Record Linkage in Consumer Products Data using Approximate String Matching and Clustering Methods

Riki Saito

December 1, 2016

Contents

1 Preface

This academic paper is submitted for the degree of Master of Science at the University of Minnesota - Twin Cities. The study was conducted under the supervision of Professor Xiaoou Li in the Department of Statistics, University of Minnesota, during Fall of 2016.

2 Background

Consumer product data is an absolute necessity for any business to do any sort of analytics. This kind of data generally pertains to sales data of products, and is typically deaggregated by product and date-time. Using consumer products data, analyses can be done in numerous ways to help inform business decisions. For example, the sales performance of products can be evaluated for specific dates or aggregated time periods (by month, season, year, etc). Estimations of the percentage of shares the products dominate the market can also be obtained and compared against the performance of competitor products in the same market. These sort of analyses are essential in business strategy and decision making.

Today, consumer product data is widely available and can be obtained easily. Many businesses often use third party for-purchase data sources from market research companies (e.g. NPD, GFK). However, before any data analysis can be conducted, preprocessing of data (and lots of it) is necessary. One of these tasks is the joining or aggregating of data from different sources. A large issue with this is the occurrence of duplicate or redundant records.

The task of identifying duplicate or redundant records is known as *record linkage*. Record linkage is a data cleaning task of identifying records that belong to the same entity in a single data set or across multiple data sets. An entity can represent people, companies, institutions, etc [5]. In this paper we focus on the application of record linkage in consumer products data.

In consumer products data, one common issue is the identification of the *Product Line*[1] from a listing of a product item known as the *Stock Keeping Unit (SKU)*[2]. The product line is a group of related products (e.g. iPhone 7), and the stock keeping unit is a distinct type of item (e.g. iPhone 7 - Rose Gold). From data where the *product line* is missing, preprocessing may be necessary to obtain information on the *product line* from the given *SKU* name, which customarily contains the *product line* name.

A foolproof way of insuring that all records get linked correctly is by "brute force" (i.e. manually linking records). Especially in data where a mislinkage is costly in the sense that it could potentially pose a serious problem (for example, misidentifying two different patients as the same person), there may be a need for some manual linkage or manual inspection of record linkage. However, a "brute force" is also very costly in terms of time and labor, and is not scalable. In scenarios with a large number of records, it is not feasible.

Let's say we have n products and we compare every product to every other product. Then we will have a total of $\frac{(n^2-n)}{2}$ comparisons. Thus the number of comparisons quickly increases with n and there a "brute force" is not scalable.

The goal of this paper is to discuss methods to automate the record linkage process (identify the *product line* of a *SKU*). Since the names of these *SKU* will contain information about the *product line (PL)* but will never have exact duplicates, we will use the *probablistic record linkage* approach. In a *probablistic record linkage*, an estimate of the *distance* between different records are obtained, and groups are computed using *distances*[3].

The process is divided into two steps. The first step is computing an estimate of *distance* between all pairs in a list of records. The method we will use is known as *approximate string matching. Approximate String*

[1] **Product Line (PL)**: a group of products comprising of different sizes, colors, or types, produced or sold under one unique product or model name (i.e. Apple iPhone 7, UE Boom 2)

[2] **Stocking Keeping Unit (SKU)**: a particular product identified by its product line as well as by its size, color, or type, where its identification is typically used for inventory purposes (i.e. Apple iPhone 7 - Rose Gold 64GB, UE Boom 2 Blue)

[3] **Distance**: an estimated numerical representation of how far apart two objects are. Can be interchangeable with **Dissimilarity**

Matching (also known as *Fuzzy String Matching*) is a pattern matching algorithm that computes the degree of similartity between two strings, and produces a quantitative metric of distance that can be used to classify the strings as a match or not a match. Using *approximate string matching*, we will compute a *distance* between all pairs of records to obtain a *distance* (or dissimilarity) matrix.

The second step is grouping records (*SKU*) into *clusters*[4] (*PL*) by applying *clustering* methods. In clustering (or cluster analysis), observations are grouped into clusters in such a way that objects in the same cluster are more similar to each other than those in other clusters. Clustering requires an input of a distance matrix, which is computed in the first step, and outputs the grouping of clusters. To compare the results and performance of our approaches, several evaluation methods such as accuracy, purity, and F-measure will be used.

In this paper, several existing and modified methods were applied to both real data and simulated data.

3 Data

We will use product listings from *Amazon.com* as an example of real data. The data is obtained from listings related to electronics and electronic accessories. We then simulated data from the real data by making assumptions about the construction of product names and distributions of product name components.

3.1 Real Data

Our set of real data consists of 282 different product listings, separated into 12 *blocks*[5] of data. In our data, since we have information about the *Brand* and *Category* of the SKUs, we can block our data by those variables, since we know that SKUs from different **categories** or *brands* never belong to the same product line. We will refer to these subsets of data grouped by *Brand* and *Category* as *blocks*. A *block* of data then will look similar to Table 2.

Table 1: Generic Block of Consumer Product Data

Product Line	SKU
Product A	Product A Bluetooth Mobile Speaker Black
Product A	Product A Bluetooth Mobile Speaker Green
Product A	Product A Bluetooth Mobile Speaker Gray
Product B	Product B Bluetooth Mobile Speaker Black
Product B	Product B Bluetooth Mobile Speaker Red
Product C	Product B Tablet Case for iPad Black
Product C	Product B Tablet Case for iPad Gray

In our data set, we have 12 *blocks* (Table 2), each with varying number of *SKUs* and *Product Lines*.

Table 2: Frequency Table of SKU and PL in Real Data by Block

Block	1	2	3	4	5	6	7	8	9	10	11	12
number of observations (SKU)	22	15	45	43	14	27	21	8	34	6	14	33
number of unique class (PL)	7	5	8	8	3	3	8	2	3	3	3	6

[4]**Cluster**: The predicted class of one or more record(s) that belong to the same entity

[5]**Block**: A set or group of observations consisting of one or more classes, grouped by higher-order variables (i.e. list of SKU within a Product Category and Brand)

3.2 Data Simulation

Using real data and general assumptions made on the data, a simulation scheme was created to generate large samples of data. In order to develop a simulation scheme, we first defined the framework of the data. In our data, **Category** and **Brand** are two levels of the data hierarchy for which we have information. Record linkage is performed for each Category-Brand *blocks*. Therefore we need a simulation scheme that generates *blocks* of data (more under section titled **Simulation**). For the simulation portion of the study we generated 1,000 repititions (blocks) of data.

4 Methods

In order to accomplish our record linkage task, we propose two steps. The first step is calculating a *distance* or dissimilarity matrix, using Approximate String Matching. The second step is calculating *clusters* of the records using Clustering methods. We will then use several evaluation measures to compare performance. In our study we will apply and evaluate record linkage methods on both real data and simulated data.

