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

Riki Saito

October 18, 2016

Contents

Introduction

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. It is necessary in aggregating or joining data by some entity that may contain duplicates but do not share a common unique identifier due to slight differences or inconsistencies. Compared to a "brute force" approach of linking records by hand, record linkage is an elegant solution to combining duplicate, redundant, or even similar (but not quite the same) records. It has applications in a wide variety of fields; for instance, in health data linking medical records of the same individuals may be necessary.

In this paper we focus on the application of record linkage in consumer Point-of-Sales (POS) Data.

Compareing products from different online stores

Background

A large issue in joining or aggregating data on consumer products is that, compared to data like medical records of individuals, there are a lot more ways of representing duplicate records, and the data structure/hierarchy may not be clearly outlined. Let's first define some key terms:

Definitions

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 5, UE Boom 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 5 64GB Black, UE Boom 2 Blue)

Record Linkage in Consumer POS Data

With the increasing size and availability of data, the need for mainting data quality is also an issue that has surged in all industries, including market analytics.

Discussion of Data Source Variety

In market analytics at Logitech, due to the variety of product categories and the global presence of the company, we obtain data from a number of different sources. As a result, we face the issue of data reported slightly differently. A major issue that we encountered were the levels of granularity or hierarchy that the data is reported in, and inconsistencies with names of the Stocking Keeping Units and Product Lines.

A typical data hierarchy in market data might appear like this:

Data Hierarchy	Description	Example
Category	General Category of products	Pointing Devices
Brand	Name of Brand or Company	Logitech
Product Line	Group of products, defined by a model name	3Button M325 Wireless Optical Scroll Mouse
Stock Keeping Unit	A product, defined by its model and features (color, size, type, etc)	3Button M325 Wireless Optical Scroll Mouse Brick Red

The problem we face is that some data sources only provide the Product Line (PL), while others will only provide the Stock Keeping Unit (SKU). Because of this, we are not able to join data at neither the Product Line nor the Stock Keeping Unit, and therefore we are not able to get an accurate measure of sales for each SKU or PL. This hinders the ability to perform market analysis and uncover meaningful insights that could be used to inform business decisions, such as increasing or decreasing production of certain SKUs or PLs.

In one of our data source, the sales were reported at the SKU level, but no data was available for the Product Line to which these SKUs belong to. Here is an example of how the data might appear:

Tables in TOC!

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

Problem here: we are not able to product Product Line Rankings and only compare sales at the SKU level

Study Goals

What we need:

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

Study goal: 1) compare traditional vs. modified approximate string matching algorithm, and 2) compare performance of different clusering methods.

Motivation: Comparison to "Brute Force"

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.

Cartesian

Comparisons

In an ideal scenario, if two products have an exact match, the result is a binary and can only take two values.

BUT we don't have approximate matches... so we use approximate match, which takes any value from range 0 - 1.

Data

The data used by Logitech is unfortunately unavailable publicly, so we have prepared our own data that have similar nature. The data we prepared comes primarily from Amazon's product offering names.

Assume no typos, data preprocessed

Amazon?

Data Simulation

$$
\begin{aligned}
&\#classes \sim Uniform(2, 8) \\
&classsize \sim \Gamma(k = 2.92, \theta = 0.61) \\
&desclength \sim \Gamma(k = 4.07, \theta = 0.32) \\
&P(Type = 1|X) = 1 - exp(-7.5(1 - x))
\end{aligned}
$$

1: Data Cleaning/Standardization

Discussion about data (string) cleaning - white space - punctuation - all lower case - UTF8 Code?

Use of Blocking Variables

Blocks: Category, Brand

reduces computation from (equations!)

Methods

In order to identify which descritions belong together, this procedure is done in two steps: 1) calculating a distance matrix using approximate string matching, and 2) grouping descriptions together using clustering.

DIAGRAM

ARGUE USE OF WEIGHTED OPTIMAL STRING ALIGNMENT

Step 1: Approximate String Matching

Essentially what we are hoping to accomplish is match and group strings that are very similar but not quite exactly the same. Some simple and intuitive solutions would be to generate a substring of one string to match another, or identify and remove irrelevant words, but these solutions may not be the best solution simply because of the varied structures of how these strings are constructed and the ambiguity in determine whether words are irrelevant or not in identifying the product line.

Since we are dealing with a large data set (~250,000 rows) with varied structures and vocabulary of words, we sought for a fast and scalable approach that is robust to different string structures and vocabularies.

