Collocations and N-grams

Promothesh Chatterjee*

Importance of Context

So far in the Bag-Of-Word model, we have ignored the order of words, but often the context hides meaning. Let's look at a sentence from one of the consumer reviews in the dataset we have been using so far.

Review: *The service was fine, but the hotel itself fell way below my expectations from such a respectable establishment as Hilton.*

Imagine another review, where I have jumbled the order somewhat.

Jumbled Review: *The hotel was fine, but the service itself fell way below my expectations from such a respectable establishment as Hilton.*

Let's create a document feature matrix for both using R package Quanteda.

```
library(quanteda)
review<- "The service was fine, but the hotel itself
          fell way below my expectations from such a
          respectable establishment as Hilton."

hotel_sample_token <- quanteda::tokens(review, what = "word",
                       remove_numbers = TRUE, remove_punct = TRUE,
                       remove_symbols = TRUE, remove_hyphens = TRUE)


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

dim(hotel_sample_dfm)
```

```
## [1] 1 8
```

```
head(hotel_sample_dfm)
```

```
## Document-feature matrix of: 1 document, 8 features (0.00% sparse) and 0 docvars.
##        features
## docs    servic fine hotel fell expect respect establish hilton
##   text1      1    1     1    1      1       1         1      1
```

Now let's do the same for the jumbled review.

```
jreview<- "The hotel was fine, but the service itself
           fell way below my expectations from such a
           respectable establishment as Hilton."

hotel_jsample_token <- quanteda::tokens(jreview, what = "word",
                        remove_numbers = TRUE, remove_punct = TRUE,
                        remove_symbols = TRUE, remove_hyphens = TRUE)


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

dim(hotel_jsample_dfm)
```

```
## [1] 1 8
```

```
head(hotel_jsample_dfm)
```

```
## Document-feature matrix of: 1 document, 8 features (0.00% sparse) and 0 docvars.
##        features
## docs    hotel fine servic fell expect respect establish hilton
##   text1     1    1      1    1      1       1         1      1
```

If you compare the two DFMs, they are essentially the same (the order of columns does not matter in a matrix). However, we know that the two reviews are fairly different. So what can we do about such issues? We have to figure out a way for incorporating context in our analysis.

Co-occurrence and Collocations of Words

As we have seen, the bag-of-words model typically ignores order of the words and that can have impact on the way text is interpreted. How can one determine if some words co-occur more frequently than can be expected by chance? Collocations refers to the words occurring frequently together. For instance, *Customer Service* is an example because they co-occur more frequently than others.

N-grams are collocated words that occur together (for instance, bi-grams are two words that occur together, tri-grams three words and so on). N grams is actually a generic term where instead of taking single words as tokens, we take some combinations of words. For instance, if we take 2 contiguous words as our token, then the resulting N gram is called a Bi-gram. Similarly, tri-grams, 4-grams etc. N grams for especially useful when we are negating an adjective which changes the meaning of the word. For instance, the if the sentence

contains a bi-gram "Not_bad", this is absolutely opposite of "bad". In a bag-of-words models, subtleties like this may be missed. Additionally, N grams can also be useful for identifying context words such as "food quality", "room service" etc. Thus, compared to numeric data, making sense of text data is often more complicated. Let's use library tokenizers to generate bigrams.

```
library(tokenizers)
tokenize_ngrams(review, n = 2, n_min = 2) # generates bigrams
```

```
## [[1]]
##  [1] "the service"               "service was"
##  [3] "was fine"                  "fine but"
##  [5] "but the"                   "the hotel"
##  [7] "hotel itself"              "itself fell"
##  [9] "fell way"                  "way below"
## [11] "below my"                  "my expectations"
## [13] "expectations from"         "from such"
## [15] "such a"                    "a respectable"
## [17] "respectable establishment" "establishment as"
## [19] "as hilton"
```

```
# An n-gram is a contiguous sequence of words
# containing at least n_min words and at most n words.
```