4.1 Step 1: Approximate String Matching

Our ultimate goal is grouping strings that are similar and separating those that are not. In practice there are various simple approaches to solving this problem. One of which mentioned is the "brute force" approach of linking records by hand, which we have already dismissed due to its lack of scalability. Another approach is extracting parts of strings that are important, using regular expressions and search patterns (e.g. extracting "Apple iPhone 7" from a list of strings). However, there are too many search patterns to consider (an upwards of as many different entities exists in the data), thus this method is also costly in terms of time and effort. This approach would also rely on good or consistent data quality. However, in our scenario we have a special data construction that would be difficult to capture by the regex approach.

A classical approach to the record linkage problem is a probabilistic record linkage method known as *Approximate String Matching* (also known as Fuzzy String Matching). It is a pattern matching algorithm that computes the degree of similarity between two strings, and produces a quantitative metric of distance that can be used to classify the strings as a match or not a match.

Works of different approximate string matching methods can be identified by different types of approaches: 1) N-gram approaches, which considers the matching-ness of all substrings of length N between two strings (notably q-gram distance, Jaccard distance [10]), 2) edit distance based approaches (Levenshtein distance, Optimal String Alignment [6]), and 3) vector space representations (Term Frequency - Inverse Document Frequency) approaches[5], which generalizes matching-ness between two strings by looking at matching words, inversely weighted by the frequency of that word across all records.

In this study we will use the edit distance based approaches. We will consider these two algorithms. The former is an existing method, and the latter is a modification/adaptation of the first:

Table 3: Approximate String Matching Algorithms Considered

Method	Description
Optimal String Alignment (OSA)	calculates the "edit distance" between two strings
Weighted Optimal String Alignment (WOSA)	calculates the "edit distance" between two strings, where edits are weighted by location and determined by a function

The Optimal String Alignment approach and the Weighted Optimal String Alignment approach are different in its assumption about the data and the computation of the distance. These two algorithms will be applied to all pairs of records for a given *block* to obtain a *distance* matrix, to be used for clustering in step 2.

4.1.1 Optimal String Alignment

We will first discuss a classical approach called Optimal String Alignment (also known as Damerau Levenshtein Algorithm). It is an algorithm that calculates the "edit distance" between two strings[6]. An "edit" is identified by one of these four:

Table 4: Edit Distance Operations[6]

Edit	Description	Example
Deletion	deletion of a single symbol	**b**eat -> eat
Insertion	insertion of a single symbol	eat -> **b**eat
Substitution	substitution of a single symbol with another	**b**eat -> **h**eat
Transposition	swapping of two adjacent symbols	be**at** -> be**ta**

In the computation of the Optimal String Alignment distance for a given string A and string B, a function $d_{A,B}(i, j)$ is defined, where for any value of i and j, $d_{A,B}(i, j)$ denotes the distance of the prefix of string A from 1 to i, and the prefix of string B from 1 to j. The pseudocode for the optimal string alignment algorithm is as follows.

Listing 1: Pseudocode for Optimal String Alignment

```
Inputs: A = string A
        B = string B

OptimalStringAlignment = function(A, B){
  // Initialize matrix
  d = matrix(0, nrow = length(A)+1, ncol = length(B)+1)

  // Compute edit distance
  for(i in 1,...,length(A)+1){
    for(j in 1,...,length(B)+1){
      if(A[i-1] == B[j-1]) cost = 0 else cost = 1
      d[i,j] = min(d[i-1,j] + 1,        // deletion
                   d[i, j-1] + 1,       // insertion
                   d[i-1, j-1] + cost))  // substitution
      if(i > 2 & j > 2 & A[i-1] == B[j-2] & A[i-2] == B[j-1]) {
        d[i,j] = min(d[i,j], d[i-2, j-2] + cost) // transposition
      }
    }
  }

  // Obtain and return optimal string alignment
  osa = d[length(A)+1, length(B)+1]
  return(osa, d)
}
```

Optimal String Alignment takes values from 0 to the maximum number of possible edits that can occur to convert one string to another (which is the maximum of the length of both strings). Therefore, a direct comparison of the OSA measure is not appropriate. To scale the OSA, we can divide by the total number of possible edits to obtain a score of range [0, 1].

Let i = 1,. . . , n be all the potential edit locations, x_i be a binomial variable where 1 indicates the occurrence of an edit operation at that location and 0 indicates no occurrence. Then edit distance can be converted into a score by dividing by n, the maximum number of possible edits (i.e. length of longer string). We can then compute a score from the optimal string alignment as such:

$$Score_{osa} = \frac{\sum_{i=1}^{n} x_i}{n}$$

The pseudocode for the Optimal String Alignment Score is as such:

Listing 2: Pseudocode for Optimal String Alignment Score

```
OptimalStringAlignment_Score = function(A, B){
  // Compute optimal string alignment
  osa, d = OptimalStringAlignment(A, B)

  // Calculate score
  n = max(length(A), length(B))
  osa_score = osa/n
  return(osa_score)
}
```

Let us consider some examples from Table 1. We will compute the Optimal String Alignment and score for these two strings.

```
A = "Product A Bluetooth Mobile Speaker Black"
B = "Product A Bluetooth Mobile Speaker Green"
```

The Optimal String Alignment is the number of edits required to transform one description to the other. Between these two strings, edit operations occur at location (column index) 36, 37, 38, 39, and 40 (`Black` → `Green`). Thus x_i takes this form

$$x_i = \begin{cases} 0, 1 \leq i \leq 35 \\ 1, 36 \leq i \leq 40 \end{cases}$$

where $\sum_{i=1}^{n} x_i = 5$. With a maximum number of edits of n = 40, our score is $\frac{5}{40}$, or 0.125 out of a maximum of 1.

The computation of distance using approximate string matching is quick and scalable, and can have a very good performance with the right selection of algorithm and parameters. However, certain algorithms may only be appropriate for a specific structure of data. In this next example we will see that Optimal String Alignment may not be the most appropriate algorithm in our case.

```
A = "Product A Bluetooth Mobile Speaker Black"
C = "Product B Bluetooth Mobile Speaker Black"
```

In this case, the Optimal String Alignment is just one (`A` → `B`). Out of a maximum number of edits of 40, this difference then only accounts for 1/40 or 0.025 out of 1.

This is a problem for us. We can see that these two SKUs do not belong to the same product line (SKU **A** belongs to product line `A` while SKU **C** belongs to product line `B`), yet the score is very small and would not be able to distinguish these two SKUs. If our goal is to identify matching product lines, this method does not quite satisfy our goal.