The method that we ultimately selected is called Fuzzy String Matching (also called Approximate String Matching). Fuzzy String Matching is a pattern matching algorithm that computes the degree of similartiy 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.

(Source)

Algorithms Considered

Method	Description	Parameters
Optimal String Alignment (OSA)	calculates the "edit distance" between two strings	none
Weighted Optimal String Alignment (WOSA)	calculates the "edit distance" between two strings, where edits are weighted by location and determined by a function	weight function

A large issue with assessing the performance of different fuzzy string matching algorithms is that, we do not have a means of checking the accuracy of the matching other than by manual inspection. Therefore, rather than blindly test algorithms and check each SKU for the accuracy of matching, we made some generalizations and assumptions about the data, and selected/tweaked algorithms and parameters that most closely aligned the assumptions.

Grouping Stocking Keeping Units (SKU) to Product Lines

As we saw in table (?), the first scenario is identifying the Product Line from the descriptions of the SKU. In general, the product descriptions are constructed with words that are specific to the Product Line (e.g. model name, type of product) as well as those specific to the SKU (color, size, etc). Out of a number of product descriptions, our goal is to identify ones that belong to the same Product Line.

Initially, we noted a clear patterns in these product descriptions, specifically in the similarity of sequence/choice

of words in descriptions that belong to the same Product Line, and its distinction with sequence/choice of words of other Product Lines. To summarize this first pattern:

1) Similar Naming Conventions shared within Similar Product Lines (presence and sequence of words)

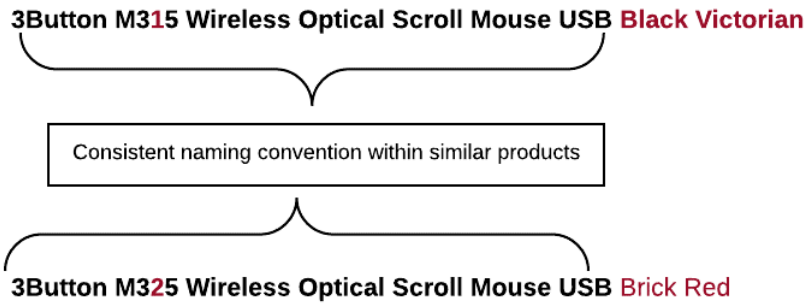

Figure 1:

For example, between these two product descriptions, a majority of the description can be matched, with differences occuring in the Product Line name (M315 vs M325) and SKU-specific words (Black Victorian vs. Brick Red)

This assumption is very straightforward. We will apply an algorithm that is robust to this assumption:

Optimal String Alignment

Also known as Damerau Levenshtein Algorithm (wikipedia), Optimal String Alignment is an algorithm that calculates the "edit distance" between two strings. An "edit" is identified by one of these four:

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**

– [Levenshtein, 1966][leven]

We will use the package `stringdist` in R to compute the edit distance of the example from above.

```
library(stringdist)

a = "M315 Wireless Optical Scroll Mouse Black Victorian"
b = "M325 Wireless Optical Scroll Mouse Brick Red"

stringdist(a, b, method = "osa")
```

```
## [1] 12
```

The edit distance is the number of edits required to transform one description to the other, which is a total of 12 edits.

Edit distance can be converted into a score by dividing by maximum number of possible edits (length of longer string).

```r
max(nchar(a), nchar(b))
```

```
## [1] 50
```

```r
stringdist(a, b, method = "osa")/max(nchar(a), nchar(b))
```

```
## [1] 0.24
```

With a maximum number of edits of 50, the score is 12/50, or 0.24, out of a maximum of 1.

Pros

It captures the difference in model name and in color variations

Why it wouldn't work in our scenario

You may have noticed that the difference in model name is only one "edit", whereas the difference in SKU variation accounts for 11 single edits. Out of the total number of edits possible (50 single-character edits), we have a score of .24 out of a total score of 1, but only .02 of this is accounted by different product line, and the rest of the .22 are captured by the difference in SKU-specifc words. If our goal is to identify matching product lines and not SKU, this method does not quite satisfy our goal.

