APFA: Automated Product Feature Alignment for Duplicate Detection

Nick Valstar, Flavius Frasincar*, Gianni Brauwers

Erasmus University Rotterdam, P.O. Box 1738, 3000 DR, Rotterdam, The Netherlands

Abstract

To keep up with the growing interest of using Web shops for product comparison, we have developed a method that targets the problem of product duplicate detection. If duplicates can be discovered correctly and quickly, customers can compare products in an efficient manner. We build upon the state-of-the-art Multi-component Similarity Method (MSM) for product duplicate detection by developing an automated pre-processing phase that occurs before the similarities between products are calculated. Specifically, in this prior phase the features of products are aligned between Web shops, using metrics such as the data type, coverage, and diversity of each key, as well as the distribution and used measurement units of their corresponding values. With this information, the values of these keys can be more meaningfully and efficiently employed in the process of comparing products. Applying our method to a real-world dataset of 1629 TV's across 4 Web shops, we find that we increase the speed of the product similarity phase by roughly a factor 3 due to fewer meaningless comparisons, an improved brand analyzer, and a renewed title analyzer. Moreover, in terms of quality of duplicate detection, we significantly outperform MSM with an F_1-measure of 0.746 versus 0.525.

Keywords: duplicate detection, automated pre-processing, product comparison, e-commerce

1. Introduction

Increasingly many people do their shopping via the Internet in the safe environment of their home (Thomas et al., 2014). The most important reason might be that the Internet allows people to visit several Web shops in a few minutes when deciding what product to buy. This is much less time consuming than in real-life where we have to consider travel time. Nevertheless, it is not an easy task to compare the large amounts of products from the many various Web shops. For example, due to differing ways of displaying product information, some products may look different, while they are in fact duplicates. Conversely, other products

*Corresponding author

Email addresses: nickvalstar@gmail.com (Nick Valstar), frasincar@ese.eur.nl (Flavius Frasincar), gianni.brauwers@gmail.com (Gianni Brauwers)

might appear the same, yet turn out to be different products. In this work we focus on solving this very problem, which is known as product duplicate detection.

Most of the time, with some effort, the human eye will spot that two products are the same when looking at both the product titles and product features. However, this takes a considerable amount of time (de Bakker et al., 2013), and it may not always be possible. Different Web shops may omit information, or may indeed have the same information, but present it under different names. In this case, one of the goals of Web shops, namely providing a fast and convenient way of shopping, is not achieved. For these reasons, the focus of this paper is to improve duplicate detection methods.

The context of this work is the television market. Note that in this context we are concerned with the sales of televisions, not the subscriptions for channels. Specifically, we consider only the online market for television selling. A dataset is used that contains over 1000 records of TV's obtained from four different Web shops (Amazon.com, Inc., n.d.; Best Buy Co., Inc., n.d.; Computer Nerds International, Inc., n.d.; Newegg Inc., n.d.) that each exhibit different product descriptions. The aim is to find the duplicate TV's between the considered Web shops. A product duplicate detection method can be used for Web shop aggregators.

1.1. Methods

Before discussing the related work in the next section, the general framework that the majority of duplicate detection methods follow is introduced. Note that not all methods use all the phases contained in this framework, but all obey the order defined by the framework. This explains to the reader the basics which are needed to understand the research question that is posed at the end of this section. Furthermore, one can use this framework as a reference throughout the rest of this paper.

Duplicate detection methods typically consist of three phases which are referred to as phases 1 through 3. However, in some papers a fourth phase can be distinguished which is performed before the other phases. Therefore, this phase is named phase 0. Phase 0 entails a prior inspection and processing of the data to extract key characteristics and information about the products and their features. A 'feature' is the general term used for an attribute of a product, such as 'Brand'. Such a feature can be represented by different 'keys' across Web shops. In one Web shop the key 'Brand Name' may be used, while in another it is simply called 'brand'. Thus, a process called feature alignment can be implemented in phase 0 where the objective is to find the keys between Web shops that represent the same feature. The resulting information from phase 0 (e.g., which keys align across Web shops) is used to speed up and improve the rest of the process. Each product description extracted from the various Web shops contains a set of keys and their corresponding values that describe the product features. As one might understand, comparing all products from each Web

shop with all products of all other Web shops quickly becomes infeasible due to the substantial amount of products. Phase 1 is therefore used to reduce the amount of comparisons. This is generally achieved through a process called 'blocking'. Based on some conditions, products are placed in groups called 'blocks'. Products that can never be duplicates (e.g., ones that have different brands or belong to the same shop) will not fall into the same block. Only products within each block are compared, significantly reducing the amount of comparisons that are made. Blocking is a fast method, but is not flawless with respect to accuracy.

The next phase, phase 2, consists of comparing all possible product-pairs that are in the same block, and calculating a similarity score for each comparison. Web shops use different keys that can represent the same feature (e.g., 'Brand', 'Company') with corresponding values (e.g., 'Motorola', 'HTC'). The challenge is to match these key-value pairs (KVP). Since it is unknown which keys correspond across Web shops, it is impossible to compare the corresponding values, thus one cannot compute the similarity score for a considered pair. In phase 2, solutions are proposed to address these problems. Lastly, once all relevant product-pairs are scored, phase 3 entails the actual decisions: which products are duplicates. This is often done by means of clustering techniques.

1.2. Research Question

Developing an automated phase 0 is the main contribution of this research. This pre-processing phase entails extracting additional information for the products and keys. The additional information can be used to produce new metrics based on the values of the keys, which are used to significantly improve the feature alignment step. Furthermore, the additional information can be used during phase 2 to both improve the model's ability to accurately match products and reduce the computation time of the algorithm. With this, we aim to improve upon the state-of-the-art MSM algorithm for product duplicate detection. MSM (van Bezu et al., 2015) performs significantly better than its predecessors (Vandic et al., 2012; de Bakker et al., 2013). This leads us to address the following research question:

- Can the MSM algorithm for product duplicate detection be improved in terms of F_1-measure and/or speed through the use of an improved and automated phase 0?

In order to answer this question, the following sub-questions are answered as well:

1. How can product feature alignment be automated and what is its value?
2. Can the calculation of the product similarity in phase 2 be improved by incorporating information from phase 0?

3. Will an automatic product feature alignment step improve the speed of the process?

The structure of this paper is as follows. In Section 2, a theoretical background is provided on duplicate detection of products from Web shops and we give an overview of the existing techniques. In Section 3, a detailed description is provided of the proposed techniques and the employed framework that are used to answer the research questions. Next, the performance of these methods are evaluated in Section 4. Lastly, conclusions are drawn based on these results, and directions are given for future work in Section 5.

2. Related Work

Entity resolution and duplicate detection are relevant in fields such as probabilistic data (Ayat et al., 2014), microarray analysis on gene data (Breitling et al., 2004), and entity disambiguation in various domains (Zhu & Iglesias, 2018; Fernández et al., 2012). Although many methods have already been proposed for the purpose of duplicate detection, the task remains a challenge (Hsueh et al., 2014; Monge, 2000). Furthermore, due to differences in used datasets and hardware, it is difficult to determine the optimal methods. Toolkits such as DuDe (Draisbach & Naumann, 2010) have been developed to ease the implementation and comparison of the various duplicate detection methods. The foundations of the theory of entity resolution are presented by Talburt (2010). Some work focuses on improving the effectiveness of duplicate detection, while others consider the scalability of the proposed solution. We aim to develop a framework that is both effective and scalable by reusing, extending, as well as proposing new solutions to the framework components. In this section, an overview is provided of the existing methods while adhering to the four-phase framework as described in the introduction for improved readability.

2.1. Phase 0: Data Pre-Processing

In (Elmagarmid et al., 2007), where the existing literature on duplicate detection in general is reviewed, there is a separate section on data preparation. However, this mostly deals with methods used for 'cleaning' the data such as standardization (e.g., changing "5 Feb '98" to a more uniform format "02/05/1998") and parsing (e.g., differentiating between strings, numbers, and Booleans). This makes the datasets easier to compare and therefore more useful. In addition, the concept of data heterogeneity, the occurence of systemic differences between databases, is discussed. Two different types of heterogeneity are considered, namely structural and lexical. Both of these types are encountered in our data. An example of structural heterogeneity is that two Web shops may represent the same feature with different keys (e.g., 'Brand' versus 'Company'). Lexical heterogeneity can occur when, for example, different Web shops use different

representations for the exact same values (e.g., '7 cd/m2', '7.0nit' , '7,0cd m^2'). Moreover, even within the same Web shop values are not always depicted consistently. In order to sensibly compare products between two Web shops, both these heterogeneity types need to be dealt with. Another example of research in this field that contains a pre-processing phase is provided by Verykios et al. (2000). Although, this work is also only concerned with 'cleaning' the data. Their research, just like much other research (Bilenko & Mooney, 2003; Jalbert, 2008) on duplicate detection, deals only with the second type of heterogeneity as was just described, so feature alignment is not necessary.

Many previously proposed models often do not include a phase 0 since most of the functions of phase 0 can be performed while processing the other phases. For example, van Bezu et al. (2015) and de Bakker et al. (2013) decide to do key matching 'on the fly' during product pairwise comparisons. When two products are compared, the lexical similarity between the names of two keys is calculated using q-grams (Sutinen & Tarhio, 1995). When this similarity passes a threshold, the keys are marked as similar and their corresponding values are compared and scored. This is done for all combinations of key-pairs between the two considered products. We discuss in Subsection 2.3 the details of this step. In the linguistic literature, this approach is intuitively called name matching, since the keys are matched based on the similarity of the key names. Figure 1 shows how this can fail. The first three keys are not matched, because the q-gram distance between the names of the keys is too low, although it is quite clear that the keys should match. In contrast, the last three keys are matched because the names of the keys agree sufficiently, while they should not be matched. This example is worked out later on. In an attempt to prevent such mistakes, in this work the focus is moved from name matching to description matching, where the descriptions are used to decide what to match. In our context, this means analyzing the values within keys in order to align features.

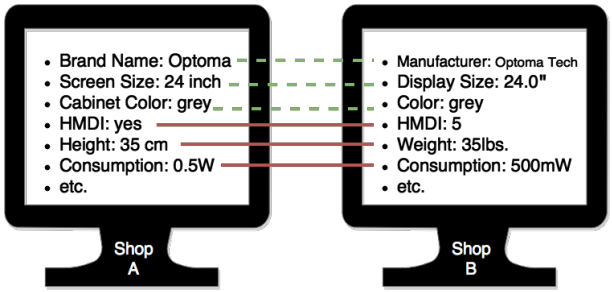

Figure 1. Illustration of how matching based only on the name of the keys can lead to incorrect key matching. Green dashed line: should be matched, but are not matched. Red full line: are matched, but should not be matched.

The idea behind incorporating a phase 0 is that the feature alignment is done beforehand and not every time for each product comparison. The first advantage is that, especially for large datasets, this saves

computational time. Instead of performing key alignment for each key-pair of each product-pair during phase 2, all key-pair matching is done beforehand during phase 0 and stored for future use. During phase 2, when comparing products, the algorithm only needs to check the previously stored results to decide which keys to compare. The second advantage is that it raises the opportunity to perform description matching. After iterating over all products in a Web shop, all values belonging to a key can be stored. After that, for a certain key, some metrics on these values (e.g., measurement units or data types) can be calculated that are used later in feature alignment. We discuss these in the rest of this section.

Nederstigt et al. (2014) use the values associated with the keys to discover the data type of said keys. To illustrate, the key named 'inputs' can in one Web shop contain string values (e.g., the kind of input: 'usb3.0', 'hdmi'), while in another Web shop the key 'inputs' can contain numeric values (e.g., the amount of inputs: '0', '4'). Even though these keys represent the 'inputs' feature, comparing them will not make sense. Yet, if one would only use name matching, the keys would certainly be matched regardless of the differing values. One can create an improved algorithm by allowing it to incorporate the additional information of the data types in its matching decisions. For the fourth example key in Figure 1, this improved algorithm would be able to detect that the data type of the left key is a string, while on the right it is a double. Consequently, this false match would be prevented. Using such information can therefore make key alignment more trustworthy. Another contribution by Nederstigt et al. (2014) is the idea of using measurement units to improve the matching process. For example, even when the data types of the key are both numeric, it is still possible that the values represent something completely different when their units do not correspond (e.g., '35 cm' versus '35lbs.'). The incorporation of measurement units would perhaps have caused the second example key in Figure 1 to be matched because of the matching unit types. Moreover, the last two keys would not be matched due to differing measurement units. Both the data types and measurement units of the keys are used by van Rooij et al. (2016) in a pre-processing phase to improve upon the MSM model. In this work, not only are both the data types and the measurement units considered as metrics, but we also use various additional metrics based on the values of the keys to obtain an even more dependable alignment method. We introduce the data distribution, coverage, diversity, and standard deviation metrics, which are explained in depth in Section 3.