4.1.2 Weighted Optimal String Alignment

The problem attributed to the Optimal String Alignment algorithm is that the occurrences of edit operations between the two strings are weighted equally regardless of its location, and thus the algorithm needs modification. This is where we can use some prior knowledge or assumptions about the kind of data we have. A generalization about product or SKU names is that model names appear towards the beginning of the string (the differences we want to highlight), while differences such as color variations or size appear towards

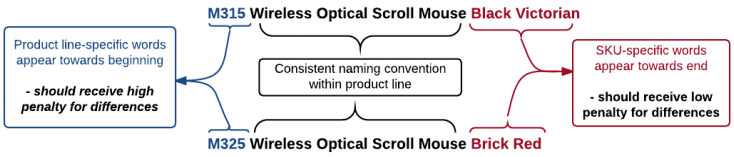

Figure 1: Data Assumptions

the end of the string (the difference we do not care about) (see Figure 1). To summarize, we have these two general assumptions about the SKU names:

1) Words specific to the SKU appear towards end of description → Should receive low weight/penalty
2) Words specific to the Product Line apper towards beginning of description → Should receive high weight/penalty

Thus we introduce weights into the Optimal String Alignment and design a modified algorithm which we will refer to as *Weighted Optimal String Alignment*. Weighted Optimal String Alignment is currently not in existence in other literatures, but it can have potentially useful applications. We will discuss its advantages in our scenario.

In Optimal string alignment, only the count of edits is considered. However, in Weighted Optimal String Alignment we consider the edits with some weights determined by location.

Listing 3: Pseudocode for Weighted Optimal String Alignment Score

```
WeightedOptimalStringAlignment_Score = function(A, B, ...){
  // Initialize
  wosa = NULL
  osa, d = OptimalStringAlignment(A, B)

  // Obtain locations where edit operations occurred
  if(ncol(d) == nrow(d)) {
    edits = difference(diagonal(d))
  } else {
    min_len = min(length(A), length(B))+1
    edits = difference(
      diagonal(
          d[1:min_len, 1:min_len]),
          d[min_len:(length(A)+1), min_len:(length(B)+1)][-1]
      )
    )
  }
  edit.at = which(edits != 0)

  // Create weight vector where f_w(x) is some function (e.g. constant, linear)
  max_len = max(length(A), length(B))
  w = f_w(1,...,max_len)

  // Calculate Score
  if(sum.right) {
    xt = Numeric(1:length(x) >= min(x == 1))
    wosa_score = w*xt/sum(w)
  } else {
    wosa_score = w*x/sum(w)
  }
  return(wosa_score)
}
```

Let w_i be the weight associated with location i, where the weights are defined by some function (e.g. linear function). Then the score for Weighted Optimal String Alignment is calculated as:

$$Score_{wosa} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

For example, let us compare the optimal string alignment approach to the weighted optimal string alignment approach. We consider these same two SKUs from a previous example:

```
A = "Product A Bluetooth Mobile Speaker Black"
B = "Product A Bluetooth Mobile Speaker Green"
```

Edit operations occur at column 36, 37, 38, 39, and 40. Previously we saw that in the optimal string alignment approach we would simply take the the sum $\sum_{i=1}^{n} x_i = 5$ divided by the total number of edits (n = 40), for a score of $\frac{5}{40} = 0.125$.

In Weighted Optimal String Alignment, we instead take the sum of the product of weights w_i and x_i. Let us for example consider *linear* weights. Since our max length is 40, let us define the (negative-slope) linear weight function as

$$w(i) = n - (i - 1), i = 1, ..., n$$

where $n = 40$. If our x_i be a binomial variable of edit at location i, then our score is $Score_{wosa} = \frac{5+4+3+2+1}{820} \approx 0.0183$. This WOSA score is significantly smaller than the OSA score (0.0183 vs. 0.125) and thus we can probabilistically identify SKU **A** and **B** as a match more confidently.

However we need to consider further modifications to our WOSA algorithm. Let us consider another set of SKUs that do not belong to the same Product Line:

```
A = "Product A Bluetooth Mobile Speaker Black"
D = "Product B Bluetooth Mobile Speaker Red"
```

Edits occur at column 9, 36 37 38 39 40. Then our score is $Score_{wosa} = \frac{32+5+4+3+2+1}{820} \approx 0.0573$. Although this score is comparatively larger than our previous example (0.0573 vs. 0.0183), arguably it is still too small to distinguish these two SKUs.

Let us consider additional parameters in our WOSA algorithm (Table 5). We have already discussed the user-selected option of the *weight function* (e.g. quadratic function, square root function). We can also consider the `type` of "item" to compute the edit distance on. In previous examples we used individual `characters`, but an alternative approach is to consider edit operations on whole `words` (i.e. if the entire word matches or not). The advantage of this is that any difference between two words (i.e. not exact spelling) disqualifies the entire word as a match. This is particularly effective for comparing product line names since they can share very small differences (e.g. iPhone 6 vs. iPhone 7). Another parameter to consider is the `sum.right` option. The sum.right' parameter will recognize all "items" after the occurrence of the first mismatch as mismatches. Mathematically we can represent this as

$$x_i = \mathbb{1}(i \geq \alpha), i = 1, ..., n$$

where α is the location of the occurrence of the first edit operation. This is another parameter designed specifically to identify different product lines.

Table 5: Parameters for Weighted Optimal String Alignment

Parameter	Description
`type`	type of "item" to compute edit distance on (by individual `character` or by whole `words`)

Parameter	Description
`weight`	weight function (`constant` (which is equivalent to OSA), `linear`, `quadratic`, `root`)
`sum.right`	flag (`T` or `F`) to recognize all "items" after the occurrence of the first mismatch as mismatches (if `T`, $x_i = \mathbb{1}(i \geq \alpha), i = 1, ..., n$)

In the same example with SKU `A` and SKU `D`, let us compute the distance with the `sum.right` option on. First edits occur at column 9. Then all occurrences on and after the 9th column will also be considered mismatches (in other words, $x_i = \mathbb{1}(i \geq 9)$). Then with linear weights ($w(i) = 40 - (i-1), i = 1, ..., 40$), our score is $Score_{wosa} = \frac{32+31+...+2+1}{820} \approx 0.6439$. Compared to our previous score of 0.0573, this is a much more probabilistically distinguishable score. Under this algorithm, the WOSA score between SKU `A` and SKU `B` remains the same, thus this modification is conveniently suitable.

For the purpose of this study, we will use `type = "item"` because this way the length of the word (i.e. number of characters) will not affect the distance. We will also use `sum.right = T`, because we want to identify difference in model (or Product Line) names, and any difference occurring in the string after that should also be considered different. Furthermore we will proceed with the linear weight function.

To produce the distance matrix for a given list of SKUs, approximate string matching algorithms are applied for every pair of SKUs.