Here is an example scenario where this poses an issue

```r
library(stringdist)

a = "M315 Wireless Optical Scroll Mouse Black Victorian"
b = "M325 Wireless Optical Scroll Mouse Brick Red"
c = "M325 Wireless Optical Scroll Mouse Black Victorian"

(a.to.c = stringdist(a, c, method = "osa")/max(nchar(a), nchar(c)))
```

```
## [1] 0.02
```

We can see that even though we want to map b to c, we end up mapping a to c with this method. The reason being that, even though b and c should belong together as the same product line, the a string is closer to c. We need a method that is robust to differences in product line, not in SKU.

Weighted Optmial String Alignment

1) Naming Convention same (presence and sequence of words describing product)
2) Words specific to the SKU towards end of description
3) Words specific to the Product Line towards beginning of description

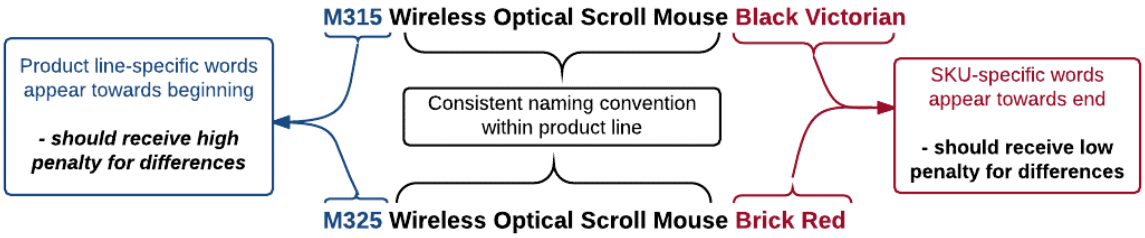

Figure 2:

Using this assumption about the data, we alter the algorithm by adding a location-based weighting.

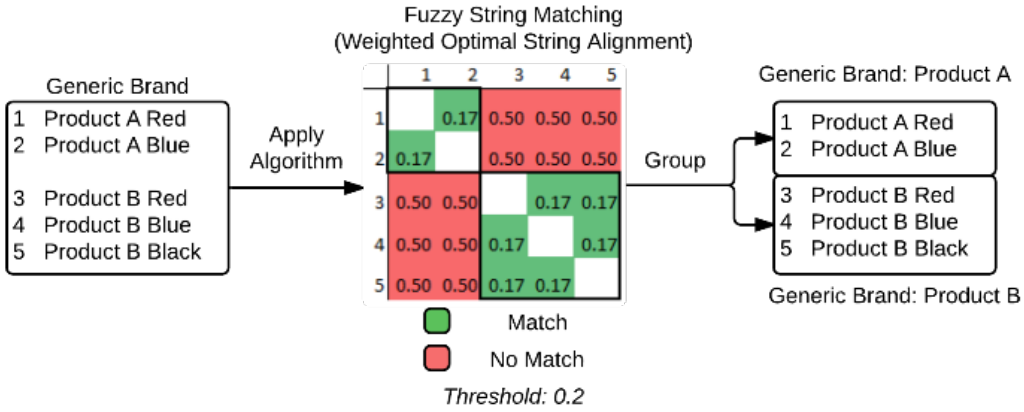

Figure 3:

Parameters

Parameter	Description	Default
Type	Option to apply algorithm by whole item or by single character. Since we want to apply algorithm to WHOLE WORDS rather than single characters, by item is ideal.	character
Weight	Weightfunction (linear, quadratic, root). In the previous example, linear was used	linear
Sum.Right	Flagfor whether or not to consider everything after the first mismatch (from the left-hand side) as mismatches (in other words, sum scores of all mismatches to the right of first mismatch)	F

Example: Amazon Products

https://www.amazon.com/dp/B00N32I2Q6/ref=twister_B01AS57B0I?_encoding=UTF8&psc=1

```
wosa <- function(a, b, type = "character", weight = "linear", sum.right = F){
  if(type == "character"){
    A <- strsplit(a, '')[[1]]
    B <- strsplit(b, '')[[1]]
    d <- array(0, c(nchar(a)+1, nchar(b)+1))
  } else if(type == "item"){
    A <- strsplit(a, ' ')[[1]]
    B <- strsplit(b, ' ')[[1]]
    d <- array(0, c(length(A)+1, length(B)+1))
  }
  for(i in seq_len(length(A))+1){
    for(j in seq_len(length(B))+1){
      if(A[i-1] == B[j-1]) cost <- 0 else cost <- 1
      d[i,j] <- min(c(
        d[i-1,j] + 1,         #deletion
                d[i, j-1] + 1,        #insertion
                d[i-1, j-1] + cost #substitution
            ))
        if(i > 2 & j > 2) if(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
    }
  }
```

```
  } else wosa$score <- 0
  wosa$weights <- w/sum(w)
  wosa$edit.loc <- edit.at
  return(wosa)
}

wosa(a, b)$score
```

```
## [1] 0.09411765
```

```
wosa(a, c)$score
```

```
## [1] 0.03764706
```

Using a Weighted Optimal String Alignment (WOSA), we are able to differentiate between a and c, while capturing an even closer distance between a and b.

