APFA: Automated Product Feature Alignment for Duplicate Detection (Technical Report)

Nick Valstar and Flavius Frasincar

Erasmus University Rotterdam

PO Box 1738, NL-3000 DR

Rotterdam, the Netherlands

nickvalstar@gmail.com, frasincar@ese.eur.nl

Abstract. This technical report contains the pseudo-code of the algorithms and details on the techniques used by the APFA framework. In addition, section 7 contains the flow chart of phase 2, and section 8 provides a running example of the full APFA algorithm.

1 Value Aggregation per Key

In this step we propose a simple algorithm that iterates over all the products and per Web shop stores all the used keys. 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. This is the building block which we use to calculate metrics in the next section. Algorithm 1 gives the pseudo code of the used algorithm.

Algorithm 1 Construct *KeyMap*. This aggregates per Web shop all unique keys including a list of corresponding values.

```
Require: Input: Set $S$ is the set of all products. Each product $p$ belongs to a shop
    ($p.shop$), and has a set of keys ($p.keys$). Each key $k$ of a product has a name
    ($k.name$) and has a value ($k.value$).
Require: Initialize: $KeyMap$ is a empty map that can contain objects (in our case
    shops) that may each be mapped to a collection of objects (in our case keys).
 1: %Iterate over all products:
 2: for all product $p \in S$ do
 3:   if $KeyMap$ not contains $p.shop$ then
 4:     $KeyMap$.add($p.shop$)
 5:   end if
 6:   %Iterate over all keys of this product:
 7:   for all key $k \in p.keys$ do
 8:     if $KeyMap$.get($p.shop$) not contains $k.name$ then
 9:       $KeyMap$.get($p.shop$).add($k.name$)
10:     end if
11:     $KeyMap$.get($p.shop$).get($k.name$).add($k.value$)
12:   end for
13: end for
Output: $KeyMap$ is now filled with information from all shops, having for each shop
a set of all the unique keys from that shop. Each of these keys has a list of all the
values that have been used with that key in that particular shop.
```

2 Adding Metrics per Key

Using the aggregated raw values we calculate certain metrics for each key individually. These metrics are used in the product feature alignment phase. The following metrics are calculated: data type, measurement unit, data distribution, coverage, diversity and standard deviation. Algorithm 2 gives the pseudo code of the used algorithm that processes the values and calculates these metrics.

Algorithm 2 Enrich *KeyMap*. This processes the values of each key of *KeyMap* and adds metrics.

```
Require: Input: KeyMap from Algorithm 1.
Require: stripStrings(list of strings) extracts the non-numerical blocks from the list
    of strings.
Require: stripDoubles(list of strings) extracts the numerical blocks from the list of
    strings and converts them to doubles.
Require: getDimensions(list of strings) checks for a fixed pattern of no, one, or two
    single composite signs (‘X’, ‘x’) in the raw values and returns resp. ‘1-dim’, ‘2-dim’,
    or ‘3-dim’.
Require: Input: unitList is a provided list with commonly used measurement units.
Require: updateUnits(key, string) keeps track of how many times each unit has oc-
    curred for this key. If the given string appears in unitList, add one count to that
    unit for the inputted key.
Require: mostOccuringUnit(key) retrieves the unit that has occurred most often in
    the values of this key. Returns “None” if no units occurred.
Require: getStandardDeviation(list of doubles) returns the standard deviation
    (spread around the mean) of the values.
 1: %Iterate over shops:
 2: for all shop $\in$ KeyMap do
 3:   %Iterate over keys of that shop:
 4:   for all key $k \in$ KeyMap.get(shop) do
 5:     %Process values of this key:
 6:     allStringValues = stripStrings(k.values)
 7:     nrStringValues = count(allStringValues)
 8:     k.StringValues = unique(allStringValues)
 9:     allDoubleValues = stripDoubles(k.values)
```

```
10:      $nrDoubleValues$ = count($allDoubleValues$)
11:      $k.DoubleValues$ = unique($allDoubleValues$)
12:      %*Retrieve DataType:*
13:      **if** $nrStringValues > nrDoubleValues$ **then**
14:         $k.datatype$ = "String"
15:      **else**
16:         $k.datatype$ = "Double" + $getDimensions(k.values)$
17:      **end if**

18:      %*Search for Measurement Units in the values*
19:      **for all** String $str \in$ allStringValues **do**
20:         **if** $unitList$ contains $str$ **then**
21:            updateUnits($k$, $str$)
22:         **end if**
23:      **end for**
24:      $k.unit$ = mostOccuringUnit($k$) %*Retrieve Measurement Unit*

25:      %*Calculate the Diversity, Coverage and Standard Deviation*
26:      $k.diversity$ = count(unique($k.values$))
27:      $k.coverage$ = count($k.values$)/$shop.size$
28:      $k.std$ = getStandardDeviation($k.allDoubleValues$)
29:   **end for**
30: **end for**
```

Output: Each *key* in *KeyMap* now has the following variables and metrics:
stringValues: a list with the unique string values stripped from the raw values of this *key*;
doubleValues: a list of unique doubles stripped from the raw values of this *key*;
datatype: the data type of this *key* (string or one-, two- or three-dimensional double);
unit: the measurement unit used for the values;
diversity: the number of unique values belonging to this *key*;
coverage: proportion of the products of this Web shop that has this *key*;
std: the standard deviation of the double values, which will be used in phase 2.

3 Alignment of Keys between Web shops

We have now come to the actual matching of the keys between Web shops. For testing the APFA framework we have used the following Web shops: [1,2,3,4]. We use all the information of each key that we have previously gathered. This includes the name of the keys, the processed string- and double-values, and the calculated metrics. 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 cross 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 *Alignments*. Algorithm 3 gives the pseudo code of the full algorithm.

Score of Metric	Weighting Parameter	Description of the Score
nameScore	nameWeight	The lexical similarity of the names of the keys.
doubleScore	doubleWeight	The distribution of the double values.
stringScore	stringWeight	The jaccard similarity of the string values.
covScore	covWeight	The difference in their coverages.
divScore	divWeight	Does one of their keys have a diversity of 1?
unitScore	unitWeight	Do their units match?

Other Parameters	Description
similarityThreshold	A threshold for the minimum required score of a key−pair.
minNameScore	A threshold for the minimum required nameScore.
stringBonus	The bonus a key−pair of type ‘String’ gets.
minContainedScore	The minimum score a key−pair gets if one of their names contains the other.

Table 1 Overview of scores, weighting parameters and other parameters used in the key-matching algorithm.

