Natural Language Processing and Text Mining from a Sports Analytics Perspective

# What is NLP?

## Definitions and Challenges

*Natural Language Processing (NLP) represents the collection of analytical techniques to distill information from vast amounts of unstructured language data*. Often the goal of NLP or text mining is information reduction to learn something about the larger body of text. An analog would be calculating the average age of a country’s population to understand a characteristic of the country. The individual ages are reduced to a single value to help you understand the country’s characteristic say, older or younger demographics rather than interviewing each person individually. Similarly, using a million tweets as your data source, you can use text mining methods to extract frequent concepts, people, places, organizations, and even attitudes without reading a single tweet. The concepts of this chapter will help you reduce noise to extract meaningful information from sports related text.

Keep in mind that text is among the most challenging of data science and analytics problems to do well. This is because human expression varies and communication theory dictates that the channel, messenger and audience may all affect the intended meaning.

As a result, when you start a natural language processing project you should understand the channel, messenger and audience and adjust your analytical methods accordingly. First, channel impacts the language used and will likewise impact the language analysis. For example, a sports contract will obviously use a lot of boiler plate legalistic language and clauses such as “For any contract procured by the Agent and signed by the Player…”. In contrast a social media channel consisting of fans will use slang, swear words and emojis consistent with in that medium. The messenger may also affect language analysis as well. For example, in Boston, sports fans regularly scream that a home run is “wicked awesome” or that an umpire call was “wicked bad.” For most English speakers “wicked” has a negative connotation so “wicked awesome” would make no sense. However, with Boston fans as the messengers “wicked” has an amplifying effect. To others outside of New England a home run for their team could be “really awesome” or a poor referee call could be described as “really bad”. Finally, the intended audience will affect language significantly as well. The language of a sports caster discussing a pivotal play in a game is dramatically different compared to two peers discussing the same play casually over a drink. Another example are the terms a coach would use with a player at the high school, collegiate and professional levels. At each level, the expectations, athleticism, and knowledge of the sport differ dictating the language to be different. These factors are just some of the challenges associated with NLP projects even outside of the sports world. However, these challenges are also part of the reason this analysis is satisfying.

## Where is it Used?

As a consumer NLP is used extensively. For example, identifying spam in an inbox is a form of document classification, using both NLP and machine learning. In terms of sports, natural language processing aptly applies to fan engagement. Sports businesses measure marketing and more specifically fan engagement to improve operations. Sports organizations would be wise to incorporate long understood aspects of traditional business marketing utilizing NLP to their industry. While document classification, human resource reviews, ticket holder propensity modeling and other areas may also utilize NLP, the examples in this chapter primarily use fan engagement and expression to demonstrate concepts.

# Language as Strings

## R’s Classes

The R programming language operates using object *classes*. The most common include Boolean, factor, numeric, integer and character. In fact, the technical name of character is “string”. Refer to table 1.0 to understand the difference of an object class. As you may expect, NLP uses the character string object class most often. This differs from factor levels which can be confusing to new R programmers. Factors have distinct, often repeatable levels such as a roster listing university attended by the athlete. In this roster, university names by themselves are not natural language meant to express an idea but instead to assign a piece of repeatable information to the athletes on the roster. In contrast, character strings may repeat but may not, and are most often used to convey varying meaning. In table 1.0, unlike the other object types, the character data value example is complete compared to the original sports context value. Thus, NLP requires a good understanding of the object class “string” and how to manipulate, correct and adjust it for analysis.

|  |  |  |
| --- | --- | --- |
| Type | Example | Sports Context |
| Boolean | TRUE or FALSE | Did LeBron James play for the Cavaliers? **TRUE** |
| Factor (with distinct “levels”) | Cleveland | The teams LeBron played for are **Cleveland**, Miami, & Los Angeles.  Levels: Cleveland, Miami, Los Angeles |
| Numeric (also called “floating point”) | 27.1 | LeBron James averages **27.1** points per game. |
| Integer | 23 | Lebron James’ jersey number is **23**. |
| Character | “Lebron James is building a new media company that aims to give a voice to Black creators” | “LeBron James is building a new media company that aims to give a voice to Black creators” |

