Implementation of string comparison algorithms in R

Authors: Prathwish Shekhar Shetty, Nikhar Gaurav

Summary:

Entity resolution is the process of identifying and matching records that denote the same entity across multiple data sources. String comparison is a vital part of an entity resolution problem. The implementation of the same is known by various other names based on the kind of operation being conducted like record linkage, data matching, duplicate detection and object identification. We have implemented hybrid and newer string matching algorithms in R and developed it into an R package "stringCompare"

For the scope of this project, we have compared strings and tested them on data from the University of Leipzig. The data set from University of Leipzig[1] has fields related to products sold on e-Commerce site Amazon and the listing for the same on Google. This data has been used to check the performance of R function implemented by us with the already existing function.

Following entity resolution algorithms to compare entities have been implemented as functions in R package "stringCompare" which generates a similarity score:

- **Monge - Elkan Comparison[2,3]:** This method separates a string into tokens based on white spaces. And then each token from one string is compared against every other token in another string using a secondary comparison method. Next, the average of the best match per token is taken to compute the Monge-Elkan Similarity score.

- **Extended Jaccard Comparison [4]:** This is an extension of the Jaccard algorithm, best suited for strings with multiple words. It uses a secondary string comparison method to calculate the similarity between all the words in the two strings to be compared. Next, a set of tokens shared between the two strings are calculated provided the similarity is above a certain threshold. Finally, to get the similarity score a ratio of shared tokens by the sum of shared tokens and unique tokens of the two strings.

- **SoftTFIDF String Comparison [4]:** The term TF represents Term Frequency and IDF represents Inverse Document Frequency.TF refers to the most frequently occurring term in a document, assuming that the same is the most relevant. IDF, on the other hand, assumes less frequently occurring words in a collection belonging to the document that is less relevant. SoftTFIDF comparison function is defined similarly to the Monge-Elkan comparison function. Assuming A and B are the two sets of tokens in the strings s1 and s2, and a secondary similarity function 'sim' is used to calculate the similarities between individual pairs of tokens (or words).

- **Needleman Wunsch Comparison [5]**: This method follows an iterative path by setting up all possible pairs of strings in a 2D matrix and representing alignments as a pathway through the array. Out of this optimum alignment is taken by connecting multiple scoring values. This method helps in finding the best fit between two strings.

Methods:

1. Monge Elkan Algorithm:

This approach of comparison involves tokenizing (i.e. making tokens based on elements that are separated by whitespaces) the two strings, and then finding the best matching pair of tokens between the two strings using a secondary comparison technique. Then the similarity between the two strings is given by the equation:

$$\text{Sim}_{\text{Monge-Elkan}}(s1, s2) = \frac{1}{|A|}\sum_{i=1}^{|A|} max_{j=1}^{|B|} sim'(Ai, Bj)$$

Where s1 and s2 denote the two strings that are being compared. A and B are the sets of tokens extracted from strings s1 and s1. |A| and |B| indicates the number of tokens in the strings respectively and *sim'* is a function that will calculate the actual similarity score (which will be greater than 0 and less than 1) between the two tokens, this can be any string comparison technique like Jaro-Winkler, Levenshtein etc. as long as the similarity between strings is given by a score in the required limits,

An example of the same can be given using two strings, s1 being "kaspersky antivirus" and s2 being "kasper antivirus".

The following are the steps in the algorithm:

1. Make a set of token A and B from strings s1 and s2
    1. A= {*'kaspersky', 'antivirus'*}
    2. B = {*'kasper', 'antivirus'*}
2. Determine the value of |A| and |B| i.e. counting the number of tokens in sets A and B
    1. |A| = 2
    2. |B| = 2
3. Compare every token from token set A with every other token from token set B using a secondary string comparison technique, here we will use Jaro-Winkler comparison
    1. $_{\text{Jaro-Winkler}}$(*'kaspersky', 'kasper'*) = 0.8889
    2. $_{\text{Jaro-Winkler}}$(*'kaspersky', 'antivirus'*) = 0.5556
    3. $_{\text{Jaro-Winkler}}$(*'antivirus', 'kasper'*) = 0.5185
    4. $_{\text{Jaro-Winkler}}$(*'antivirus', 'antivirus'*) = 1.0000
4. Determine the best matching pair of words for the tokens of set A. Here it is:
    1. *'kaspersky'* and *'kasper' with Jaro-Winkler score of 0.8889*
    2. *'antivirus'* and *'antivirus'* with Jaro-Winkler score of 1.000
5. Use the equation above to determine the similarity score. Hence