Since a collocation analysis allows us to identify words that co-occur, it could be used to identify proper names simply on the basis of their capitalization. Let's use the function textstat_collocations from the another library from the Quanteda suite, quanteda.textstats.

```
library(quanteda)
library(readtext)# https://readtext.quanteda.io/articles/readtext_vignette.html
library(tidyverse)# For loading bunch of packages such as dplyr, stringi etc.

# Load up the .CSV data.
hotel_raw <- readtext("C:/Users/u0474728/Dropbox/Utah Department Stuff/Teaching/Text Analysis/Summer 20

set.seed(1245654)# for replicability
hotel_raw <- slice_sample(hotel_raw,n=5000)# take a small sample

my_corpus<-corpus(hotel_raw, text_field = "text")

toks_reviews <- tokens(my_corpus, remove_punct = TRUE)

tstat_col_caps <-  tokens_select(toks_reviews, pattern = "^[A-Z]",
                                 valuetype = "regex",
```

```
                                    case_insensitive = FALSE,
                                    padding = TRUE) %>%
                quanteda.textstats::textstat_collocations(min_count = 10, size = 2)
head(tstat_col_caps, 10)
```

```
##        collocation count count_nested length    lambda        z
## 1    times square   264            0      2 10.138017 53.20470
## 2    central park    79            0      2  8.459749 47.44918
## 3    union square   118            0      2  8.639495 46.79117
## 4          when i   209            0      2  3.664619 41.55358
## 5   grand central    49            0      2  8.706633 40.50214
## 6  state building    48            0      2 10.315801 36.71092
## 7       york city    38            0      2  6.979220 34.70389
## 8    empire state    73            0      2 12.039650 34.32995
## 9    penn station    35            0      2  9.957324 33.62767
## 10    grand hyatt    28            0      2  7.682984 32.66338
```

Let's understand what the R code is doing. After the usual preprocessing, the **`tokens_select()`** function is being used to select tokens (i.e., individual words) from a pre-existing object called **`toks_news`**. The **`pattern`** argument is set to "^[A-Z]", which is a regular expression pattern that matches any word that starts with a capital letter. The **`valuetype`** argument is also set to "regex", indicating that the pattern argument should be treated as a regular expression. Regular expressions (also called regex) are a set of pattern matching commands used to detect string sequences in text data[1]. The **`case_insensitive`** argument is set to **`FALSE`**, indicating that the matching of capital letters should be case sensitive. The **`padding`** argument is set to **`TRUE`**, indicating that the selected tokens should include one word of padding on either side. The **`%>%`** operator is then used to "pipe" the output of **`tokens_select()`** into the **`textstat_collocations()`** function. This function identifies frequently occurring pairs of words (i.e., collocations) in the selected tokens. The **`min_count`** argument is set to 100, meaning that the function will only identify collocations that occur at least 100 times in the selected tokens. The **`size`** argument can be used to even identify three word proper nouns.

Overall, this code selects all words from a pre-existing object that start with a capital letter, and then identifies frequently occurring collocations within those words.

Issues with N-gram Tokenization

Adding n-grams significantly increases the number of columns in the document feature matrix and consequently aggravates the problem of dimensionality and sparsity. Thus, while looking at words in combinations do help us deal with inherent challenges of text data, there is a significant cost associated with it.

Let's look at this issue in our dataset.

[1]For a simple exposition of string manipulations and regex, see: https://www.hackerearth.com/practice/machine-learning/advanced-techniques/regular-expressions-string-manipulation-r/tutorial/

```
hotel_raw_token2 <- quanteda::tokens(hotel_raw$text,
                            what = "word",
                            remove_numbers = TRUE,
                          remove_punct = TRUE,
                          remove_symbols = TRUE)

#More preprocessing
hotel_raw_token2<-hotel_raw_token2 %>%
  tokens_remove(stopwords(source = "smart")) %>%
  tokens_wordstem() # Note I am not doing tokens_tolower

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

#Create N Grams with library quanteda
hotel_ngrams<-hotel_raw_token2 %>%
  tokens_ngrams(n=1:2)%>%
    dfm()
dim(hotel_raw_dfm) # The original document feature matrix
```

```
## [1]  5000 15374
```

```
dim(hotel_ngrams)
```

```
## [1]   5000 193429
```