4.2 Step 2: Unsupervised Clustering

Using the distance matrix obtained for a given block or list of SKUs, we apply clustering methods to predict the class of the observations. Since in reality we do not know the true classes (Product Line) of the observations (SKUs), we need unsupervised methods. We do not even know the number of clusters, so methods such as K-means would not be appropriate. Here we will introduce the use of two clustering methods: Hierarchical Clustering, and Density-Based Spectral Clustering (DBSCAN). We will also introduce an adaptive (2-stage iterative) process where the weights used in the *weighted optimal string alignment* algorithm is updated based on the results of the clustering (and run until convergence of clusters or a user-specified max number of iteration is reached).

4.2.1 Hierarchical Clustering

Hierarchical clustering, as the name implies, is a method that builds clusters in a hierarchical manner. Hierarchies can be built in two ways. In the Agglomerative approach, the hierarchy is built from the bottom up where each observation starts in its own clusters, and is paired/grouped as the observations are traced up on the hierarchy. The Divisive approach works in the opposite manner, where we start with a single large cluster of all observations, and observations are split recursively from the top down and branching off into new clusters, creating a hierarchy[9]. Hierarchical clustering has a variety of linkage criteria for determining the distance between sets of observations. In our study we will consider single-linkage and complete-linkage.

In the **single-linkage** clustering approach, the distance between two clusters (or set of observations) is determined by the distance of the closest members. This means that between two clusters we only consider the points that have shortest distance possible. In this method the rest of the cluster and the overall structure of the cluster is not taken into consideration.

In the **complete-linkage** clustering approach, the distance between two clusters (or set of observations) is determined by the distance of the furthest members. In this approach, the overall structure of the cluster is considered. One issue with this approach is that the linkage will be sensitive to outliers, and single observations within cluster that are further from the rest of the points can largely affect the cluster's linkage with other clusters.

Let us consider the agglomerative (bottom-up) approach where all observations initially start as its own cluster. Then at any given point of the bottom-up approach, for all pairs of clusters A and B clusters where $A \neq B$ and given a distance matrix d, the clusters to be joined next is determined by the following for the mentioned two methods:

$$\text{single-linkage} \rightarrow min(d(x, y) : x \in A \ \& \ y \in B)$$
$$\text{complete-linkage} \rightarrow max(d(x, y) : x \in A \ \& \ y \in B)$$

Listing 4: Pseudocode for Agglomerative Hierarchical Clustering

```
Inputs: d = distance matrix
        method = clustering method (single vs. complete)

AgglomerativeHierarchicalClustering = function(d, method){
  while (all observations NOT in one cluster) {
    // Find 'closest' clusters
    A, B = ClosestClusters(d, method)
    // Merge clusters
    MergeClusters(A, B)
    // Update distance matrix by deleting the rows/columns corresponding to clusters A and B from
        d matrix, add row/column corresponding to the new cluster
    UpdateDistanceMatrix(d, A, B)
  }
}
```

The arrangements of clusters in hierarchical clustering can be visualized using a *Dendrogram*, a type of tree diagram (see Figures 2 and 3). With the height (or distance) on the y-axis, the top of the tree represents all observations under a single cluster, and from the top down each node represents a split of the cluster into smaller clusters at the corresponding height.

There are two ways of obtaining clusters from hierarchical clustering; one is to supply a number of clusters to obtain, and the other is to supply a cutoff value to cut the dendrogram at. The former requires prior knowledge on the number of clusters, and in our scenario we do not have such information, so we proceed with the second way.

It is critical that an appropriate cutoff is selected: for example, if a cutoff is too high, then all the observations will be placed in one cluster; if the cutoff is too low, every single observation will be in its own cluster.

In Figures 2 we compare the OSA and WOSA distance matrices using single-link Hierarchical Clustering (complete-link Hierarchical Clustering in Figure 3).

One complication with the use of hierarchical clustering in our scenario is the lack of an appropriate way of selecting the right cutoff value. In general, hierarchical clustering is an exploratory method that allows for visual exploration as in these dendrograms. Determine a cutoff value is most commonly determined by inspecting the dendrogram and selecting the cutoff where the vertical branches (uninterrupted by splits) is the longest.

4.2.2 Density-Based Spectral Clustering of Applications with Noise (DBSCAN)

The problem with hierarchical clustering is its sensitivity to the shape or structures of the cluster. The single-linkage hierarchical clustering will only consider the closest members between two clusters and the complete-linkage will only consider the furthest members between two clusters. We now consider another clustering method known as *density-based spectral clustering of applications with noise* (DBSCAN)[4].

DBSCAN browses the neighborhood of each point for other points within a given space, and only links clusters when a certain number of *density-reachable* points are within the neighborhood. In the DBSCAN algorithm, each point can be considered as a 1) core point, 2) density-reachable point, and/or 3) outlier.

