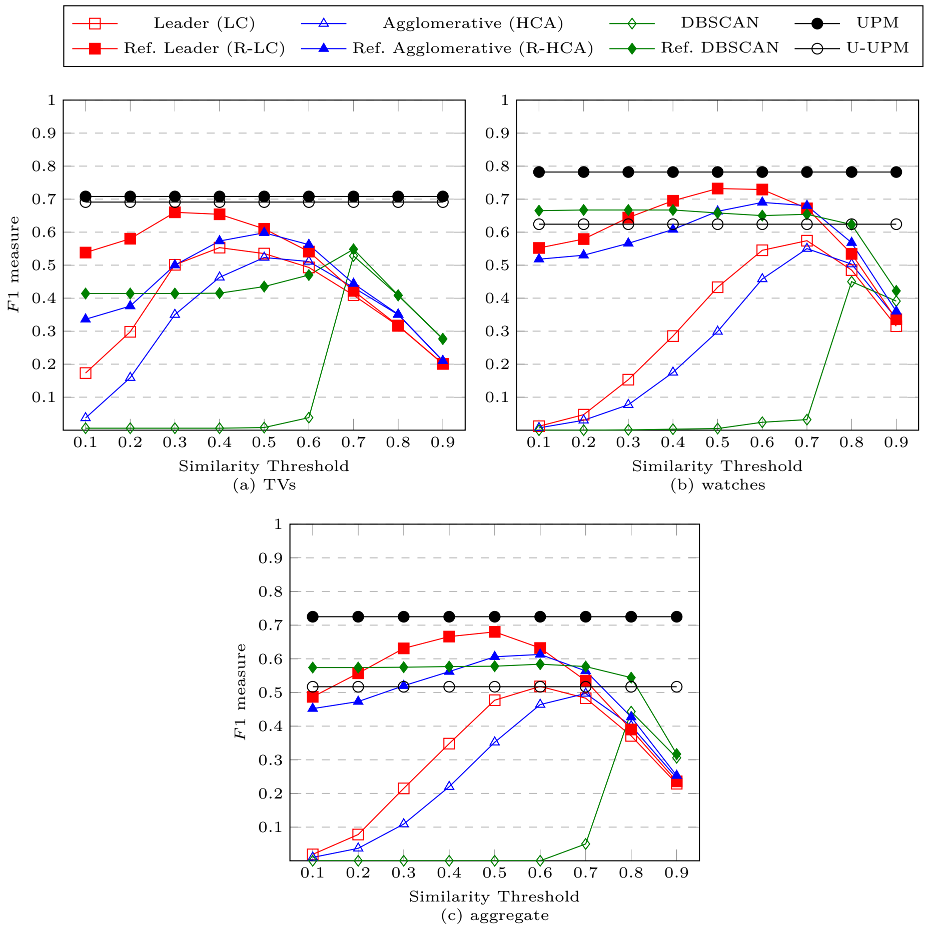

Fig. 8 Performance comparison of UPM and U-UPM against 3 baseline clustering algorithms and their refined versions for the last 3 Skroutz datasets of Table 3. The solid colored markers indicate the execution of the post-processing verification stage.

tuations of the similarity threshold τ, highlight yet another significant advantage of the proposed method against the adversary clustering algorithms.

4.3.2 Comparison with String Similarity Measures

We continue the comparison of UPM against several well-established string similarity measures. More specifically, we employed three token-based measures, namely, Cosine similarity, Jaccard index and Dice coefficient and their weighted versions, where the plain token counts are replaced by their respective inverse document frequency. We also included the Edit Distance measure, which is one of the most popular character-based similarity measures. Table 4 contains the definitions of the aforementioned measures, given two strings t and t'. Regarding the last row, Edit Similarity derives from the normalized Edit Distance measure, which is equal to the minimum number

Table 4 Adversary string similarity measures

Measure	Baseline	Weighted
Cosine Similarity	$\mathcal{S}_{cos} = \frac{\lvert t \cap t' \rvert}{\sqrt{\lvert t \rvert}\sqrt{\lvert t' \rvert}}$	$\mathcal{S}_{wcos} = \frac{\sum_{w \in (t \cap t')} idf_w^2}{\sqrt{\sum_{w \in t} idf_w^2}\sqrt{\sum_{w \in t'} idf_w^2}}$
Jaccard Index	$\mathcal{S}_{jacc} = \frac{\lvert t \cap t' \rvert}{\lvert t \cup t' \rvert}$	$\mathcal{S}_{wjacc} = \frac{\sum_{w \in (t \cap t')} idf_w^2}{\sum_{w \in (t \cup t')} idf_w^2}$
Dice Coefficient	$\mathcal{S}_{dice} = \frac{\lvert t \cap t' \rvert}{\lvert t \rvert + \lvert t' \rvert}$	$\mathcal{S}_{wdice} = \frac{\sum_{w \in (t \cap t')} idf_w^2}{\sum_{w \in t} idf_w^2 + \sum_{w \in t'} idf_w^2}$
Edit Similarity	$\mathcal{S}_{edit} = 1 - \frac{ED(t,t')}{max\{\lvert t \rvert, \lvert t' \rvert\}}$	–

of single character edit operations (i.e., insertion, deletion, and substitution) needed to transform t into t'. Consequently, there is no weighted version for Edit Similarity.

Figures 9 and 10 illustrate the matching performance of the string similarity measures of Table 4 for the 9 PriceRunner datasets. Similarly to the presentation of the previous subsection, a curve with uncolored markers represents a baseline method, whereas the curve with the same color and colored markers shows the performance of the weighted counterpart. Based on our earlier conclusions and to provide fair results, we applied the blocking condition of Eq. 7 during the evaluation of a similarity measure. In other words, we prevented the comparison of the products that originate from the same vendor and we considered that they do not match. Without this blocking condition, the results presented below would be much worse.

For this reason, U-UPM was not included in these diagrams; comparing it with string similarity metrics which adopt the blocking condition, would be both unfair and meaningless. In that case, the inclusion of the same metrics without the application of the blocking condition would be necessary. However, such an inclusion would double the number of the plotted curves in all diagrams, rendering them almost impossible to read.