But more parameters

```
wosa(a, b,  type = "item", weight = "linear", sum.right = T)$score
```

```
## [1] 1
```

```
wosa(a, c,  type = "item", weight = "linear", sum.right = T)$score
```

```
## [1] 1
```

Now let us look at an example from amazon

```
wosa("Bose SoundLink Color Bluetooth Speaker (Blue)", "Bose SoundLink Color Bluetooth Speaker (Black)",
```

```
## $score
## [1] 0.04761905
##
## $weights
## [1] 0.28571429 0.23809524 0.19047619 0.14285714 0.09523810 0.04761905
##
## $edit.loc
## [1] 6
```

Assessment of all parameters?

Step 2: Unsupervised Clustering

Single Average Complete Ward

Multidimensional Scaling

Spectral Clustering

```r
#load packages
pacman::p_load(dplyr, tidyr, foreach, qualV, stringdist, dbscan, kernlab)

#source some functions
source("C:/Users/rjsai/Dropbox/UMN Courses/Plan B/measure_functions.R")
source("C:/Users/rjsai/Dropbox/UMN Courses/Plan B/str_cleaner.R")

lappend <- function(lst, obj) {
  lst[[length(lst)+1]] <- obj
  return(lst)
}

#load data
sku <- read.csv("C:/Users/rjsai/Dropbox/UMN Courses/Plan B/data/sku_list.csv", stringsAsFactors = F)

#build list
sku.list <- sku %>%
  group_by(Brand, Category) %>%
  do(grp=data.frame(.)) %>%
  lapply(function(x) {(x)})


# step 1) create distnace matrices using the following methods:
# optimal string alignment (item vs character?)
# weighted optimal string alignment (parameters)
# Params:
# s - strings
# method - one of three "qgram", "osa", "wosa"
dist.mat <- function(s, type = "item", weight = "constant", sum.right = T){
  require(stringdist)
  dist <- array(0, c(length(s), length(s)))
  for(j in 1:length(s)) {
    for(k in j:length(s)) {  #computing only half the matrix to remove redundant calculations
      dist[j,k] <- wosa(s[j], s[k], type = type, weight = weight, sum.right = sum.right)$score
    }
  }
  #fill lower triangle
  dist[lower.tri(dist)] <- t(dist)[lower.tri(dist)]   #fill lower triangle with upper triangle
  return(dist)
}

## Test for one each
type = "item" #lets assume by item i.e. no consideration of typos, assume well-structured names
weight = "linear" #just consider constant vs linear
sum.right = T
dist.list = list()

#dist matrix for all groups
for(i in 1:length(sku.list$grp)){
  dist.list[[length(dist.list)+1]] <- dist.mat(sku.list$grp[[i]]$SKU,
    type =  type, weight = weight, sum.right = sum.right)
}
```

```r
#2 clustering

#multidimensional scaling
#mdsfit <- cmdscale(dist.list[[2]], eig = TRUE, k = 2)
#x <- mdsfit$points[, 1]
#y <- mdsfit$points[, 2]
#plot(x, y)

#spectral
# sp.grps = list()
# for(i in 1:length(dist.list)){
#   dist.temp = dist.list[[i]]
#   #cap max number of cluster to half of length
#   k = max(2, ceiling(nrow(dist.temp)/10)):floor(nrow(dist.temp)*.5)
#   sigma = sapply(k, function(x) specc(dist.temp, centers=x, kpar = "scale", iterations = 500)@kernelf
#   k.best = k[which.min(sigma)]
#   #append groups
#   db.grps[[i]] = specc(dist.temp, centers=k.best)@.Data
# }

## DBSCAN
db.grps = list()
for(i in 1:length(dist.list)){
  dist.temp = dist.list[[i]]
  db <- dbscan(dist.temp, eps = .4, minPts = 2)
  db$cluster
  #append groups
  db.grps[[i]] = db$cluster
}

## Hierarchical
methods = c("single", "complete", "average", "ward.D", "ward.D2")
cutoff = .2


hs.grps = hc.grps = ha.grps = hw.grps = hw2.grps = list()
for(i in 1:length(dist.list)){
  dist.temp = as.dist(dist.list[[i]])
  hs.grps[[i]] = hclust(dist.temp, method = "single") %>% cutree(h = cutoff)
  hc.grps[[i]] = hclust(dist.temp, method = "complete") %>% cutree(h = cutoff)