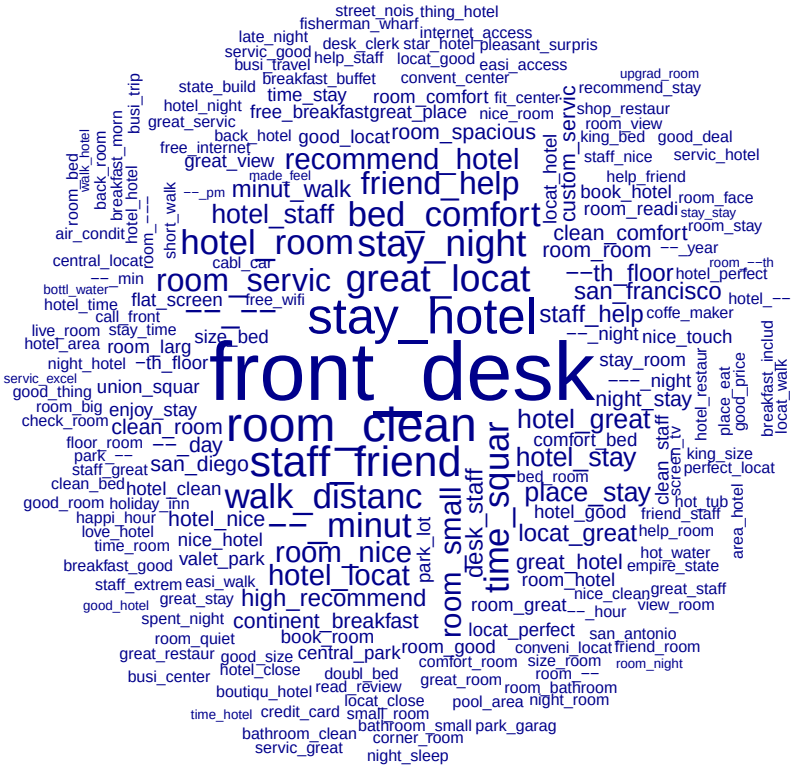
Look at the jump in dimensions when we create just bigrams. Imagine what will happen with tri-grams and 4 grams etc.

Practical Ways to do N grams

We saw that if you take all N grams in a dataset, it is not pragmatic. So one way to deal with this is consider customized N grams. Considering customized N grams keeps a check on issue of dimensionality but making meaningful N grams is an iterative process.

So what is the method to customize N grams? There is no hard and fast rule as it is very context dependent (and hence it helps if you have knowledge of the domain of the generated text). However, we can do a few things. First, let's look at the the word cloud of just bigrams (First, I will modify R the code used to above for generating N grams).

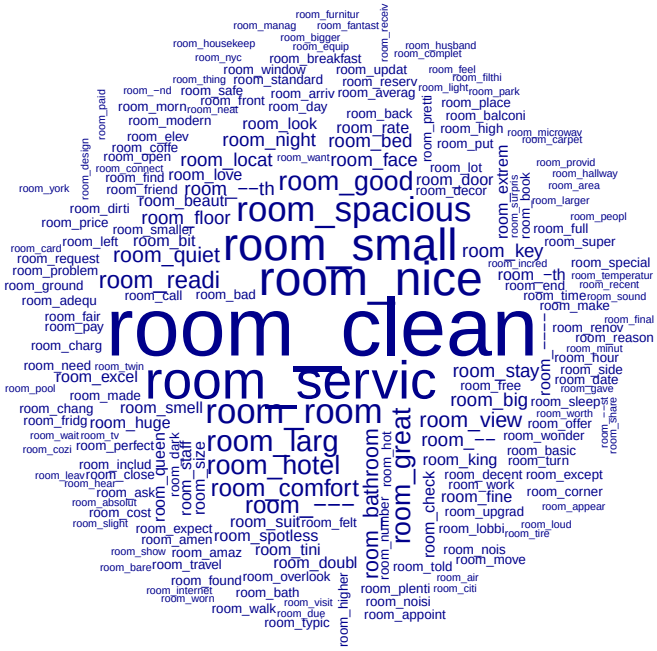
```r
hotel_ngrams<-hotel_raw_token2 %>%
  tokens_ngrams(n=2)%>%
    dfm()

library(quanteda.textplots)# for word clouds
textplot_wordcloud(hotel_ngrams, min_size = 0.5,
  max_size = 4,
  min_count = 3,
  max_words = 200,
  color = "darkblue",
  font = NULL,
  adjust = 0,
  rotation = 0.1,
  random_order = FALSE,
  random_color = FALSE,
  ordered_color = FALSE,
  labelcolor = "gray20",
  labelsize = 1.5,
  labeloffset = 0,
  fixed_aspect = TRUE,
  comparison = FALSE)
```



We can see that front_desk, stay_hotel, room_clean etc. are some of the bigrams that come with high frequency. This will give you an idea as what some bigrams that should be considered are.