Algorithm 3 Matching keys. This algorithm aligns keys between Web shops. Those pairs of keys that have sufficiently similar names and corresponding values are matched and are stored in *Alignments*.

```
Require: Input: KeyMap from Algorithm 2.
Require: Initialize: Alignments is a empty map that can contain pairs of two objects
    (in our case two shops) where each of these pairs may be mapped to a collection
    of objects (in our case key-pairs, that in turn consist of two keys and a score).
Require: The parameters from Table 1.
Require: jaccardSimilarity(list of strings, list of strings). Range 0-1.
Require: tTest(list of doubles, list of doubles) tests using a paired, 2-sample t-statistic
    whether their respective list of double values can originate from the same distribu-
    tion. Returns the p-value, denoting the probability that they come from the same
    distribution. Range 0-1.
Require: nameSimilarity(String, String) returns the q-gram distance between the
    strings, with q=3. Range 0-1.
Require: weightedAverage(String, Double, Double, Double, Double, Double, Double)
    calculates a weighted average of the given scores.
Require: addMatchingKey(shop, shop, key, key) stores the given matching key pair
    in the variable alignments for the given combination of shops.
 1: %Iterate over shops:
 2: checkedShops = null
 3: for all shop1 ∈ KeyMap do
 4:   checkedShops.add(shop1)
 5:   %Iterate over shops (excluding the already checked shops):
 6:   for all shop2 ∈ KeyMap - checkedShops do
 7:     %We now repeatably search for the best match of keys between shop1 and shop2
    and assign it as a matched key. Once the score for the best match is not sufficient,
    we stop the search for matching keys between these two Web shops.
 8:     assigning = true
 9:     assignedKeys1 = null
10:     assignedKeys2 = null
11:     while assigning do
12:       bestPairKey1 = null
```

```
13:     bestPairKey2 = null
14:     highestPairScore = -1
15:     %Iterate over keys of shop1: (key1 stands for a key from shop1)
16:     for all key key1 $\in$ KeyMap.get(shop1) - assignedKeys1 do
17:       bestMatchingKey2 = null
18:       highestKey2Score = -1
19:       %Iterate over keys of shop2: (key2 stands for a key from shop2)
20:       for all key key2 $\in$ KeyMap.get(shop2) - assignedKeys2 do
21:         %We now calculate the score between key1 and key2.
22:         %Datatype. If datatypes do not agree, we do not consider the pair.
23:         if key1.datatype != key2.datatype then
24:           break %(Skip to next iteration in the for loop of line 20.)
25:         else
26:           pairDatatype = key1.datatype
27:         end if

28:         %Name similarity.
29:         nameScore = nameSimilarity(key1.name,key2.name)
30:         if key1.name contains key2.name OR key2.name contains key1.name
   then
31:           nameScore = max{namescore, minContainedScore}
32:         end if
33:         if nameScore < minNameScore then
34:           %When the similarity does not pass a certain threshold, we do not
   consider the pair.
35:           break %(Skip to next iteration in the for loop of line 20.)
36:         end if

37:         %Coverage. We take the negative squared error of the difference of their
   coverage.
38:         covScore = $-(key1.coverage - key2.coverage)^2$
```

```
%Diversity
if $key1.diversity == 1$ OR $key2.diversity == 1$ then
    $divScore$ = -1 %We punish very small diversity
else
    $divScore$ = 0
end if
%Calculate the similarity of the values.
if $pairDatatype$ == "STRING" then
    $stringScore$ = jaccardSimilarity($key1.stringValues$, $key2.stringValues$)
    $isString$ = 1
else
    $distribution\_p\text{-}value$ = tTest($key1.doubleValues$, $key2.doubleValues$)
    $doubleJaccardScore$ = jaccardSimilarity($key1.doubleValues$, $key2.doubleValues$)
    $doubleScore = \max\{distribution\_p\text{-}value, doubleJaccardScore\}$
    $isString$ = 0
end if

%Measurement Unit
if $key1.unit$ == 'none' OR $key2.unit$ == 'none' then
    $unitScore$ = 0
else
    if $key1.unit == key2.unit$ then
        $unitScore$ = 1
    else
        $unitScore$ = -1
    end if
end if
```

```
65:           %which key2 matches this key1 best?
66:           finalScore = weightedAverage(keyScore, covScore, divScore, doubleScore, stringScore, isString, unitScore)
67:           if finalScore > highestKey2Score then
68:             highestKey2Score = finalScore
69:             bestMatchingKey2 = key2
70:           end if
71:         end for
72:         %which key1 has the best match?
73:         if highestKey2Score > highestPairScore then
74:           highestPairScore = highestKey2Score;
75:           bestPairKey1 = key1;
76:           bestPairKey2 = bestMatchingKey2;
77:         end if
78:       end for

79:       %Assign the best found key-pair between shop1 and shop2 (only if it passes a threshold.)
80:       if highestPairScore ≥ similarityThreshold then
81:         addMatchingKey(shop1, shop2, bestPairKey1, bestPairKey2)
82:         %We now continue comparing, but leave out these two keys.
83:         assignedKeys1.add(bestPairKey1)
84:         assignedKeys2.add(bestPairKey2)
85:         assigning = true
86:       else
87:         assigning = false
88:       end if
89:     end while
90:   end for
91: end for
```

Output: The variable *alignments* now contains the matching key-pairs between each combination of two Web shops, including their scores.

4 Preparation of Alignments

In order to incorporate the obtained information up until now, the aligned keys from *Alignments* are to be enriched first. Recall that the output from the previous algorithm in the previous section is the variable *Alignments* which contains pairs of aligned keys including their scores. The keys contain information that was used for matching, but not all their information is useful in phase 2. This section does not add anything new, rather it has been added for readability reasons, for it would make the previous or next section verbose when incorporating it therein. For a complete overview of the preparation of *Alignments* we refer the reader to the Algorithm below.

Algorithm 4 Prepare *Alignments*. This goes through the aligned key-pairs and enriches them with information that is used in the product comparison phase.

```
Require: Input: Alignments from Algorithm 3.
Require: Input: Enriched KeyMap from Algorithm 2.
Require: enrichAlignments(String, String, String, String, List of Strings, List of
    Strings, String, Double) enriches Alignments with the subsequent names, shops,
    possible units, datatype and score of its matched keys.
 1: %Iterate over (aligned) pairs:
 2: for all keypair ∈ Alignments do

 3:     %Get both keys
 4:     Key k1 = keypair[1]
 5:     Key k2 = keypair[2]

 6:     %Get info from keys
 7:     name1 = k1.name
 8:     name2 = k2.name
 9:     shop1 = k1.shop
```

```
10:     $shop2 = k2.shop$
11:     $units1 = k1.possibleUnits$
12:     $units2 = k2.possibleUnits$
13:     $std = \min\{k1.std,\ k2.std\}$
14:     $datatype = k1.datatype$
15:     $pairscore = pair.score$
16:     %Add info to Alignments
17:     enrichAlignments($name1,\ name2,\ shop1,\ shop2,\ units1,\ units2,\ datatype,$
      $pairscore$)
18: end for
```

Output: Each *keypair* in *Alignments* now has the following variables:

name1/name2: key name of first/second key;

shop1/shop2: shop of first/second key;

units1/units2: all units that have been found in the values of the first/second key. Note that this time it is not the canonical representation as shown earlier, but rather the raw units, because we need the raw units for stripping purposes;

datatype: datatype of both keys;

std: the smallest standard deviation of both sets of double values;

pairscore: score of this keypair as calculated in Algorithm 3.