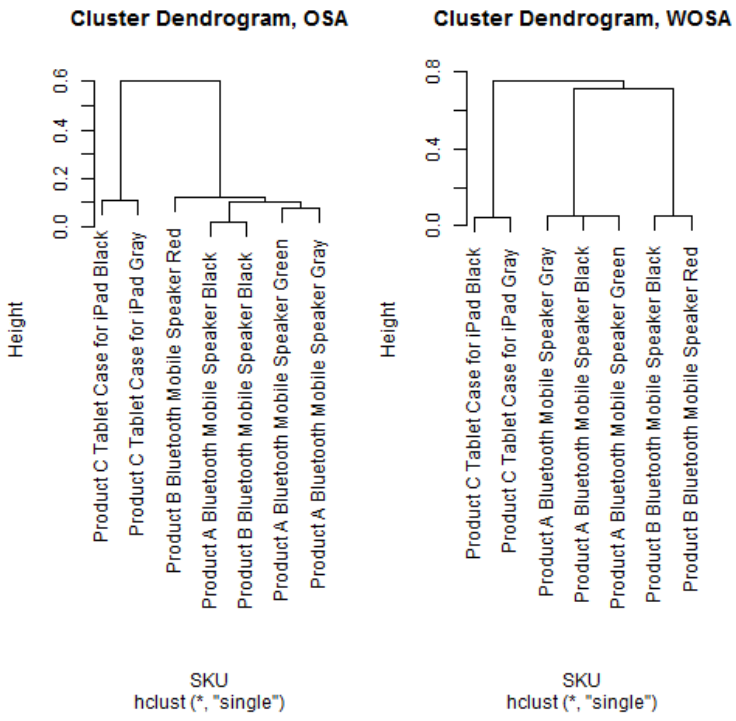

Figure 2: Dendrogram of Single-Link Hierarchical Clustering

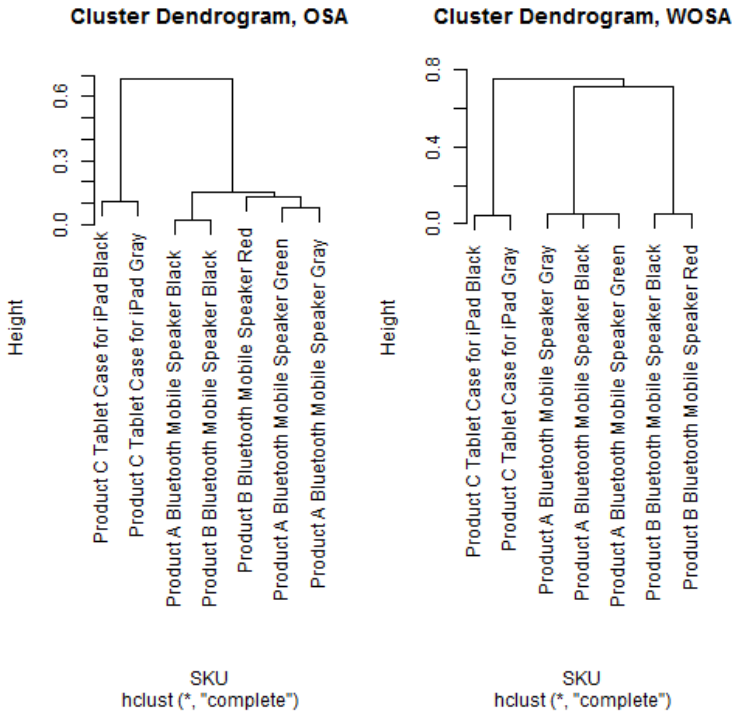

Figure 3: Dendrogram of Complete-Link Hierarchical Clustering

For each point p not yet visited, we inspect its neighborhood. If there exist x points within a distance ϵ and is of size greater than or equal to a minimum n points, then point p is considered a core point with x directly reachable points and considered a cluster C. Then for each directly reachable points p' we repeat the same process and all points directly reachable from points p' are added to cluster C until no other points can be added. If the initial condition is not satisfied for point p, then it is considered an outlier[4].

Listing 5: Pseudocode for Agglomerative Hierarchical Clustering

```
Input: d = distance matrix
       epsilon = size/boundary of neighborhood
       min_points = minimum points required in neighborhood of epsilon

DBSCAN = function(d, epsilon, min_points){
    C = 0
    for each unvisited point P in dataset
          mark P as visited
          sphere_points = regionQuery(P, epsilon)
          if sizeof(sphere_points) < min_points
                ignore P
          else
                C = next cluster
                expandCluster(P, sphere_points, C, epsilon, min_points)
}

expandCluster(P, sphere_points, C, epsilon, min_points):
     add P to cluster C
     for each point P' in sphere_points
           if P' is not visited
                mark P' as visited
                sphere_points' = regionQuery(P', epsilon)
                if sizeof(sphere_points') >= min_points
                     sphere_points = sphere_points joined with sphere_points'
                if P' is not yet member of any cluster
                     add P' to cluster C
```

Note: Pseudocode adopted from[4]

The DBSCAN algorithm has a few notable advantages over the hierarchical clustering. One of which is that, whereas the hierarchical clustering method requires a cutoff value to determine clusters after the cluster hierarchy and thus requiring one to either have prior knowledge (or make a guess) about an appropriate cutoff value or manually tune the cutoff value ad hoc, the DBSCAN algorithm uses parameters established pre-computation to determine clusters. Another advantage is that DBSCAN is less sensitive to irregular cluster shapes/structures compared to hierarchical clustering methods.

4.2.3 Adaptive (Weighted) Optimal String Alignment with DBSCAN

We also propose a method that combines the first and second step using an iterative updating process to build an *adaptive* record linkage procedure. We will utilize *weighted optimal string alignment* and *DBSCAN* to construct a 2-stage adaptive procedure. We will refer to this algorithm as the *Adaptive (Weighted) Optimal String Alignment with DBSCAN*.

Initially, the distance matrix and clusters are obtained as usual during the first iteration. Then using the results of the cluster, we repeat the process with a modification on the *WOSA* (*Adaptive WOSA or AWOSA*) algorithm by computing and applying a penalization (or reweighting) vector λ_i as such:

$$Score_{awosa} = \frac{\sum_{i=1}^{n} \lambda_i w_i x_i}{\sum_{1}^{n} w_i}$$

Note that in the initial step, λ_i is simply equivalent to 1, for all value of i (thus equivalent to the original WOSA algorithm). Additionally, if the specified weights w_i take a value of 1 for all i, this is then equivalent to the original OSA algorithm.

λ_i is a representation of the percent uniqueness of the record at column i (where columns are `character` or `words` as specified in the WOSA algorithm). It is the fraction of the number of mismatching pairs of `items` over the total the number of pairs of `items` at the ith location in a set of records.

For $Z \in z_1, ..., z_n$ where Z is a set records in cluster c of size n, and i denotes the ith column in a record z (with a maximum value of m), we compute percent uniqueness λ_i as such:

$$\lambda_i = \frac{\sum_{k=1}^{n} \sum_{j<k}^{n} \mathbb{1}(z_j(i) \neq z_k(i))}{\binom{n}{2}}; i = 1, .., m; j, k = 1, ..., n$$

Table 6: Example: λ_i in a set of records

Location (i)	1	2	3	4	5	6	7
SKU 1	Product	A	Bluetooth	Mobile	Speaker	Midnight	Black
SKU 2	Product	A	Bluetooth	Mobile	Speaker	Dark	Green
SKU 3	Product	A	Bluetooth	Mobile	Speaker	Dark	Blue
λ_i	0	0	0	0	0	2/3	1

A vector of penalty factors λ_i are given to all pairs of records in a given set. For cluster c, all pairs within the cluster are given the penalization vector λ_i, which is computed specifically for cluster c. This is done for all clusters. For records belonging to different clusters, the associated λ_i vector will be equal to 1 for all i (thus equivalent to the original WOSA algorithm).

This 2-stage procedure is repeated until one of two stopping criteria is met. The first condition is the maximum number of iterations to run (`max.iter`). The second condition is the convergence of estimated clusters (i.e. if the clusters estimated from the current iteration exactly matches the clusters from the previous iteration).

Listing 6: Adaptive (Weighted) Optimal String Alignment with DBSCAN

```
Input: x = records
       max.iter = maximum number of iterations
       ... = additional arguments for approximate string matching/clustering algorithms

Adaptive_WOSA_DBSCAN = function(x, max.iter, ...){
  iter = 0; lambda0 = 1

  // Initial distance matrix (d0) and clusters (clust0) using lambda0 = 1
  d0 = AWOSA(x, lambda0 ...)
  clust0 = DBSCAN(d0, ...)
  // Loop until convergence of cluster OR reach max iteration
  while(iter < max.iter){
    iter = iter + 1
    // Compute lambda to reweigh weights
    for each cluster in clust0 {
      lambda[cluster] = ComputeLambda(x[cluster])
    }
    // recompute distance matrix (d) and clusters (clust) using new lambda
    d = AWOSA(string_list, lambda, ...)
    clust = DBSCAN(d)
    // Stopping criteria
    if(all(clust == clust0)) break
```

```
    // If no convergence, repeat
    d0 = d
    clust0 = clust
  }
  return(clust)
}
```

The motivation behind the *adaptive* approach is that after the initial iteration, the algorithm focuses its attention to the validity of clusters by reweighting and reexamining its distance. Thus this aims to reduce the number of false positives that have occurred in the estimation of clusters (i.e. pairs in clusters that do not actually belong together).