*Table 1.0 demonstrates the common object classes in R.*

Additionally, each object class can be contained within an object *type*. You are likely familiar with object types including vectors, matrices, data frame, lists and arrays. For simplicity, in this chapter data will be in data frames and occasionally simple vectors. For new R programmers think of data frames as a data worksheet like Excel and a vector as a single column in that sheet. The string manipulation functions covered in this section are not exhaustive but should give you a solid foundation to build upon while also covering typical uses.

## Base R string manipulation Example

R has multiple basic functions for dealing with strings. In this section we will apply basic sting functions to a single tweet before applying the functions to a larger body of text.

Consider the following public tweet shown in illustration 1.0 below from this original tweet. <https://twitter.com/missevans__/status/1179078078561226755>

A picture containing bird

Description automatically generated

*Illustration 1.0 An example tweet held in R as a string object.*

When this is pulled into an R session as a character string it will appear as the text below.

@missevans\_\_: Y’all I’m really about to dance on the Cleveland Cavaliers court starting next week \U0001f631\U0001f631\U0001f631

To begin, as the data scientist you may want to remove the unrecognized Unicode characters which are actually the screaming face emojis. R does not natively understand most emojis, especially since they are updated often. Thus, if you are working in a channel that frequently employs them, it is worthwhile to remove, or substitute them. In the following code chunk the character string, `tweet`, is passed to the `gsub(search, substitute, object)` function. This stands for Global Substitution and works with regular expressions to match a character pattern then substitute it to another. The gsub syntax accepts a pattern to search for, then the replacement substitution text and finally the object to search within.

# Create a single example

tweet <- '@missevans\_\_: Y’all I’m really about to dance on the Cleveland Cavaliers court starting next week \U0001f631\U0001f631\U0001f631'

tweet <- gsub('\U0001f631','',tweet)

As shown in the output, the function identified the Unicode text, replaced it with nothing and performed this operation within the last parameter on the `tweet` object. You could replace the identified pattern with any word or characters such as “emoji screaming face” so as to keep track of the emoji expression. You can apply more substitutions for contractions such as “y’all” and “I’m” although the `qdap` package has a function for this. As with many operations in R there are multiple libraries and functions to perform substitutions but this code gives a basic method.

[1] "@missevans\_\_: Y’all I’m really about to dance on the Cleveland Cavaliers court starting next week "

You will notice now that this example tweet is a retweet and the body of text is after a “:”. Often when performing analysis, you need to remove extraneous information like this because it may hinder results. In this case, the ` strsplit(object, search, parameters)` string split function is helpful. It is applied to an object, searches for a pattern and returns a list of text that has been separated by the search parameter. In this example you have a list of one, because there was one object, with two elements for text on either side of the split. The meaningful text is the second element of text and can be accessed with indexing, square brackets, shown below.

tweet <- strsplit(tweet, ':', fixed = T)

[1] "@missevans\_\_"

[2] " Y’all I’m really about to dance on the Cleveland Cavaliers court starting next week "

tweet <- tweet[[1]][2]

Another good base function to know when performing text analysis is `tolower(object)`. Often word frequency can be useful in learning about author intent, effort and topics in a large number of documents. For example, if you had 100,000 tweets mentioning a sports team, you may want to aggregate terms to learn about the subjects in the tweets. In fact, we will do this later in the chapter! R will interpret terms with different capitalization as distinct, making aggregation difficult. The `tolower()` function solves this. Let’s now clean up this tweet a bit more using `tolower()`

tweet <- tolower(tweet)

tweet

[1] " y’all i’m really about to dance on the cleveland cavaliers court starting next week "

Examining the resulting tweet object you now see that “Cleveland” has been made lowercase among other changes. This step is used to improve word frequency analysis especially in channels where users take shortcuts and may not use proper grammar. It may be the case in the collection of 100,000 tweets one user wrote “Cleveland cavaliers” and another “Cleveland Cavaliers”. In that case, the `tolower()` functions unifies the word tokens so that “cavaliers” is correctly counted twice.