  #ha.grps = c(ha.grps, hclust(dist.temp, method = "average") %>% cutree(h = cutoff))
  #hw.grps = c(hw.grps, hclust(dist.temp, method = "ward.D") %>% cutree(h = cutoff))
  #hw2.grps = c(hw2.grps, hclust(dist.temp, method = "ward.D2") %>% cutree(h = cutoff))
}


#evaluation
## Precision, Recall, Fmeasure
clustEval = function(clust, group){
  class = unique(group)

  confusionMatrix = table(group, clust)
  groupMarginal = rowSums(table(group, clust))
```

```
  #Purity http://stackoverflow.com/questions/9253843/r-clustering-purity-metric
  purity = sum(apply(confusionMatrix, 2, max))/length(clust)
  #High purity is easy to achieve when the number of clusters is large - in particular, purity is 1 if

  ## Precision
  allPos = sum(sapply(groupMarginal, function(x) choose(x, 2)))
  #True positives are only the pairs that are of same type
  truePos = sum(sapply(confusionMatrix, function(x) choose(x, 2)))
  precision = truePos/allPos

  ## Recall
  #false negatives can be found by looking at pairs that should be grouped together, but are not
  falseNeg = sum(apply(confusionMatrix, 1, function(row)
    sum(sapply(1:(length(row)-1), function(i) row[i]*sum(row[(i+1):length(row)])))
  ))
  recall = truePos/(truePos + falseNeg)

  ##F measure
  Fm = 2*precision*recall/(precision+recall)

  return(c(purity = purity, precision = precision, recall = recall, Fmeasure = Fm))
}

db.eval = hs.eval = hc.eval = NULL
for(i in 1:length(dist.list)){
  temp.class = sku.list$grp[[i]]$Product.Line.Class
  db.eval = rbind(db.eval, c("group" = i, "size" = length(temp.class), clustEval(db.grps[[i]], temp.clas
  hs.eval = rbind(hs.eval, c("group" = i, "size" = length(temp.class), clustEval(hs.grps[[i]], temp.clas
  hc.eval  = rbind(hc.eval, c("group" = i, "size" = length(temp.class), clustEval(hc.grps[[i]], temp.cla
}

db.eval
```

```
##       group size    purity precision    recall  Fmeasure
##  [1,]     1   22 0.9090909 1.0000000 1.0000000 1.0000000
##  [2,]     2   15 0.9333333 0.8500000 0.8500000 0.8500000
##  [3,]     3   45 0.9111111 0.7962963 0.7962963 0.7962963
##  [4,]     4   43 1.0000000 0.9633028 0.9633028 0.9633028
##  [5,]     5   14 1.0000000 1.0000000 1.0000000 1.0000000
##  [6,]     6   27 1.0000000 1.0000000 1.0000000 1.0000000
##  [7,]     7   21 1.0000000 1.0000000 1.0000000 1.0000000
##  [8,]     8    8 1.0000000 1.0000000 1.0000000 1.0000000
##  [9,]     9   34 1.0000000 1.0000000 1.0000000 1.0000000
## [10,]    10    6 1.0000000 1.0000000 1.0000000 1.0000000
## [11,]    11   14 1.0000000 0.8888889 0.8888889 0.8888889
## [12,]    12   33 0.3636364 1.0000000 1.0000000 1.0000000
```

```
hs.eval
```

```
##       group size    purity   precision      recall    Fmeasure
##  [1,]     1   22 1.0000000 0.818181818 0.818181818 0.818181818
##  [2,]     2   15 1.0000000 0.850000000 0.850000000 0.850000000
##  [3,]     3   45 1.0000000 0.009259259 0.009259259 0.009259259
##  [4,]     4   43 1.0000000 0.954128440 0.954128440 0.954128440
```

```
##  [5,]     5   14 1.0000000 0.615384615 0.615384615 0.615384615
##  [6,]     6   27 1.0000000 1.000000000 1.000000000 1.000000000
##  [7,]     7   21 1.0000000 1.000000000 1.000000000 1.000000000
##  [8,]     8    8 1.0000000 1.000000000 1.000000000 1.000000000
##  [9,]     9   34 1.0000000 1.000000000 1.000000000 1.000000000
## [10,]    10    6 1.0000000 1.000000000 1.000000000 1.000000000
## [11,]    11   14 1.0000000 0.888888889 0.888888889 0.888888889
## [12,]    12   33 0.3636364 1.000000000 1.000000000 1.000000000
```