The first conclusion that derives from these diagrams is that, in all datasets, the similarity metrics with IDF token weights performed much better than their standard expressions. For instance, in the *aggregate* dataset of Fig. 10c, the effectiveness of $\mathcal{S}_{wcos}$, $\mathcal{S}_{wjacc}$, and $\mathcal{S}_{wdice}$ were $F1 = 0.43@\tau = 0.6$, $F1 = 0.42@\tau = 0.4$, and $F1 = 0.41@\tau = 0.5$, respectively. Compared to the corresponding $F1$ scores of $\mathcal{S}_{cos}$, $\mathcal{S}_{jacc}$, and $\mathcal{S}_{dice}$ these values were higher by 139%, 121%, and 128%, respectively. Similar (or even greater) differences in performance were observed for the rest of the datasets as well.

The second conclusion concerns the poor performance of Edit Similarity; there was no experiment where the $F1$ score of this measure exceeded 0.2. This poor effectiveness is not surprising. Since Edit Similarity is a character-based measure, it assigns high similarity scores to nearly identical tokens. However, nearly identical tokens may represent different products according to our early discussion. Consequently, these high scores are assigned erroneously.

UPM prevailed over its adversary approaches in all examined cases. The largest performance gap was measured in the *CPUs* dataset (Fig. 9a), where the $F1$ score achieved by UPM (0.657) was more than two times greater than the respective values of $\mathcal{S}_{wcos}$ (0.319), $\mathcal{S}_{wjacc}$ (0.310), and $\mathcal{S}_{wdice}$ (0.312). The difference between UPM and the baseline similarity metrics was even greater, since $\mathcal{S}_{cos}$, $\mathcal{S}_{jacc}$, and $\mathcal{S}_{dice}$ scored

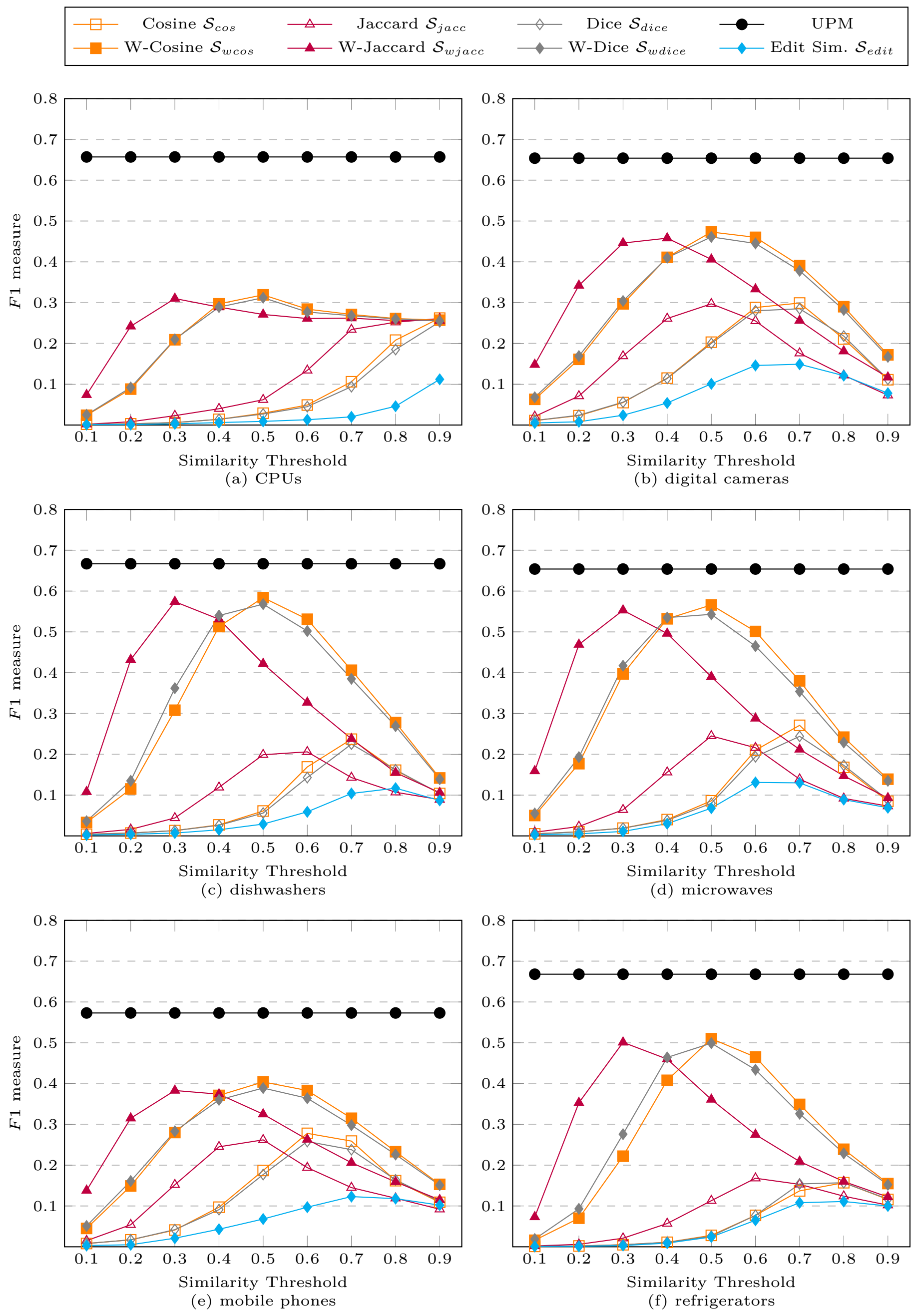

Fig. 9 Performance comparison of UPM against 7 string similarity measures for the first 6 PriceRunner datasets of Table 3.