    $\text{sim}_{\text{monge-elkan}}$(*'kaspersky antivirus','kasper antivirus'*) = $\frac{1}{2}(0.8889 + 1.0000)$= <u>0.9445</u>

2. Extended Jaccard:

Extended Jaccard algorithm is an extension of the Jaccard coefficient which calculates the similarity between two strings, but firstly splitting the two strings into a set of tokens, and then getting a similarity score but considering the ratio of the intersection of the two token sets by the union of the two token sets. And the Jaccard coefficient is given by the formula

$$\text{sim}_{\text{jacccard}}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In the above equation, A and B is the set of tokens taken from strings s1 and s2. And $|A \cap B|$ is the number of tokens that are similar in the set A and B, while $|A \cup B|$ is the total number of tokens from both the strings.

In cases when the strings are made up of several words then we can take the tokens to be words and then we can extend the basic Jaccard coefficient by introducing a secondary similarity function. Where a secondary similarity function can be used to compare the words from both the sets of tokens. Once we use a secondary function we can extend the Jaccard coefficient to match longer strings, then the new similarity equation becomes:

$$\text{sim}_{\text{Extended-jacccard}}(s1, s2) = \frac{|S|}{|S|+|U_A|+|U_B|}$$

Where the terms in the above equation is given by:

$$S = \{(a_i, b_j) | a_i \in A \wedge b_j \in B : sim'(a_i, b_j) \geq \theta\}, (0 < \theta < 1)$$
$$U_A = \{a_i | a_i \in A \wedge b_j \in B \wedge (a_i, b_j) \notin S\}$$
$$U_B = \{b_j | a_i \in A \wedge b_j \in B \wedge (a_i, b_j) \notin S\}$$

Here s1 and s2 are the two strings being compared. And A and B are the set of tokens extracted from strings s1 and s2. We generate another set of tokens called 'S' which is a shared token set i.e. includes all the tokens from A and B that are similar to each other by a degree greater than or equal to the threshold giving by the user. The similarity between the two tokens can be calculated by using a similarity function that returns a user with a score in the range of 0 to 1.

An example of the same can be given using two strings, s1 being "kaspersky is awesome" and s2 being "kasper are antivirus".

The following are the steps in the algorithm:

1. Make a set of token A and B from strings s1 and s2
    1. A= {*'kaspersky', 'is', 'awesome'*}
    2. B = {*'kasper', 'are', 'antivirus'*}
2. Compare every token from token set A with every other token from token set B using a secondary string comparison technique, here we will use Jaro-Winkler comparison

    Jaro-Winkler(*'kaspersky', 'kasper'*) = 0.8889
    Jaro-Winkler(*'kaspersky', 'are'*) = 0.6296
    Jaro-Winkler(*'kaspersky', 'antivirus'*) = 0.5556
    Jaro-Winkler(*'is', 'kasper'*) = 0.5556
    Jaro-Winkler(*'is', 'antivirus')* = 0.5370
    Jaro-Winkler(*'awesome', 'kasper'*) = 0.5317
    Jaro-Winkler(*'awesome', 'are'*) = 0.6507
    Jaro-Winkler(*'awesome', 'antivirus'*) = 0.4179
    Jaro-Winkler(*'is', 'are'*) = 0.0000
3. Generate the sets of tokens S, U_A and U_B

    S= { *'kaspersky'*, '*kasper'* }, U_A= {'*is*', '*awesome'*}, U_B= {'*are*', '*antivirus*'}
4. Generate the values of $|S|$, $|U_A|$ and $|U_B|$ after looking at the sets

    $|S|$= 2, $|U_A|$ = 2, $|U_B|$ = 2
5. Use the values in the equation

    $\text{sim}_{\text{extendedJaccard}}$*("kaspersky is awesome", "kasper are antivirus")* = $\frac{2}{2+2+2}$ = <u>0.3333</u>

3. Soft TF/IDF:

TFIDF consists of TF (Term Frequency) and IDF (Inverse Document Frequency). TF is measure of how frequent a term is in document aka. the number of times a word appears in a document, divided by the total number of words in that document. Mathematically it can be written as:

$$TF(t) = \frac{Number\ of\ times\ term\ t\ appears\ in\ a\ document}{Total\ number\ of\ terms\ in\ the\ document}$$

IDF which measures how important a term is. As TF considers all the terms as equally important hence the common words like "the", "is", "of", "that", etc get the highest weight and undermines the other important words. Thus to scale up the rare words which are more important we calculate IDF aka calculating the log of the ratio of total no. of strings provided divided by a number of strings with that term in it. Mathematically it can be written as:

$$IDF(t) = log\_e \frac{Total\ number\ of\ strings}{Number\ of\ strings\ with\ term\ t\ in\ it}$$

Soft TF/IDF is the softer implementation where it calculates a weighted score using a secondary similarity like "Jaro-Winkler" for two strings as compared to TF/IDF which calculates for multiple strings. As part of Soft TF/IDF, initially, a weighted TF/IDF value is calculated. Mathematically this can be defined as:

$$TFIDF(t) = \log_e(1 + No.of\ times\ term\ t\ appears\ in\ the\ string) * \log_e \frac{Total\ no.of\ strings}{No.of\ strings\ with\ term\ t\ in\ it}$$

Each individual term in two strings string1 and string2 being compared are taken as a pair by matching each term t from string1 with each term s from string2 and calculates a score which is a product of secondary similarity score by "Jaro-Winkler" (degree) for both terms t and s from string string1 and string2 with weighted TF/IDF score for both s and t. Mathematically it can be defined as:

$$Score(s,t) = degree * TFiDF(t) * TFIDF(s)$$

All the scores obtained are added together to get a weighted score which is the final SoftTFIDF score for two strings being compared. For example,

1. Taking two strings:
   string1= "Boston is awesome" string2= "Boston is best"

2. Calculating a weighted TFIDF score for all the terms:

string1,		**string2,**	
TFIDF(*'Boston'*)=	0.0000	TFIDF('*Boston'*)=	0.0000
TFIDF(*'is*')=	0.0000	TFIDF('*is'*)=	0.0000
TFIDF('*awesome'*)=	0.7071	TFIDF('*best'*)=	0.7071

3. Calculating the comparison score by comparing terms from string1 and string2 by using secondary similarity score from "Jaro-Winkler".

Score("Boston", "Boston")=0	Score("Boston", "is")= 0	Score("Boston", "awesome")= 0
Score("is", "Boston")=0	Score("is", "is")= 0	Score("is", "awesome")= 0
Score("best", "Boston")=0	Score("best", "is")= 0	Score("best", "awesome")= 0.2976

So, the final weighted score is 0.2976

4. Needleman Wunsch:

This algorithm finds the similarity score between two strings namely string1 and string2 by finding the best alignment between the pair of terms t and s taken from string1 and string2 respectively i.e. every term of string1 is paired with every term of string2 and score is calculated by finding best alignment between them.

The process begins by generating a similarity matrix (sim) from terms s and t taken from string1 and string2 respectively as pairs where +1 is assigned for every matching alphabet and -1 for every mismatch.

For example:

1. Let term t taken from string1 be "GCATGCU" and term s was taken from string2 be "GACTTBD". So the similarity matrix generated would be as below.

	G	**C**	**A**	**T**	**G**	**C**	**U**
G	**1**	**-1**	**-1**	**-1**	**1**	**-1**	**-1**
A	**-1**	**-1**	**1**	**-1**	**-1**	**-1**	**-1**
T	**-1**	**-1**	**-1**	**1**	**-1**	**-1**	**-1**
T	**-1**	**-1**	**-1**	**1**	**-1**	**-1**	**-1**
A	**-1**	**-1**	**1**	**-1**	**-1**	**-1**	**-1**
C	**-1**	**1**	**-1**	**-1**	**-1**	**1**	**-1**
A	**-1**	**-1**	**1**	**-1**	**-1**	**-1**	**-1**