Let's use some customized bigrams here. Let me pick all phrases starting with the word 'room'.

```
hotel_ngrams<-hotel_raw_token2 %>%
  tokens_ngrams(n=2)# Doing this step as tokens_compound works only on tokens

cust_bigram<- tokens_compound(hotel_ngrams, phrase("room*"))
cust_bigram<- tokens_select(cust_bigram, phrase("room_*"))

cust_bigram<-dfm(cust_bigram)

textplot_wordcloud(cust_bigram, min_size = 0.5,   max_size = 4,
  min_count = 3,   max_words = 200,   color = "darkblue",   font = NULL,
  adjust = 0, rotation = 0.1, random_order = FALSE,   random_color = FALSE,
  ordered_color = FALSE,   labelcolor = "gray20",   labelsize = 1.5,
  labeloffset = 0,   fixed_aspect = TRUE,   comparison = FALSE)
```



Let's understand what the R code is saying.

1. After creating the bigrams, the **`tokens_compound`** function is used to combine the bigrams that contain the word "room" into a single token, using the **`phrase`** argument with a wildcard character (*).
2. The **`tokens_select`** function is used to select only the bigrams that match the pattern "room_*", which means that the bigram should start with the word "room" and end with any other word.
3. The resulting tokens are then converted to a document-feature matrix (DFM) using the **`dfm`** function.
4. Finally, the **`textplot_wordcloud`** function is used to create a word cloud visualization of the bigrams. The visualization shows the most frequent bigrams in the data, with the size of each bigram representing its frequency. The **`min_size`**, **`max_size`**, **`min_count`**, and **`max_words`** arguments control the appearance of the word cloud.

Overall, this code performs text preprocessing, feature extraction, and visualization to identify frequent bigrams related to hotel rooms in the given text data.

So you can do this exercise with different words to find out what are the most frequently occurring n grams, and which ones can be important for your analysis. Essentially, it is next to impossible to work with all n grams, you have to strategize which n grams to use. The strategies can include using wordclouds, their frequency distributions, then select the relevant words and see the visualization, pick the important bi-grams or tri-grams. The domain knowledge will also give you more hints as to which n grams to use for your subsequent analysis.

Not adjacent collocates Unfortunately, words that collocate do not have to be immediately adjacent but can also encompass several slots which makes it harder to retrieve of collocates that are not adjacent. We can try a few techniques using Quanteda.

```
library(quanteda.textstats)
library(quanteda.textplots)

text_sentences <- hotel_raw$text %>%
  tolower() %>%
  paste0(collapse= " ") %>%
  stringr::str_split(fixed(".")) %>%
  unlist() %>%
  stringr::str_squish()

text_tokens <- tokens(text_sentences, remove_punct = TRUE) %>%
  tokens_remove(stopwords("english"))
# extract collocations
text_coll <- textstat_collocations(text_tokens, size = 2, min_count = 100)
text_coll[1:10,1:5]
```

```
##          collocation count count_nested length   lambda
## 1         front desk  1065            0      2 7.961520
## 2       times square   315            0      2 6.919765
## 3  friendly helpful   311            0      2 5.105948
## 4    staff friendly   420            0      2 4.201447
## 5     across street   260            0      2 6.679613
## 6    great location   398            0      2 3.636392
## 7  walking distance   370            0      2 9.060015
## 8    within walking   174            0      2 6.405168
## 9        --th floor   188            0      2 5.803832
## 10        next door   185            0      2 4.867748
```

1. The **text_sentences** variable is created by taking the **text** column of the **hotel_raw** dataset, converting it to lowercase using the **tolower** function, concatenating all sentences into a single string using **paste0**, splitting the string into individual sentences using the period character as a delimiter with the **str_split** function from the **stringr** package, converting the resulting list to a character vector using **unlist**, and removing any extra white spaces between words using **str_squish** from the **stringr** package.
2. The **text_tokens** variable is created by tokenizing the **text_sentences** with the **tokens** function from the **quanteda** package, removing punctuation with the **remove_punct** argument, and removing English stopwords with the **stopwords** function from the **quanteda** package.
3. The **textstat_collocations** function from the **quanteda.textstats** package is used to extract collocations (pairs of words that occur together more often than would be expected by chance) from the

text_tokens data. The **size** argument specifies the size of the collocation (in this case, 2-word collocations), and the **min_count** argument sets a minimum threshold for the frequency of each collocation. The resulting collocations are stored in the **text_coll** variable.

4. The last line of code prints the first 10 rows and 5 columns of the **text_coll** variable, which shows the most frequent collocations and their frequencies.

Overall, this code performs text preprocessing, tokenization, collocation extraction, and outputting the most frequent collocations in the given text data.

We can also check co-occurrence counts.

```
options(stringsAsFactors = FALSE)
library(quanteda)

hotel_corpus <- corpus(hotel_raw$text, docnames = hotel_raw$doc_id)
# original corpus length and its first document
ndoc(hotel_corpus)
```

```
## [1] 5000
```

```
substr(as.character(hotel_corpus)[1], 0, 200)
```

```
##
## "I had some reservations about staying here however it suited our needs. Basically, we wanted a bed
```

```
corpus_sentences <- corpus_reshape(hotel_corpus, to = "sentences")

ndoc(corpus_sentences)
```

```
## [1] 48705
```

```
as.character(corpus_sentences)[1]
```

```
##                                                hotel-reviews.csv.26420.1
## "I had some reservations about staying here however it suited our needs."
```

```
# Preprocessing of the corpus of sentences
corpus_tokens <- corpus_sentences %>%
  tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
  tokens_tolower()

# calculate multi-word unit candidates
hotel_collocations <- textstat_collocations(corpus_tokens, min_count = 25)
```

```r
hotel_collocations <- hotel_collocations[1:250, ]

corpus_tokens <- tokens_compound(corpus_tokens, hotel_collocations)

minimumFrequency <- 10

# Create DTM, prune vocabulary and set binary values for presence/absence of types
binDTM <- corpus_tokens %>%
  tokens_remove("") %>%
  dfm() %>%
  dfm_trim(min_docfreq = minimumFrequency) %>%
  dfm_weight("boolean")

# Matrix multiplication for cooccurrence counts
coocCounts <- t(binDTM) %*% binDTM
as.matrix(coocCounts[192:195, 192:195])
```

```
##              features
## features      appointment   or have family
##   appointment          11    1    3      1
##   or                    1 1740  133     24
##   have                  3  133 1944     28
##   family                1   24   28    372
```

This R code performs several tasks related to text analysis and natural language processing using the **quanteda** package. Here's an overview of what each section of the code is doing:

1. Creating a corpus from a raw text data frame (**hotel_raw**), with each document labeled by a unique **doc_id**.
2. Reshaping the original corpus into a corpus of sentences.
3. Preprocessing the corpus of sentences by tokenizing the text and applying several text cleaning steps such as removing punctuation, numbers, and symbols, and converting all tokens to lowercase.
4. Identifying multi-word units (collocations) in the text using the **textstat_collocations** function, and extracting the top 250 most frequent collocations.
5. Identifying compound words in the text that match the extracted collocations, and replacing them with a single token.
6. Creating a document-term matrix (DTM) from the tokenized text, removing infrequent terms that appear in fewer than **minimumFrequency** documents, and converting the resulting DTM to a binary format indicating the presence or absence of each term in each document.

7. Calculating co-occurrence counts between all pairs of terms in the DTM using matrix multiplication.

   Statistical significance

   In order to not only count joint occurrence we have to determine their significance. Different significance-measures can be used. I am using a script to calculate the co-occurrence significance measures. If you are interested in the details, there is an appendix which contains the details.

```
# Read in the source code for the co-occurrence calculation
source("C:/Users/u0474728/Dropbox/Utah Department Stuff/Teaching/Text Analysis/Summer 2024/Course Materi
# Definition of a parameter for the representation of the co-occurrences of a concept
numberOfCoocs <- 15
# Determination of the term of which co-competitors are to be measured.
coocTerm <- "marriott"

coocs <- calculateCoocStatistics(coocTerm, binDTM, measure="LOGLIK")
# Display the numberOfCoocs main terms
print(coocs[1:numberOfCoocs])
```

Visualization of co-occurrence

```
##      courtyard             jw        rewards         points     properties
##       148.30352       83.10065       60.55257       39.13767       34.37454
##     waterfront    renaissance           this        typical          at_the
##       33.64631       26.70149       23.71171       21.58775       21.37566
##   indianapolis staying_at_the      residence         member         recent
##       21.15330       20.41790       18.76620       18.24143       17.76537
```

In this R code, the first line reads in the source code for the function that calculates co-occurrence statistics. This function has been defined in a separate R script file, which is then sourced into the current R session.