Results

Evaluation

Fscore Recall Precision

Discussion

good if assumptions are met (need to be careful if assumptions are met)

Conclusion

location weighting can be applied to other algorithms

Appendix

References

[leven]: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.

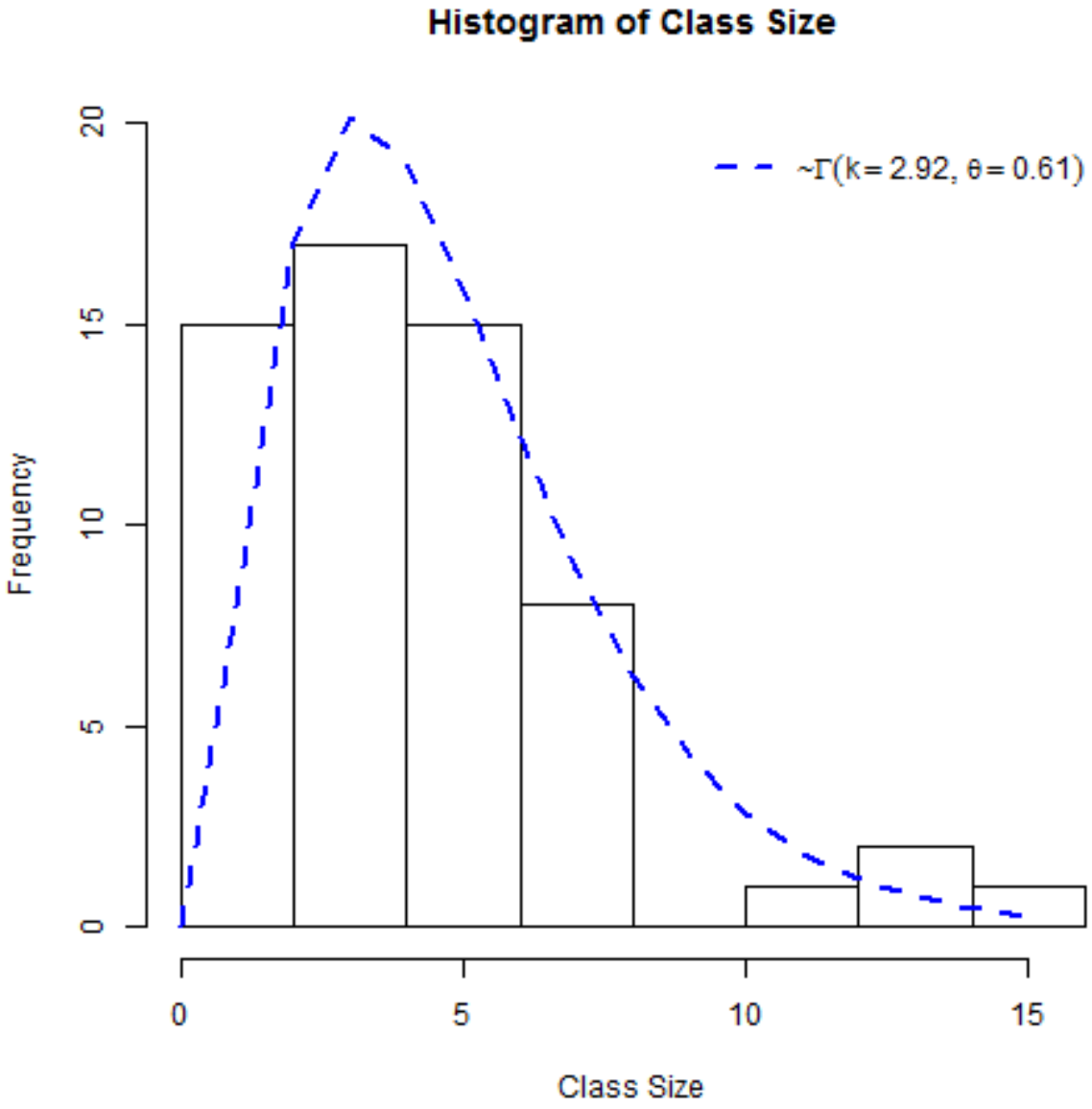
Histogram of Class Size
~Γ(k = 2.92, θ = 0.61)
Frequency
0
5
10
15
20
0
5
10
15
Class Size