2.2. Phase 1: Blocking

The goal of blocking is reducing the number of product-pairs that are to be compared. Papadakis et al. (2013) have laid the foundations of this method. The idea is to assign products to blocks based on certain conditions. To illustrate blocking, consider an easy example where we assign products to blocks

using the condition that their brand must correspond, so that in each block every product has the same brand. Consequently, products that do not have any blocks in common are not to be compared. The major advantage is that the more blocks you have, the smaller the amount of comparisons. For example, 1 large block of 50 requires $1 \times \binom{50}{2} = 1225$ product comparisons, while 5 blocks of 10 would require only $5 \times \binom{10}{2} = 225$ product comparisons. The risk, however, is that a potential duplicate ends up in different blocks due to a mistake in the Web shop (misspellings) or in the blocking selection (too strict or incorrect conditions). The probability of this occurring can be reduced by creating blocks based on more than one condition. The side effect is that candidate pairs are likely to fall in multiple blocks and thus need to be compared more than once. In turn, this issue can be dealt with by keeping track of whether a pair has already been compared or not. In general, the sizes of the blocks have no limit, while some specific practical applications do require such a limit. For example, due to privacy or scalability issues. As such, Fisher et al. (2015) propose a clustering-based framework that allows one to control the sizes of the blocks.

If the available data adheres to a certain schema, the blocking phase can be performed in a relatively straightforward manner, since one can always use the same key or set of keys for the conditions in the blocking process. However, as has been mentioned before, Web shop data is generally highly heterogeneous with no clearly defined schema. In such cases, schema-agnostic methods can be implemented that simply extract all available information from the products and use that information for the blocking process (Papadakis et al., 2015; Simonini et al., 2019).

Different methods can be used to make the blocking phase more scalable. For example, in (Kolb et al., 2012), a map-reduce algorithm is proposed for implementing the sorted neighborhood blocking method via multiple passes. Instead of scaling existing methods, other works focus on introducing more scalable approaches. For example, Vandic et al. (2020) propose a scalable approach that improves the blocking phase using model words extracted from the product titles and descriptions. Model words are words that contain alphabetic characters as well as numeric characters, such as 'TV100'. Additionally, the proposed blocking method by Vandic et al. (2020) is applicable when more than two Web shops are to be compared, which is also the case in this paper.

After blocking is finished, all products that have at least one block in common are compared with each other. From now on, pairs that are to be compared are referred to as candidate pairs. So within a block of size 10 the number of candidate pairs is $\binom{10}{2} = 45$. Note that not all works use blocking. In those cases we say, for uniformity purposes, that all products are put into a single large block. Instead of blocking, a simple condition based on the brands of products is used by van Bezu et al. (2015). During the pairwise

comparison phase only pairs with the same brand are compared. Such a heuristic can be seen as a form of blocking since it produces similar effects on the actual comparisons made.

2.3. Phase 2: Pairwise Product Comparison

Phase 2 entails comparing all products that need to be compared using a comparison algorithm. Firstly, the comparison algorithm takes as input a set of candidate pairs. Then, the algorithm computes for each candidate pair a similarity score ranging between 0 and 1. Perhaps the clearest pairwise comparison method is the HSM method proposed by de Bakker et al. (2013). HSM largely builds on TMWM developed by Vandic et al. (2012) where model words play an important role. HSM was built for only two Web shops, so pairs were immediately assigned as being duplicates and not scored for similarity. Given two products, it first checks whether the model words from the titles match. If they do, assign them as duplicates. Otherwise, loop through the keys and check if it can find keys that lexicographically match sufficiently. For matching keys, compute the similarity of the corresponding values. For all other non-matching keys, compute the similarity of all the model words of the corresponding values taken all together. If the average of the similarity scores is above a certain threshold, the products are considered duplicates.

The authors of MSM (van Bezu et al., 2015) improved upon the HSM model in several ways. First and foremost, they have extended the model from two to multiple Web shops so that a product may be duplicated across two or more shops. Consequently, product pairs are scored on their similarity, rather than immediately assigning them to clusters. The similarity score ranges between 0 and 1 with the following exception: if two products are from the same shop they are not compared and have a similarity of $-\infty$. Furthermore, in the case that two products have differing brands, a similarity score of 0 is given. A final improvement is that the very flexible q-grams measure is used for comparing keys and for comparing values. For comparing the model words of the keys that were not previously compared, the same technique as in HSM is used, namely simply calculating the percentage of model words that those keys have in common in their corresponding values.

van Dam et al. (2016) propose to use Locality-Sensitive Hashing (LSH) (Indyk & Motwani, 1998) to significantly reduce the number of product comparisons made during this phase. Based on model words extracted from the product titles, binary vector representations are produced. Vector representations are not uncommon for duplicate detection and entity resolution. For example, in (Yang et al., 2019), unsupervised feature generation models are used to produce vector embeddings of places for the purpose of place deduplication. For the LSH model, the product feature vectors are used as input such that the algorithm can pre-select product-pairs as duplicates with a high accuracy. Consequently, the number of necessary

comparisons is reduced significantly. Further extensions of this LSH method are proposed by Hartveld et al. (2018). These extensions consist of data cleaning and incorporating additional information from the key-value pairs. Compared to the MSM algorithm, Hartveld et al. (2018) present a 95% reduction in the number of computations with only a 6% reduction in the F_1-measure.

As stated earlier in Subsection 2.1, another way of making the pairwise comparison method substantially more efficient is by storing the matched keys from the pre-processing phase. The comparisons are then also likely to make more sense, since we have identified corresponding keys between Web shops. Using this knowledge about keys, we plan to treat the comparison between two numerical values differently than between two strings, instead of using q-grams for everything. As an illustration, the q-gram similarity between 1000 and 1001 is the same as between 5000 and 1000, while the differences between these values are rather dissimilar. Also, q-grams cannot properly deal with rounding ('19.9' vs. '20' gives a q-gram similarity of 0, while it perhaps should be much higher). Lastly, based on the similarity of the keys as described in Subsection 2.1 we weigh some comparisons more than others. When we are less certain that two keys are a match, we weigh their value comparison less. With these adjustments, we aim to improve upon the MSM algorithm in terms of F_1-measure.

2.4. Phase 3: Clustering

Once all candidate pairs are scored, the final phase can be performed, namely the actual assignment of duplicates, which is done via clustering. Clustering is a process in which similar data records are classified into groups (Jain et al., 1999). There is significant work on clustering techniques with applications in many different fields of research (Saxena et al., 2017). Clusters in the context of duplicate detection are groups of duplicate data records. A clustering algorithm that does not require a predefined amount of clusters must be used in the case of duplicate detection, since we do not know how many duplicates there are in total (Hassanzadeh et al., 2009). Elmagarmid et al. (2007) and Hassanzadeh et al. (2009) provide an overview of clustering techniques specifically for duplicate detection. However, the main issue with previous solutions is that they often take a database perspective that includes structured data. Since we work with semi-structured Web data, the phases explained in the previous subsections are necessary before a clustering algorithm can be used.

MSM (van Bezu et al., 2015) uses agglomerative hierarchical clustering (Tan et al., 2006). All products start in their own cluster of size one, after which clusters that are sufficiently similar are merged based on a certain linkage strategy. MSM uses single linkage, where the distance between two clusters is defined as the distance between the closest two points of the clusters. The distance between products is measured as 1

minus the similarity score, which is also known as the dissimilarity. The dissimilarity ranges between 0 and 1, with the exception of ∞ as introduced earlier. It is not a proper mathematical distance function, since a distance of 0 does not necessarily mean that the products are the same. Of course, the closer it is to 0, the higher is the probability that the products are duplicates. The intuitive downside of this method is that each time it only looks at one product of a cluster (the one most similar) while the rest is ignored, even though they might be very different. This can cause products with very low similarity to end up together. An exception can be made to this regard. Namely, when one of the distances between the points in two clusters is set equal to ∞, complete linkage can be used. Complete linkage simply calculates the largest distance between the clusters, rather than the smallest. In that case, they will not be merged, which has solved the issue. The downside of complete clustering is that one product (the furthest) has all the influence in the merging process and the rest is ignored, while these might be very similar to each other. A third method is the so-called average linkage, which is defined as the average of all the distances between all possible points between the clusters. When one of the distances is set to ∞, the average distance is automatically set to ∞ as well. This method creates a balance between the aforementioned downsides of single- and complete linkage.

3. Methodology

The structure of the methodology is based on the order of the posed research questions. First, in Subsection 3.1, a novel pre-processing algorithm is presented that matches product features between Web shops. Second, in Subsection 3.2, an algorithm is proposed that performs pairwise comparison similar to MSM (van Bezu et al., 2015), while incorporating information from the improved pre-processing step. Figure 2 illustrates the various steps of APFA that are explained in this section.

After developing the proposed steps, we implement the two algorithms into a framework so that it can be tested and optimized. By a framework we mean the environment in which the algorithms are run and the evaluation is performed. The framework that is used was proposed by Vandic et al. (2020). The programming languages Java and Scala are used in the framework, while Spark is employed for distributed computing.

Due to space limitations, the pseudocode of all proposed algorithms throughout this section can be found in the technical report, which can be accessed online (Valstar & Frasincar, 2019). The software (including the data) is made available in GitHub (Valstar & Frasincar, 2019).

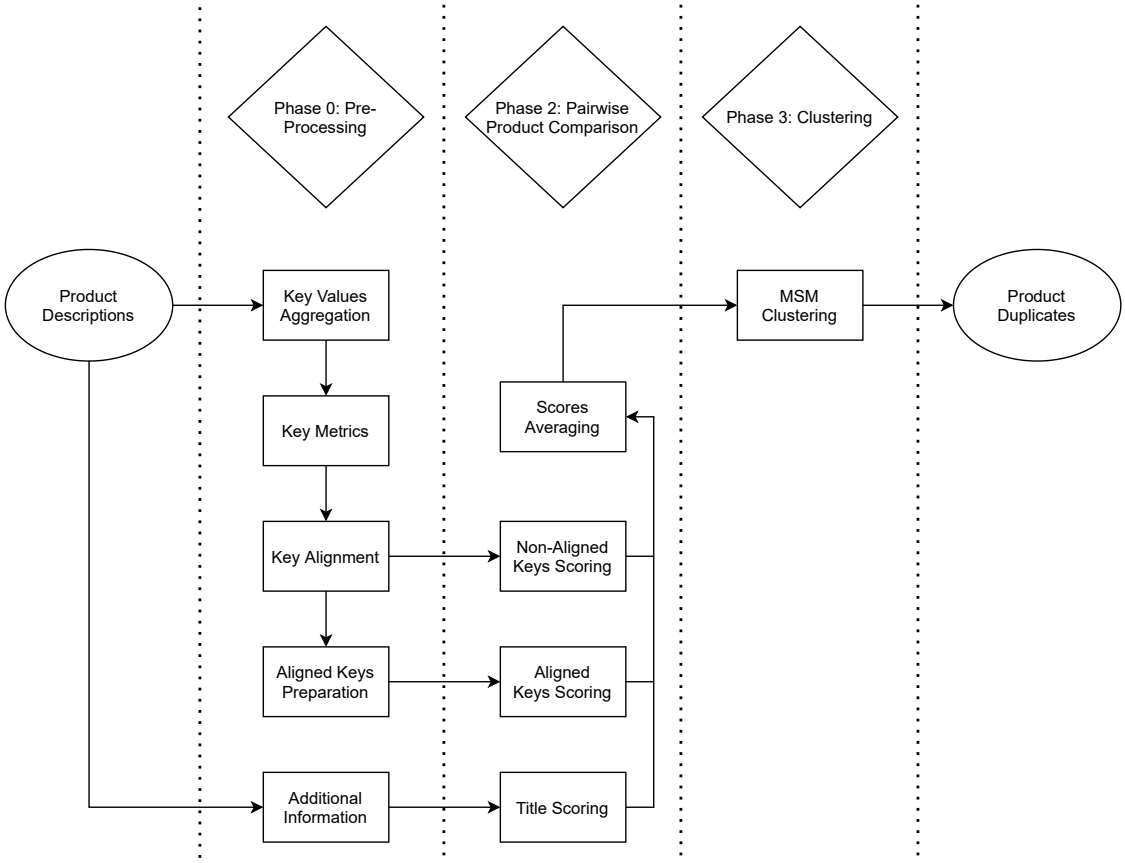

Figure 2. A flowchart illustrating the various processing steps utilized in APFA. Note that phase 1 (blocking) is missing since it is also not used in MSM.