Lastly, you should notice that we have extra spacing in our tweet because of the global substitution applied earlier. After you have performed many string manipulations like substitutions, splits, accounting for unknown characters i.e. emojis, and dealing with contractions it is a good idea to remove the extra space. NLP is computationally intensive so removing extraneous bytes is a good idea. Further, an extra blank space can be considered a term in some methods which will hinder your analysis. Using the `trimws(object)` function will trim the white space from both, left or right sides of your text as shown below.

tweet <- trimws(tweet, which = 'both')

tweet

[1] "y’all i’m really about to dance on the cleveland cavaliers court starting next week"

Although not quite string manipulation it is often useful to locate patterns, such as words, within a body of text. The base functions grep(pattern, object, ignore.case = T) and grepl(pattern, object, ignore.case = T) are valuable aides for locating the presence of words. Each is passed a regular expression pattern, then the object with additional parameters. The difference between `grep` and `grepl` is that the former will tell you the location and the latter will provide a logical return if the pattern is found, hence the “L” in the function. Consider the following small collection of two tweets.

tweets <- c('#Cavs Collin Sexton matched a career-high with 31 points. He’s now averaging more than 20 per game','#Mavericks guard Seth Curry is now shooting .442 from three-point range in his career, moving him into 2nd place in @NBA history')

tweets

[1] "#Cavs Collin Sexton matched a career-high with 31 points. He’s now averaging more than 20 per game"

[2] "#Mavericks guard Seth Curry is now shooting .442 from three-point range in his career, moving him into 2nd place in @NBA history"

Let’s find the presence of the term “cavs” and “point” using `grep`. After running the code you will see that “cavs” is in the first tweet and the function returns a 1. It was recognized because of the third parameter which ignores the case. In the second function call the return was 1 and 2. This is because the term “point” exists in the first and second tweet. As you will soon find out, the integer returns can be used to quickly find the documents with specific terms among thousands of documents.

grep('cavs', tweets, ignore.case = T)

[1] 1

grep('point', tweets)

[1] 1 2

Contrast these functions with the code below which does not add the third parameter. Although the term “Mavericks” is contained in the second tweet, it is not identified because the search pattern is “mavericks”. This is another example of why `tolower` can be useful.

grep('mavericks', tweets)

integer(0)

Next let’s use `grepl` to identify the presence of a text string. Although the input of the function is the same as `grep` the additional “L” means the function will return a Boolean (TRUE or FALSE) result for every document searched. Whereas the previous commands provided a 1, 1 and 2 and a 0 respectively the `grepl` will provide two outs as shown below.

grepl('mavericks', tweets, ignore.case = T)

[1] FALSE TRUE

Many of the functions used thus far are performing a match based on “regular expressions” as you get into more complex situations such as searching for “one term OR another” or “one term AND another” you will need to explore the usefulness of regular expressions more in depth.

Lastly, keep in mind the functions here are basic string manipulation functions that you will utilize throughout the chapter and in your own analysis. For the most part we will make these straightforward and easily understood. However, there are more which may need to be applied depending on your use case. Thus, it is a great idea to research regular expressions, and the most popular packages `stringi` and `stringr` for this type of manipulation.