Figure 4:

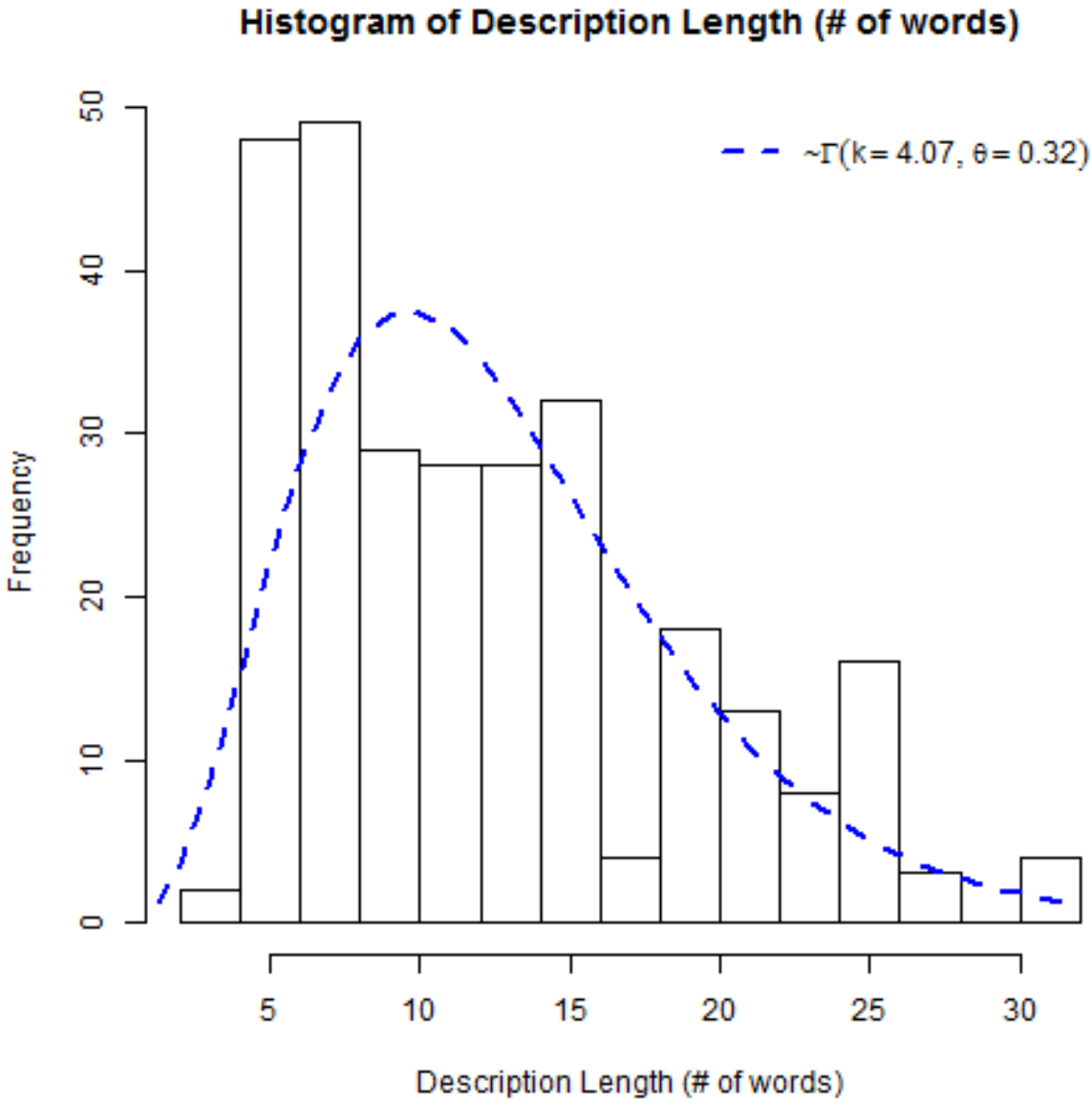
Histogram of Description Length (# of words)
~Γ(k = 4.07, θ = 0.32)
Frequency
0
10
20
30
40
50
5
10
15
20
25
30
Description Length (# of words)