3.1. Phase 0: Pre-Processing

In order to answer the first research subquestion, an automatic feature alignment phase is developed. In the previous section, it was explained how keys are often matched while they should not be matched, and other keys are not matched while they should be. This is caused by the fact that, in earlier works, merely the names of the keys are used in order to detect key matches. In this section, we present several other metrics based on the values of the keys that can be used for key matching in addition to the key names.

In Subsection 3.1.1, it is explained how all values per key in one Web shop are aggregated in order to extract valuable information. Next, in Subsection 3.1.2, it is explained how the various metrics for the keys are extracted. Using these metrics, in Subsection 3.1.3, it is demonstrated how the keys between the different Web shops are matched. These matched keys are then enriched with information, as shown in Subsection 3.1.4. Finally, a brand-analyzer and title-analyzer are presented in Subsection 3.1.5 that are designed to extract additional information about each individual product independent of the key alignments. Both the

product information, as well as the key alignments, are then used further on in Phase 2: Pairwise Product Comparison, presented in Subsection 3.2.

3.1.1. Value Aggregation per Key

We have argued earlier that metrics based on the values of the keys in addition to the name of the key, may provide valuable information for feature alignment. In this step, we propose a simple algorithm that iterates over all the products and stores all the used keys per Web shop. More importantly, for these keys, a list of all used values that correspond to a key is stored. For now, all values are seen as strings, and no processing is done. These values are used to calculate metrics in the next section. Figure 3 shows a visualization of the inputs and outputs of the algorithm.

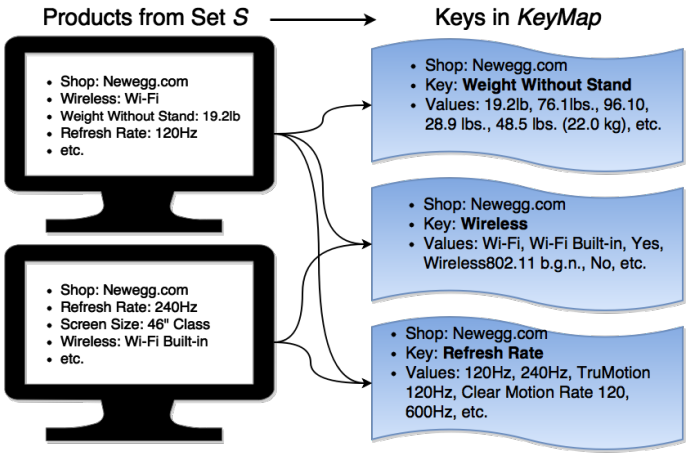

Figure 3. Value aggregation per key. From the products per Web shop, all keys are collected containing all their corresponding values.

3.1.2. Adding Metrics per Key

Using the aggregated raw values for each key, we calculate certain metrics. These metrics are used in the product feature alignment phase in Subsection 3.1.3. In addition to the data type and measurement unit metrics used by van Rooij et al. (2016), we propose four new key metrics: data distribution, coverage, diversity, and standard deviation.

Before doing any calculations, we clean the raw values of each key and split them. Raw values (e.g., 'LCD HD', '40inch × 22.3inch') often consist of several parts which we refer to as 'blocks'. Note that this is separate from the blocking phase for duplicate detection. The blocks are separated by spaces, commas, composite symbols (e.g., 'x', '+'), and other punctuation marks. For the ('LCD HD', '40inch × 22.3inch') example, the blocks would be ('LCD', 'HD', '40inch', '22.3inch'). When, after splitting, a block contains

both numeric and non-numeric characters (e.g., '40inch'), another split is made so that each numeric part forms a block, and each non-numeric part forms a block. For example, splitting ('40inch') would result in ('40', 'inch'). Figure 4 shows what this looks like for the example above. The colored cubes at the bottom are the final blocks. The coloring is explained later in this section. All blocks from all values of a key are taken together and are treated equally. The first three metrics are calculated based on these collected blocks.

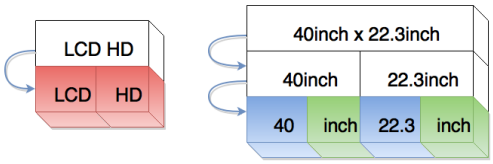

Figure 4. Blocking. Raw values are split into blocks.

Each block is now categorized into a data type, which is a two step process. Figure 5 visualizes this process. Recall that a block by construction must contain either only numeric characters or only non-numeric characters. Thus, blocks can be classified as quantitative or qualitative. The values of quantitative blocks are then simply converted into a 'double' value, or in some programming languages referred to as a float (e.g., '1', '2', '5.5', '-10.25'). On the other hand, for each qualitative block, it is checked whether the value occurs in a unit measurement list. If it does, the block is denoted as being a unit measure. All other qualitative blocks are classified as a so-called 'string'. More details on unit measurements are provided later in this section.

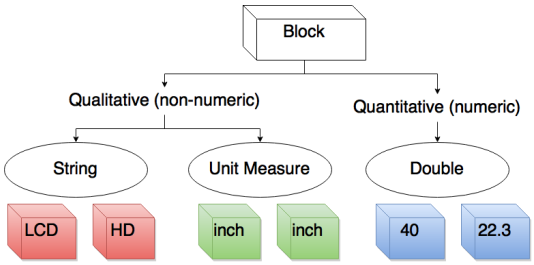

Figure 5. Block data type hierarchy. Oval shapes are data types. At the bottom there are examples of the colored blocks from the previous figure that correspond to these data types.

Summarizing, per key, all values are split into blocks and each block is classified into a data type. From the data types of these blocks, the data type of the key is decided. A key can be either a string or a double. For doubles, the dimensionality is also specified as is shown in Figure 6. When a key is of type double, it means that it describes a quantitative value, such that many of its values look like '2', but also values such as '180Hz'. The latter, although not purely numeric, is clearly describing a quantitative class, which is why unit measures are employed for blocks. Intuitively, when a key has a lot values with blocks of data

type ‘unit measure’, the key is likely to describe something quantitative. This reasoning brings us to the following: if for a key the number of ‘double’ blocks plus the number of ‘unit measure’ blocks is larger than the number of ‘string’ blocks, the key is regarded as a double, and otherwise as a string. By default, keys of type double are one-dimensional, but when the large majority (>90%) of the values of a key share the pattern of including one or two single composite signs (‘X’, ‘x’) between values, the key is considered to be two- or three-dimensional. For example, (‘40inch × 22.3inch’) would indicate a two-dimensional key.

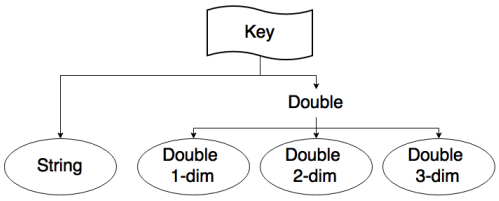

Figure 6. Key data type hierarchy. Oval shapes are data types.

The measurement unit list is provided online (List with Measurement Units, n.d.) and contains the most commonly used measurement units. Such a list can be easily found and implemented in any context, making it an elegant way of incorporating prior knowledge into the automated framework. Units are not always represented similarly between Web shops, or even within Web shops. This is dealt with by translating the raw units into a canonical representation (e.g., ‘lb’, ‘lbs.’, ’pound’, ‘pounds’ all become ‘pounds’). See Table 1 for conversion examples. The canonical representation of the unit that occurs most often in the blocks of a key is marked as the unit of that key. To illustrate, consider a key that has the following counts for (raw) unit measurement blocks: 50 blocks of ‘lb’, 50 blocks of ‘lbs.’, 50 blocks of ‘pound’, and 75 blocks of ‘watt’. Then, the canonical form ‘pounds’ has the majority with 150 blocks and thus will be denoted as the unit measure of the considered key. Thus, the canonical form can now be used instead of the raw units, so that in phase 0 keys may be matched more precisely. Furthermore, in phase 2 values of keys may be compared when they share the same unit.

Table 1 List of Units. Raw units are translated into a canonical form.

Canonical Representation	(raw) Units			
pounds	lb	lbs.	pound	pounds
nit	cd/m2	cd-m2	nit	
inch	inches	″	inch	
watt	w	watt	watts	
hours	hour	hours	hrs	
degrees	degree	°	degrees	
etc.				

The next metric is the data distribution metric. The data distribution is a slightly different metric, since it is not calculated individually per key. Instead, once in the key matching phase, when comparing two keys that are classified as 'double', we test using a standard t-test whether their respective sets of double values can originate from the same distribution. For example, {'0.3','0.5','1.0'} will not likely come from the same distribution as {'10','14','28'}. We deem a t-distribution valid, as the samples are usually larger than 100 values and there is no obvious reason for another distribution. The assumption is that when the values of two keys are distributed very differently, there are 2 possibilities: the keys do not represent the same feature or the feature is not represented in the same manner. We discuss both of these possibilities. An exception to the first possibility would be the case that one Web shop would sell only large TV's and another only small TV's. However, this is not very likely as shops typically try to offer their products to a wide range of customers. For the second possibility, the keys do represent the same feature, while having different distributions. This is for example possible when the keys are represented in different units. When the units are given, the unit metric will catch this. However, when the units are not given, the distribution metric will correctly reject this key-pair to be matched. To clarify, consider the following: ('50cm' versus '0.5m') will be rejected by the unit metric and the distribution metric, while ('50','60' versus '0.5','0.6') will only be rejected by the distribution metric. Although 50cm is equal to 0.5m, keep in mind that in this case 'rejected' means that the keys should not be matched, because comparing their values is incorrect. On the other hand, ('500W' versus '500Hz') will just be rejected by the unit metric. Therefore, both of these metrics are needed. Note that by 'rejected' we mean that the key-pair will get a negative score for that metric and therefore will probably not be denoted as a matching key-pair. In the next subsection, we go into more detail on scoring key-pairs for the purpose of feature alignment.

Lastly, diversity, coverage, and standard deviation are relatively straightforward metrics. Diversity is

the number of unique values a key has. To illustrate, a day of the week will have a diversity of 7, while a month entry will have a diversity of 12. The idea is that when two keys have contrasting/similar diversities, we can adjust the similarity score of the key-pair accordingly. On the other hand, diversity may also be used for the importance of the matched feature. A diversity of 1 (e.g., all values are 'yes') does not add any discriminating power when comparing corresponding keys. Such keys can be left out, which is inspired by (Koller & Sahami, 1997) where the selection of important features is stressed. We deliberately choose to consider the raw values as to maintain a fair comparison between Web shops. When splitting the values into blocks, this may increase the number of unique values more for a Web shop that uses values that consist of several blocks, rather than just of one block. Coverage of a key is the proportion of products of a Web shop that have that particular key included in their description. This may therefore be calculated as the number of raw values a key has, divided by the number of products a Web shop has. Important features such as brand will always be included in the description of the product, while details such as shipping size will only be included occasionally, either because it is not known for that product or it is left out by accident. Whatever the reason, it may suggest that such a key is not as important as a key that is always included. This metric may therefore also help to prevent aligning keys between Web shops that have very different coverages and therefore probably not the same meaning. Finally, the standard deviation (spread around the mean) is calculated based on the stripped double values of the key. This last metric is used in phase 2 when comparing values of matched keys, and more details follow in Subsection 3.2. Figure 7 shows the entire process for one of the keys from the example used in Figure 3 in the previous section.

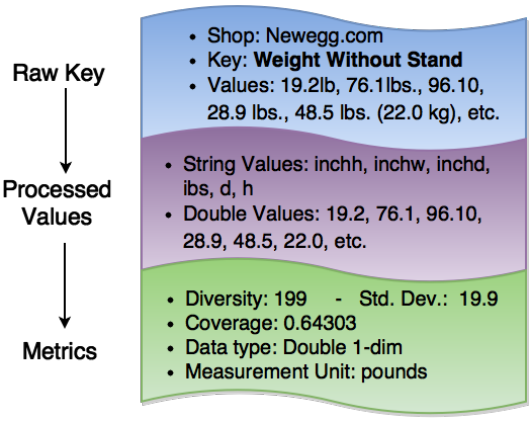

Figure 7. Adding metrics per key. Processes the raw values of a *key* and calculates metrics. Note that for reasons explained earlier, detected units (in this case: 'lb','pound','lbs.') are not included in 'String Values'.