4.3 Evaluation

There are several ways to compare the clusters to the actual class (Product Line) of the observations (SKUs). Since not one evaluation measure is better than the rest, we will consider a few of them simultaneously. The evaluation measures we will consider are the accuracy, purity, and F1 Score[13].

The simplest of measures is the accuracy. In clustering, accuracy is defined as:

$$Accuracy = \frac{TP + TN}{P + N}$$

Where P = all positives (number of within-cluster pairs), N = all negative (number of between-cluster pairs), TP = true positives (number of within-cluster pairs that are true), and TN = true negatives (number of between-cluster pairs that are true).

Another measure that is commonly used in evaluating the performance of clusters is known as purity, which is a measure of how "pure" each cluster is. In the calculation of purity, for each cluster we count the frequency of the most frequently-appearing class. That is then summed and then divided by the total number of observations. Mathematically it is defined as:

$$Purity = \frac{1}{n} \sum_{q=1}^{k} \max_{1 \leq j \leq l} n_q^j,$$

where n is the total number of observations, $q = 1, .., k$ is the cluster index, l is the number of true classes, $j = i, .., l$ is the class index, and n_q^j is the frequency of class j in cluster q.

The problem with purity is that it is insensitive to the number of clusters. Hypothetically, if every observation was in its own cluster, then all clusters will have 100% purity, and thus the total purity will also be 100%.

We will introduce one more measure, known as the F_1 score (or F-measure). The F_1 score considers the harmonic mean of the precision and recall, which are not considered in neither accuracy nor purity. The F_1 score is calculated as:

$$F - score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

where

$$Precision = \frac{TP}{TP + FP}$$

and

$$Recall = \frac{TP}{TP + FN}$$

In general, purity is not a good measure of evaluation due to the forementioned issue, so we will primarily consider accuracy and F1 Score in evaluating the performance of our methods.

5 Results

We will evaluate the measures from the two Approximate String Alignment algorithms (OSA and WOSA) combined with three clustering approaches (single-link/complete-link Hierarchical Clustering, DBSCAN), and on the Adaptive (Weighted) Optimal String Alignment with DBSCAN algorithm. These procedures will be applied on two data sets; one is the real data obtained from Amazon listings, and the other is simulated data. For the simulated data we will also assess the run time of clustering methods.

In general, WOSA will have a tendency to give a smaller score than OSA for our kind of data, so for Hierarchical clustering, we have fixed the cutoff value at 0.3 for OSA and 0.2 for WOSA. For the DBSCAN, we have fixed the ϵ value at 0.3 for DBSCAN.

5.1 Real Data

The forementioned Approximate String Matching and Clustering Methods were applied on the real data. Here are the results of the evaluation measures and run time.

Table 7: Results (Real Data): Optimal String Alignment

	Accuracy	Purity	Precision	Recall	F1 Score	Average Time (seconds)
Hierarchical Clustering (Single-link)	0.9254	0.9470	0.8525	0.8301	0.8148	0.8133
Hierarchical Clustering (Compete-link)	0.9254	0.9470	0.8525	0.8301	0.8148	0.8175
DBSCAN	0.8663	0.8560	0.8283	0.9653	0.8454	0.8167
Adaptive OSA with DBSCAN	0.8796	0.9055	0.8671	0.8696	0.8167	3.1708

Table 8: Results (Real Data): Weighed Optimal String Alignment

	Accuracy	Purity	Precision	Recall	F1 Score	Average Time (seconds)
Hierarchical Clustering (Single-link)	0.9277	0.9470	0.9359	0.8447	0.8243	1.003
Hierarchical Clustering (Compete-link)	0.9277	0.9470	0.9359	0.8447	0.8243	1.003
DBSCAN	0.9311	0.9135	0.8717	0.9721	0.8970	1.008
Adaptive WOSA with DBSCAN	0.9800	0.9636	0.9513	0.9468	0.9443	3.018

The first notable observation is that the results from single-linkage vs. complete-linkage hierarchical clustering gives the same measures for all evaluations, and presumptuously the same clustering results. This means that the structure of clusters (as discussed in the Methods section) does not have much influence on the cluster linkages. Another possible explanation is that observations in each cluster is fairly compact (close together), and separated well from other clusters.

We now compare DBSCAN to Hierarchical clustering. DBSCAN does not always outperform Hierarchical Clustering, but the best results from this whole table comes from DBSCAN with WOSA, with the highest accuracy and F1 score measures. However, DBSCAN has a considerably higher measure of recall than precision (i.e. high number of false positives), which arguably is a cause for concern.

As previously mentioned, the adaptive method WOSA with DBSCAN aims to reduce the false positives (which is used to measure precision). In the OSA algorithm the reduction in recall does not justify the increase in precision. However, with WOSA the results are remarkable as it improves the precision substantially with a relatively lower decrease in recall, and also considerably improves the F1 score. The one downside to the Adaptive WOSA w/ DBSCAN approach is the time consumption (it takes about three times as long to run this algorithm compared to all others).

5.2 Simulation

Using real data and general assumptions made on the data, we created a simulation scheme to generate large samples of data. In order to develop a simulation scheme, we first define the framework of the data.

In our data, *Category* and *Brand* are two levels of the data hierarchy for which we have information. Record linkage is performed for each Category-Brand groups. Therefore, we need a simulation scheme that generates random SKU names within a defined Category-Brand, or *Blocks*. Thus we can ignore those two levels.

Therefore, we need a simulation scheme that generates a list (or *Block*) of SKUs and its class. In a given block there will be a random number of classes, each of some random size, where the individual observations are SKU names. Each of these SKU names will consist of a random number of words, and the names are constructed randomly.

In general, the SKU names are constructed with words that are either specific to the Product Line (e.g. model name, type of product) or words that are specific to the SKU (color, size, etc).

These are the mentioned components of the data values that we need to randomly generate:

1. **Number of Classes (Product Lines)**: In a given group, the number of classes vary. We will use a discrete uniform distribution using the range of the number of classes in the real data (min = 2, max = 8).
2. **Size of class**: For each of the classes generated in **1.**, the number of SKUs in the class also varies (see Figure 5 in Appendix). We roughly estimated this distribution using a Gamma distribution.
3. **Length of SKU**: After class size is determine, we generate SKU names. Since not all SKU names have the same number of words, we generate the SKU length randomly (see Figure 6 in Appendix). We roughly estimated this distribution using a Gamma distribution.
4. **Word type**: When generating a SKU name, words are sampled from a list of words produced from the original data. These words are separated into two lists: 1) words that describe the product (Desc), and 2) words that describe features of the SKU (SKU); e.g. colors, size, etc. Using the real data, we computed a bernoulli probability of word type as a function of location x (see Figure 7 in Appendix). We roughly estimated this distribution using an exponential function.