2. A scoring matrix (score) is created where length of column is length(term s) + 1 and length of row is length(term t) +1. The first index of column and row is considered gap and assigned a value of -1. The below given algorithm is then used to generate values of scoring matrix (score) where similarity matrix (sim) is used to populate data by adding to previous value if there is a match or d=-1 is added for mismatch.

```
d=-1
for i=0 to length(s)
Score(i,0) ← d*i
for j=0 to length(t)
Score(0,j) ← d*j
for i=1 to length(s)
for j=1 to length(t)
{ Match ← Score(i-1,j-1) + Sim(s[i], t[j])
  Delete ← Score(i-1, j) + d
  Insert ← Score(i, j-1) + d
  Score(i,j) ← max(Match, Insert, Delete)
}
```

3. This will generate the scoring matrix (score) as given below:

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]
[1,]	0	-1	-2	-3	-4	-5	-6	-7
[2,]	-1	1	0	-1	-2	-3	-4	-5
[3,]	-2	0	0	1	0	-1	-2	-3
[4,]	-3	-1	-1	0	2	1	0	-1
[5,]	-4	-2	-2	-1	1	1	0	-1
[6,]	-5	-3	-3	-1	0	0	0	-1
[7,]	-6	-4	-2	-2	-1	-1	1	0
[8,]	-7	-5	-3	-1	-2	-2	0	0

The score obtained here for term t and s is 0 which is the final value obtained in the end. In the end, a weighted score is obtained is obtained by adding all the scores obtained for a pair of all terms from string1 with string2 using Needleman Wunsch.

Data Preparation:

The functions were developed in R and put to test against a dataset that was taken the University of Leipzig. This was an entity resolution benchmark dataset that has been used in various benchmarking evaluations. We decided to use a dataset that contained the names of various products from e-commerce website Amazon and Google. The dataset obtained from the website had three CSV files, one containing the product names from Amazon and another having product names from Google and the last CSV having a matching of all similar products.

Steps used in tidying the data for the analysis:

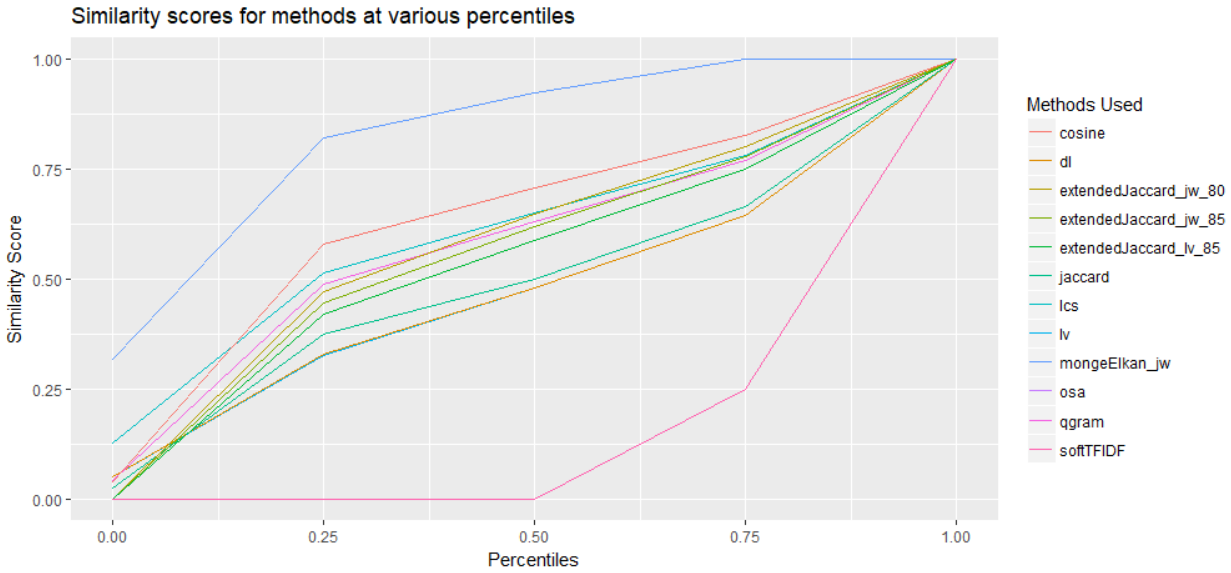
1. Import the three CSV files into R environment
2. Change the encoding of the product names to 'latin-ascii'
3. Create a dataset of matching entities after merging the three datasets
4. Create a dataset of unmatching entities by creating a cross join of all the values from Amazon and Google that do not match each other
5. Create a sample of 100,000 values to be used in the analysis