0.262, 0.261 and 0.255, respectively. Notice the almost identical performances of both the weighted and the baseline string similarity measures. Another interesting point

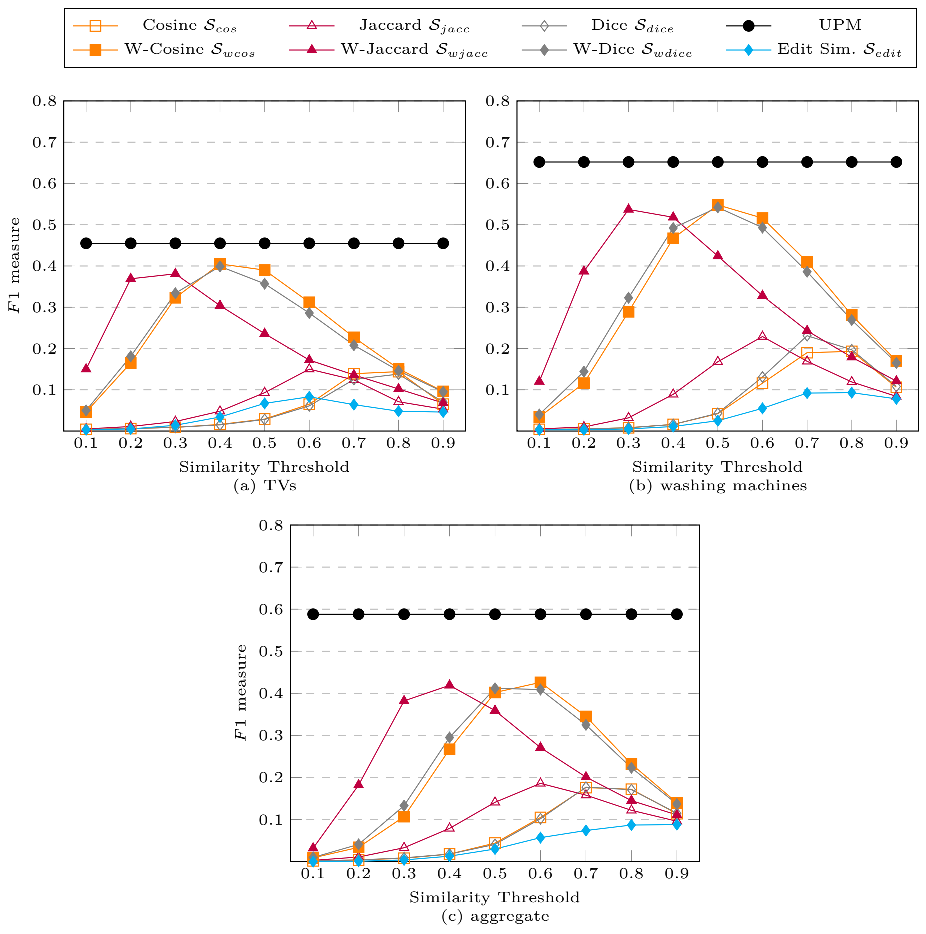

Fig. 10 Performance comparison of UPM against 7 string similarity measures for the last 3 PriceRunner datasets of Table 3.

that worths commenting is the similarity between the curves of the Cosine measure and Dice coefficient. This similarity was observed in all of our tests. However, the Cosine measure was always above Dice coefficient, even by infinitesimal margins that rarely exceeded the limits of 1–3%.

On the other hand, the smallest difference in the $F1$ scores was observed in the dataset that contained *TVs* (Fig. 10a). UPM achieved $F1 = 0.455$, roughly 12% higher than the $F1 = 0.405$ of the weighted Cosine measure. The other two methods, $\mathcal{S}_{wjacc}$ and $\mathcal{S}_{wdice}$, scored $F1 = 0.381$ and $F1 = 0.405$, respectively. Regarding the baseline similarity measures, their accuracy in this dataset was remarkably poor, as none of them achieved $F1$ values greater than 0.2.

Furthermore, UPM outperformed all the string similarity metrics in the large and heterogeneous *aggregate* dataset (Fig. 10c), since its $F1$ value was measured equal to 0.588. Compared to the value of 0.426 that was achieved by the weighted Cosine measure, it derives a significant accuracy difference of about 38%. Similarly to the previous cases, the other two weighted metrics performed slightly worse (0.419