5 Additional Information per Product

In this section in the technical appendix we explain the brand analyzer and the title analyzer which were omitted in the report. Before moving on to the actual product comparison, we gather information about the products. Till so far the focus has been the keys of the products, such that we have collected information per key. In contrast, in this section we gather information per product, which proves of great worth in performing product comparison. We now discuss the collection of brands and the processing of the product titles.

Probably the most distinguishing feature of a television, and perhaps of any product, is its brand. We therefore wish to pay particular attention to extracting the brand from each product. For each Web shop it holds that the brand is given in a certain key which we call the ‘brand key’. However, finding this is not easy, since just like any other feature, the brand key is not always represented the same between Web shops. Moreover, strangely enough, its coverage is seldom 100%, so we cannot extract a brand for each product.

The way how the authors of MSM deal with this, is by using the title of the product, since the brand is often included there. They use a small list with the most common brands such as ‘Sony’, ‘Samsung’, ‘Philips’, and search for these brands in the title. This way, the brands of the majority of products can be found. Before comparing two products, the brands are both extracted and compared. When the brands agree or are unknown, the comparison continues, but otherwise this product-pair is rejected, as two televisions from different brands are never duplicates. We go into more detail on this so-called brand-heuristic in the next section.

There are two downsides to this approach, which we now discuss and improve upon. An obvious first downside is that the extraction of the brand from the title happens ‘on the fly’ during the product comparison phase, as they do not have a pre-processing phase. Consequently, for each combination of two products, both titles have to be processed each time, which requires a fair amount of computational time. In APFA we propose to extract the brand once for each title in phase 0, which is in turn used in phase 2.

Another improvement is that we do not only turn to the title to extract the brand, but also at the product brand key if it can be found. This gives rise to a higher probability that the brand will be found for two reasons. Firstly, when a brand is not included in the title, it will most likely be included in the brand key. Secondly, the used brand list does not contain uncommon brands such as 'JVC', 'Westinghouse', 'Hannspree', so these are not stripped from the title, but may be included in the brand key.

We now explain how the brand key is found for each Web shop. We simply iterate through all enriched keys and per key we count how many of the brands in the brand list from MSM occur in the string values of that key. Then, per Web shop, the key with the most brands contained in its string values is most likely the brand key of that Web shop. Optimally, we now have the brand key for each Web shop, giving rise to two opportunities. At first, as said before, we can take the brand from the brand key when it can not be found in the title. Actually, the algorithm first looks at the brand key since this is less time consuming. Secondly, whenever we encounter a new brand in the brand key, we include it into the brand list. Through this iterative process, the probability of finding a less common brand in the title grows.

Only very few products now remain where a brand has not been found. For these, a second list containing the common and uncommon television brands has been provided [5] so that even uncommon brands can be stripped from the product title. Such lists can be found without difficulty for many contexts, so it is a neat way of improving the method by injecting external information. Such an extensive list is large, so it would certainly not be feasible to use it for all products. Using it for only the few remaining products is a good alternative. In the evaluation we evaluate this improved gathering of brands and compare it with the current brand extraction of MSM.

Besides brands, there is more valuable information contained in product titles. A title should give a customer a quick overview of the product, providing the brand and some important features. For our context, examples of such features are resolution, size and sharpness. Gathering such information from the titles is valuable in product comparison. Since we aim to keep our approach applicable for other contexts we wish to extract not only these three features, but any feature. Therefore we employ an earlier used technique, namely collecting all model words from the title. Recall that a model word is a word containing both numbers and letters. All model words can thus be split up into numeric and non-numeric parts, were the latter often is a unit measurement of some type. In such fortunate cases, when our method recognizes a unit, it is converted to the canonical unit representation as was explained earlier, and both unit and value are stored. If no unit can be found, the raw model word is stored entirely. Figure 1 illustrates this, using one example of a TV from Web shop Newegg.com [2]. For this example 'class(64.9' and 'TFTV4537' are not recognized and thus stored as *titleRest*, while the other two ('inch: 65.0' and 'pixels: 720.0') are stored as *titleUnits*. Both *titleUnits*, as well as *titleRest* are used in the scoring of product-pairs.

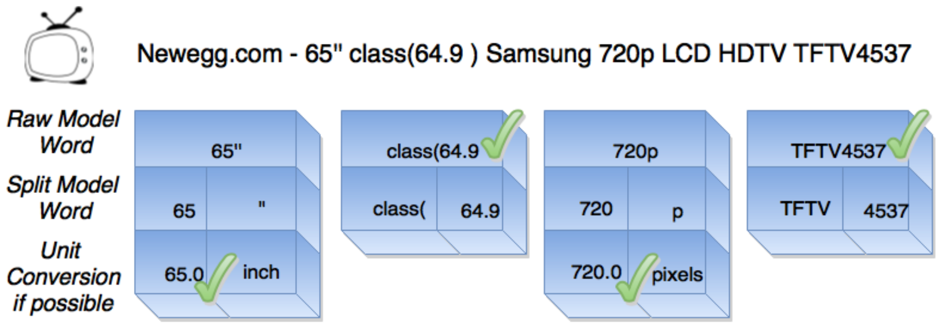

Fig. 1. Title Analyzer in phase 0. Model words are extracted, split up into numeric and non-numeric parts, and converted to a known unit plus value if possible. A green tick denotes what is being stored.

6 Phase 2: Pairwise Product Comparison

Now the keys are aligned, we continue to subquestion 2 and develop an algorithm for product comparison while incorporating information from phase 0. We distinguish four steps 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*).

Step c, which concerns the scoring based on titles has been implemented differently in APFA than in MSM. In the latter, when comparing two products, the model words are extracted from both titles and the Jaccard similarity between the sets of model words is calculated, resulting in a score between 0 and 1. In our approach, we use the results of the second part of Subsection 5 where the titles of all products are analyzed. Recall from Figure 1 that for each product title we have a list of units (*titleUnits*) and a list of remaining model words (*titleRest*). We discuss how these are used in product duplicate detection now.

Figure 2 shows an example of two different products, where the left one is the same as was used in Figure 1. The left one has two *titleUnits*: 'inch' and 'pixels'. The right TV has 'inch' twice and 'pixels'. Because they both have exactly one value for 'pixels', these are compared by checking whether the absolute distance between the values is less or equal to 1. If so, they are scored with a 1, otherwise 0. Because the right TV has two values for 'inch' both are compared. If at least one matches, a full score of 1 is granted, 0 otherwise. The reasoning for this is as follows: when two values are given for the same unit measure, one is probably a more detailed version of the other (compare '6490' from this example to '65'). Another reason for providing two values could be that in fact they do represent a different feature. In that case, when one matches between the two product, this is most likely the same feature.