3.1.3. Alignment of Keys between Web shops

The final stage of phase 0 is the actual matching of the keys. All the information of each key that has been previously gathered is incorporated in this process. This includes the name of the keys, the processed string- and double-values, and the calculated metrics. Technically, the name of the Web shop of a key also plays a role in key matching since keys within the same Web shop should not be matched. This is because products within the same Web shop are not to be compared, following the assumption made by van Bezu et al. (2015) that Web shops do not contain duplicates.

As stated before, MSM does not employ a separate phase for key matching, but does this 'on the fly' during product comparison. We now explain how this approach works and compare this with our approach. A first difference is that, in MSM, key matching occurs every time when two products are compared, while in our approach, key matching happens only once beforehand. The second difference is that, in key matching, MSM uses only information about those two products, while our model uses information about keys gathered from all products, due to the metrics being based on the cumulative information in the keys.

We now elaborate on how the score between two keys is calculated. In MSM, the q-gram similarity between the names of the keys is calculated. When this similarity is higher than a pre-specified threshold, the keys are matched. In this paper, this is just a small part of the scoring, since the score is also based on the processed string- and double-values, and the previously calculated metrics. Each of these scores that the total similarity score consists of will now be discussed. Recall that the input for the matching algorithm is a pair of keys from different shops, where a key has the information as visualized in Figure 7.

The first step in the matching of two keys is to check the data types. When the data types do not agree, the algorithm immediately rejects this key-pair (e.g., one of the keys is of type 'String' while the other is of type 'Double 2-dim'). These keys most likely do not represent the same feature. Even if they do, when arriving in phase 2, the comparison between a double value and a string value will give a delusive answer. Thus, it is better not to compare products based on this key at all. The same holds for different dimensionalities. Note that, technically, the keys are not scored based on data types, but this step merely acts as a test whether this key-pair should be scored and matched at all.

Just as in MSM, the lexical similarity of the names of the keys based on the q-gram measure is now calculated. This score, ranging from 0 to 1, is used in the calculation of the final score. Furthermore, this score also acts as a test, since it is required to pass a certain predefined threshold. The reasoning is that even when a pair has a high similarity score overall, we are hesitant to match these keys if the names of the keys are not alike. In order to evaluate whether this hesitation is appropriate, the model is also implemented

with a threshold parameter set to 0 to compare the results. One more test that is performed is when the model checks whether one of the names is contained in the other name. This test is necessary since in many cases it means that one Web shop is merely more elaborate than the other on the used terminology. As such, both keys still denote the same feature. Often, however, such pairs score low using q-grams (e.g., 'Weight' vs. 'Weight (Approximate)' scores 0.42, 'Labor' vs. 'Labor warranty' scores 0.47), so that these key-pair may not pass the threshold. Therefore, a set minimum score is given when one key name is contained in the other. This minimum score is a parameter that is trained using a grid-search. Note that when a name is contained in the other, it is still allowed to score higher than the minimum. This occurs when the names are very similar (e.g., "Length" vs. "Length cm", "Size" vs. "Sizes", "Res max" vs. "Res max.").

The next part of the similarity score is the part determined by the coverage metric. The coverage score is calculated by taking the negative squared error of the difference of their coverages, as can be seen in Equation 1. The closer the coverages of the keys, the higher the score, ranging between -1 and 0. The reasoning behind taking squares is that it results in a relatively smaller punishment when the coverages are quite similar. This coverage score is used, because we reason that a key with high/low coverage is deemed more/less important by the Web shop. We assume that this works in a similar manner across Web shops. Note that the size of the Web shop is taken into account, since coverage has been divided by the number of products in the shop. For the diversity metric, the model punishes pairs with a negative score when at least one key has a diversity value of one. When there is no variation in the values of a key of a certain Web shop, there is little value in using such a key.

$$covScore = -(key1.coverage - key2.coverage)^2 \tag{1}$$

$$divScore = \begin{cases} -1 & key1.diversity == 1 \; OR \; key2.diversity == 1 \\ 0 & Otherwise \end{cases} \tag{2}$$

The similarity of the values is calculated differently when dealing with strings or doubles. In case of strings, a variant of the Jaccard (Phillips, 2013) similarity as seen in Equation 3 is used, where A and B are sets of strings. This formula is adjusted so that instead of dividing by the amount of unique strings the sets have in total, they are divided by the number of strings the smallest set has, resulting in Equation 4. This is done to avoid penalizing bigger shops that have more different products, and therefore more different unique values, which results in larger sets of strings. Comparing a large shop with a small shop would then always result in a low Jaccard similarity. Finally, Equation 5 is used to calculate the score for the string

similarity, using the (adjusted) Jaccard similarity as just described.

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3}$$

$$\text{Jaccard adjusted}(A, B) = \frac{|A \cap B|}{\min\{|A|, |B|\}} \tag{4}$$

$$stringScore = \text{jaccardSimilarity}(key1.stringValues,\ key2.stringValues) \tag{5}$$

When dealing with doubles, a paired, 2-sample t-test is performed to test whether the sets of double values could have originated from the same distribution. Only the first dimension is considered even when the values are two- or three-dimensional, so that the comparison is fair. This results in a score between 0 and 1, where 0 means that the probability that these sets of values came from the same distribution is very small, and 1 indicates a high probability. When this probability is high, we reason that these keys are likely describing the same feature, since their values are distributed similarly. When the sets are not distributed similarly, but nevertheless have multiple values in common, we still want to give a high score. Therefore, we also calculate the (adjusted) Jaccard similarity and take the maximum of both measures.

$$\begin{aligned} doubleScore = \max\{&\text{p-value-t-test}(key1.doubleValues,\ key2.doubleValues), \\ &\text{jaccardSimilarity}(key1.doubleValues,\ key2.doubleValues\} \end{aligned} \tag{6}$$

The final metric is based on the measurement unit, which is scored according to Equation 7. When both keys have the same unit, a positive score is awarded, because this makes it more likely that they refer to the same feature. When the units are not alike, this strongly suggests the features differ, so a negative score is given. In the case that one or both units are missing, the two features receive a zero score. Due to the fact that units are generally used more often in combination with doubles than with strings, the final measurement score score is adjusted as to maintain a fair comparison. Therefore, all strings are given a bonus score, as illustrated in Equation 8.

$$unitScore = \begin{cases} 0 & key1.unit == \text{`}none\text{'}\ OR\ key2.unit == \text{`}none\text{'} \\ 1 & key1.unit == key2.unit \\ -1 & Otherwise \end{cases} \tag{7}$$

$$isString = \begin{cases} 1 & key1.datatype == `String' \ AND \ key2.datatype == `String' \\ 0 & key1.datatype == `Double' \ AND \ key2.datatype == `Double' \end{cases} \quad (8)$$

The final total score is a weighted average of all the previously discussed scores. To ensure optimal weighting, parameters are introduced. Based on multiple training sets, these parameters are trained and optimal values are found. This is further explained in Section 4. Finally, Equation 9 gives the formula used for the calculation of the final score between two keys. The final score has to pass a threshold (*similarityThreshold*) in order for a key-pair to be a candidate for being a match. We say candidate, since it will not be matched when one of either keys has a higher final score with another key.

$$finalScore = \sum_{i=1}^{7} \alpha_i \beta_j$$

$$\alpha = \{keyWeight, doubleWeight, stringWeight, covWeight, divWeight, unitWeight, isString\}$$

$$\beta = \{keyScore, doubleScore, stringScore, covScore, divScore, unitScore, stringBonus\} \quad (9)$$

Summarizing, for any combination of two Web shops, a score is calculated for each of the combinations of keys, after which the pair with the highest score is assigned as being a 'match'. Next, both keys of that pair are removed and the process is repeated from the beginning for these shops. This stops when the highest score does not reach a certain threshold. When that happens, the respective pair is not marked as a match, and the algorithm continues with another combination of Web shops. All found matches between each combination of shops are stored in a variable called *Alignments*.

3.1.4. Preparation of Alignments

In order to incorporate the information obtained up until now, the aligned keys from *Alignments* are to be enriched first, just as was done to the keys from *KeyMap* in Subsection 3.1.2. Recall that the output from the last algorithm in the previous section is the variable *Alignments* which contains pairs of aligned keys including their scores. The keys (as can be seen in Figure 7) contain information that was used for matching, but not all their information is useful in phase 2. At that stage, when comparing products, we focus on the individual values belonging to the products of the keys that are to be compared. We are no longer interested in the collective values that have been aggregated earlier, such as all the raw, double and string values. For each matched key-pair, only the name of the respective keys, the Web shops, all used units per key, the minimum standard deviation, the mutual data type, and the total final score are kept. In

the next section, it is revealed to what purpose these metrics are kept.

3.1.5. Additional Information per Product

In addition to the keys and metrics explained previously, information about the brand and title of the products is also extracted. In this section, a concise overview of the brand-analyzer and title-analyzer is provided. For the intricate details, the interested reader can refer to the technical report (Valstar & Frasincar, 2019).

An important feature for any product is the brand. It can be considered a good indicator of products being duplicates, since products with differing brands will generally not be the same. However, the brand is not always a feature that is included in a product listing. As such, a brand-analyzer is employed to identify the brand for each product. The brand-analyzer is an improved version of the brand extraction process of MSM. It essentially consists of extracting the brand from the product title and/or identifying a 'brand key' for each web shop and using that to extract the brands of the products.

While the brand-analyzer only extracts the brand from the title, additional valuable information can be found. A title provides a quick overview of the product that may include information about important product features. In the context of TV's, examples of such features are resolution, size and sharpness. Similarly to the blocking procedure previously described for the extracted keys, the title-analyzer extracts model-words from the title of a product, and splits them up into numeric and non-numeric parts. If the model identifies a unit in the non-numeric part, it is converted to a canonical unit representation, similar to before. The extracted features are stored for later use during the pairwise product comparison phase.

3.2. Phase 2: Pairwise Product Comparison

We move on to phase 2 now, deliberately not going into phase 1. Phase 1, the blocking phase, has not been used by van Bezu et al. (2015), so for fair comparison it is not used here either. Now that the keys are aligned, we continue to subquestion 2 and develop an algorithm for product comparison while incorporating information from phase 0. Four separate steps are implemented in this phase. A pair of products is scored on the values of their matching keys in a manner conforming to the information we have on these matching keys (*Step a*). Next, the product-pair is also scored on the values of the keys that were not aligned (*Step b*) and on their titles (*Step c*). Finally, a weighted average is calculated and is assigned to the product-pair in consideration (*Step d*). In the remainder of this section, detailed descriptions of these steps are provided, while the changes that have been made compared to MSM (van Bezu et al., 2015) are also explained.

In this phase, all combinations of two products are compared and scored, resulting in a large amount of comparisons ($\binom{n}{2}$) and thus computational time. When linearly increasing the number of products, the amount of comparisons grows quadratically for large n ($\binom{a*n}{2}/\binom{n}{2} \rightarrow a^2$). In order to keep this feasible, two heuristics are implemented that were used in MSM as well, namely the shop and brand heuristic. If two products are from the same shop, they are not to be compared and get a similarity of $-\infty$. In the case that two products have differing brands, it is highly unlikely that these are duplicates, so a similarity of 0 is given, and no further comparison is done. Depending on the clustering algorithm of phase 3, a pair with similarity 0 can occasionally end up together as duplicates, which is not the case with $-\infty$. Consequently, we are a bit more flexible with non-matching brands, as this can be due to misspelling, while shops usually do not have duplicates. Note that the shop is always provided for each product, and the brand has been extracted from the features and title of each product as has been previously described in Subsection 3.1.5.

Only when both conditions of the above heuristics are met, we continue to *Step a*: scoring the product-pair on the values of their matching keys. Two different scores are involved for each KVP (key-value pair), namely the key-score and the value-score. The former quantifies the certainty that the keys actually represent the same feature, and the latter is the similarity of the values. In the calculation of both scores there are major differences between MSM and our proposed approach. We discuss these now.

For the key matching, recall that in MSM the keys are matched 'on the fly' during the comparison of two products. All combinations of keys between the two products are scored based on the lexical similarity of their names, keeping only those key-pairs with sufficient key-score. A key may be matched with multiple other keys. In our approach, the key matching has already been performed in phase 0. Keys are not matched merely based on their lexical similarity, but also on their collective values, units, data types, and more. Keys cannot be matched multiple times. In practice, given two products from two different Web shops, all key-pairs that have been labeled in phase 0 to be matches between these two Web shops are retrieved. Those key-pairs for which both keys appear in these specific products are kept. Recall that it is not always possible to use all labeled key-pairs, as many products have missing key-value pairs. Now, only for the remaining matching key-pairs, a value-score is calculated.

When scoring the values of key-value pairs for the purpose of product scoring, in MSM the value-score is simply calculated using q-grams on their values, where, besides cleaning for some punctuation marks, no processing is done. In contrast, in our approach, we make a distinction between data types and treat them differently. Recall that in key matching we used strings and doubles, where the latter is further specified to be one-, two- or three-dimensional (e.g., '*10*x*5*x*2 cm*^{3}'.). We now briefly discuss how keys that are strings

are dealt with, and after that we more elaborately discuss doubles.

When dealing with a key-pair that has been denoted to be a string, consider the following three values of the 'USB' input of Newegg.com: 'Yes', 'USB 2.0 (JPEG, MPEG-4/DivX HD)', and '1 (Side)', where we encounter, respectively, a Boolean, a qualitative, and a quantitative value. It is not only difficult to compare such values, but meaningless as well. Therefore, we do not score key-pairs on their string similarity.

When the data type is a one-dimensional double, an attempt is made to extract the double value by processing the raw value and converting it to a double. We now explain how the processing of a double works, and how two doubles are scored. The first part of the processing is removing everything between brackets, because although it may be useful additional information for the customer, it is not essential for our task. Consider for example the following value for the key 'Output Power' from *Newegg*: '*20W (10W + 10W, THD 10%)*', where obviously we only wish to extract the '*20W*', or even better only '*20*'. The latter can be said because we may assume that the keys that are to be compared have values using the same units of measurement, which the previously discussed key matching algorithm has made sure of. This brings us to the second processing step, namely stripping the value of all measurement units that have been used for that key in that specific Web shop. The third processing step deals with composite values. Some Web shops are more detailed than others, making it more difficult to compare their values. To illustrate this, consider again the key 'Output Power' and compare a product from *Newegg* with a product from *Bestbuy*, having for a certain product-pair respective values '*7W + 7W*' and '*14W*', quite certainly representing the same audio output which naturally consists of a left and a right audiobox. When our method encounters a '+' sign, the sum is calculated. As of now, we have not included other composite signs, leaving this to be an interesting direction for future work. The final step is simply extracting the numeric part from what is left so that it can be converted to a double. These four processing steps help identify duplicates as well as non-duplicates, because each step attempts to assure a fair comparison between values.

Recall that due to inconsistency within Web shops, even when a key has been appointed to be of type double, it does not imply that this holds for all products, meaning it is not always possible to extract a double value for both keys. The compiler of the programming software is able to check whether a value is numeric or not. In those unfortunate cases that it is not a double while it should be, it is better to not consider this key-pair at all, so that it neither has a positive nor negative contribution to the scoring of these products. To illustrate, consider the feature 'USB-inputs' with the following values that are not necessarily equal, yet not conflicting as well: *'1', 'Yes', 'Unknown', 'usb2.0'*. In such cases, simply comparing these would result in a meaningless score and thus weaken the reliability of the eventual score of the product

similarity. Therefore, when at least one is not of type double, the key-pair is not scored on this part.

For two- or three-dimensional doubles, we simply take the first value and continue as if it were one-dimensional. Note that this does not mean that the information of the extra dimensions is ignored, since these were an important part of the key matching performed in phase 0. For product comparison, taking only the first value usually suffices (e.g., Resolution '1920 × 1080' vs. '1024 × 768' becomes '1920' vs. '1024', resulting in the same answer). However, this might not always be the case (Item Dimensions '36.8 × 24.4 × 10.4' vs. '36.8 × 21.6 × 2.0' becomes an incorrect comparison), so we strongly suggest to follow up on this in future research and compare all individual values of multi-dimensional values.

Given the two processed double values, a similarity score is calculated, which is a different approach from taking q-grams. Namely, we take the absolute (numeric) difference of the two double values and evaluate whether this is sufficiently small. Specifically, when it is smaller than the *AllowedDifference* parameter (see Equation 10), a score of 1 is rewarded, and 0 otherwise. As stated earlier, some Web shops are more precise than others, so when the width of one product of a (duplicate) pair is '30-6/7 inch', it might very well be the case that the product in the other Web shop has a width of '31 inch'. Furthermore, due to imperfect processing, values may be cut off (e.g., '30-6/7' to '30') instead of rounded, causing a difference of 1 inch. Our reasoning in such cases is that we wish to allow for an absolute difference up to (and including) 1. However, we must bear in mind the distribution of values. For example, when the double values of 'USB-inputs' are all taken from ('0', '1', '2', and '3'), we obviously do not want to allow for a difference of 1. Therefore, the minimum of 1 and half of the smallest standard deviation of the values of the keys is taken. Consequently, the more dense the values are distributed, the lower the *AllowedDifference* and thus the stricter we are.

$$AllowedDifference = min\{1, 0.5 * \sigma_{key1}, 0.5 * \sigma_{key2}\} \tag{10}$$

The final scoring method for the aligned keys is the same for all data types. Namely, for each key-pair, the value-score is weighted by multiplying it with the key-score. This is intuitively clear when recalling that the key-score can be seen as the certainty that this key-pair represents the same feature and thus has to be compared. The more certain we are that the comparison of values between two keys is valid, the larger the weight it gets. As mentioned earlier, the authors of MSM have used the lexical similarity of the names of the keys for the key-score, whereas in our approach a more composite score is used that also considers the aggregated values of the keys and several metrics. When all matched key-pairs between the two products are scored, the weighted average of their scores is the output of *Step a*, the *alignedScore*. The amount of

key-pair comparisons that have been made, is stored in *alignedCount*.

In *Step b*, the keys that were not scored in *Step a* are evaluated. In other words, the keys that are present in at least one product of the pair, but do not have a matching key in the other product. Specifically, per product, the values of all these remaining keys are aggregated together, resulting in a set of values for both products. From both sets, all model words are extracted. Model words are words containing at least one numerical value. When a model word occurs twice for one product, it is given a double weight, as it seems to be important. Thus, duplicates are kept, so technically we are not dealing with proper sets here, but with bags. Then, the score for this step (*restScore*) is calculated with the Jaccard similarity from Equation 3, where A and B are not sets but bags (possibly containing duplicates), where the set operators ($\cap, \cup$) must be interpreted loosely, meaning that duplicates are retained. This step has been kept the same as in MSM, where model words proved to be very useful in this context. The smallest size of both bags of model words is stored as *restCount*.

Before moving on to the titles, the weighted average of the scores that were achieved by the aligned keys and the rest keys is calculated using Equation 11, resulting into *featuresScore*. *restWeight* is used as a weighting parameter. When one of *alignedCount* or *restCount* is zero, the other one gets full weight. When both counts are zero, this is dealt with in *step d*.

$$featuresScore = alignedScore \times (1 - restWeight) + restScore \times restWeight \tag{11}$$

Step c concerns the scoring based on titles. For each title, a number of features are extracted as described in Subsection 3.1.5, which are compared to determine a score. The total amount of features compared for two titles is denoted as *titleCount*, and the score produced is defined as *titleScore*. The details of the proposed scoring technique can be found in the technical report (Valstar & Frasincar, 2019). In *Step d*, we calculate a final score for the product-pair in question, which is a weighted average of the score for the features (*featuresScore*) and the titles (*titleScore*), using the weighting parameter μ, as shown in Equation 12. There are several exceptions to this calculation of the final score which we discuss now. When *alignedCount* or *titleCount* is low, it means that there was not much information to be gained from the features or title. To illustrate, consider two televisions that are only compared on their resolution, a quite general feature that many televisions have in common. Consequently, when their resolutions agree, it does not add much value. Therefore, when *alignedCount* is lower than the parameter *minAlignedCount*, only the *titleScore* is used.

The same holds for the *titleCount*. When both do not pass the threshold, the final score is set to zero.

$$finalScore = \mu \times titleScore + (1 - \mu) \times featuresScore \tag{12}$$

Coming back to the example presented before, when two televisions have a different resolution, it does add valuable information, since it implies quite strongly that these are not duplicates. Therefore, even when the *alignedCount* does not pass the threshold, but the *featuresScore* is low enough, it is still used in the same manner as by default (Equation 12). The same holds for the title. Specifically, an individual score is low when it is lower than the parameter ε, which is used in the phase 3 clustering algorithm as well. Simplified, when two products have a similarity lower than ε, they are not considered in the clustering algorithm. For that reason we consider scores lower than this threshold 'low'.

The final scores are used as the input for the clustering method of phase 3. We use the same method as MSM (van Bezu et al., 2015), so we do not go into the details. However, a quick explanation may help the reader understand what the phase entails. The higher the score between two products, the more likely that these end up in the same cluster and be denoted as duplicates. As said before, when their score is lower than ε, they are not considered, but when they pass it, they are not necessarily marked as duplicates either, since it is still possible that one of these products is part of a different product-pair with a higher score. Product-pairs with the highest score are put together in a cluster and this process iteratively continues until no more product-pairs exist with a score higher than ε. Finally, all products in a cluster are marked as duplicates.

4. Evaluation

In this section, we evaluate the proposed method using the same real-world dataset as was referred to in the previous section. Our dataset consists of 1629 TV's from 4 different Web shops, namely 163 from Amazon.com, 672 from BestBuy.com, 744 from Newegg.com, and 20 from TheNerds.net. We know the actual duplicate products within this dataset, so it is possible to evaluate the performance of our algorithm. We know that there are 1262 unique TV's in the entire dataset, although we have not used this knowledge when implementing the model.

In Subsection 4.1, the feature alignment algorithm is evaluated. In Subsection 4.2, we evaluate the full algorithm. Finally, in Subsection 4.3, the speed of the proposed algorithm is investigated. We start each subsection with the corresponding subquestion belonging to the research question posed in Subsection 1.2.

4.1. Feature Alignment

In this first part of the evaluation, we answer the first subquestion of this work: How can product feature alignment be automated and what is its value? As explained in Subsection 1.1, product features can be represented by different keys across different Web shops. Examples of such keys and their corresponding values are provided in Subsections 1.1 and 2.1. Throughout Subsection 3.1, we have described an algorithm that matches the keys such that the features of different products are aligned, and the corresponding values can be compared. This algorithm has been intrinsically tested against a gold standard on the correct feature alignments between Web shops. This means that, independent of the other phases, we test how the algorithm performs solely on key matching, not yet taking into account the effect it has on duplicate detection. In order to evaluate the algorithm, we compare it with a gold standard that contains the correct matching key-pairs between Web shops. Specifically, 'correct' means that at least two out of three people indicated that a particular key-pair is a match, while the third person did not contradict this. This means that they did not include either keys in their list of matching keys. In the case that they would have included one of the keys with another key, it would count as a conflict, so that the key-pair is not included in the gold standard. This inter-annotated agreement (IAA) has been performed to create a gold standard between the two largest Web shops, Newegg.com and BestBuy.com, who together account for 87% of all products. The ratio of the number of occurrences where two out of three agree versus the occurrences where three people agree is 0.53. Naturally, we have implemented the key-matching algorithm for all Web shops, but throughout the entire evaluation of this phase we only consider the two Web shops mentioned before.

In this evaluation, we use the conventional way of defining true positives, false positives, false negatives, and true negatives. Important to realize is that we look at key-pairs, not at single keys. A true positive (TP) is therefore defined as a key-pair that has been marked correctly as being a match. A false positive (FP) is a key-pair that we deemed to be a match, while it is not according to the gold standard. A false negative (FN) is a key-pair that should be denoted as a match, but is not, either because neither of these keys have been matched at all, or because they have been incorrectly matched to a different key. Finally, a true negative (TN) is that two keys are not matched, when indeed they should not be matched. Furthermore, we use the commonly used performance measures Precision, Recall, and the F_1-measure. The Precision as in Equation 13 is the ratio of our correctly matched key-pairs versus all our matched key-pairs, while Recall as in Equation 14 is the ratio of our correctly matched key-pairs versus all matching keys according to the gold standard. Lastly, the F_1-measure, shown in Equation 15, is the harmonic mean between the two where both are deemed equally important.

$$\text{Precision} = TP/(TP + FP) \tag{13}$$

$$\text{Recall} = TP/(TP + FN) \tag{14}$$

$$F_1\text{-Measure} = 2 \times \text{Precision} \times \text{Recall}/(\text{Precision} + \text{Recall}) = 2 \times TP/(2 \times TP + FP + FN) \tag{15}$$

Before we evaluate this phase using these performance measures, we give a detailed evaluation of the parameters that were used in the framework. Ideally one would use multiple datasets to train the parameters, and test whether the proposed method works out of sample as well, and not just in the given dataset. Due to a lack of multiple datasets, we make use of a bagging bootstrapping technique (Breiman, 1996). For each bootstrap sample, approximately 63% of the original dataset is sampled as a training set and the remaining approximate 37% as the test set. The training set is chosen in such a way that the ratios of duplicates and non-duplicates in the training set and the test set are approximately the same as in the original dataset. On the training set we perform a grid search with steps of 0.1 to find out the combination of choices for the different parameters that results in the highest performance measure in the training set. Finally, when the best parameters are found on the training set, these are used in the key-matching algorithm on the test set, and the final performance measures are calculated. This process is repeated for 50 bootstraps, each having a different training and test set. Summarizing, for each bootstrap, an optimal parameter set is found based on a training set, and using this parameter set, performance measures are calculated for a test set.

In Table 2, we provide the means of the found optimal values for each parameter and their standard deviation. Note that these parameters are the same as the ones discussed in Subsection 3.1.3. We now shortly recap the meaning of these parameters, interpret the found means of the optimal values, and elaborate on the stability using the standard deviation.

The weighting parameters can only be interpreted relatively, as the six of them together are used for

Table 2 Analysis of the Optimal Parameters of phase 0. Means and standard deviations are calculated based on the results obtained from 50 bootstraps.

Weighting Parameters (belonging to score)	Mean	Standard Deviation	Other Parameters	Mean	Standard Deviation
nameWeight (nameScore)	0.926	0.044	*minNameScore*	0.700	0.000
doubleWeight (doubleScore)	1.700	0.000	*minContainedScore*	0.702	0.014
stringWeight (stringScore)	2.048	0.133	*stringBonus*	0.060	0.093
covWeight (covScore)	0.272	0.131	*similarityThreshold*	1.632	0.047
divWeight (divScore)	0.000	0.000			
unitWeight (unitScore)	0.000	0.000			

the calculation of the weighted average of the final score. When one of these weights is higher than the rest, it means that the corresponding score is relatively more important than the other scores. *nameScore* is the lexical similarity between the names of the keys. Its weight is quite stable, with its standard deviation of 0.044, and only having values 0.9 and 1.0. *nameScore* ranges from 0 to 1, but any key-pair having a *nameScore* lower than *minNameScore* is immediately dismissed, so technically its range is [*minNameScore*, 1]. *minNameScore* is extremely stable with a mean of 0.7 and a standard deviation of 0, meaning that for all 50 bootstraps, 0.7 was the optimal value. It seems that 0.8 would be too strict, meaning that too many key-pairs would be prematurely dismissed while some may actually be correct matches. In turn, 0.6 would be too lenient to too many key-pairs, resulting in some incorrectly matched keys. The third aspect that played a role in the lexical name similarity was that we grant a score of at least *minContainedScore* when one of the key names is contained in the other. It is remarkable that for 49 bootstraps, the best value for *minContainedScore* was 0.7, and once it was 0.8. As a result, when the name of the key is contained in the other name of the key, this grants at least a score equal to the *minNameScore*, so that the considered key-pair is not dismissed. This is an interesting finding, as apparently such keys sometimes are (correctly) matched, while otherwise they would not even be considered. We conclude that they must have been correctly matched occasionally, for otherwise when it would not result into correct matches and/or cause wrong matches, a lower value for *minContainedScore* would have been found to be more optimal in the training samples.

doubleScore is the score describing how well the distributions of the double values of the keys match each other. *doubleWeight* has been set to 1.7 for all 50 bootstraps, being a surprisingly constant factor in the weighting process. Together with *stringWeight*, they account for the most influence in the calculation of the final score. This is an interesting result, as it confirms that the values of keys are very important in deciding whether two keys are representing the same feature, and not only the names of the keys as was done in MSM. *stringWeight* is not a very stable parameter, with its standard deviation of 0.133. Still, 80% of its values are 2.0 and the rest ranges between 2.0 and 2.4. A parameter that is strongly related to *doubleWeight* and *stringWeight* is *stringBonus*. This bonus is granted to key-pairs that are both of type strings, as a compromise to the score that keys with doubles may receive when they have matching measurement units. It seems not to be very important, with its mean of 0.060 and standard deviation of 0.093. As the value does not follow a standard distribution (it is bounded at 0 and discrete), we cannot easily say whether it is significantly different from zero. What we do know is that about half of its values are 0, indicating that it is redundant, while the other half are either 0.1 or 0.2. At any rate, it is valuable to include this parameter, so that the potential structural difference between key-pairs of type strings and of type doubles is not captured

in the difference between *doubleWeight* and *stringWeight*, so that these may still be meaningfully compared to the other 4 weighting parameters.

The coverage, denoting how well a key is represented in a Web shop, has proven to be useful in key-matching with an average *covWeight* of 0.272 and a standard deviation of 0.131. The values range between 0.2 and 0.5, but are mostly 0.2. The *covScore* is always a negative score, which describes the difference of the coverages of the keys. The further the coverages are apart, the lower (so the more negative) the score. Our reasoning is that when a key is important for a product, it is included more often in the product description, so that its coverage is relatively high. If one Web shop deems a key important, the other Web shop will most likely do as well, given that the keys represent the same feature. This is an important finding, because it means that coverage is useful in feature alignment.

The *divScore* does not get any weight having been weighted 0 for all 50 bootstraps. This score is set to 0 by default, but becomes -1 if and only if at least one of the two keys has a diversity of 1, meaning that it has only 1 unique value for that key. A possible explanation for the *divWeight* being 0 is that the coverage metric captures the issue, as keys with such a low diversity often have low coverage, so that the reason for having only one unique value is simply the small amount of products that has a value for this key. More surprisingly is that the unit metric seems to be of little value in the process of key matching. Apparently, the measurement units were used consistently across the tested Web shops or they were missing so often that the neutral score of 0 was given regularly. Looking for a reason more closely in this context, we observe that most features were simply in inches, Hz, or pixels and their values were so different that the low *doubleScore* (that compares their value distributions) would prevent these from being matched anyway. This may explain why checking for units proved to be redundant.

The last parameter is *similarityThreshold*. The final score is compared to this threshold at the end of each key-pair comparison. If it passes it, the key-pair becomes a candidate pair and the algorithm continues iterating. In the end, the candidate pair with the highest score is marked as a match. On average, the optimal value for *similarityThreshold* is 1.632, and its values are either 1.6 or 1.7. With a standard deviation of 0.047, it is a stable parameter, which was to be expected as it controls the amount of key-pair matches that are formed, an important part in both Precision and Recall. This is the case, for having a too low value (<1.6) generates too many FP's, while a too high value (>1.7) gives rise too many FN's. Interpreting this value in the light of all other optimal parameters suggests that when a key-pair has a *doubleScore* or *stringScore* of respectively about 0.6 or 0.5, it is sufficient to be a candidate key-pair, given that their coverages are not too far apart. Note that this holds because the *nameScore* must be higher than 0.7.

As explained before, MSM does not have a separate phase for key matching, but matches keys during the product comparison phase. When the lexical similarity of two keys is higher than a threshold, the keys are compared, or in other words: the keys are matched temporarily. In order to compare our results with MSM, we have simulated this technique on the same 50 bootstraps as in APFA. The performances obtained without a feature alignment phase (MSM) and with a feature alignment phase (APFA) are shown in Table 3. Though both Precision and Recall are higher, APFA's higher F_1-measure is mostly due to higher Precision and less due to higher recall. The interpretation is that both methods are able to detect the correct matching keys almost equally well, whereas APFA is more selective than MSM, resulting in less incorrect matches. We apply a Wilcoxon signed rank test (Wilcoxon, 1945) to check whether the difference in F_1-measure is significant. Specifically, we test whether both are equal, compared to the alternative that APFA's F_1-measure is higher. Indeed, even against a significance level of 0.001, the null hypothesis that both are equal is rejected. From this we conclude that by incorporating a pre-processing phase, features can be aligned significantly better.

Table 3 Average performance measures without a feature alignment phase (MSM) and with a feature alignment phase (APFA). The last column is the one-sided p-value of the Wilcoxon signed rank test (H_0: $\mu_{APFA} = \mu_{method}$ versus H_A: $\mu_{APFA} > \mu_{method}$).

Method	F_1-measure	Precision	Recall	p-value
MSM	0.572	0.533	0.618	0.000
APFA	0.767	0.817	0.725	x

4.2. Duplicate Detection

In this subsection, we answer the second subquestion of this work: Can the calculation of the product similarity in phase 2 be improved by incorporating information from phase 0? Before evaluating, there are some notes to be made about the difference with the previous section so that it is clear how the evaluation is performed. Where we had a gold standard for correct key-pairs only for two Web shops, we know the exact correct product duplicates for the complete dataset. We do perform bootstraps in the same manner and calculate the performances measures using the same formulas, but the calculation of TP, FP, TN, and FN needs some explanation. As we have more than two Web shops, it is possible to have a duplicate across three or more shops. A cluster can thus contain one (no duplicates), two, three, or four products. When a cluster contains three products (A, B, C), we see this as three product-pairs (AB, BC, AC). All three pairs are then separately evaluated. MSM does this in the same manner, so that a fair comparison is possible.

One shortcoming of the evaluation of this phase is that it cannot be tested intrinsically for APFA,

as it must use the results of phase 0, namely the feature alignments. Ideally, we would use as input the correct matching key-pairs between each combination of Web shops. That way, phase 2 could be tested independently of phase 0. Unfortunately, creating such a gold standard by hand is, besides tedious, extremely difficult. Each Web shop has roughly 80 keys, resulting in a large amount of possibilities. Moreover, besides the key matching, phase 2 also uses, for example, the data type and units used for each key and the weights for each matched key-pair from phase 0. Nevertheless, keeping the research question of this work in mind, it is not harmful to test phase 2 extrinsically, as the main goal is to improve duplicate detection using a pre-processing phase. Summarizing, in this subsection we use the results of phase 0 as well, so we evaluate the full algorithm, rather than only phase 2.

Just as in the previous subsection, we first evaluate the optimal values for the parameters of this phase. Table 4 shows the mean and standard deviation of the parameters acquired over 50 bootstraps using a grid search between 0 and 1 with steps of 0.1. Two exceptions are *minAlignedCount* and *minTitleCount*, where a grid search between 1 and 10 was used with steps of 1. Zero is not included, because in those cases the score is not used at all as has been explained in Subsection 3.2. We now recap the usage of these parameters, interpret the optimal values, and evaluate the stability using the standard deviation.

Table 4 Analysis of the optimal parameters of phase 2 over 50 training sets.

Parameter	Mean	Standard Deviation
minAlignedCount	4.18	0.63
restWeight	0.11	0.03
minTitleCount	2.00	0.00
titleRestWeight	0.62	0.04
μ	0.50	0.09
ε	0.22	0.04

minAlignedCount is used as a threshold, representing the minimum required amount of compared key-pairs between two products. Specifically, only the aligned pairs from phase 0 are counted, not the 'rest' keys that could not be matched. This parameter was introduced with the idea that when too few comparisons are made, the attained score is not sufficiently reliable. As can be seen in Table 4, on average the *minAlignedCount* is 4.18 with a standard deviation of 0.63. All values are in the range 3-5, so the spread is not large. Still, there seems to be some flexibility. It is an important result that *minAlignedCount* is higher than 1, providing evidence that setting a required amount of comparisons is useful in duplicate detection.

restWeight, although very small, has never been set to 0.0 throughout all bootstraps, implying that there is some value incorporating it in the feature score. The *restScore* is quite unreliable, as it does not look at keys, but only at values. For example, it may incorrectly give a full score to two equal values, while one represents a minimum resolution and the other is a maximum resolution. Nevertheless, it provides an overall indication whether there are few or many values matching between two products. Taking all model words from all keys and comparing these takes quite some computational time. Especially in APFA, where comparing aligned keys is very quick (we discuss this in the next section), calculating the *restScore* is relatively time consuming, while it only represents 10% of the feature score. We therefore conclude that using the 'rest' keys is certainly useful, but we suggest to leave it out when speed is required.

minTitleCount is set to 2.0 for all 50 bootstraps, making this a very stable parameter. The clear interpretation is that when only one comparison can be made between the titles of two products, it is not reliable. Intuitively this is true, since knowing that two televisions have one feature in common (e.g., resolution, screen rate, size) does not provide much evidence that they are duplicates. In the calculation of the *titleScore*, a weighted average is taken of the scores attained by the *titleUnits* and the *titleRest*. *titleUnits* are model words that contain a value and a unit, while *titleRest* contains any other remaining model words. *titleRestWeight* has an average optimal value of 0.62, 39 times being set to 0.6 and 11 times to 0.7. To understand why the rest is relatively more important than the units, we consider some examples of the *titleRests*: '4K', 'E291A1', '47G2', 'class(64', '3D-ready', 'E423', 'cd/m2'. Some of these may represent a product code which may either be internally used within that Web shop or used for televisions in general. The latter may explain why *restScore* gets so much weight. However, we expect that we deal with both cases as some codes have only 4 characters and others have about 14, while a unified code usually has a fixed format. At any rate, in general it is very helpful in duplicate detection. Especially in non-automated methods, it is possible to gain close to perfect results when extracting the code from each product, and then only using that in duplicate detection. In automated methods it is still possible to use the product code, but other model words which are not codes may interfere with this.