The method used to compare the performance of the package:

1. Run each of the functions developed for the packages and the functions available in the 'stringdist' against the names of the products from the two sources data is available in the 'Matched dataset' and 'Unmatched Dataset'
2. Calculate the percentile scores for each method at various intervals to look at the spread for the 'Matched dataset'
3. By looking at the plot take an acceptable error percentage, in our case, we accept an error of 15%, hence by splitting the matched data as 85% accurate and 15% error we get our rate of true positives and False Negative
4. Now considering the similarity score at 15% we can determine the number of instances in the 'Unmatched dataset' that fall over the set threshold, hence this becomes the False Positive
5. Similarly, we can check for number of instances that fall below the threshold, hence giving us our True Negatives
6. Now using the above-computed numbers, we can calculate the accuracy and precision of the algorithms used

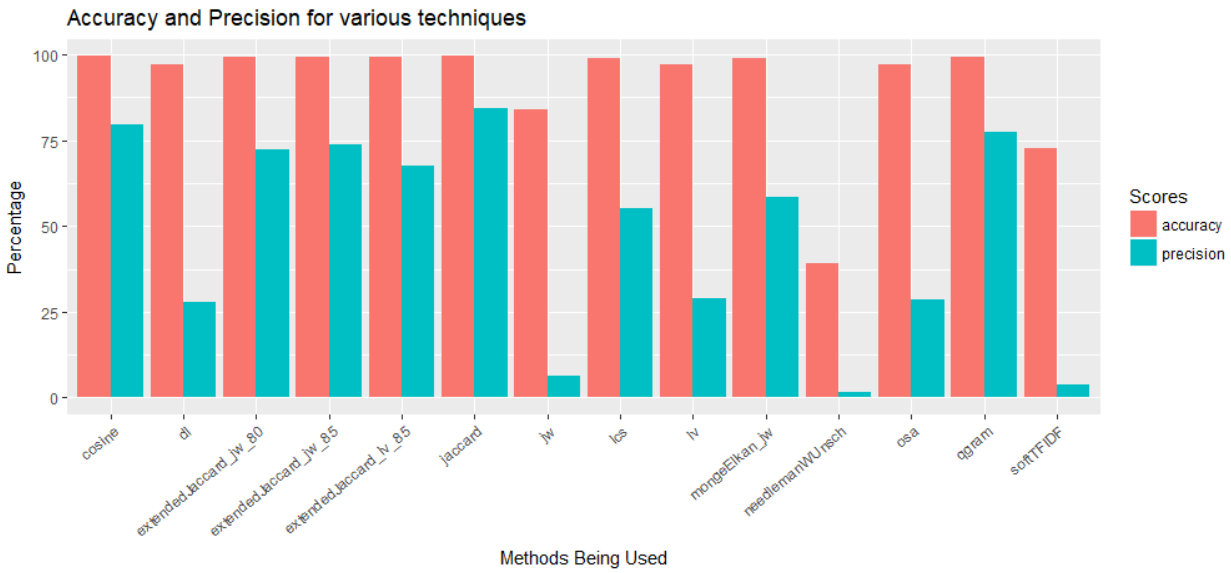
Results:

1. Plotting the similarity scores for each of the algorithms at various percentiles



Ideally, we would want to see all the values at 1 as we are running the comparison on a matched dataset. But since the names do not exactly match we can see values ranging from 0 to 1. We can see from the above plot that Monge-Elkan algorithm using Jaro-Winkler is exhibiting a very high similarity as expected. But we see that the other methods used by us like Extended Jaccard shows lower similarity scores for the matched datasets, which is not the ideal behaviour. But we cannot determine the performance of the algorithms by just the similarity score as each algorithm has different methods to penalize inaccuracy leading to lower scores.Also, we notice that softTFIDF has a score of zero till 50th percentile and then quickly shoots up.