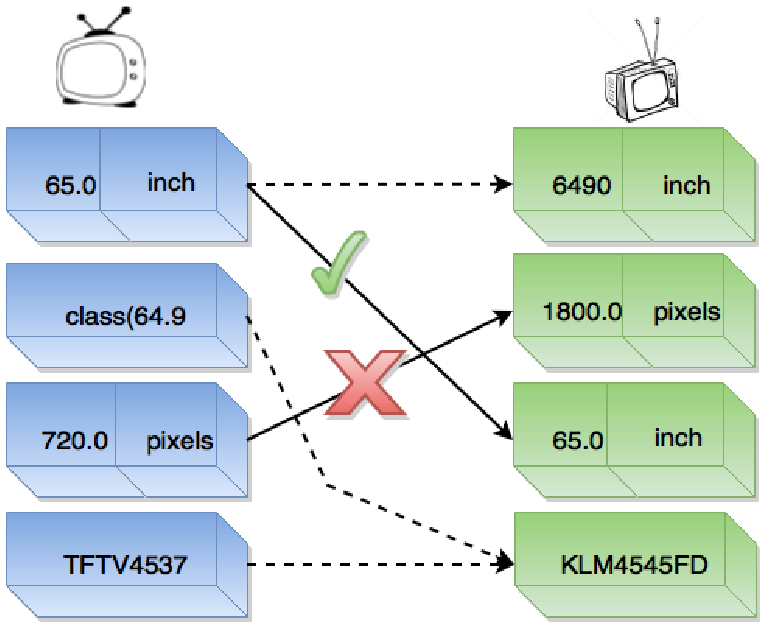

Fig. 2. Title Analyzer in phase 2.

The same approach is taken for the values of *titleRest*, which are all compared to each other between the two products, as they do not have a certain unit. If at least one match can be found, a score of 1 is given, 0 otherwise. In the used example the scores for the *titleUnitScore* and *titleRestScore* are resp. 0.5 and 0, and the counts for the *titleUnitCount* and *titleRestCount* are resp. 2 and 1. The *titleCount* is 3, and the *titleScore* is calculated as a weighted average of the scores of the units and the rest, as can be seen in Equation 1, where *titleRestWeight* acts as a weighting parameter. When one of *titleUnitCount* or *titleRestCount* is zero, the other one gets full weight. When both counts are zero, this is dealt with in *step d*.

$$titleScore = titleUnitScore \times (1 - titleRestWeigth) + titleRestScore \times titleRestWeigth \quad (1)$$

To keep track of the scores, counts, and used parameters per step in this section we provide Table 2.

Table 2 Phase 2: Overview of Scores, Counts and Parameters (p)

Step	Variable	Description
a	alignedScore	Average score of the comparisons of matched key-pairs
a	alignedCount	Total amount of comparisons between matched key-pairs
a	stringComparing (p)	Defines whether we score strings or not
b	restScore	Average score of the comparisons of non-matched key-pairs
b	restCount	Total amount of comparisons between non-matched key-pairs
b	restWeight (p)	Denotes the weight of the restScore compared to alignedScore
c	titleUnitScore	Average score of the unit-comparisons within a title
c	titleUnitCount	Total amount of the unit-comparisons within a title
c	titleRestScore	Average score of the rest-comparisons within a title
c	titleRestCount	Total amount of the rest-comparisons within a title
c	titleRestWeight (p)	Denotes the weight of the titleRestScore compared to titleUnitScore
c	titleCount	Sum of titleUnitCount and titleRestCount
d	minTitleCount (p)	titleCount has to be higher/equal to this threshold
d	titleScore	Weighted average of titleUnitScore and titleRestScore
d	minAlignedCount (p)	alignedCount has to be higher/equal to this threshold
d	featuresScore	Weighted average of alignedScore and restScore
d	mu (p)	Denotes the weight of the titleScore compared to featuresScore
d	finalScore	Weighted average of titleScore and featuresScore
d	varepsilon (p)	Parameter used in clustering and as a threshold for ‘low’ scores

This section ends with Algorithm 5, the pseudocode of the methodology of this subsection. Also, in Appendix 7 the flow chart is shown that belongs to Algorithm 5. Finally, in Appendix 8 we provide a running example of the full method of this section.