Figure 5:

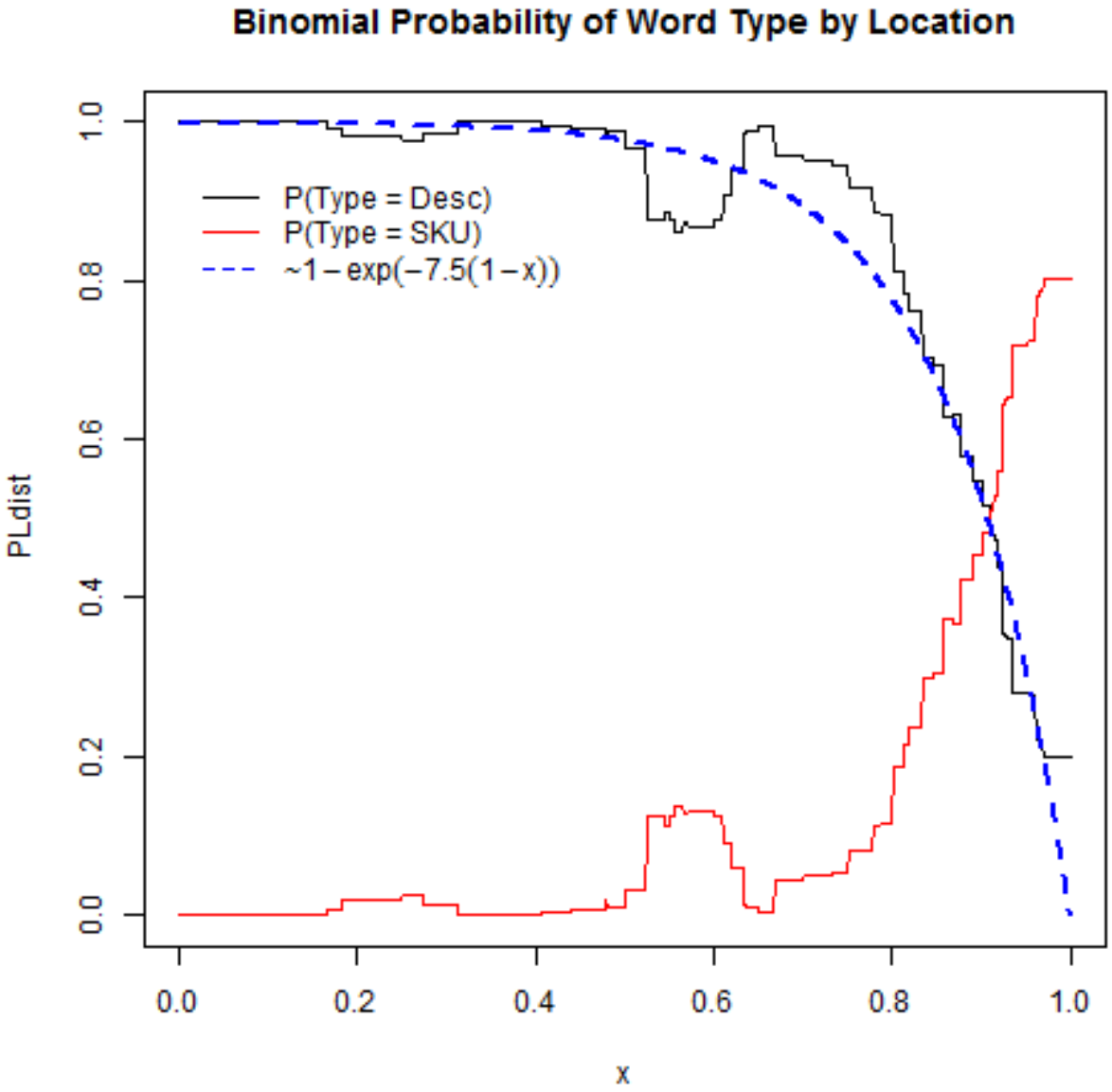
Binomial Probability of Word Type by Location
P(Type = Desc)
P(Type = SKU)
~1 – exp(–7.5(1 – x))
PLdist
0.0
0.2
0.4
0.6
0.8
1.0
x

Figure 6: