Conceptual Understanding of LSA

Promothesh Chatterjee*

Context and Term Co-Occurrence Matrix

As we have seen in the previous section, the two fundamental issues in NLP are synonymy and polysemy. Both of these issues relate to the context of the sentence. One way to determine the context is by creating a term co-occurrence matrix (TCM). TCM is a square matrix where all the terms occur in rows as well as columns, so that when some terms appear together, they may define some context. Term co-occurrence matrix is actually derived from document-term matrix DTM. Let's first understand the concept of TCM with a simple example.

```
library(quanteda)
txt2 <- c("a a a b b c", "a a c e", "a c e f g")
txt2 %>% tokens() %>% dfm() %>% fcm(context = "document", count = "frequency")
```

```
## Feature co-occurrence matrix of: 6 by 6 features.
##         features
## features a b c e f g
##        a 4 6 6 3 1 1
##        b 0 1 2 0 0 0
##        c 0 0 0 2 1 1
##        e 0 0 0 0 1 1
##        f 0 0 0 0 0 1
##        g 0 0 0 0 0 0
```

As can be seen, 'a' co-occurs with 'a' thrice within the first document, once in the second document and zero times in the third document, so a total of 4 co-occurrences of 'a' with 'a'. Similarly for 6 co-occurrences of 'a' with 'b' and so on.

Let's revisit our previous example and generate the DTM before generating TCM.

```
library(quanteda)
a<-"Before winter, we generally see a spike in snow blower sales,
the warm forecast will sure reduce sales.."
b<-"The temperature forecast for the coming months is high,
this will impact snow removal business"

abc<-c(a,b)
abc_token <- tokens(abc,
                    remove_numbers = TRUE, remove_punct = TRUE,
                    remove_symbols = TRUE)

abc_dfm<-dfm(abc_token)

abc_dfm1<-as.matrix(abc_dfm)
abc_dfm1[1:2,1:12]
```

```
##        features
## docs    before winter we generally see a spike in snow blower sales the
##   text1      1      1  1         1   1 1     1  1    1      1     2   1
##   text2      0      0  0         0   0 0     0  0    1      0     0   2
```

Now let's look at a snapshot of term co-occurrence matrix.

```
tcm<-fcm(abc_dfm, context = "document", count = "frequency")
tcm<-as.matrix(tcm)
tcm[1:12,1:12]
```

```
##            features
## features    before winter we generally see a spike in snow blower sales the
##   before         0      1  1         1   1 1     1  1    1      1     2   1
##   winter         0      0  1         1   1 1     1  1    1      1     2   1
##   we             0      0  0         1   1 1     1  1    1      1     2   1
##   generally      0      0  0         0   1 1     1  1    1      1     2   1
##   see            0      0  0         0   0 1     1  1    1      1     2   1
##   a              0      0  0         0   0 0     1  1    1      1     2   1
##   spike          0      0  0         0   0 0     0  1    1      1     2   1
##   in             0      0  0         0   0 0     0  0    1      1     2   1
##   snow           0      0  0         0   0 0     0  0    0      1     2   3
##   blower         0      0  0         0   0 0     0  0    0      0     2   1
##   sales          0      0  0         0   0 0     0  0    0      0     1   2
##   the            0      0  0         0   0 0     0  0    0      0     0   1
```

If the words, 'blower' and 'removal' show high co-occurrence, they are likely to be synonyms or contextually related words. Let's understand the concept of Latent Semantic Analysis using term co-occurrence matrix (although in computations LSA actually uses term-document matrix not term co-occurrence matrix) and how it help us address issues of synonymy and polysemy.

LSA assumes there are hidden (latent) concepts in the data that map onto the words.

Latent Concepts	Words
Concept 1	word1
concept 2	word2
.	.
.	.
.	.
concept n	wordp

All combinations of mapping are possible (for instance, concept 2 can map onto all the words, whereas concept 1 may map onto wordp only) but word co-occurrences with high frequency are likely related to a particular

concept. Thus, relevant mappings are established based on the term co-occurrence structure in the data. For instance, let's say we have a collection of documents from two fields- Marketing and Finance. Terms such as brand, customer, churn, satisfaction would be mapped onto latent construct -Marketing whereas terms such as stocks, debentures, capital, leverage would be mapped onto Finance in the term co-occurrence matrix. Once the terms associated with a latent construct are identified, they would not be treated as independent entities but as a single dimension. So that is how you get the latent concepts and reduce sparsity. For example,

$Marketing = \alpha(brand) + \beta(customer) + \gamma(churn) + \delta(satisfaction)$

Mechanics of Latent Semantic Analysis

In the actual computation of LSA, Term Document Matrix (which is transpose of Document Term Matrix) is used. LSA attempts to find a lower dimensional approximation of Term Document Matrix by using a matrix factorization algorithm called Singular Value Decomposition (SVD). We look at the mathematical details in the next section, for now, let's understand the big picture.

Singular Value Decomposition is computed by factorizing the Term Document Matrix into product of three matrices:

$$U_{t\times n}, \Sigma_{n\times n}, (V_{d\times n})^T$$

where t is the number of terms/words,

d is the number of documents,

n is the min of (t,d),

U is the left singular vector of words,

V is the right singular vector of documents,

U and V have orthonormal columns, i.e., UU'=V'V = I

Σ is a n x n diagonal matrix (singular value matrix) = diag($\sigma_1, \sigma_2....\sigma_n$)

Let's look at a simplified example. Consider a 6 x 5 term document matrix.

	D1	D2	D3	D4	D5
w1	1	1	0	0	2
w2	0	2	0	1	1
w3	1	0	0	0	0
w4	0	3	0	2	0
w5	2	0	3	1	1
w6	0	0	2	0	3

If we perform a SVD factorization on the above matrix, our output will have three matrices,

U, ,V^T

		U		
-0.25712	-0.0630233	-0.318479	0.6422	0.64431
-0.29882	-0.424245	0.279408	0.288256	-0.27668
-0.0617946	0.0348624	-0.256965	0.532873	-0.698197
-0.332141	-0.770245	-0.033234	-0.289481	0.0425223
-0.658944	0.307838	-0.529603	-0.358901	-0.126779
-0.54429	0.35607	0.687668	0.0890258	0.0543349

		Σ		
5.14662	0	0	0	0
0	4.10518	0	0	0
0	0	2.52217	0	0
0	0	0	0.926352	0
0	0	0	0	0.663583

		V^T		
-0.318035	-0.35969	-0.645576	-0.315168	-0.503366
0.143116	-0.784922	0.383085	-0.403612	0.231854
-0.64811	0.05576	-0.210909	-0.125551	0.718746
0.493627	0.378115	-0.276839	-0.701254	0.212049
-0.463311	0.3293	0.561564	-0.479841	-0.362358

Singular value matrix helps us determine how to reduce size of the three component matrices by restricting the matrices U, Σ, V^T to their first k<n rows.

$X = U_{t\times k}, \Sigma_{k\times k}, (V_{d\times k})^T$

From a practical perspective, the key is to determine 'k', which would be reasonable for the problem. Top k values will capture the essence of data. Typically in large datasets, around k=300 is chosen.

In our example, if we choose to retain k=3 singular vectors or three latent dimensions (in other words, we are saying σ values 5.146, 4.105, 2.522 capture the important dimensions), U matrix will have 6 rows and 3 columns, Σ matrix will have 3 rows and 3 columns and V^T matrix will have 3 rows and 5 columns.

If we multiply the three truncated matrices, we get a matrix of the original dimensions but this process serves four important purposes: a) capturing latent meaning, b) noise reduction, c) high-order co-occurrence, and d) sparsity reduction.

	U	
-0.25712	-0.0630233	-0.318479
-0.29882	-0.424245	0.279408
-0.0617946	0.0348624	-0.256965
-0.332141	-0.770245	-0.033234
-0.658944	0.307838	-0.529603
-0.54429	0.35607	0.687668

	Σ	
5.14662	0	0
0	4.10518	0
0	0	2.52217

		V^T		
-0.318035	-0.35969	-0.645576	-0.315168	-0.503366
0.143116	-0.784922	0.383085	-0.403612	0.231854
-0.64811	0.05576	-0.210909	-0.125551	0.718746

Once we multiply the three matrices, we get a 6 x 5 latent semantic space. Thus, the SVD transformation creates a semantic space out of the TDM. The SVD combines vectors in a manner such that semantically related are closest to each another, thereby preserving a big chunk of original information in fewer dimensions. LSA allows us to look at latent concepts and many other things in the latent semantic space. Note that U is now the reduced matrix representation of words and V is reduced matrix representations of documents.

What Can We Do With LSA?

In the semantic space, a simple analysis could be computing cosine similarity. For instance, we can find which terms are in proximity by calculating the cosine similarity between vectors in the $\Sigma \times U$ matrix. We can also look at relation between documents by computing cosine similarity between vectors in the $\Sigma \times V^T$ matrix.

Further analysis such as cluster analysis can be performed to group terms or documents. Since both terms and documents are in the same latent semantic space, we can find terms that are most related to a given document,thereby providing document labels. Similarly, we can find documents most related to particular term of interest, thus grouping documents together.

We can also project new content into our existing semantic space. These projected content are called pseudo-documents which can be compared to existing terms or documents or to each other. We will explore some of these with R code next.

An excellent source for understanding how to conduct LSA in R is: *A Guide to Text Analysis with Latent Semantic Analysis in R with Annotated Code: Studying Online Reviews and the Stack Exchange Community (Gefen et al. 2017)*

LSA in R

Let's understand these issues with our original hotel review example. Here is a snapshot of preprocessed tfidf weighted term document matrix. Note that I have taken a small sample of the original dataset, otherwise the computations are very resource intensive.

```
suppressMessages(library(tidyverse))
set.seed(1234)
hotel_raw <- read_csv("C:/Users/u0474728/Dropbox/Utah Department Stuff/Teaching/Text Analysis/Summer 201
hotel_raw<-hotel_raw[sample(nrow(hotel_raw), 100), ]# randomly select 100 rows
hotel_raw_token2 <- tokens(hotel_raw$Description, what = "word",
                       remove_numbers = TRUE, remove_punct = TRUE,
                       remove_symbols = TRUE)


#Create DFM
hotel_raw_dfm<-hotel_raw_token2 %>%
  tokens_remove(stopwords(source = "smart")) %>%
  #tokens_wordstem() %>%
  tokens_tolower() %>%
  dfm()

hotel_tfidf<-t(dfm_tfidf(hotel_raw_dfm, scheme_tf = "prop", scheme_df = "inverse", base = 10))
hotel_tfidf1<-as.matrix(hotel_tfidf)
hotel_tfidf1[1:6,1:6]
```

```
##           docs
## features        text1      text2      text3       text4       text5 text6
##   hotel    0.00539527 0.00242484 0.01438739 0.001926882 0.002248029     0
##   rated    0.05000000 0.00000000 0.00000000 0.000000000 0.000000000     0
##   romantic 0.07614394 0.00000000 0.00000000 0.000000000 0.000000000     0
##   hotels   0.02134680 0.00000000 0.00000000 0.000000000 0.000000000     0
##   boston   0.03252575 0.00000000 0.00000000 0.000000000 0.000000000     0
##   agree    0.05000000 0.00000000 0.00000000 0.000000000 0.000000000     0
```

Now let us fit the LSA model using r package lsa. By specifying 'dims=dimcalc_share()', R chooses the number of dimensions to retain by default. To specify a specific number of dimensions, replace dimcalc_share() with the number (e.g., 3)

```r
library(lsa)
library(LSAfun)
hotel_LSAspace <- lsa(hotel_tfidf, dims=dimcalc_share())

dim(hotel_LSAspace$tk)
```

```
## [1] 2114   38
```

```r
dim(hotel_LSAspace$dk)
```

```
## [1] 100  38
```

```r
length(hotel_LSAspace$sk) # Because the $sk matrix
```

```
## [1] 38
```

```r
#only has values on the diagonal, R stores it as a numeric vector.

hotel_LSAspace$tk[1:5,1:5]
```

```
##                   [,1]        [,2]         [,3]         [,4]         [,5]
## hotel    -0.0053575613 0.047730066 -0.020669178  0.041699286 -0.024602666
## rated    -0.0006375985 0.006326478 -0.002961274  0.007046346 -0.002316362
## romantic -0.0017304447 0.012231198 -0.008185786  0.017922970 -0.006983861
## hotels   -0.0035723975 0.020372618 -0.023505182  0.042475259 -0.018504887
## boston   -0.0030096839 0.305038577  0.088484202 -0.080266128 -0.017767895
```

```r
hotel_LSAspace$dk[1:5,1:5]
```

```
##               [,1]       [,2]        [,3]       [,4]        [,5]
## text1 -0.007072649 0.05656720 -0.02546342 0.05840793 -0.01804098
## text2 -0.006155764 0.01993496 -0.02743174 0.05495960 -0.03256147
## text3 -0.006127037 0.03798812 -0.05445636 0.10621403 -0.07811181
## text4 -0.005461308 0.02452736 -0.03413193 0.05691717 -0.02910521
## text5 -0.003328859 0.03143325 -0.02710506 0.06500906  0.03940381
```

```r
hotel_LSAspace$sk[1:10]
```

```
##  [1] 0.5546318 0.4470671 0.4299404 0.4144554 0.3894249 0.3780577 0.3483496
##  [8] 0.3409937 0.3384872 0.3262508
```

From lsa package's documentation: A document-term matrix M is constructed from a given text base of n documents containing m terms. This matrix M of the size m×n is then decomposed via a singular value decomposition into: term vector matrix T (constituting left singular vectors), the document vector matrix D (constituting right singular vectors) being both orthonormal, and the diagonal matrix S (constituting singular values).

These matrices are then reduced to the given number of dimensions k=dims to result into truncated matrices Tk, Sk and Dk — the latent semantic space. If these matrices Tk,Sk,Dk were multiplied, they would give a new matrix Mk (of the same format as M, i.e., rows are the same terms, columns are the same documents), which is the least-squares best fit approximation of M with k singular values.

In the code above, the function lsa outputs three matrices combined in the LSAspace object as a list. We can glimpse at small subsets of these matrices. The hotel_LSAspace$tk refers to the term matrix, hotel_LSAspace$dk refers to the document matrix, and hotel_LSAspace$sk represents the singular value matrix.

As mentioned previously, to find which are proximate terms we can calculate the cosine similarity between vectors in the $\Sigma \times U$ matrix. So let's first multiply the truncated Σ, U matrices.

```
tk2 = t(hotel_LSAspace$sk * t(hotel_LSAspace$tk))
tk2[1:10,1:3]
```

```
##                   [,1]        [,2]         [,3]
## hotel    -0.0029714740 0.021338542 -0.008886514
## rated    -0.0003536324 0.002828360 -0.001273171
## romantic -0.0009597597 0.005468166 -0.003519400
## hotels   -0.0019813653 0.009107927 -0.010105827
## boston   -0.0016692664 0.136372709  0.038042930
## agree    -0.0003536324 0.002828360 -0.001273171
## deluxe   -0.0005372098 0.003472798 -0.002744376
## suite    -0.0030200814 0.008965294 -0.008166514
## bathtub  -0.0003536324 0.002828360 -0.001273171
## shower   -0.0014796099 0.008486377 -0.010038196
```

In the truncated output, we can see that "address", "taxi", "changed" have similar values across first, second and third dimension, indicating that they co-occur across documents. The terms "times" and "side" seem close in the third dimension but are separated in the first and second dimension. Before exploring cosine similarity of terms, let's also multiply the $\Sigma \times V^T$ matrices for document comparison.

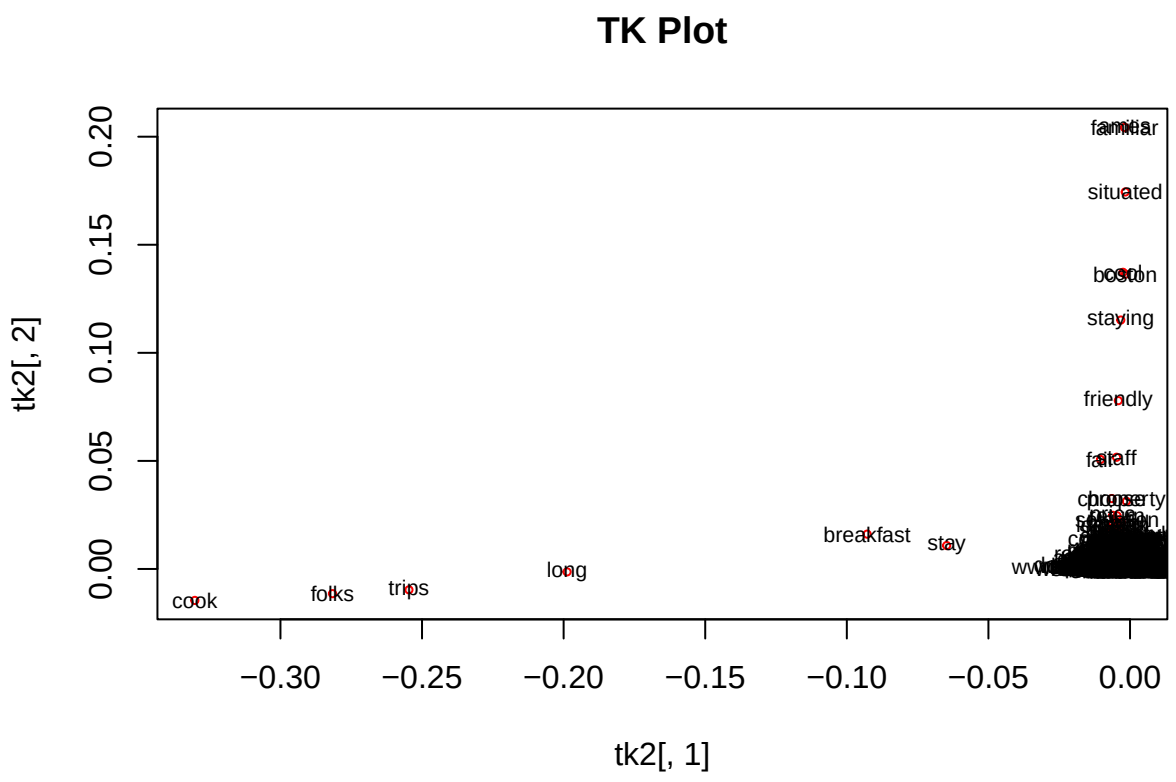
```
dk2 = t(hotel_LSAspace$sk * t(hotel_LSAspace$dk))
dk2[1:10,1:3]
```

```
##                [,1]        [,2]         [,3]
## text1  -0.003922716 0.025289333 -0.010947754
## text2  -0.003414182 0.008912266 -0.011794012
```

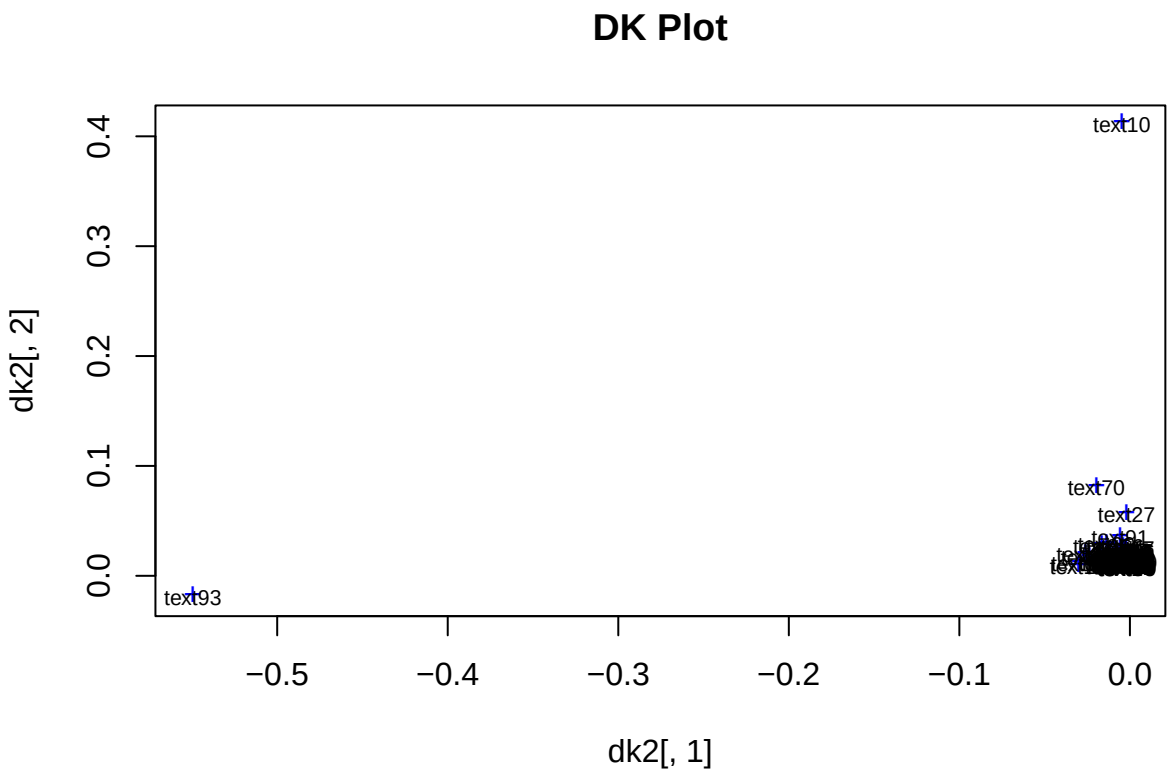
```
## text3  -0.003398250 0.016983240 -0.023412985
## text4  -0.003029015 0.010965374 -0.014674693
## text5  -0.001846291 0.014052771 -0.011653557
## text6  -0.012213427 0.018272651 -0.021530661
## text7  -0.002955748 0.016589305 -0.020180638
## text8  -0.002407817 0.012287127 -0.009296915
## text9  -0.003355055 0.012897480 -0.017015047
## text10 -0.004671463 0.410879555  0.118299503
```

Let's visualize the terms and documents in two dimensions

```
plot(tk2[,1], y= tk2[,2], col="red", cex=.50, main="TK Plot")
text(tk2[,1], y= tk2[,2], labels=rownames(tk2) , cex=.70)
```




```
# This can be done with the documents too. The added parameter cex determines text size.
plot(dk2[,1], y= dk2[,2], col="blue", pch="+", main="DK Plot")
text(dk2[,1], y= dk2[,2], labels=rownames(dk2), cex=.70)
```



Let's look at cosine similarity between terms

```
# Create a cosine similarity between two Terms
myCo <- costring('nice','hotel', tvectors= tk2)
myCo
```

```
## [1] 0.5090996
```

```
myCo1 <- costring('hotel','view', tvectors= tk2)
myCo1
```

```
## [1] 0.1525145
```

We can compute cosine similarity between all the terms and look at some.

```
myTerms2 <- rownames(tk2)
myCosineSpace2 <- multicos(myTerms2, tvectors=tk2)
#breakdown=TRUE forces data into lower case
myCosineSpace2[1:7,1:7]
```

```
##              hotel     rated   romantic    hotels     boston     agree
```

```
## hotel    1.0000000 0.3802361 0.22844235 0.3219929 0.31951003 0.3802361
## rated    0.3802361 1.0000000 0.69433087 0.1913072 0.17547002 1.0000000
## romantic 0.2284423 0.6943309 1.00000000 0.1080618 0.08072534 0.6943309
## hotels   0.3219929 0.1913072 0.10806181 1.0000000 0.02943820 0.1913072
## boston   0.3195100 0.1754700 0.08072534 0.0294382 1.00000000 0.1754700
## agree    0.3802361 1.0000000 0.69433087 0.1913072 0.17547002 1.0000000
## deluxe   0.5205006 0.8937253 0.69251241 0.1975940 0.13787787 0.8937253
##             deluxe
## hotel    0.5205006
## rated    0.8937253
## romantic 0.6925124
## hotels   0.1975940
## boston   0.1378779
## agree    0.8937253
## deluxe   1.0000000
```

```
#write.csv(myCosineSpace2, file="termCosineResults.csv")#save file for further analysis
```

Similar analysis can be done for the documents

```
myDocs2 <- rownames(dk2)
myCosineSpace3 <- multicos(myDocs2, tvectors=dk2)
myCosineSpace3[1:6,1:6]
```

```
##            text1     text2       text3      text4       text5       text6
## text1 1.00000000 0.4616150  0.14313797  0.5478252  0.08036898  0.15814582
## text2 0.46161505 1.0000000  0.19527209  0.6953808  0.30628526  0.12848683
## text3 0.14313797 0.1952721  1.00000000 -0.0181833 -0.01180386  0.04263251
## text4 0.54782520 0.6953808 -0.01818330  1.0000000  0.22170871  0.32165168
## text5 0.08036898 0.3062853 -0.01180386  0.2217087  1.00000000 -0.06047703
## text6 0.15814582 0.1284868  0.04263251  0.3216517 -0.06047703  1.00000000
```

We can also look at neighbors of terms or documents

```
neighbors("text5", n=10, tvectors = dk2)
```

```
##     text5    text87    text77     text8    text39    text16     text7    text67
## 1.0000000 0.7561178 0.5446148 0.4708451 0.4632018 0.4437828 0.4316226 0.4263877
##    text84    text42
## 0.4185908 0.4161790
```

```
neighbors("food", n=10, tvectors = tk2)
```

```
##        food       views         top    discount outstanding    thinking
##   1.0000000   0.9341879   0.9294263   0.9259645   0.9250094   0.9070905
##        tips         los     angeles   daughters
##   0.8981656   0.8963602   0.8963602   0.8963602
```

Using the function plot_neighbors, we find the immediate neighbors visually.

```
plot_neighbors("location", n=5, tvectors= tk2)
```

```
##                   x          y           z
## location 0.42382530 0.72741343  0.43950727
## great    0.02053259 0.97546259 -0.03786572
## coffee   0.91874963 0.14835848  0.28689220
## fine     0.88324356 0.11704005  0.37058975
## back     0.50273512 0.06777924  0.84881353
```

There are many more functions that are available in different R packages, look up the Gefen et al. 2017 article if you are interested.

To summarize, LSA uses singular value decomposition to break down TDM into three smaller component matrices. The number of singular vectors can be reduced to a create a less sparse representation of the original such that hidden semantic content can be discovered. We can further use the reduced document matrix as input for subsequent analysis.