The final score is calculated as a weighted average of the *featuresScore* and the *titleScore*. μ is the weight that the title gets. The optimal values for μ range between 0.3 and 0.7, being the least stable parameter in the whole algorithm. At the end of this section, we provide a detailed evaluation of μ.

ε has two interpretations as it has two usages. Most importantly, it is used in the clustering algorithm as a threshold. The dissimilarity of a product-pair must be lower than this threshold in order to be a candidate pair. The found average value for ε in APFA is significantly lower than MSM (0.22 vs 0.52). This has a

clear interpretation. When ε is lower, less products are clustered since a lower dissimilarity is required. In other words, in APFA a higher product similarity between keys is required to become clustered. APFA is therefore more selective than MSM, so we expect a relatively higher Precision measure for APFA.

The second usage of ε has been explained at the end of Subsection 3.2. Recall that when the *titleCount* does not pass *minTitleCount* (or when *alignedCount* does not pass *minAlignedCount*) it is deemed as too unreliable and its score is not used in the scoring, with one exception. Namely, when the count is too low, but at the same time the score is lower than ε. In that case, it is used in the scoring, since we reason that finding non-matching values for a feature contains more information than finding matching values. At first we used a separate parameter for this threshold, but found that most of the time its values were the same as the values for ε. It is not strange that these are related, keeping in mind that according to the first usage of ε, it acts as a threshold whether product-pairs are to be matched or not. Related or not, it is an important finding that low scores should be considered even when their count is low, because that information is valuable in detecting non-duplicates.

We have now come to the final evaluation of the proposed algorithm for duplicate detection. We have calculated the F_1-measure, Precision, and Recall based on 50 different test sets. For each bootstrap, the data is pre-processed in phase 0 before it moves on to phase 2 where the product similarities are calculated. A slightly higher Precision than Recall is achieved, and the F_1-measure lays around 0.75. The interpretation of the Precision is that of all matched product-pairs by APFA, roughly 76% is correct, and the interpretation of Recall is that about 73% of the actual duplicates are detected.

We now compare these results to MSM in Table 5. Again, using a Wilcoxon signed rank test (Wilcoxon, 1945), we check whether the difference in the F_1-measure is significant. APFA significantly outperforms MSM even with a significance level of 0.001. Especially the Precision has risen substantially, which is in accordance with our expectations, as APFA is more selective than MSM.

Table 5 Duplicate detection: Average performance measures for MSM and APFA. The last column is the one-sided p-value of the Wilcoxon signed rank test (H_0: $\mu_{APFA} = \mu_{method}$ versus H_A: $\mu_{APFA} > \mu_{method}$).

Method	F_1-measure	Precision	Recall	p-value
MSM	0.525	0.472	0.592	0.000
APFA	0.746	0.763	0.731	x

Lastly, we elaborate on the relation and relative importance between the product features and the product title. As such, we have evaluated the algorithm once using only features and once using only titles in the product comparison phase. This is done using the conventional 50 bootstraps, and the results for both

methods are shown in Table 6. As expected, for both APFA and MSM, the F_1-measure is higher when optimally combining the features and titles. For both methods, the title on its own does better than only the features. The importance of the title is even stronger in MSM as the reported optimal value for μ is 0.65 (van Bezu et al., 2015), while for APFA it is 0.5. We conclude by saying that APFA significantly outperforms MSM not only for features and titles together, but also when considering them individually. Answering the question of this subsection, using a pre-processing phase certainly improves the quality of duplicate detection.

Table 6 Performance measure for features and titles separately. Averages are taken over 50 bootstrap samples. The last column is the one-sided p-value of the Wilcoxon signed rank test (H_0: $\mu_{APFA} = \mu_{MSM}$ versus H_A: $\mu_{APFA} > \mu_{MSM}$).

		Average F_1-measure		
Description	Restriction	APFA	MSM	p-value
Only Features	μ=0	0.521	0.392	0.000
Only Titles	μ=1	0.605	0.440	0.000
Feature & Titles	(No restr.)	0.746	0.525	0.000

4.3. Speed of the Proposed Algorithm

In this final subsection, we answer the third subquestion: Will an automatic product feature alignment step improve the speed of the process? Before looking at the speed of the process, we take a closer look at two components that make up most of the computational time. Firstly, there is a difference between MSM and APFA concerning the number of product-pair comparisons. Recall that both methods use a so-called Brand Heuristic, which checks whether two products have the same brand. If they are not of the same brand, they will certainly not be duplicates, so they do not need to be compared. In Table 7, we show how the amount of product comparisons may be reduced using a Brand Heuristic. Specifically, we see that MSM already reduces the number of comparisons by 75% compared to not using a Brand Heuristic. In turn, APFA performs 43% less product comparisons than MSM. In Subsection 3.1.5, we described our approach to finding the brand compared to the heuristic of MSM and we discussed two advantages, namely completeness and speed. Indeed, because we search in both title and features of each product, we discover a brand more often than the MSM method. The decrease of 43% is fully due to completeness. Moreover, in our approach we search for the brand only 1629 times, namely once for each product, while in MSM this must happen for each potential product-pair: 419.010 times. However, due to smart caching, this still can be processed relatively quickly.

The second component that takes significant computational time is the number of key comparisons within

Table 7 Product-pair Comparisons. Reducing the amount of comparisons using the brand heuristic.

Method	#Product Comparisons
(No Brand-Heur.)	419,010
MSM	96,890
APFA	55,149

Table 8 Key Comparisons for each product-pair comparison.

Method	#Key Comparisons
MSM	900
APFA	10

one product-pair comparison. Recall that in MSM, given two products, all their keys are compared on their lexical key similarity, after which their values are compared when their key similarity is high enough. On average, a product has 30 keys, so this requires 900 comparisons. Again, string similarities including all lexical key similarities between products are stored in the cache, so these do not have to be re-calculated. Nevertheless, on average the cache has to be accessed 900 times for each product comparison (which is 96.890 times in total). In contrast, in APFA only the matched key-pairs that were found in the pre-processing phase are being compared. On average, there are 20 matched key-pairs between two Web shops (on shop level), and on average 10 of them can actually be used on product level, because many products have missing keys. Table 8 summarizes these results.

After comparing the number of comparisons, we now move on to the actual computational time. All runs were performed on a PC with a 2.53GHz Intel Core 2 Duo processor and 2GB of RAM. MSM uses an extensive framework of caching for the product similarity phase. Caching is appropriate in this framework, because when performing multiple bootstraps and/or different parameter combinations, many calculations have been done before. MSM's caching framework stores string similarities as well as similarity scores of product-pairs. APFA has been implemented in the same framework so that caching works in a similar way for the product comparison phase. In Figure 8, we compare the durations of ten sequential bootstraps of MSM and APFA on the training set with one parameter combination. For both methods, we notice a clear downward trend, strongly suggesting that this is due to caching. Specifically, the decrease in time from bootstrap 1 to 2 is about 50%, which makes sense as the overlap of two random subsets that contain two-thirds of the original set, is 50%.

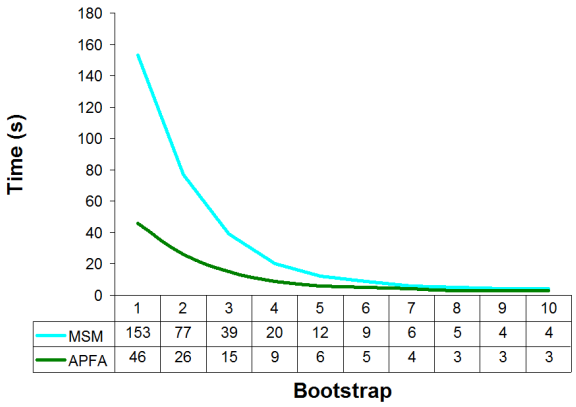

Figure 8. Computational time of the product comparison phase compared.

Phase 0 takes on average 18 seconds on the training set with one parameter combination. This duration ranges from 13 to 20 seconds depending on the bootstrap sample and parameter combination. Ignoring caching, so looking only at one bootstrap, when running the full algorithm (phase 0 and phase 2), APFA is still significantly faster than MSM ($18 + 46 = 64$ seconds versus 153 seconds). When increasing the number of products in the dataset, the time of the pre-processing phase will not increase much. This is because more products does not mean more keys, but rather more values per key, barely causing any additional computational time for the feature alignment phase. Therefore, APFA is certainly scalable, as with increasing size, the duration of the pre-processing phase diminishes relative to the rest of the process. Our expectation is that when caching could be achieved for phase 0, that APFA (including phase 0) will be faster than MSM for each sequential bootstrap.

The answer to the third research subquestion is two-fold: incorporating a pre-processing phase greatly improves the speed of one run. When multiple bootstraps on the same dataset are run, the pre-processing phase makes it slower because a caching framework has not yet been implemented for that phase. However, bootstrapping is usually only done in research, when (a subset of) the correct answers are known, and one wants to evaluate an algorithm. In practice, in a real-world duplicate detection problem, this is not the case, so a single run on the full dataset is all that needs to be done. In that case a pre-processing phase greatly improves the speed as well.

5. Conclusion & Future Work

In this work, we have proposed a method for the problem of product duplicate detection. The context has been finding duplicate television between different Web shops. We have improved upon the state-of-the-art

MSM algorithm for product duplicate detection (van Bezu et al., 2015), mainly by developing an automated pre-processing phase that occurs before the similarities between products are calculated. This way, we managed to outperform MSM both in terms of F_1-measure and in speed. In this section, we summarize our contributions and present our most important findings.

The first step was to incorporate a pre-processing phase, so that features can be aligned between Web shops in an automated way. The most important result is that not only the names of the keys, but the values of the keys are very important in deciding whether two keys are representing the same feature. For keys containing qualitative values, the Jaccard similarity of the values of both keys has been compared, while for keys containing numeric values, their distributions are compared. Considering unit measurements proved not to be useful in feature alignment, because when units are different, this is already captured by the difference in their value distributions. Coverage, representing the ratio of how many products have a certain key within a Web shop, has been found to be useful in feature alignment. When exploiting this metric, the diversity metric which represents the number of unique values of a key, is redundant, as it is captured by the coverage, because keys with such a low diversity often have low coverage as well. Lexical key matching based on their names has also been improved by granting a minimum score to key-pairs of which one of their key-names is contained in the other. By testing on a real-world dataset of TV's, it has been shown that we significantly outperform MSM with an F_1-measure of 0.767 versus 0.572 for feature alignment.

The second step has been to improve the quality of duplicate detection, using the gained information from the pre-processing phase. Given two products, we now only compare the keys there were matched earlier, instead of comparing all keys with each other. Moreover, depending on the discovered data type in the pre-processing phase, we treat key-pairs differently, so that the comparisons are more meaningful. Comparing non-aligned keys is still useful, but may be omitted for efficiency purposes. As the product titles are important, we have improved the title analyzer. Rather than comparing all model words in one large set, we make a distinction between model words with units and the rest, so that the units can be aligned between the two products and scored separately. An important finding is that a required minimum amount of comparisons of both aligned keys and between titles is necessary to prevent unreliable high scores based on too little information. On the other hand, we found that low scores do provide important information in rejected non-duplicates, even when the amount of performed comparisons is low. Applying our duplicate detection method to the same real-world dataset of TV's, we find that we significantly outperform MSM with an F_1-measure of 0.746 versus 0.525.

Since speed is important in the process of duplicate detection, we have enhanced the pre-processing phase in such a way that it reduces the running time of the product similarities phase. MSM makes on average 900 comparisons per product-pair, while APFA brings this down to 10 on average. Moreover, when increasing the amount of keys, the former grows quadratically, while the latter linearly. Because brands play an important role in distinguishing between TV's, we have developed an improved brand finder, which not only extracts the brand from the product title, but also from the discovered brand key of each product. This leads to a decrease of 43% in the amount of product comparisons, saving much computational time. Incorporating a pre-processing phase improves the speed of the product similarities phase by roughly a factor of 3. When increasing the size of the dataset, the time of the pre-processing phase will relatively diminish compared to the rest of the process, achieving even higher efficiency.