The second line of code sets the numberOfCoocs parameter to 15. This parameter will be used later in the analysis to specify the number of most frequent co-occurring terms to include in the results.

The third line of code sets the coocTerm parameter to "marriott". This is the term for which we want to measure co-occurrences with other terms in the corpus.

These parameter settings are important for specifying the analysis to be conducted and will be used in subsequent code to calculate and display co-occurrence statistics for the specified coocTerm.

One way to expand the semantic context of a target term is to compute "secondary co-occurrence" terms for every co-occurring term associated with the target term. This generates a graph that can be visualized using specialized layout algorithms such as Force Directed Graph.

To analyze and visualize network graphs in R, the igraph-package can be used. A three-column data-frame can be used to create any graph object. Each row in the data-frame represents a triple, encoding edge information for two nodes (source, sink) and an edge-weight value.

In the case of a term co-occurrence network, each triple consists of the target word, a co-occurring word, and the significance of their joint occurrence. These values can be denoted as "from," "to," and "sig," respectively.

```
resultGraph <- data.frame(from = character(), to = character(), sig = numeric(0))
```

The network collection process for the target term consists of two steps. Initially, we collect all the significant co-occurrence terms related to the target term. Subsequently, we gather all the co-occurrences of the co-occurrence terms obtained from the first step.

The intermediate outcomes for each term are temporarily stored as triples named "tmpGraph." Using the "rbind" command, which concatenates data-frames vertically, all "tmpGraph" objects are appended to the final network object that is stored in "resultGraph."

```
# The structure of the temporary graph object is equal to that of the resultGraph
tmpGraph <- data.frame(from = character(), to = character(), sig = numeric(0))

# Fill the data.frame to produce the correct number of lines
tmpGraph[1:numberOfCoocs, 3] <- coocs[1:numberOfCoocs]
# Entry of the search word into the first column in all lines
tmpGraph[, 1] <- coocTerm
# Entry of the co-occurrences into the second column of the respective line
tmpGraph[, 2] <- names(coocs)[1:numberOfCoocs]
# Set the significances
tmpGraph[, 3] <- coocs[1:numberOfCoocs]

# Attach the triples to resultGraph
resultGraph <- rbind(resultGraph, tmpGraph)

# Iteration over the most significant numberOfCoocs co-occurrences of the search term
for (i in 1:numberOfCoocs){

  # Calling up the co-occurrence calculation for term i from the search words co-occurrences
  newCoocTerm <- names(coocs)[i]
  coocs2 <- calculateCoocStatistics(newCoocTerm, binDTM, measure="LOGLIK")

  #print the co-occurrences
  coocs2[1:10]

  # Structure of the temporary graph object
  tmpGraph <- data.frame(from = character(), to = character(), sig = numeric(0))
  tmpGraph[1:numberOfCoocs, 3] <- coocs2[1:numberOfCoocs]
  tmpGraph[, 1] <- newCoocTerm
  tmpGraph[, 2] <- names(coocs2)[1:numberOfCoocs]
```

```
  tmpGraph[, 3] <- coocs2[1:numberOfCoocs]

  #Append the result to the result graph
  resultGraph <- rbind(resultGraph, tmpGraph[2:length(tmpGraph[, 1]), ])
}
```