Algorithm 5 Product Comparison. Here we compare and score all possible product-pairs between all combinations of Web shops, using information from phase 0.

```
Require: Input: Set $S$ is the set of all products. Each product $p$ belongs to a
    shop ($p.shop$), and has a set of keys ($p.keys$). Each key $k$ of a product has a
    name ($k.name$) and has a value ($k.value$).
Require: Input: Enriched *Alignments* from Algorithm 4.
Require: Initialize: *productSimilarities* is an empty map that can contain pairs
    of two objects (in our case two products) where each of these pairs has a
    score.
Require: shopSelection(*Map*, *String*, *String*) Retrieves only those key-pairs
    from the map *Alignments* that occur between the given shops.
Require: stripString(*String*) Cleans the string from punctuation marks and
    excessive spaces.
Require: scoreString(*String*, *String*) Returns the q-gram similarity between the
    two strings. Range 0-1.
Require: checkBoolean(*String*, *String*) Returns 0 or 1 when the values can be
    compared and are resp. not equal or equal, -1 when they cannot be compared.
Require: stripDouble(*String*) Cleans it, removes units, processes composites,
    strips the numerical value and converts it to a double.
Require: stripFirstDouble(*String*) Similar to stripDouble(), but now for the
    first dimension of the value.
Require: checkDouble(*Double*, *Double*) Checks whether both arguments are
    doubles.
Require: scoreDouble(*Double*, *Double*) Returns 1 if the absolute difference be-
    tween the doubles is less than the AllowedDifference, 0 otherwise.
```

```
Require: getMW(List of keys) First retrieves all the values that correspond to
    the inputted keys and then extracts all the model words from them.
Require: modelWordsScore(List of Strings, List of Strings) Outputs the
    restScore: the fraction of matching strings of the given lists of strings. Also
    outputs therestCount: the smallest size of both lists of strings.
Require: addSimilarity(product, product, Double) Adds the final score to the
    given product-pair and stores it in productSimilarities.
 1: %Iterate over all combination of two products:
 2: for all product $p1 \in S$ do
 3:   for all product $p2 \in S$ do
 4:     %Two prior checks: different Shops & equal Brands
 5:     if p1.shop == p2.shop then
 6:       %We assume that there are no duplicates within Web shops.
 7:       addSimilarity(p1, p2, $-\infty$)
 8:       break %(Skip to next iteration in the for loop of line 3.)
 9:     end if
10:     if productInfo.get(p1).brand != productInfo.get(p2).brand then
11:       %When products have different brands, they cannot be duplicates.
12:       addSimilarity(p1, p2, 0)
13:       break %(Skip to next iteration in the for loop of line 3.)
14:     end if
15:     %Step a: Calculate a score for those keys that can be aligned.
16:     remainingKeys1 = p1.keys
17:     remainingKeys2 = p2.keys
18:     alignedCount = 0
19:     totalWeight = 0
20:     totalScore = 0
21:     keypairs = shopSelection(alignments, p1.shop, p2.shop) %Select match-
    ing pairs between these shops
22:     for all $keypair \in keypairs$ do
23:       if $keypair.name1 \notin remainingKeys1$ OR $keypair.name2 \notin remain-$
    $ingKeys2$ then
24:         %One of the given products does not have this key provided. It is
    missing or unknown.
25:         break %(Skip to next iteration in the for loop of line 22.)
26:       end if
27:       rawvalue1 = p1.get(keypair.name1)
28:       rawvalue2 = p2.get(keypair.name2)
29:       if keypair.datatype == "STRING" then
30:         s1 = stripString(rawvalue1)
31:         s2 = stripString(rawvalue2)
32:         if stringComparing == true then
33:           score = scoreString(s1, s2)
```

```
34:          else
35:             score = checkBoolean(rawvalue1,rawvalue2)
36:             if score== $-1$ then   %No comparison can be made.
37:                break %(Skip to next iteration in the for loop of line 22.)
38:             end if
39:          end if
40:       else
41:          if keypair.datatype == “DOUBLE_1dim” then
42:             d1 = stripDouble(rawvalue1)
43:             d2 = stripDouble(rawvalue2)
44:          else  %2 or 3 dimensional. Take only first value
45:             d1 = stripFirstDouble(rawvalue1)
46:             d2 = stripFirstDouble(rawvalue2)
47:          end if
48:          if checkDouble(d1,d2) == false then
49:             score = checkBoolean(rawvalue1,rawvalue2)
50:             if score== $-1$ then %we cannot compare these values, so we
    will not score them.
51:                break %(Skip to next iteration in the for loop of line 22.)
52:             end if
53:          else
54:             score = scoreDouble(d1, d2)
55:          end if
56:       end if
57:       thisWeight = keypair.pairscore
58:       totalWeight = totalWeight + thisWeight
59:       weightedScore = thisWeight $\times$ score
60:       totalScore = totalScore + WeightedScore

61:       alignedCount = alignedCount + 1
62:       remainingKeys1.remove(keypair.name1)
63:       remainingKeys2.remove(keypair.name2)
64:    end for
65:    if (alignedCount $>$ 0) then
66:       alignedScore = totalScore / totalWeight;
67:    else
68:       alignedScore=0
69:    end if
70:    %Step b. Calculate a score for the rest ( those features that could not be
    aligned.)
71:    modelWords1 = getMW(remainingKeys1)
72:    modelWords2 = getMW(remainingKeys2)
73:    [restScore, restCount] = modelWordsScore(modelWords1, model-
    Words2)

74:    %Step c. Calculate a score for the similarity of the titles.
75:    productInfo1 = productInfo.get(p1) //info that was gathered for each
    product in phase 0.
76:    productInfo2 = productInfo.get(p2)
```

```
    %Units. Units that were found in the titles.
    titleUnitScore = 0
    units1 = productInfo1.titleUnits
    units2 = productInfo2.titleUnits
    for all unit1 ∈ units1 do
      for all unit2 ∈ units2 do
        if unit1 == unit2 then %we found matching units. now we compare their values.
          score = 0
          values1 = unit1.values %the values that belong to this unit for this product title.
          values2 = unit2.values
          for all Double value1 ∈ values1 do
            for all Double value2 ∈ values2 do
              if abs(value1-value2)<= 1 then
                score = 1
              end if
            end for
          end for
          titleUnitScore = titleUnitScore + score
          titleUnitCount = titleUnitCount + 1
        end if
      end for
    end for
    if titleUnitCount > 0 then
      titleUnitScore = titleUnitScore / titleUnitCount
    end if
    %Rest. Other model words in titles.
    rest1 = productInfo1.titleRest
    rest2 = productInfo2.titleRest
    titleRestScore = 0
    if size(rest1) == 0 OR size(rest2) == 0 then
      titleRestCount = 0
    else
      titleRestCount = 1
      for all String r1 ∈ rest1 do
        for all String r2 ∈ rest2 do
          if r1 contains r2 OR r2 contains r1 then
            titleRestScore = 1 %One match is enough.
          end if
        end for
      end for
    end if
    titleCount = titleUnitsCount + titleRestCount
    %Step d. Weighted average of the scores of steps a,b,c.
    %Keys
    if alignedCount × restCount > 0 then %Both were scored.
      featuresScore = alignedScore × (1-restWeigth) + restScore × restWeigth
```

```
123:       else %At least one of them is not scored.
124:          $featuresScore = alignedScore + restScore$
125:       end if %Note that in practise always at least one is scored.
126:       %Titles
127:       if $titleUnitCount \times titleRestCount > 0$ then %Both were scored.
128:          $titleScore = titleUnitScore \times (1-titleRestWeigth) + titleRestScore \times titleRestWeigth$
129:       else %At least one of them is not scored.
130:          $titleScore = titleUnitScore + titleRestScore$
131:       end if %Note that in practise always at least one is scored.

132:       %Final Score
133:       if $titleCount > minTitleCount$ then
134:          if $alignedCount >= minAlignedCount$ then
135:             $finalScore = \mu \times titleScore + (1-\mu) \times featuresScore$
136:          else
137:             if $featuresScore < \varepsilon$ then %When it is a bad score, we still want to use it. reasoning: finding an inequality says more than finding an equality.
138:                $finalScore = \mu \times titleScore + (1-\mu) \times featuresScore$
139:             else
140:                $finalScore = titleScore$
141:             end if
142:          end if
143:       else
144:          if $alignedCount >= minAlignedCount$ then
145:             if $titleScore < \varepsilon$ then %When it is a bad score, we still want to use it. reasoning: finding an inequality says more than finding an equality.
146:                $finalScore = \mu \times titleScore + (1-\mu) \times featuresScore$
147:             else
148:                $finalScore = featuresScore$
149:             end if
150:          else
151:             $finalScore = 0$
152:          end if
153:       end if

154:       addSimilarity($p1$, $p2$, $finalScore$)
155:    end for
156: end for
```

Output: The variable *productSimilarities* now contains a score for all pairs of products between all combinations of Web shops.