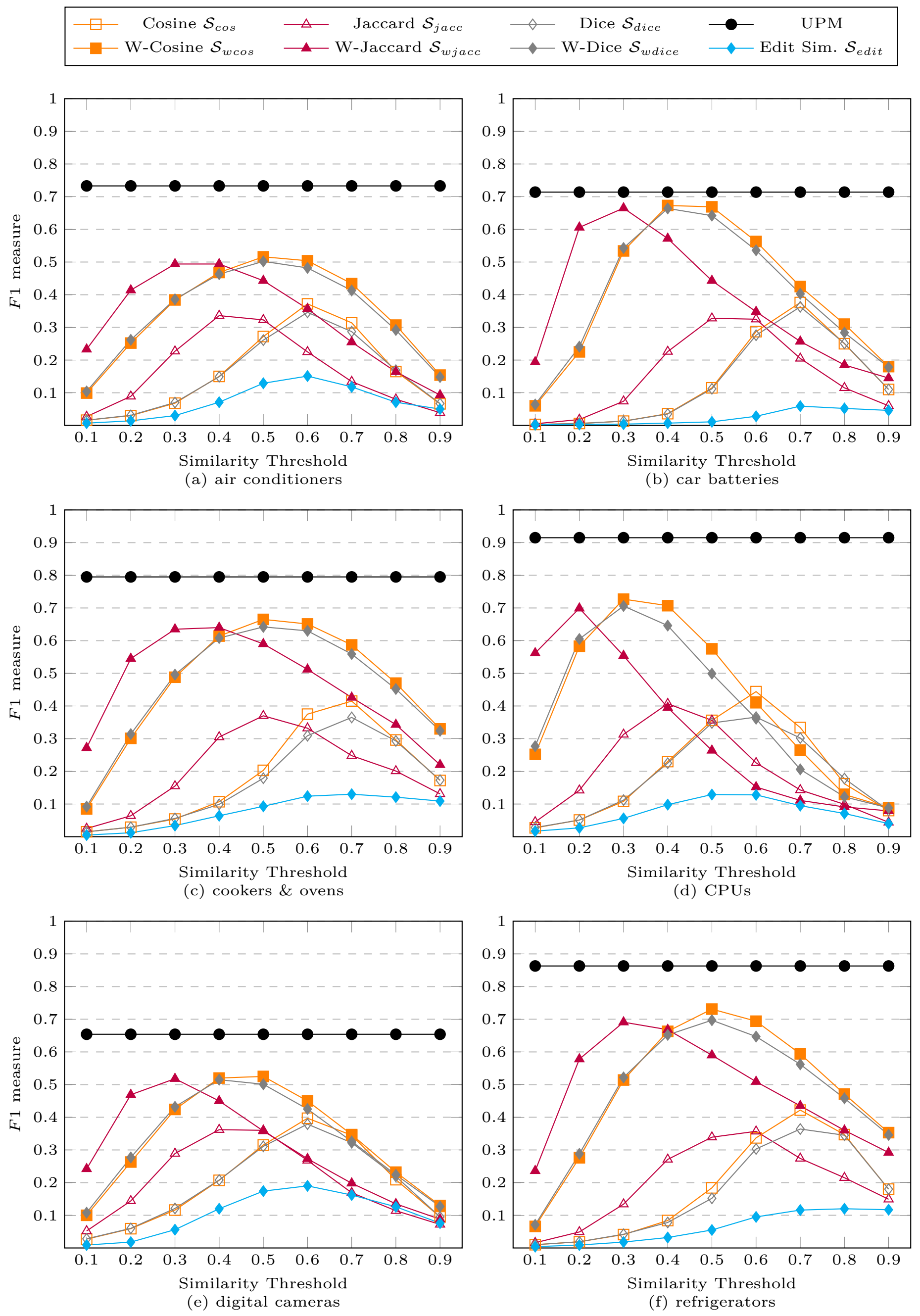

Fig. 11 Performance comparison of UPM against 7 string similarity measures for the first 6 Skroutz datasets of Table 3.

and 0.412 for $\mathcal{S}_{wjacc}$ and $\mathcal{S}_{wdice}$, respectively), whereas the baseline methods were unacceptably inaccurate; their $F1$ scores were consistently lower than 0.2.

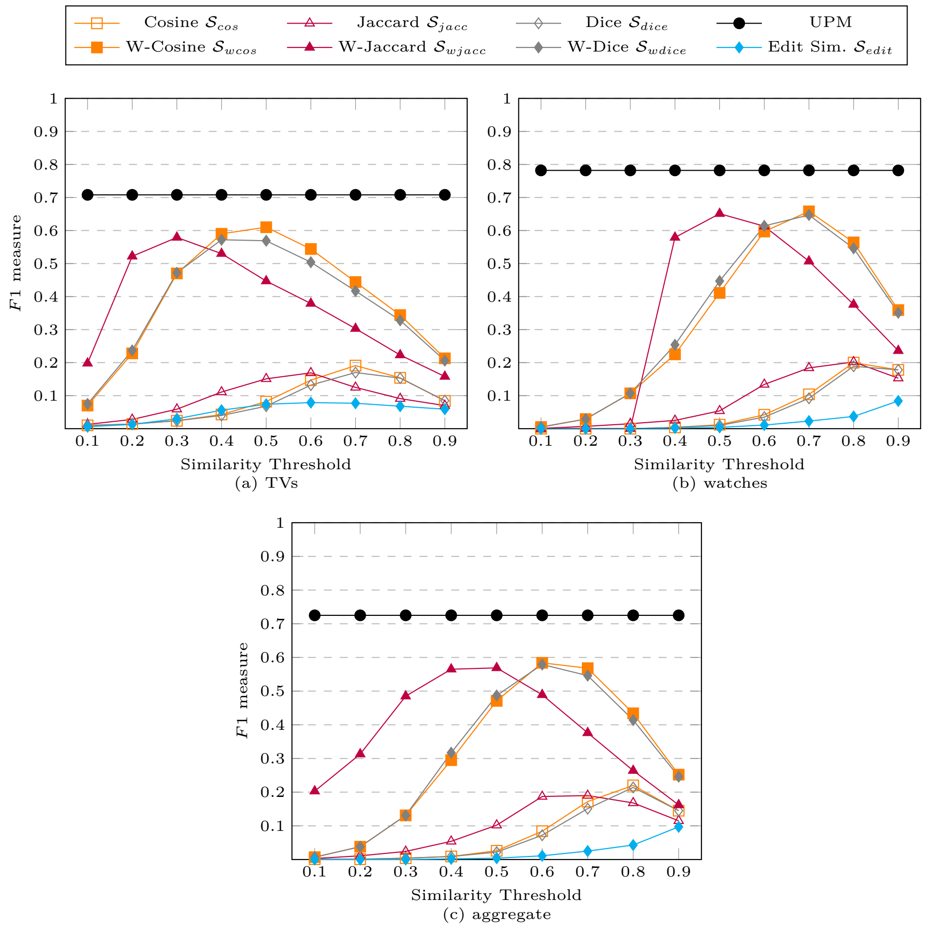

Fig. 12 Performance comparison of UPM against 7 string similarity measures for the 9 Skroutz datasets of Table 3.

These results demonstrate the superiority of UPM against their adversary methods in multiple different categories of products, including the large and heterogeneous *aggregate* dataset. The performance of UPM is improved even further in the datasets which originate from the Skroutz platform, illustrated in Figures 11 and 12). In two cases, namely, *CPUs* (Fig.11d) and *refrigerators* (Fig. 11f), UPM approached 100% precision, with its $F1$ being equal to 0.92 and 0.863, respectively. In the same datasets, the effectiveness of $\mathcal{S}_{wcos}$ was roughly 0.72 (-28%) and 0.73 (-26%), respectively, whereas $\mathcal{S}_{wjacc}$ achieved 0.70 (-31%) for *CPUs* and 0.69 (-33%) for *refrigerators*. Furthermore, $\mathcal{S}_{wdice}$ was slightly more accurate than $\mathcal{S}_{wjacc}$, since its $F1$ scores were greater by approximately 1%.