Some sample results

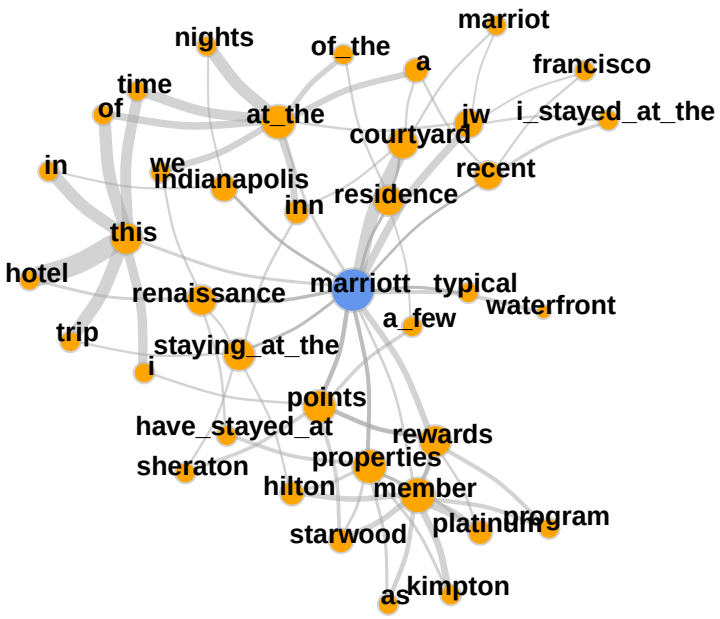
```
# Sample of some examples from resultGraph
resultGraph[sample(nrow(resultGraph), 6), ]
```

```
##             from       to       sig
## 89       typical     eggs 16.710820
## 311 indianapolis     east 19.746874
## 142           jw    santa  7.929767
## 84        points sheraton 30.706662
## 134       points     good 14.012084
## 82            jw  marriot  9.057155
```

Using iGraph

```
require(igraph)

# set seed for graph plot
set.seed(1)

# Create the graph object as undirected graph
graphNetwork <- graph.data.frame(resultGraph, directed = F)

# Identification of all nodes with less than 2 edges
verticesToRemove <- V(graphNetwork)[degree(graphNetwork) < 2]
# These edges are removed from the graph
graphNetwork <- delete.vertices(graphNetwork, verticesToRemove)

# Assign colors to nodes (search term blue, others orange)
V(graphNetwork)$color <- ifelse(V(graphNetwork)$name == coocTerm, 'cornflowerblue', 'orange')

# Set edge colors
E(graphNetwork)$color <- adjustcolor("DarkGray", alpha.f = .5)
# scale significance between 1 and 10 for edge width
E(graphNetwork)$width <- scales::rescale(E(graphNetwork)$sig, to = c(1, 10))

# Set edges with radius
```

```r
E(graphNetwork)$curved <- 0.15
# Size the nodes by their degree of networking (scaled between 5 and 15)
V(graphNetwork)$size <- scales::rescale(log(degree(graphNetwork)), to = c(5, 15))

# Define the frame and spacing for the plot
par(mai=c(0,0,1,0))

# Final Plot
plot(
  graphNetwork,
  layout = layout.fruchterman.reingold, # Force Directed Layout
  main = paste(coocTerm, ' Graph'),
  vertex.label.family = "sans",
  vertex.label.cex = 0.8,
  vertex.shape = "circle",
  vertex.label.dist = 0.5,          # Labels of the nodes moved slightly
  vertex.frame.color = adjustcolor("darkgray", alpha.f = .5),
  vertex.label.color = 'black',     # Color of node names
  vertex.label.font = 2,            # Font of node names
  vertex.label = V(graphNetwork)$name,      # node names
  vertex.label.cex = 1 # font size of node names
)
```



This R code generates a graph visualization of a network based on co-occurrence data. The graph is created using the "igraph" package.

Here is what each line of code does:

- `require(igraph)`: loads the igraph package.
- `set.seed(1)`: sets a random seed value for reproducibility.
- `graphNetwork <- graph.data.frame(resultGraph, directed = F)`: creates an undirected graph object from the data-frame "resultGraph".
- `verticesToRemove <- V(graphNetwork)[degree(graphNetwork) < 2]`: identifies all nodes with less than 2 edges.
- `graphNetwork <- delete.vertices(graphNetwork, verticesToRemove)`: removes the identified nodes and their corresponding edges from the graph.
- `V(graphNetwork)$color <- ifelse(V(graphNetwork)$name == coocTerm, 'cornflowerblue', 'orange')`: sets the color of the target term node to blue and all other nodes to orange.
- `E(graphNetwork)$color <- adjustcolor("DarkGray", alpha.f = .5)`: sets the color of edges to dark gray with a transparency level of 50%.
- `E(graphNetwork)$width <- scales::rescale(E(graphNetwork)$sig, to = c(1, 10))`: scales the edge width between 1 and 10 based on the significance level.

- `E(graphNetwork)$curved <- 0.15`: sets the curve of the edges.
- `V(graphNetwork)$size <- scales::rescale(log(degree(graphNetwork)), to = c(5, 15))`: sizes the nodes based on the degree of networking between 5 and 15.
- `par(mai=c(0,0,1,0))`: sets the margins for the plot.
- `plot(graphNetwork, layout = layout.fruchterman.reingold, main = paste(coocTerm, ' Graph'), vertex.label.family = "sans", vertex.label.cex = 0.8, vertex.shape = "circle", vertex.label.dist = 0.5, vertex.frame.color = adjustcolor("darkgray", alpha.f = .5), vertex.label.color = 'black', vertex.label.font = 2, vertex.label = V(graphNetwork)$name, vertex.label.cex = 1)`: generates the plot using the Fruchterman-Reingold force-directed layout algorithm. It also sets various node and label parameters such as shape, size, distance, and font size. Finally, it labels the nodes with their names and displays the plot with a main title.

To sum it up, we have looked at how to incorporate context in a meaningful way using n grams and visualization techniques. We will come back to the topic when we are discussing word embeddings.

Reference https://tm4ss.github.io/docs/Tutorial_5_Co-occurrence.html

Appendix R Script for co-occurence statistics

```
calculateCoocStatistics <- function(coocTerm, binDTM, measure = "DICE") {

  # Ensure Matrix (SparseM} or matrix {base} format
  require(Matrix)
  require(slam)
  if (is.simple_triplet_matrix(binDTM)) {
    binDTM <- sparseMatrix(i=binDTM$i, j=binDTM$j, x=binDTM$v, dims=c(binDTM$nrow, binDTM$ncol), dimnam
  }

  # Ensure binary DTM
  if (any(binDTM > 1)) {
    binDTM[binDTM > 1] <- 1
  }

  # calculate cooccurrence counts
  coocCounts <- t(binDTM) %*% binDTM

  # retrieve numbers for statistic calculation
  k <- nrow(binDTM)
  ki <- sum(binDTM[, coocTerm])
```

```
  kj <- colSums(binDTM)
  names(kj) <- colnames(binDTM)
  kij <- coocCounts[coocTerm, ]

  # calculate statistics
  switch(measure,
         DICE = {
           dicesig <- 2 * kij / (ki + kj)
           dicesig <- dicesig[order(dicesig, decreasing=TRUE)]
           sig <- dicesig
         },
         LOGLIK = {
           logsig <- 2 * ((k * log(k)) - (ki * log(ki)) - (kj * log(kj)) + (kij * log(kij))
                          + (k - ki - kj + kij) * log(k - ki - kj + kij)
                          + (ki - kij) * log(ki - kij) + (kj - kij) * log(kj - kij)
                          - (k - ki) * log(k - ki) - (k - kj) * log(k - kj))
           logsig <- logsig[order(logsig, decreasing=T)]
           sig <- logsig
         },
         MI = {
           mutualInformationSig <- log(k * kij / (ki * kj))
           mutualInformationSig <- mutualInformationSig[order(mutualInformationSig, decreasing = TRUE)]
           sig <- mutualInformationSig
         },
         {
           sig <- sort(kij, decreasing = TRUE)
         }
        )
  sig <- sig[-match(coocTerm, names(sig))]
  return(sig)
}
```

This R code defines a function **calculateCoocStatistics** that takes three parameters: **coocTerm**, **binDTM**, and **measure**. The purpose of this function is to calculate co-occurrence statistics for a given term (**coocTerm**) in a binary document-term matrix (**binDTM**), using a specified statistical measure (**measure**).

The function first ensures that the **binDTM** is in a suitable format for the calculation. If it is not in the sparse matrix format, it is converted to this format. The function then ensures that the **binDTM** is binary by setting any values greater than 1 to 1.

The function then calculates the co-occurrence counts for all terms in the **binDTM**. It also retrieves some numbers (**k**, **ki**, **kj**, **kij**) that will be used to calculate the co-occurrence statistics.

The function then calculates the co-occurrence statistics based on the specified **measure**. The available measures are "DICE", "LOGLIK", "MI", and a default option that sorts the co-occurrence counts in descending

order. The resulting statistics are sorted in descending order and returned as output, with the `coocTerm` removed from the list.

Overall, this function is useful for computing co-occurrence statistics and can be used to explore relationships between terms in a corpus.