2. Calculating the Accuracy and Precision of each algorithm



methods	accuracy	precision
osa	97.06417	28.450051
lv	97.11155	28.806048
dl	97.00987	28.052805
lcs	98.92201	55.194805
qgram	99.48766	77.326802
cosine	99.52912	79.668349
jaccard	99.60513	84.246575
jw	83.95854	6.439394
mongeElkan_jw	99.03554	58.558559
extendedJaccard_jw_80	99.39980	72.293814
extendedJaccard_jw_85	99.42547	73.806366
extendedJaccard_lv_85	99.29516	67.502987
softTFIDF	72.77887	3.879235
needlemanWUnsch	39.20533	1.768141

From the above plot and table, we can clearly see that the algorithms Monge-Elkan and Extended Jaccard in the package have consistently higher accuracy and precision than most of the algorithms available in the 'stringdist' package. Also, softTFIDF has a high accuracy but is very low in precision. From the scores of accuracy and precision for Needleman Wunsch, we can say that it is not best suited for this type of data.

Discussion:

Lesson Learned:

This project, it helped us in getting the better understanding of package building process and have a first-hand experience on technical issues that can arise. Also, we have understood various algorithms that can be used in entity resolution, which could come in handy while pursuing topics in the realm of Natural Language Processing etc. We realized the limitations of R in handling string related operations.

Implementation wise it helped us in understanding the concepts of entity resolution through various string comparison algorithms and their importance in vast data science field. The biggest lesson learned here is that the string operations in R are not as efficient as compared to other languages.

Next Steps:

As an enhancement to this project, we would like to make these functions that are more computationally efficient and are operational when running against large datasets, to achieve this we would like to convert all the functions that are currently running in R to C++ and then call it to R via

RCpp. And upon successful completion of that, we would like to make use of OpenMP to run the compiled files ever faster. Also, we will work towards publishing this package on CRAN.

Statement of Contribution:

Prathwish Shetty:

Performed complete implementation of algorithms “Monge-Elkan” and “Extended Jaccard”. Created the package “stringCompare” and tested it thoroughly. Performed data comparison for implemented algorithms with other preexisting algorithms. Created R markdown file.

Nikhar Gaurav:

Performed complete implementation of algorithms “softTFIDF” and “needlemanWunsch”. Created the final report for the project.

References:

1. https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution
2. Monge, A.E., Elkan, C.P. (1996) The field matching problem: Algorithms and applications
3. Christen, P. (2012). Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution and Duplication Detection
4. http://sepwww.stanford.edu/public/docs/sep112/bob2/paper_html/node3.html
5. https://en.wikipedia.org/wiki/Needleman–Wunsch_algorithm

Appendix:

Monge-Elkan Algorithm Code:

```
mongeElkan <- function(string1, string2,method) {
  string1_count <- stringr::str_count(string1, "\\S+")
  string2_count <- stringr::str_count(string2, "\\S+")

  list1 <- matrix(nrow = string1_count, ncol = string2_count + 1)

  for (i in (1:string1_count)) {
    for (j in (1:string2_count)) {
      list1[i, j] <-
        (stringdist::stringsim(unlist(strsplit(
          string1, "\\s+"
        ))[i], unlist(strsplit(
          string2, "\\s+"
        ))[j], method = method))
    }
    list1[i, j + 1] <- max(list1[i, ], na.rm = T)
  }
  mean(list1[, string2_count + 1])
}
```

Extended Jaccard Algorithm Code:

```
extendedJaccard<-function(string1,string2,method,threshold){

  string1_count <- stringr::str_count(string1, "\\S+")
  string2_count <- stringr::str_count(string2, "\\S+")
  string_mat<-matrix(nrow=string1_count+1,ncol=string2_count+1)

  for (i in (1:(string1_count))) {
    for (j in (1:string2_count)) {
      if ((stringdist::stringsim(unlist(strsplit(string1, "\\s+"))[i],
                                 unlist(strsplit(string2, "\\s+"))[j],
                                 method))>=threshold){
        string_mat[i,j]<-1
      }
      else{
        string_mat[i,j]<-0
      }
    }
  }

  for (i in (1:string1_count)){
    string_mat[i,(string2_count+1)]<-if(sum(string_mat[i,]>0,na.rm = T)>0) 1 else 0
  }

  for (j in (1:string2_count)){
    string_mat[(string1_count+1),j]<-if(sum(string_mat[,j]>0,na.rm = T)>0) 1 else 0
  }

  nu_str1<-sum(string_mat[,string2_count+1],na.rm=T)
  nu_str2<-sum(string_mat[string1_count+1,],na.rm=T)
  (nu_str1+nu_str2)/(nu_str1+nu_str2+(string1_count-nu_str1)+(string2_count-nu_str2))
}
```