Moreover, in the case of the *aggregate* dataset, UPM outperformed the weighted measures $\mathcal{S}_{wcos}$, $\mathcal{S}_{wjacc}$ and $\mathcal{S}_{wdice}$ by approximately 24%, 28%, and 25%, respectively. The performance gap was substantially larger compared to the baseline string similarity methods $\mathcal{S}_{cos}$, $\mathcal{S}_{jacc}$ and $\mathcal{S}_{dice}$, since the $F1$ scores of UPM were greater by 3.3, 3.8, and 3.4 times, respectively.

Similarly to the clustering algorithms, the maximum accuracy of the string similarity metrics was measured for various values of τ. This observation was valid for both the baseline and the weighted versions. For example, $\mathcal{S}_{cos}$ achieved its peak performance for 5 different values of τ, ranging from 0.5 to 0.9, whereas its weighted counterpart scored the highest $F1$ values for 4 different values of τ, ranging from 0.3 to 0.6. Similar conclusions can be drawn for the rest of the opponent measures.

Four datasets, that is, *CPUs, digital cameras, refrigerators*, and *TVs* have been crawled from both product comparison platforms. The examination of all methods on these datasets leads to the conclusion that the effectiveness does not primarily depend on the category itself. Instead, it is rather affected by how accurately the vendors describe their products. For instance, the $F1$ score of UPM for the *CPUs* of PriceRunner and Skroutz was 0.657 and 0.920, respectively, showing a difference of about 40%. On the contrary, this difference was below 1% for the *digital cameras.* Moreover, UPM performed better on the *refrigerators* rather than the *CPUs* of PriceRunner, whereas the opposite occurred on the respective datasets of Skroutz.

4.4 Efficiency Evaluation

This subsection contains the experimental measurements of the efficiency of the proposed algorithm in comparison with the aforementioned clustering and string similarity methods. Since the execution times depend heavily on the employed data structures and the applied algorithms, we initially discuss in brief some crucial implementation details before the presentation of the results.

As stated earlier in Subsection 3.2.2, UPM is based on two lexicon structures L_w and L_c that are utilized to store and look-up for tokens and combinations, respectively. We implemented both lexicons by using standard hash tables with chains (linked lists) for collision resolution. Regarding the forward index R, it is merely an array of product objects. The forward lists $r_{p,w}$ and $r_{p,c}$ of each product $p \in R$ are also standard, expandable arrays of pointers that point to the corresponding entries within L_w and L_c, respectively.

The similarity between two titles t and t' is a key operation for all methods, including the proposed verification stage. To improve its efficiency, all product titles were tokenized in advance, that is, before a matching algorithm was deployed. The tokens of each title were then stored within in-memory arrays that were sorted in lexicographical order. In this way i) we avoided re-tokenizing the same title each time it was needed, and ii) we were able to compute the intersection between two titles (that is, find their common tokens) in time $O(|t| + |t'|)$, linear to the sum of the length of the titles. Without these simple actions, the computation of the value of any similarity measure between t and t' would be very expensive.

Regarding the weighted string similarity measures, the inverse document frequency (IDF) for all tokens was pre-computed and stored within a lexicon structure, whose form was very similar to that of L_w. Nonetheless, to retrieve the IDF of a token during the calculation of a similarity value, a look-up operation is required. Consequently, we expect the weighted versions of the similarity measures to be slower than the baselines, due to these look-ups.

After these necessary explanations, we proceed with the presentation of the efficiency measurements of the involved methods. Figure 13 depicts the running times (in seconds) of UPM against the 6 aforementioned clustering algorithms and the 7

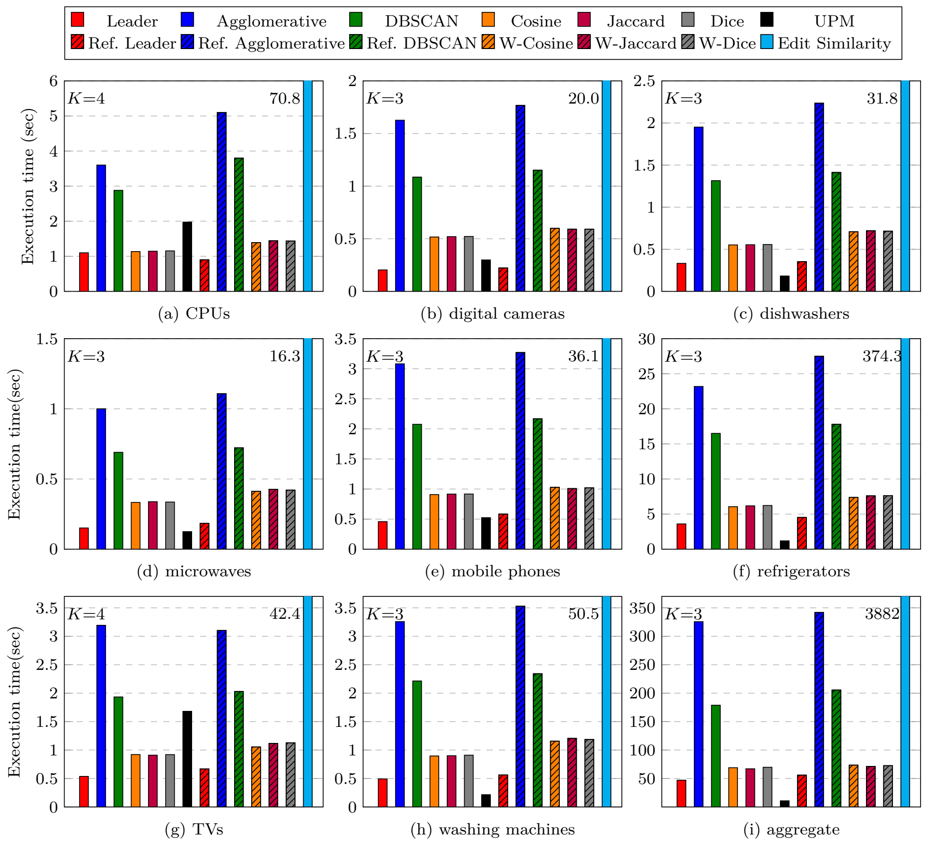

Fig. 13 Efficiency comparison of various products matching methods for the 9 PriceRunner datasets of Table 3.

string similarity measures for the 9 datasets of PriceRunner. The times presented here have been measured for the value of τ for that each method achieved its maximal performance. For example, according to Fig. 5a the peak accuracy of R-LC in the dataset which contained *CPUs* was obtained for $\tau = 0.3$; hence, the running time of this method in this specific dataset was measured for $\tau = 0.3$.