To generate the sample SKU (after length of SKU is randomly generated), for each word location we obtain the `type` of word randomly from the bernoulli distribution as a function of location, and then randomly select a word from a list of words associated with the `type` of word. These selected words are then concatenated to generate the sample SKU.

The data components are broken down as the following:

Table 9: Simulation Scheme

Component	Distribution
# Classes (Product Lines)	$\sim Uniform(2, 3, 4, 5, 6, 7, 8)$
Class Size (# SKUs in Class)	$\sim \Gamma(k = 2.92, \theta = 0.61)$, rounded to nearest integer
Length of Description (# Words in SKU)	$\sim \Gamma(k = 4.07, \theta = 0.32)$, rounded to nearest integer
Word Type (Product vs. SKU-descriptive)	$\sim$ Bernoulli(p), where p = P(Type = Product $\mid$ x) $= 1 - exp(-7.5(1 - x))$, x = column location of word as a proportion of total length

This is an example of what the simulated data looks like (first 10 rows):

```
##                                                              SKU Class
## 1  surface mobile bose 35mm bass x300 ii male 10inch ltecoralmi...     1
## 2  surface mobile bose 35mm bass x300 ii male 10inch aquatic mi...     1
## 3  surface mobile bose 35mm bass x300 ii male 10inch miniblackp...     1
## 4  surface mobile bose 35mm bass x300 ii male 10inch genbabyblu...     1
## 5  surface mobile bose 35mm bass x300 ii male 10inch matte dark...     1
## 6  surface mobile bose 35mm bass x300 ii male 10inch atmosphere...     1
## 7  surface mobile bose 35mm bass x300 ii male 10inch sage minib...     1
## 8  splashproof bass bass ii parallax 2015 machine backlit 106 m...     2
## 9  splashproof bass bass ii parallax 2015 sugarplum backlit 106...     2
## 10 splashproof bass bass ii parallax 2015 wired backlit 106 dee...     2
```

After obtaining 1,000 blocks of data generated using the simulation scheme, we obtained the evaluation measures and run time:

Table 10: Results (Simulation): Optimal String Alignment

	Accuracy	Purity	Precision	Recall	F1 Score	Average Time (seconds)
Hierarchical Clustering (Single-link)	0.9307	1.0000	0.9770	0.7129	0.7894	0.7185
Hierarchical Clustering (Compete-link)	0.9307	1.0000	0.9770	0.7129	0.7894	0.7185
DBSCAN	0.9298	0.9016	0.8313	1.0000	0.8877	0.7190
Adaptive OSA with DBSCAN	0.9760	0.9628	0.9319	1.0000	0.9571	4.2860

Table 11: Results (Simulation): Weighed Optimal String Alignment

	Accuracy	Purity	Precision	Recall	F1 Score	Average Time (seconds)
Hierarchical Clustering (Single-link)	0.9688	1.0000	0.9980	0.8702	0.9123	0.7156
Hierarchical Clustering (Compete-link)	0.9688	1.0000	0.9980	0.8702	0.9123	0.7156
DBSCAN	0.9976	0.9954	0.9917	1.0000	0.9951	0.7163
Adaptive OSA with DBSCAN	0.9993	0.9987	0.9980	1.0000	0.9988	2.5984

We see that simlarly to the results on real data, single-linkage and complete-linkage clustering gives the same results. In general, the WOSA evaluation measures are better than the OSA measures for hierarchical clustering.

The hierarchical clustering methods appear to give perfect purity and a very high precision, but not so well in recall. This means that the method is able to accurately separate observations that do not belong together, but fails to identify a large number of observations that do belong together. One cause of this could be from the cutoff value being too conservative (too low). However, for OSA we are using a cutoff value of 30%, which in general is already on the larger side, so perhaps the OSA algorithm is not appropriate.

On the other hand, DBSCAN has a good performance overall, especially with WOSA. It is able to almost always accurately predict the correct class, without sacrificing precision or recall.

Again we see that Adaptive WOSA with DBSCAN out performs all other methods, on all measures. These results are analogous to the results on real data.

6 Conclusion

Our record linkage procedure on consumer product data was performed in two steps. In the first step we obtained the distance matrix using two different methods of approximate string matching (Optimal String Alignment vs. Weighted Optimal String Alignment). In the second step we computed clusters using three methods of clustering (Single-link Hierarchical Clustering vs. Complete-link Hierarchical Clustering vs. Density-Based Spectral Clustering of Applications with Noise). Additionally, we implemented a 2-stage adaptive procedure (Adaptive Weighted Optimal String Alignment with DBSCAN). Several evaluation measures were considered, but final evaluation was primarily focused on the accuracy and F-measure.

Optimal String Alignment is a classical approach and is a widely known method for tackling the record linkage problem. However, due to the unique nature of our data construction, we proposed the Weighted Optimal String Alignment approach, which accounts for the special data structure through location-weighting feature. For the reason that the WOSA was designed to suit this special kind of data, we expect WOSA to perform better than OSA. This is reflected in the results as on all accounts WOSA performed better than or just as well as OSA did on all evaluation measures (Table 7 vs. Table 8 for real data, Table 10 vs. Table 11 for simulated data). On average, the WOSA approach had a 4.2% improvement in accuracy and 4.7% improvement in F1 Score over the OSA for real data, and 3.7% in accuracy and 8.7% in F1 Score for simulated data.

We proposed three methods for clustering, the first two of which fall under the family Hierarchical Clustering methods (single-linkage vs. complete-linkage). The motivation behind comparing these two methods is the comparison of these two methods to the sensitivity of the cluster shapes/structures in our data. However, our results between single-linkage and complete-linkage clustering is almost indistinguishable. In general, hierarchical clustering methods gives a high measure in most evaluation measures, but performs poorly on recall, which is a result of the high false negative rate. Thus in general hierarchical clustering suffers from a high Type II Error (falsely separating pairs that in reality belong together).

This issue with hierarchical clustering is resolved with the use of DBSCAN as its results produced a much higher measure of recall, while sacrificing precision. However, in general the reduction in precision is justified by the even larger improvement in recall, which reflects on the higher measure of F1 score. The improvement of DBSCAN over Hierarchical Clustering is more prevalent with WOSA than with OSA.

The Adaptive WOSA with DBSCAN had the best results (specifically with WOSA), with a 98% accuracy and .9443 F1 Score on real data (compared to the runner-up DBSCAN approach with 93.11% accuracy and .8970 F1 Score. The results are analogous in the simulated data as well.