7 Flow Chart Diagram phase 2

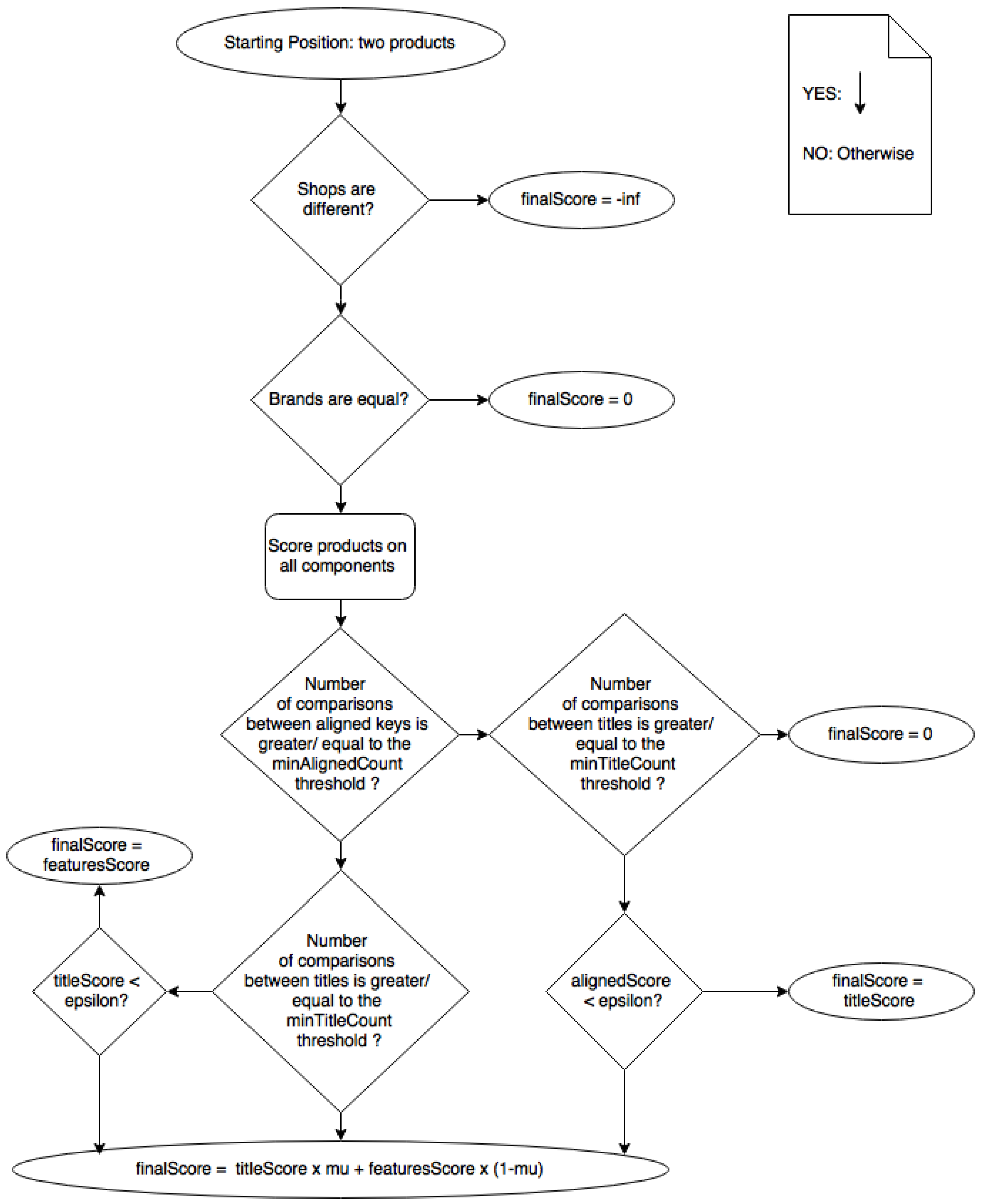
Starting Position: two products
Shops are different?
finalScore = -inf
Brands are equal?
finalScore = 0
Score products on all components
Number of comparisons between aligned keys is greater/ equal to the minAlignedCount threshold ?
Number of comparisons between titles is greater/ equal to the minTitleCount threshold ?
finalScore = 0
finalScore = featuresScore
titleScore < epsilon?
Number of comparisons between titles is greater/ equal to the minTitleCount threshold ?
alignedScore < epsilon?
finalScore = titleScore
finalScore = titleScore x mu + featuresScore x (1-mu)
YES:
NO: Otherwise

Fig. 3. Flow chart diagram of phase 2.

8 Running Example

Below in Figures 4, 5 and 6 we give a running example of the full method of both feature alignment and pairwise product comparison. The numbers 1-5 correspond to Algorithms 1-5 from the methodology, while 5a-5d correspond to *step a- step d* of Algorithm 5. For readability purposes we show only one key-pair that is being matched. Next, the matched key-pair is used in the scoring of one feature of a product-pair.

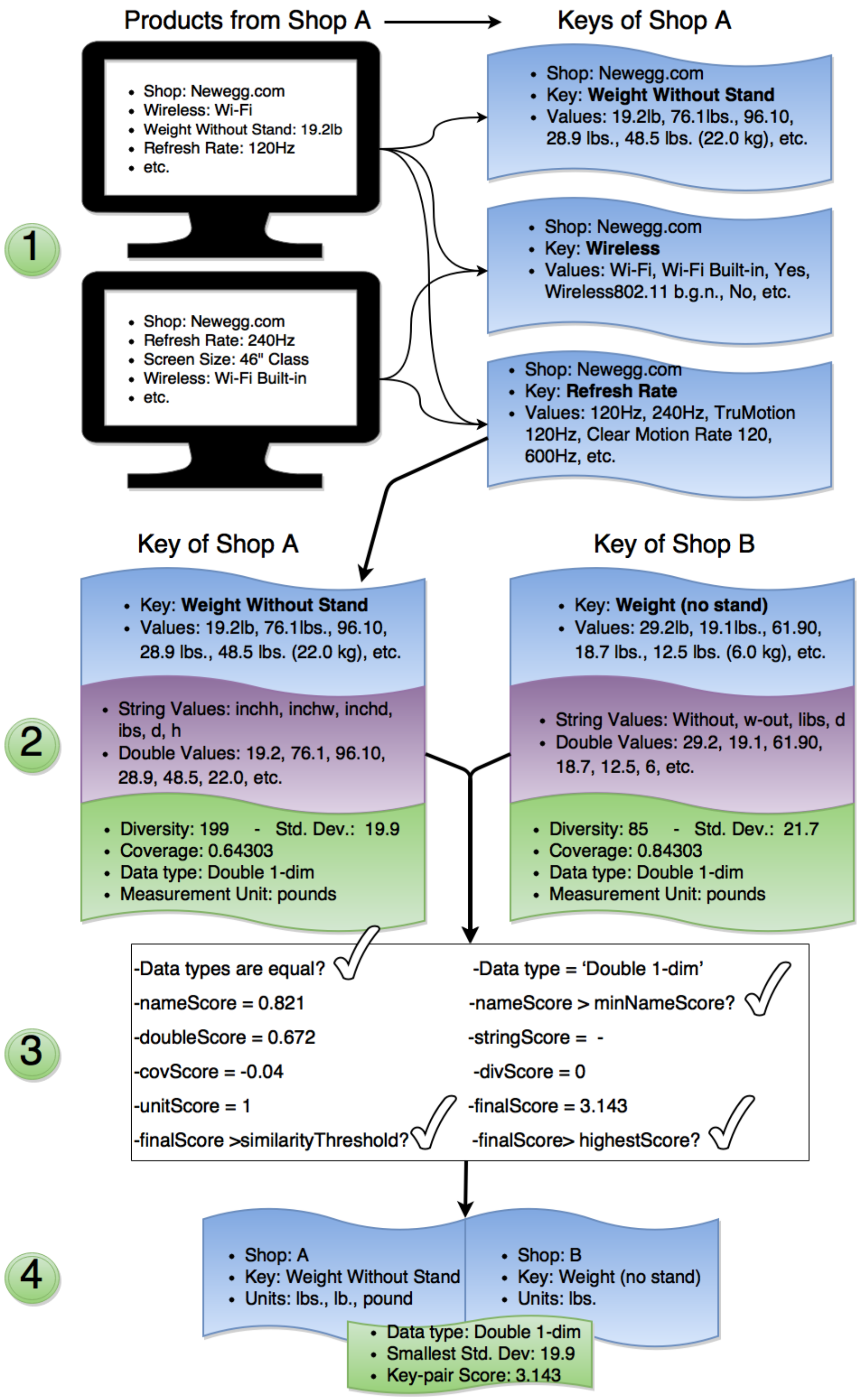
Products from Shop A
Keys of Shop A
Shop: Newegg.com
Wireless: Wi-Fi
Weight Without Stand: 19.2lb
Refresh Rate: 120Hz
etc.
Shop: Newegg.com
Refresh Rate: 240Hz
Screen Size: 46" Class
Wireless: Wi-Fi Built-in
etc.
Shop: Newegg.com
Key: **Weight Without Stand**
Values: 19.2lb, 76.1lbs., 96.10, 28.9 lbs., 48.5 lbs. (22.0 kg), etc.
Shop: Newegg.com
Key: **Wireless**
Values: Wi-Fi, Wi-Fi Built-in, Yes, Wireless802.11 b.g.n., No, etc.
Shop: Newegg.com
Key: **Refresh Rate**
Values: 120Hz, 240Hz, TruMotion 120Hz, Clear Motion Rate 120, 600Hz, etc.
1
Key of Shop A
Key of Shop B
Key: **Weight Without Stand**
Values: 19.2lb, 76.1lbs., 96.10, 28.9 lbs., 48.5 lbs. (22.0 kg), etc.
String Values: inchh, inchw, inchd, ibs, d, h
Double Values: 19.2, 76.1, 96.10, 28.9, 48.5, 22.0, etc.
Diversity: 199 - Std. Dev.: 19.9
Coverage: 0.64303
Data type: Double 1-dim
Measurement Unit: pounds
Key: **Weight (no stand)**
Values: 29.2lb, 19.1lbs., 61.90, 18.7 lbs., 12.5 lbs. (6.0 kg), etc.
String Values: Without, w-out, libs, d
Double Values: 29.2, 19.1, 61.90, 18.7, 12.5, 6, etc.
Diversity: 85 - Std. Dev.: 21.7
Coverage: 0.84303
Data type: Double 1-dim
Measurement Unit: pounds
2
-Data types are equal? ✓
-nameScore = 0.821
-doubleScore = 0.672
-covScore = -0.04
-unitScore = 1
-finalScore >similarityThreshold? ✓
-Data type = 'Double 1-dim'
-nameScore > minNameScore? ✓
-stringScore = -
-divScore = 0
-finalScore = 3.143 ✓
-finalScore> highestScore? ✓
3
Shop: A
Key: Weight Without Stand
Units: lbs., lb., pound
Shop: B
Key: Weight (no stand)
Units: lbs.
Data type: Double 1-dim
Smallest Std. Dev: 19.9
Key-pair Score: 3.143
4

Fig. 4. Running Example part 1: Feature Alignment

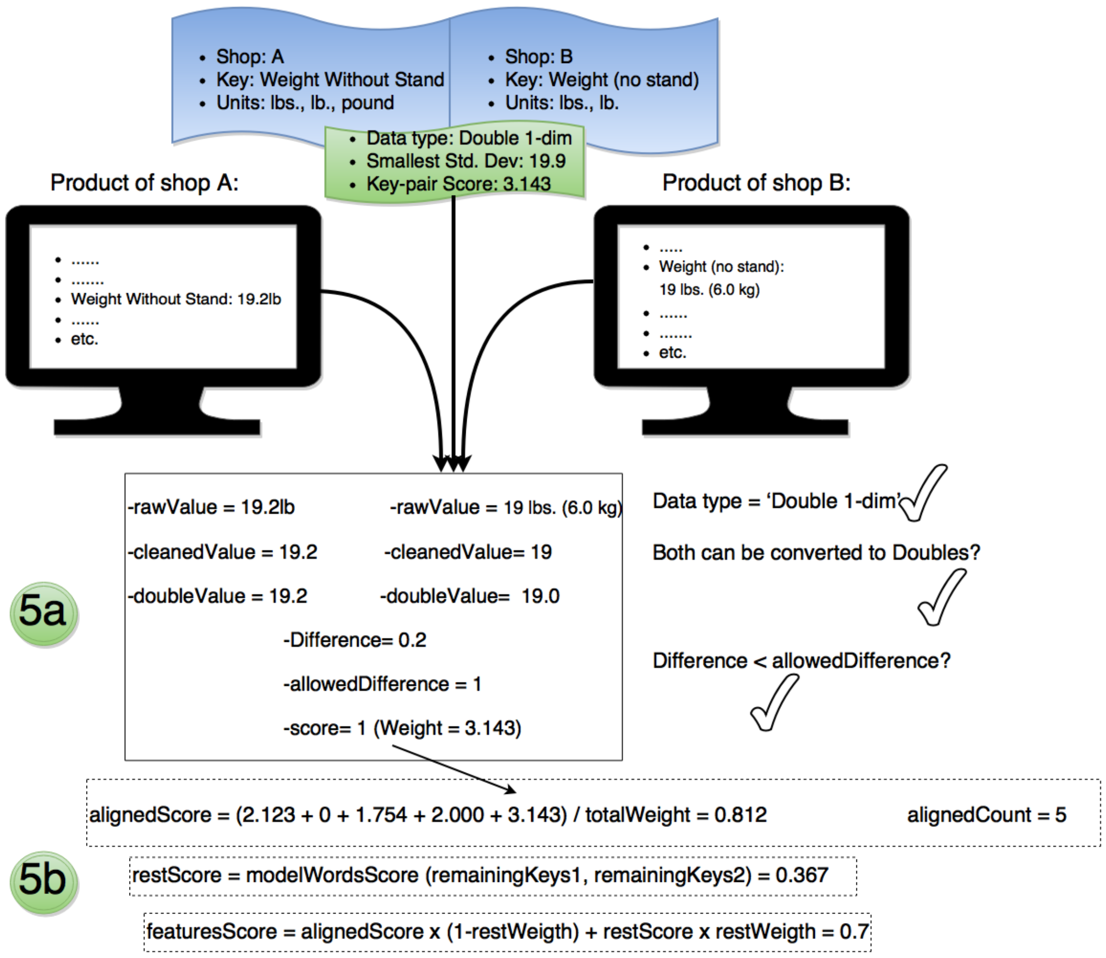
Shop: A
Key: Weight Without Stand
Units: lbs., lb., pound
Shop: B
Key: Weight (no stand)
Units: lbs., lb.
Data type: Double 1-dim
Smallest Std. Dev: 19.9
Key-pair Score: 3.143
Product of shop A:
Weight Without Stand: 19.2lb
etc.
Product of shop B:
Weight (no stand): 19 lbs. (6.0 kg)
etc.
-rawValue = 19.2lb
-rawValue = 19 lbs. (6.0 kg)
-cleanedValue = 19.2
-cleanedValue= 19
-doubleValue = 19.2
-doubleValue= 19.0
-Difference= 0.2
-allowedDifference = 1
-score= 1 (Weight = 3.143)
Data type = 'Double 1-dim'
Both can be converted to Doubles?
Difference < allowedDifference?
5a
alignedScore = (2.123 + 0 + 1.754 + 2.000 + 3.143) / totalWeight = 0.812
alignedCount = 5
5b
restScore = modelWordsScore (remainingKeys1, remainingKeys2) = 0.367
featuresScore = alignedScore x (1-restWeigth) + restScore x restWeigth = 0.7

Fig. 5. Running Example part 2: Pairwise Product Comparison (Aligned keys and Rest keys)

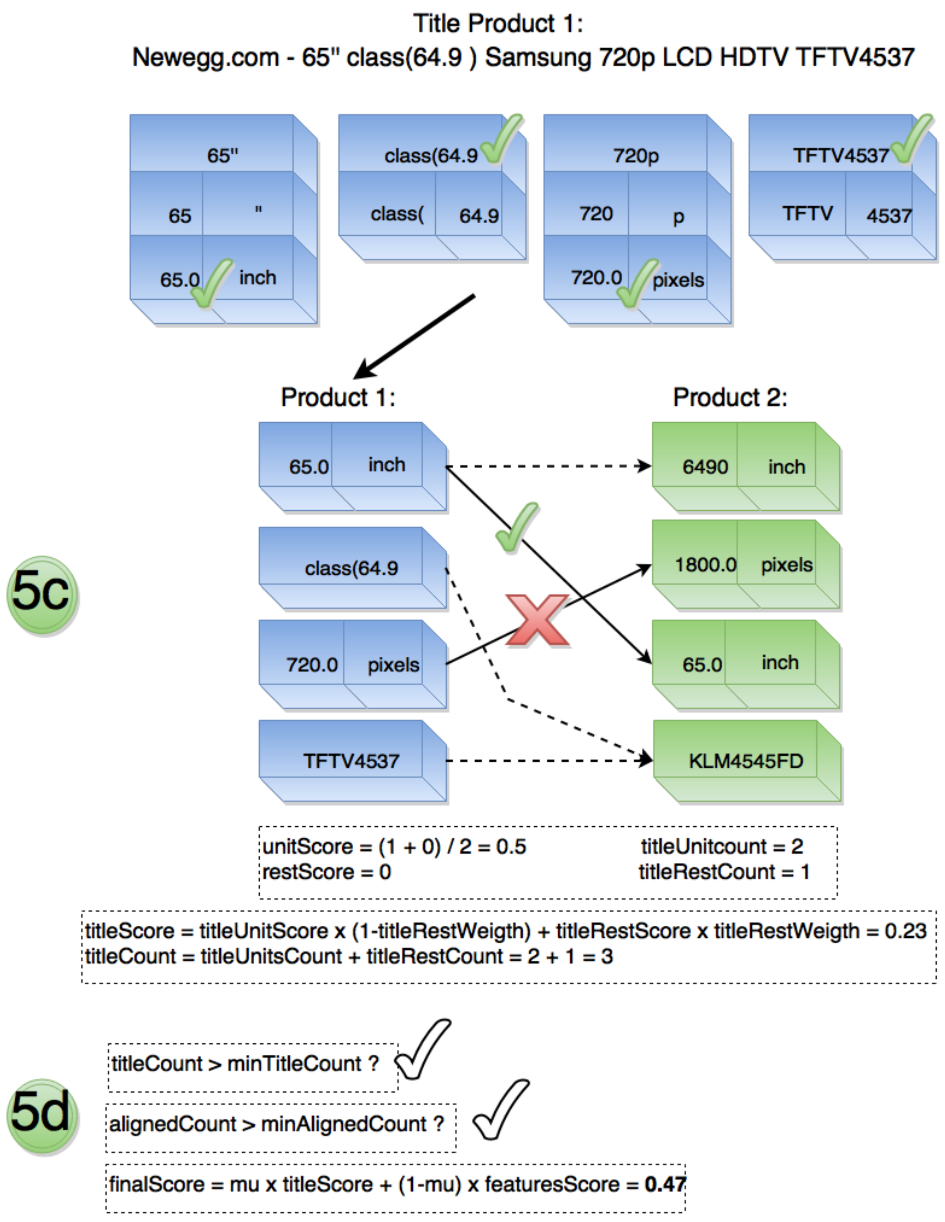
Title Product 1:
Newegg.com - 65" class(64.9) Samsung 720p LCD HDTV TFTV4537
65"
65
"
65.0
inch
class(64.9
class(
64.9
720p
720
p
720.0
pixels
TFTV4537
TFTV
4537
Product 1:
Product 2:
65.0
inch
class(64.9
720.0
pixels
TFTV4537
6490
inch
1800.0
pixels
65.0
inch
KLM4545FD
5c
unitScore = (1 + 0) / 2 = 0.5
restScore = 0
titleUnitcount = 2
titleRestCount = 1
titleScore = titleUnitScore x (1-titleRestWeigth) + titleRestScore x titleRestWeigth = 0.23
titleCount = titleUnitsCount + titleRestCount = 2 + 1 = 3
5d
titleCount > minTitleCount ?
alignedCount > minAlignedCount ?
finalScore = mu x titleScore + (1-mu) x featuresScore = 0.47

Fig. 6. Running Example part 3: Pairwise Product Comparison (Title Analyzer and final Scoring)

References

1. Inc. Amazon.com. *http://www.amazon.com.*

2. Newegg Inc. *http://www.newegg.com.*

3. Best Buy Co., Inc. *http://www.bestbuy.com.*

4. Computer Nerds International, Inc. *http://www.thenerds.net.*

5. List of Television Manufacturers. *https://en.wikipedia.org/wiki/List_of_television_manufacturers.*