The first comment on these diagrams concerns Edit Similarity, which was much slower than the other approaches that participate in our evaluation. Presenting its high execution times in common plots with the other methods would distort the visual perception of the results. Hence, we have chosen to use the running time of the second slowest method as an upper bound for the vertical axis of the diagrams and write the execution time of Edit Similarity in the top-right part of each diagram. Moreover, in the lop-left part, we report the value of K that determines the population of the computed k-combinations and affects the running times of UPM. As stated earlier, K was set according to Eq. 8 in conjunction with the last column of Table 3.

Remarkably, the qualitative form of all 9 diagrams is nearly identical. That is, the slowest method in all cases was Edit Similarity, followed by R-HCA, HCA, R-

DBSCAN and DBSCAN. The token-based string similarity metrics were faster than all the aforementioned methods. More specifically, the 3 baseline measures (Cosine similarity, Jaccard index, and Dice coefficient) consumed roughly equal times, and this is valid for their weighted versions, too. Furthermore, the baseline and the refined Leader clustering algorithms outperformed all the other clustering methods and string similarity measures.

The element that differentiates these diagrams was the efficiency of UPM. According to the last column of Table 3, the products within the *CPUs* (Fig. 13a) and *TVs* (Fig. 13g) datasets were described by titles that were longer compared to the ones of the rest categories. Therefore, Eq. 8 dictates that we set $K = 4$ for these categories. Under these circumstances, UPM was 2–3 times slower than LC, R-LC, and the 6 token-based string similarity measures. This is anticipated, since a high value of K requires a large number of combinations to be constructed and scored.

Nevertheless, in the other cases where we set $K = 3$, UPM outperformed all the opponent methods by significant margins. The results demonstrate that the larger the dataset is, the greatest the performance gap becomes. For example, in the *refrigerators* dataset, which includes 11291 titles, UPM consumed only 1.2 seconds, and it was 3 times faster than the second faster method, namely, Leader clustering. In contrast, in the *aggregate* dataset that was more than three times larger than *refrigerators* (about 35.3 thousand titles), UPM was completed in roughly 11 seconds, that is, 4.3 times faster than LC (47 sec). This means that for $K = 3$, UPM has a better scaling compared to LC and R-LC.

Regarding the other clustering methods, Agglomerative clustering was the slowest among all the opponent methods (with the exception of Edit Similarity), since in the *aggregate* dataset, HCA and R-HCA consumed about 325 and 342 seconds, respectively. In other words, these algorithms were approximately 30–31 times slower than UPM. On the other hand, DBSCAN was considerably more efficient than Agglomerative clustering. More specifically, in the same dataset, DBSCAN and R-DBSCAN were completed in 179 and 205 seconds, respectively.

Concerning the baseline string similarity measures, $\mathcal{S}_{cos}$, $\mathcal{S}_{jacc}$, and $\mathcal{S}_{dice}$ required about 69, 67, and 70 seconds, respectively, whereas their weighted counterparts $\mathcal{S}_{wcos}$, $\mathcal{S}_{wjacc}$ and $\mathcal{S}_{wdice}$ consumed 73, 71, and 73 seconds, respectively. These increased durations can be expressed by delays of 4–6%. The precise execution times of all methods on the *aggregate* dataset of PricRunner are reported in the second column of Table 5. Moreover, the speed differences between UPM and all the opponent approaches are displayed in the third column of Table 5.

These speed differences are utilized to assist us investigate the time overhead introduced by the execution of the verification stage. The key point here is how effective the baseline clustering algorithm was in terms of matching accuracy, because this affects the number of the correcting operations. More specifically, if the initial method performs well at first place, then the refinement algorithm has only a few corrections to make, therefore, we expect it to be swift. In the opposite case, the algorithm must iterate through multiple invalid clusters and move many products to other clusters, or create numerous additional clusters.

In the *aggregate* dataset, R-LC, R-HCA, and R-DBSCAN required 56, 342, and 205 seconds, respectively, and they were slower than LC, HCA, and DBSCAN by 17%, 5%, and 14.5%, respectively. In contrast, the corresponding delays for the *refrigerators* dataset were 25%, 17%, and 8%. As stated earlier, the delay of the verification

Table 5 Efficiency comparison of various methods on the two *aggregate* datasets

Method	**PriceRunner Aggregate**		**Skroutz Aggregate**	
	Time (sec)	Gain (×)	Time (sec)	Gain (×)
UPM	11	–	635	–
Leader Clustering	47	4.3	1733	2.7
Refined Leader	56	5.1	1995	3.1
Agglomerative Clustering	325	29.5	11712	18.4
Refined Alglomerative	342	31.1	11940	18.8
DBSCAN	179	16.3	5359	8.4
Refined DBSCAN	205	18.6	6800	10.7
Cosine Measure	69	6.3	2443	3.8
Weighted Cosine	73	6.6	2655	4.2
Jaccard Index	67	6.1	2406	3.8
Weighted Jaccard	71	6.5	2625	4.1
Dice Coefficient	70	6.4	2560	4.0
Weighted Dice	73	6.6	2601	4.1
Edit Similarity	3882	352.9	125333	197.4

algorithm depends heavily on the effectiveness of the baseline clustering algorithm, a property that makes it hard to predict its duration.