We have seen some astonishing results, but there are still more approaches that can be explored. For example, the idea of location-weighting utilized in the Weighted Optimal String Alignment can be extend to other

approximate string matching algorithms. Furthermore, the iterative/adaptive approach can be generalized to other combinations of approximate string matching and clustering methods as well.

7 Appendix

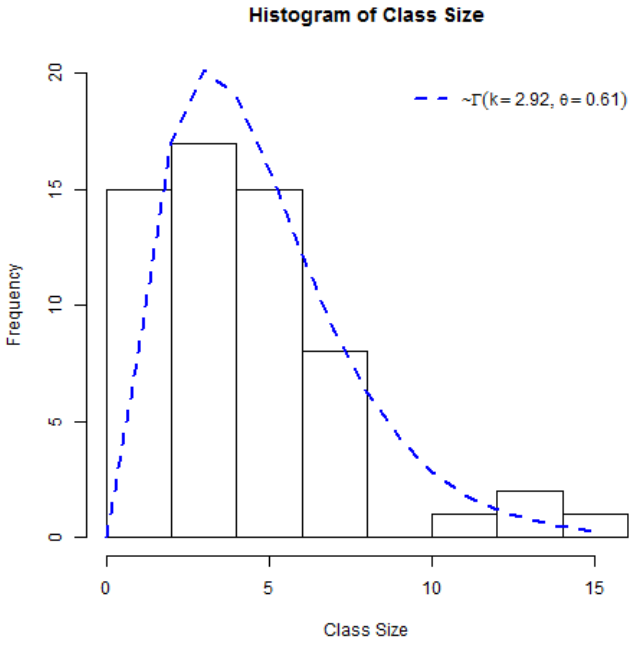

Figure 4: Simulation Scheme: Histogram of Class Size

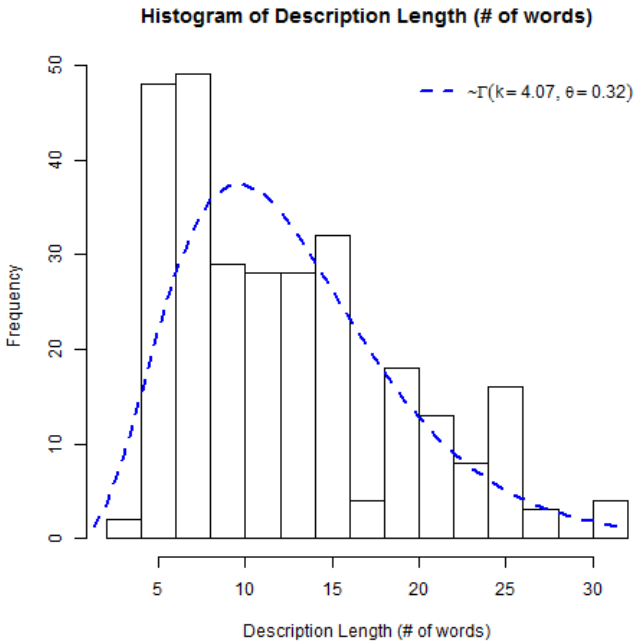

Figure 5: Simulation Scheme: Histogram of Description (Word) Length

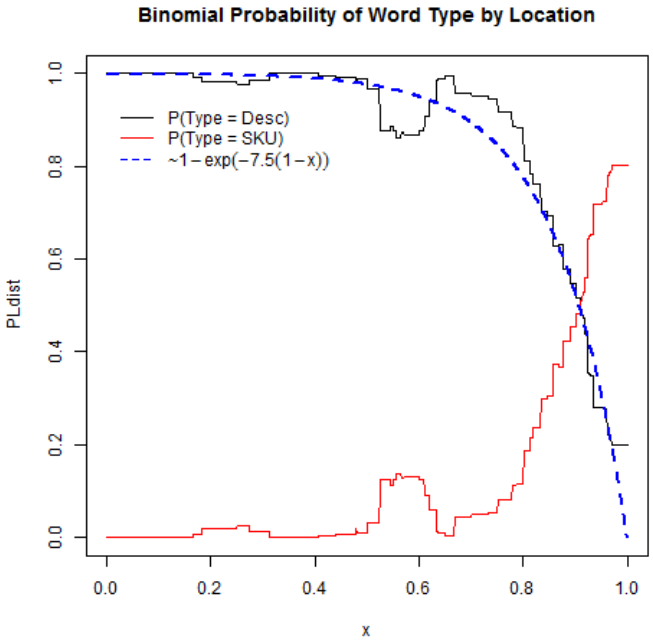

Figure 6: Simulation Scheme: Binomial Probability Function as a function of Location (X)

8 References

[1] Ariel, A., Bakker, B. F. M., de Groot, M., van Grootheest, G., van der Laan, J., Smit, J., & Verkerk, B. (2014). Record Linkage in Health Data : a simulation study.

[2] Christen, P., & Goiser, K. (2007). Quality and complexity measures for data linkage and deduplication. Quality Measures in Data Mining, 151, 127-151. https://doi.org/10.1007/978-3-540-44918-8_6

[3] Christen, P. (2012). A survey of indexing techniques for scalable record linkage and deduplication. IEEE Transactions on Knowledge and Data Engineering, 24(9), 1537-1555. https://doi.org/10.1109/TKDE.2011.127

[4] Ester, Martin, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." Kdd. Vol. 96. No. 34. 1996.

[5] Gu, L., & Baxter, R. (2003). Record linkage: Current practice and future directions. Cmis, 03/83. Retrieved from http://festivalofdoubt.uq.edu.au/papers/record_linkage.pdf

[6] Vladimir I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Doklady Akademii Nauk SSSR, 163(4):845-848, 1965 (Russian). English translation in Soviet Physics Doklady, 10(8):707-710, 1966.

[7] Ng, A. Y., Jordan, M. I., & Weiss, Y. (2001). On spectral clustering: Analysis and an algorithm. Advances in Neural Information Processing Systems 14, 849-856. https://doi.org/10.1.1.19.8100

[8] Sauleau, E. a, Paumier, J.-P., & Buemi, A. (2005). Medical record linkage in health information systems by approximate string matching and clustering. BMC Medical Informatics and Decision Making, 5, 32. https://doi.org/10.1186/1472-6947-5-32

[9] Schütze, Hinrich. "Introduction to Information Retrieval." Proceedings of the international communication of association for computing machinery conference. 2008.

[10] Ukkonen, Esko. "Approximate string-matching with q-grams and maximal matches." Theoretical computer science 92.1 (1992): 191-211.

[11] van der Loo, M. (2014). stringdist: an R Package for Approximate String Matching. The R Journal, 6(1), 111-122.

[12] Winkler, William E. "Matching and record linkage." Business survey methods 1 (1995): 355-384.

[13] Zhao, Y., & Karypis, G. (2002). Evaluation of Hierarchical Clustering Algorithms for Document Datasets, 515-524.