SoftTFIDF Algorithm Code:

```
softTFIDF<- function(string1, string2) {
  m<-tolower(gsub('[[:punct:] ]+',' ',string1))
  n<-tolower(gsub('[[:punct:] ]+',' ',string2))
  sp <- unlist(str_match_all( m, "\\S+" ) )
  sp2 <- unlist(str_match_all( n, "\\S+" ) )
  str_df <- data.frame( "string"=character(),"word" = character(), "count" = integer(), "total"=integer() , stringsAsFactors=FALSE)
  flag<- 0
  for(j in 1:length(sp))
  {
    for(i in 1:length(sp))
    {
      if(sp[j]==sp[i]){
        flag<- flag+1}
    }
    str_df[nrow(str_df) + 1, ] <- c( "string1",sp[j], flag,length(sp) )
    flag<-0
  }
  for(j in 1:length(sp2))
  {
    for(i in 1:length(sp2))
    {
      if(sp2[j]==sp2[i]){
        flag<- flag+1}
    }
    str_df[nrow(str_df) + 1, ] <- c( "string2",sp2[j], flag,length(sp2) )
    flag<-0
  }
  FinalCount<- unique( str_df)
  IDF  <-FinalCount %>% group_by(word) %>% count() %>% mutate( idf= log(2/n) )
  TF_Idf <-left_join(FinalCount,IDF, by="word")
  scalar1 <- function(x) {x / sqrt(sum(x^2))}
  TF_IDF_Final <- TF_Idf %>% mutate(TF=as.numeric(count)/as.numeric(TF_Idf$total), TF_IDF= scalar1(log(1+as.numeric(count))* log(2/as.numeric(n))))
  sim<- matrix(data = 0, nrow = length( unique(sp2)), ncol =length( unique( sp)), dimnames = list( unique(sp2),unique(sp)))
  usp<-unique(sp)
  usp2<- unique(sp2)
  value<-0
  for(i in 1:length(usp2))
    for(j in 1:length(usp))
    {
      deg<-stringdist::stringsim(usp[j],usp2[i] ,"jw")
      if( deg >= 0.5){
        sim[i,j] <-  deg * TF_IDF_Final[j,8] * TF_IDF_Final[length(usp)+i,8]
      }
      else {
        sim[i,j]<- 0
      }
      value <- value + sim[i,j]
    }
  value<-1-value
  if(is.nan(value)){
    value<- 1
  }
  return(value)
}
```

Needleman Wunsch Algorithm:

```
needlemanwunsch<- function(string1, string2) {
m<-tolower(gsub('[[:punct:] ]+',' ',string1))
n<-tolower(gsub('[[:punct:] ]+',' ',string2))
sp <- unlist(str_match_all( m, "\\S+" ) )
sp2 <- unlist(str_match_all( n, "\\S+" ) )
sim<- matrix(data = 0, nrow = length( unique(sp2)), ncol =length( unique( sp)), dimnames = list( unique(sp2),unique(sp)))
usp<-unique(sp)
usp2<- unique(sp2)
value<-0
NMWScore <- function(A,B) {
strA<- strsplit(A,"")[[1]]
strB<-strsplit(B,"")[[1]]
sim<- matrix(data = 0, nrow = length(strB), ncol =length(strA), dimnames = list(strB,strA))
for(i in 1:length(strB))
for(j in 1:length(strA))
{
if(strB[i]==strA[j])
sim[i,j]<-1
else
sim[i,j]<--1
}
F <- matrix(0, nrow =length(strB)+1  , ncol =length(strA)+1)
d<- -1
for ( i in 0:length(strB))
F[i+1,1]<- d*i
for ( j in 0:length(strA))
F[1,j+1]<- d*j
for(i in 1:length(strB))
for(j in 1:length(strA))
{
match  = F[i,j] + sim[i,j]
delete = F[i,j+1] + d
insert = F[i+1,j] + d
F[i+1,j+1] = max(match, delete, insert)
}
score<- F[i+1,j+1]
return(score)
}
for(i in 1:length(usp2))
for(j in 1:length(usp))
{
sim[i,j] <-  NMWScore(usp[j],usp2[i])
value <- value + sim[i,j]
}
return(value)
}
```