Another significant point is that, in some cases, the refined version of a clustering algorithm appears to be faster than the baseline. Although this is surprising, it is explained by the fact that in Fig. 13 we present the running times for that value of τ a method achieves the best of its accuracy. However, different values of τ lead to different execution times and, as discussed in the previous subsection, the application of the verification stage may maximize the $F1$ scores of a method for different values of τ. For example, in Fig. 13a it appears that R-LC is faster than LC. Nevertheless, Fig. 5a shows that R-LC and LC achieve their peak performance for $\tau = 0.3$ and $\tau = 0.5$, respectively, and this explains the presented execution times. On the other hand, in this dataset, R-HCA and R-DBSCAN consumed 5.1 and 3.8 seconds, respectively, and they were slower than the corresponding baselines by 41.6% and 35.7%, respectively.

The efficiency measurements were also positive for our proposed algorithm in the 9 datasets that originated from Skroutz. Figure 14 contains 9 diagrams that illustrate the execution times of all methods for each dataset, and their form is similar to the one of the diagrams of Fig. 13. In particular, LC and R-LC were the strongest opponents for UPM, whereas DBSCAN and HCA (and their refined versions) were considerably slower. The three baseline token-based string similarity measures were roughly equally fast, whereas their weighted variants were slightly less efficient. Furthermore, Edit Similarity was prohibitively slow.

So, in the case of *watches*, which is the second largest dataset of our tests, UPM consumed 503 seconds and outperformed LC and R-LC by 45% and 118%, respectively. On the other hand, the three baseline similarity measures were about 2.6 times slower than UPM, and their weighted variants 2.9 times. In general, UPM was the fastest approach in 7 out of 9 datasets; it was outperformed by LC and R-LC only in the datasets that contained *air conditioners*, where it was executed with $K = 4$, and *CPUs*, where $K = 3$.

Regarding the largest dataset of our evaluation, the *aggregate* dataset, the precise running times of all methods are shown in the fourth column of Table 5 and

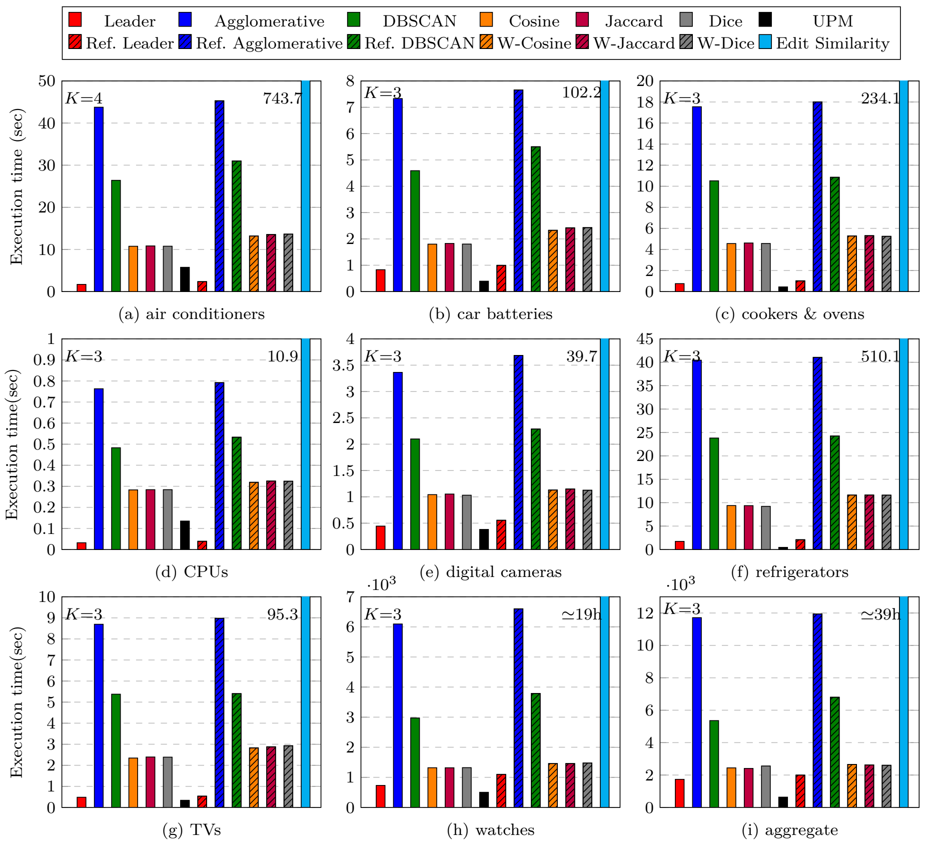

Fig. 14 Efficiency comparison of various products matching methods for the 9 Skroutz datasets of Table 3.

have been plotted in Fig 14i. The results show that UPM was considerably faster compared to all adversary approaches. More specifically, our algorithm consumed in total 635 seconds and it was about 3, 18, and 11 times faster than R-LC, R-HCA, and R-DBSCAN, respectively. In comparison with the similarity measures, UPM outperformed the 3 baseline methods by 3.8–4 times, and their weighted variants by 4.1–4.2 times.

5 Conclusions

In this paper we introduced UPM, a clustering-based unsupervised algorithm for matching product titles from different data sources. This problem is particularly important for the e-commerce industry, since it facilitates the retrieval of products and the comparison of their features and prices. UPM implements multiple novel elements, the most important of which are:

– It does not perform pairwise comparison of the titles, thus, it avoids the quadratic complexity of this procedure. Instead, it constructs variable-sized combinations

of the titles' words and scores each one of them according to several criteria, including their frequency, their average distance from the beginning of the titles, their length, etc.
- It partially identifies the semantics of the title words, namely, possible model descriptors, technical characteristics, etc., and scores them accordingly.
- It introduces *blocking conditions*, that is, logical expressions that prevent the co-existence of products with common attributes within a cluster. In this paper, we formulated a simple blocking condition which states that *"a cluster cannot contain multiple products from the same vendor"*.
- According to this condition, a post-processing verification stage is deployed with the aim of correcting the erroneous matches of the previous stages. This algorithm moves products from one cluster to another, creates new clusters and merges others. Apart from its substantial benefits in matching accuracy, its greatest advantage is that it is applicable to all clustering methods, provided that a blocking condition exists, or can be formulated.