The algorithm is not limited to this particular dataset or even to the television market. The only requirement is that the dataset has some sort of title and some features with values, which is often the case. Even if there is no title, APFA performs reasonably well with an F_1-measure of 0.521 on this dataset. For future research, we recommend trying the algorithm on several different datasets, preferably with varying types of products. Furthermore, it will be important to do more research on the last phase, namely the clustering algorithm. In (Benjelloun et al., 2008), a different approach than the one we use now is proposed, which is called 'matching and merging'. When a match occurs, the source code of those entities is immediately removed and we continue with the merged version. Especially when the dataset becomes larger, we suggest using map-reduce algorithms, two of which are given in (Rastogi et al., 2013), that focus on computing connected components for large graphs, and in (Jin et al., 2013), that proposes a distributed algorithm for single-linkage hierarchical clustering.

We give two directions to improve the key matching algorithm. The first is to research to what extent the matching keys obey the transitivity relation. If transitivity holds sufficiently, this knowledge may be implemented in the pre-processing phase. Specifically, after aligning shop A with B and shop B with C, we may use the transitivity relation to make several alignments between shop A and C already, that way further bringing down the number of key-pair comparisons when aligning the keys. A second direction is to evaluate whether it is profitable to perform key matching within Web shops before the actual key matching between Web shops. We suspect that it may prove useful, as we encountered some keys within Web shops that in fact represent the same feature (e.g., 'warranty terms' with 'warranty term', 'shipping:' with 'shipping'). If these could be merged, we would have more values so that key matching could be improved. Moreover, during the product similarity phase, the merged key may be used, rather than only one of the keys. As the

latter is the case, we are currently missing all information from the other key.

Furthermore, much may still be left to be exploited from the blocking phase, with it being the method of reducing the number of product comparisons. The pre-processing phase may provide valuable information for the forming of the blocks, e.g., finding out which features are important for block splitting and what these features are across different Web shops. This will further enhance the effectiveness of the blocking method, so we recommend to combine the proposed pre-processing phase with the blocking phase.

Lastly, in this paper we have only considered the performance of APFA in comparison with the algorithm that it is meant to improve upon, namely MSM. APFA utilizes various handcrafted features and processing steps to link product keys. An interesting approach to explore and compare APFA with is the implementation of deep learning methods for this problem. Similarly to how *Word2Vec* (Mikolov et al., 2013) can be used to calculate similarity scores between words and find synonyms, a deep learning implementation could be used to define key similarity scores and find matching keys. Not only could this idea be implemented for the purpose of calculating key similarity scores during phase 0, but also during phase 2 to calculate similarity scores between products. There already is significant work done on linking similar knowledge from different sources using deep learning techniques (Xu et al., 2016). Furthermore, deep learning approaches could also be explored for the other parts of the process like the clustering phase (Aljalbout et al., 2018).

Acknowledgment

Wouter Bender, Tim Grevelink, Kai Huiskamp, and Ruben Timmermans contributed in the early stages of this work with their insights in the field of Value-Driven Similarity Detection. Thomas Breugem, Bert Wassink, Gino Mangnoesing, and Wim Kuipers contributed with their research on applying Data Types and Measurement Units for the purpose of key matching.

References

Aljalbout, E., Golkov, V., Siddiqui, Y., Strobel, M., & Cremers, D. (2018). Clustering with Deep Learning: Taxonomy and New Methods. arXiv preprint arXiv:1801.07648.

Amazon.com, Inc. (n.d.). URL: `http://www.amazon.com`.

Ayat, N., Akbarinia, R., Afsarmanesh, H., & Valduriez, P. (2014). Entity Resolution for Probabilistic Data. *Information Sciences*, *277*, 492–511.

de Bakker, M., Frasincar, F., & Vandic, D. (2013). A Hybrid Model Words-Driven Approach for Web Product Duplicate Detection. In *25th International Conference on Advanced Information Systems Engineering (CAiSE 2013)* (pp. 149–161). Springer volume 7908 of *LNCS*.

Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Whang, S., & Widom, J. (2008). Swoosh: a Generic Approach to Entity Resolution. *The VLDB Journal*, *18*, 255–276.

Best Buy Co., Inc. (n.d.). URL: `http://www.bestbuy.com`.

van Bezu, R., Borst, S., Rijkse, R., Verhagen, J., Frasincar, F., & Vandic, D. (2015). Multi-component Similarity Method for Web Product Duplicate Detection. In *30th Annual ACM Symposium on Applied Computing (SAC 2015)* (pp. 761–768). ACM.

Bilenko, M., & Mooney, R. J. (2003). Adaptive Duplicate Detection Using Learnable String Similarity Measures. In *9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003)* (pp. 39–48). ACM.

Breiman, L. (1996). Bagging Predictors. *Machine Learning*, *24*, 123–140.

Breitling, R., Armengaud, P., Amtmann, A., & Herzyk, P. (2004). Rank Products: A Simple, yet Powerful, New Method to Detect Differentially Regulated Genes in Replicated Microarray Experiments. *FEBS Letters*, *573*, 83–92.

Computer Nerds International, Inc. (n.d.). URL: `http://www.thenerds.net`.

van Dam, I., van Ginkel, G., Kuipers, W., Nijenhuis, N., Vandic, D., & Frasincar, F. (2016). Duplicate Detection in Web Shops Using LSH to Reduce the Number of Computations. In *31st Annual ACM Symposium on Applied Computing (SAC 2016)* (pp. 772–779). ACM.

Draisbach, U., & Naumann, F. (2010). DuDe: The Duplicate Detection Toolkit. In *8th International Workshop on Quality in Databases (QDB 2010)*.

Elmagarmid, A. K., Ipeirotis, P. G., & Verykios, V. S. (2007). Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, *19*, 1–16.

Fernández, N., Fisteus, J. A., Sánchez, L., & López, G. (2012). IdentityRank: Named Entity Disambiguation in the News Domain. *Expert Systems with Applications*, *39*, 9207–9221.

Fisher, J., Christen, P., Wang, Q., & Rahm, E. (2015). A Clustering-Based Framework to Control Block Sizes for Entity Resolution. In *21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2015)* (pp. 279–288). ACM.

Hartveld, A., van Keulen, M., Mathol, D., van Noort, T., Plaatsman, T., Frasincar, F., & Schouten, K. (2018). An LSH-Based Model-Words-Driven Product Duplicate Detection Method. In *30th International Conference on Advanced Information Systems Engineering (CAiSE 2018)* (pp. 409–423). Springer volume 10816 of *LNCS*.

Hassanzadeh, O., Chiang, F., Lee, H. C., & Miller, R. J. (2009). Framework for Evaluating Clustering Algorithms in Duplicate Detection. *Proceedings of the VLDB Endowment*, *2*, 1282–1293.

Hsueh, S., Lin, M., & Y., C. (2014). A Load-Balanced MapReduce Algorithm for Blocking-based Entity-resolution with Multiple Keys. In *12th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2014)* (pp. 3–9). Australian Computer Society volume 152.

Indyk, P., & Motwani, R. (1998). Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *30th Annual ACM Symposium on Theory of Computing (STOC 1998)* (pp. 604–613). ACM.

Jain, A. K., Murty, M. N., & Flynn, P. J. (1999). Data Clustering: A Review. *ACM Computing Surveys*, *31*, 264–323.

Jalbert, N. (2008). Automated Duplicate Detection for Bug Tracking Systems. In *2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN 2008)* (pp. 52–61). IEEE.

Jin, C., Patwary, M. M. A., Agrawal, A., Hendrix, W., Liao, W., & Choudhary, A. (2013). DiSC: A Distributed Single-Linkage Hierarchical Clustering Algorithm using MapReduce. In *4th International Workshop on Data Intensive Computing in the Clouds (DataCloud 2013)*.

Kolb, L., Thor, A., & Rahm, E. (2012). Multi-Pass Sorted Neighborhood Blocking with MapReduce. *Computer Science -*

Research and Development, *27*, 45–63.

Koller, D., & Sahami, M. (1997). Hierarchically Classifying Documents using Very Few Words. *Stanford InfoLab*.

List with Measurement Units (n.d.). URL: `http://www.convert-me.com/en/unitlist.html`.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv preprint arXiv:1301.3781.

Monge, A. E. (2000). Matching Algorithms within a Duplicate Detection System. *IEEE Data Engineering Bulletin*, *23*, 14–20.

Nederstigt, L. J., Aanen, S. S., Vandic, D., & Frasincar, F. (2014). FLOPPIES: A Framework for Large-Scale Ontology Population of Product Information from Tabular Data in E-commerce Stores. *Decision Support Systems*, *59*, 296–311.

Newegg Inc. (n.d.). URL: `http://www.newegg.com`.

Papadakis, G., Alexiou, G., Papastefanatos, G., & Koutrika, G. (2015). Schema-Agnostic vs Schema-Based Configurations for Blocking Methods on Homogeneous Data. *41st International Conference on Very Large Data Bases (VLDB 2015)*, *9*, 312–323.

Papadakis, G., Ioannou, E., Palpanas, T., Niederee, C., & Nejdl, W. (2013). A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces. *IEEE Transactions on Knowledge and Data Engineering*, *25*, 2665–2682.

Phillips, J. M. (2013). Jaccard Similarity and Shingling. University of Utah.

Rastogi, V., Machanavajjhala, A., Chitnis, L., & Das Sarma, A. (2013). Finding Connected Components in Map-Reduce in Logarithmic Rounds. In *IEEE International Conference on Data Engineering 2013 (ICDE 2013)* (pp. 50–61). IEEE.

van Rooij, G., Sewnarain, R., Skogholt, M., van der Zaan, T., Frasincar, F., & Schouten, K. (2016). A Data Type-Driven Property Alignment Framework for Product Duplicate Detection on the Web. In *17th International Conference on Web Information Systems Engineering (WISE 2016)* (pp. 380–395). Springer volume 10042 of *LNCS*.

Saxena, A., Prasad, M., Gupta, A., Bharill, N., Patel, O. P., Tiwari, A., Er, M. J., Ding, W., & Lin, C.-T. (2017). A Review of Clustering Techniques and Developments. *Neurocomputing*, *267*, 664–681.

Simonini, G., Papadakis, G., Palpanas, T., & Bergamaschi, S. (2019). Schema-Agnostic Progressive Entity Resolution. *IEEE Transactions on Knowledge and Data Engineering*, *31*, 1208–1221.

Sutinen, E., & Tarhio, J. (1995). On Using q-Gram Locations in Approximate String Matching. In *Third Annual European Symposium (ESA 1995)* (pp. 327–340). Springer volume 979 of *LNCS*.

Talburt, J. R. (2010). *Entity Resolution and Information Quality*. Morgan Kaufmann.

Tan, P., Steinbach, M., & Kumar, V. (2006). *Introduction to Data Mining*. Pearson International Edition, 1st edn.

Thomas, I., Davie, W., & Weidenhamer, D. (2014). *Quarterly Retail e-commerce Sales 3rd Quarter 2014*. U.S. Census Bureau News.

Valstar, N., & Frasincar, F. (2019). Technical Report and Software of APFA. URL: `https://github.com/nickvalstar/APFAProject`.

Vandic, D., Frasincar, F., Kaymak, U., & Riezebos, M. (2020). Scalable Entity Resolution for Web Product Descriptions. *Information Fusion*, *53*, 103–111.

Vandic, D., Van Dam, J., & Frasincar, F. (2012). Faceted Product Search Powered by the Semantic Web. *Decision Support Systems*, *53*, 425–437.

Verykios, V., Elmagarmid, A., & Houstis, E. (2000). Automating the Approximate Record-matching Process. *Information Sciences*, *126*, 83–98.

Wilcoxon, F. (1945). Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, *1*, 80–83.

Xu, B., Ye, D., Xing, Z., Xia, X., Chen, G., & Li, S. (2016). Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)* (pp. 51–62). ACM.

Yang, C., Hoang, D., Mikolov, T., & Han, J. (2019). Place Deduplication with Embeddings. In *2019 World Wide Web Conference (WWW 2019)* (pp. 3420–3426). ACM.

Zhu, G., & Iglesias, C. A. (2018). Exploiting Semantic Similarity for Named Entity Disambiguation in Knowledge Graphs. *Expert Systems with Applications*, *101*, 8–24.