UPM was exhaustively evaluated in terms of both matching accuracy (i.e., $F1$ scores) and efficiency (execution times), against 3 successful clustering algorithms and 7 string similarity measures. The experiments were conducted by using 18 datasets that we created by crawling two popular product comparison platforms. The results demonstrated that in all examined cases, UPM achieved a much higher product matching quality compared to the adversary approaches. With only a few exceptions, which are explainable, UPM was also the fastest method among all.

In addition, we applied the aforementioned verification algorithm to the output of the 3 clustering algorithms to examine its impact on the accuracy of these algorithms. The experiments showed that our post-processing procedure is able to refine the produced clusters and improve the performance of any clustering algorithm by a significant margin. More specifically, in some datasets, the application of the verification stage led to substantial gains that exceeded 250%.

References

Akritidis L, Bozanis P (2018) Effective Unsupervised Matching of Product Titles with k-Combinations and Permutations. In: Proceedings of the 14th IEEE International Conference on Innovations in Intelligent Systems and Applications (INISTA), pp 1–10

de Bakker M, Frasincar F, Vandic D (2013) A hybrid model words-driven approach for web product duplicate detection. In: Proceedings of the International Conference on Advanced Information Systems Engineering, pp 149–161

Bär D, Biemann C, Gurevych I, Zesch T (2012) UKP: Computing Semantic Textual Similarity by Combining Multiple Content Similarity Measures. In: Proceedings of the 1st Joint Conference on Lexical and Computational Semantics, pp 435–440

Bilenko M, Mooney RJ (2003) Adaptive Duplicate Detection using Learnable String Similarity Measures. In: Proceedings of the 9th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), pp 39–48

Chaudhuri S, Ganjam K, Ganti V, Motwani R (2003) Robust and Efficient Fuzzy Match for Online Data Cleaning. In: Proceedings of the 2003 ACM International Conference on Management of Data (SIGMOD), pp 313–324

Christen P (2008) FEBRL: a Freely Available Record Linkage System with a Graphical User Interface. In: Proceedings of the 2nd Australasian Workshop on Health Data and Knowledge Management, pp 17–25

Dhillon IS, Guan Y, Kulis B (2007) Weighted graph cuts without eigenvectors a multilevel approach. IEEE Transactions on Pattern Analysis and Machine Intelligence 29(11):1944–1957

Dunn JC (1973) A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. Journal of Cybernetics 3(3):32–57

Elmagarmid AK, Ipeirotis PG, Verykios VS (2007) Duplicate Record Detection: a Survey. IEEE Transactions on Knowledge and Data Engineering 19(1):1–16

Ester M, Kriegel HP, Sander J, Xu X, et al. (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd International Confernece on Knowledge Discovery and Data Mining (KDD), pp 226–231

Filippone M, Camastra F, Masulli F, Rovetta S (2008) A survey of kernel and spectral methods for clustering. Pattern recognition 41(1):176–190

Gomaa WH, Fahmy AA (2013) A Survey of Text Similarity Approaches. International Journal of Computer Applications 68(13)

Gopalakrishnan V, Iyengar SP, Madaan A, Rastogi R, Sengamedu S (2012) Matching product titles using web-based enrichment. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM), pp 605–614

Hua W, Wang Z, Wang H, Zheng K, Zhou X (2015) Short text understanding through lexical-semantic analysis. In: In Proceedings of the 31st IEEE International Conference on Data Engineering, pp 495–506

Islam A, Inkpen D (2008) Semantic Text Similarity using Corpus-Based Word Similarity and String Similarity. ACM Transactions on Knowledge Discovery from Data (TKDD) 2(2):10

Jain AK, Murty MN, Flynn PJ (1999) Data clustering: a review. ACM Computing Surveys (CSUR) 31(3):264–323

Köpcke H, Thor A, Thomas S, Rahm E (2012) Tailoring entity resolution for matching product offers. In: Proceedings of the 15th International Conference on Extending Database Technology, pp 545–550

Li C, Lu J, Lu Y (2008) Efficient merging and filtering algorithms for approximate string searches. In: Proceedings of the 24th IEEE International Conference on Data Engineering (ICDE), pp 257–266

Londhe N, Gopalakrishnan V, Zhang A, Ngo HQ, Srihari R (2014) Matching Titles with Cross Title Web-search Enrichment and Community Detection. Proceedings of the VLDB Endowment 7(12):1167–1178

Lu J, Lin C, Wang W, Li C, Wang H (2013) String Similarity Measures and Joins with Synonyms. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp 373–384

Lu W, Robertson S, MacFarlane A (2005) Field-weighted XML Retrieval Based on BM25. In: Proceedings of International Workshop of the Initiative for the Evaluation of XML Retrieval, pp 161–171

MacQueen J, et al. (1967) Some methods for classification and analysis of multivariate observations. Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability 1(14):281–297

Manning CD, Raghavan P, Schütze H (2008) Introduction to Information Retrieval. Cabridge University Press

Ng AY, Jordan MI, Weiss Y (2002) On spectral clustering: Analysis and an algorithm. In: Proceedings of Advances in Neural Information Processing Systems, pp 849–856

Shen W, DeRose P, Vu L, Doan A, Ramakrishnan R (2007) Source-Aware Entity Matching: a Compositional Approach. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE), pp 196–205

Sneath PH (1957) The application of computers to taxonomy. Microbiology 17(1):201–226

Sorensen TA (1948) A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. Biologiske Skrifter 5:1–34

Wang J, Li G, Fe J (2011a) Fast-join: An efficient method for fuzzy token matching based string similarity join. In: Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE), pp 458–469

Wang J, Li G, Yu JX, Feng J (2011b) Entity Matching: How Similar is Similar. Proceedings of the VLDB Endowment 4(10):622–633

Xiao C, Wang W, Lin X, Yu JX, Wang G (2011) Efficient Similarity Joins for Near-duplicate Detection. ACM Transactions on Database Systems 36(3):15

Xu R, Wunsch DC (2005) Survey of clustering algorithms. IEEE Transactions on Neural Networks 16(3):645–678