# Basic Text Processing Workflow

Now that you have a basic understanding of string manipulations it is time to explore a basic framework for tackling a natural language processing analysis. Due to the difficulty associated NLP stemming from the diversity of language, the first step is to clearly define the aim of the project. It may be high level such as explore and define methods for gauging social media fan engagement to something more precise such as analyze scouting reports alongside other athletic data to model on field performance. Without a problem definition you will be doing “curiosity analysis” with no direction. Given the challenges of natural language analytics, strive to be as succinct in possible and be willing to iterate and adjust along the way. Once you have a problem reasonably defined this should lead you towards a channel and specific pieces of text for analysis. This may be online reviews, contracts or something else but is rarely the entire Internet or some vast collection of unrelated and diverse documents. Next you need to preprocess the documents which entails organization and feature extraction. A simple example would be collecting 10000 tweets mentioning a player. Once organized into a *corpus*, or collection of related documents, you can extract features such as sentiment analysis from those documents. The features or values extracted varying depending on the type of analysis you expect to perform. It could be as simple as counting the occurrence of a term or as complex as creating a modeling matrix for use in a deep neural net model to classify documents. In any case, once the appropriate features have been extracted from the documents, you then run the analysis and finally seek to address the problem definition. Once again, addressing the problem statement may be as simple as providing a visual like a word cloud or as complex as using the output of the analysis in a customer propensity machine learning model.

Corpus: a collection of related documents to be analyzed. Multiple collections are called a corpora.

In review, the basic steps of a NLP project are outlined below.

1. Problem Definition
2. Identify Text Sources
3. Preprocessing
4. Analytics
5. Reach Insight or Recommendation

In this chapter we are focusing on fan engagement in social media. The methods can be applied to other types of documents yet are not an exhaustive set of approaches. However, the methods used in the chapter are popular useful and satisfying to explore.

Thus in the chapter our steps are as follows.

1. Problem Definition – Understand fan engagement for various NBA teams using multiple common methods and marquee players.
2. Identify Text Sources – We will focus on a collection of tweets collected daily throughout the 2019-2020 NBA season.
3. Preprocessing & Feature Extraction - String manipulation and organization into a document term matrix to get term frequency
4. Analytics – Build visualizations such as bar charts, word clouds and pyramid plots. Perform word associations and sentiment analysis.
5. Reach Insight or Recommendation – Within the provided text identify the most discussed teams, terms, and corresponding sentiment representing the twitter dialog of fans, and sports professionals.

# Basic Visualizations

Now that we have an intended goal of understanding fan engagement in the NBA let’s load the `tm` or “text mining” library to perform more robust string manipulation and organization. In addition, data manipulation packages `slam`, `dplyr` and `qdapRegex` are loaded below. Lastly, `ggplot2`, `ggthemes` , `plotrix` and `wordcloud` are loaded to create some insightful visuals from the manipulated strings.

library(tm)

library(slam)

library(dplyr)

library(qdapRegex)

library(ggplot2)

library(ggthemes)

library(plotrix)

library(wordcloud)

When performing NLP tasks, it is often a good idea to ensure strings are not considered factors. Factors are distinct levels such as “small”, “medium” and “large”. The levels are text but not quite in the manner of free form natural language. The environment option code below ensures strings are not factors and turns off scientific notation.

options(stringsAsFactors = F, scipen = 999)

Let’s download our data directly from an open repository. The repository (https://github.com/kwartler/NLP\_SportsAnalytics) has multiple monthly files in this format so you can expand and change the analysis as you become more familiar with concepts throughout this chapter. In fact, as you embark on your own NLP analysis, you would need to identify your sources, instead of merely downloading the curated data set below. You could gauge fan engagement aggregated from multiple social channels or comparatively by social channel. For simplicity the rest of the chapter uses corpora only from twitter specifically from the 2019-2020 NBA season.

gitPage <- 'https://github.com/kwartler/NLP\_SportsAnalytics/blob/master/data/raw/A\_Oct2019.csv?raw=true'

tweets <- read.csv(url(gitPage))

*If your R system struggles with such a large data set, if you are testing the code or if your Internet connection is slow, you may declare a smaller number of rows using* tweets <- read.csv(url(gitPage),nrows = 1000) *or use one of the “tiny” data sets also listed in the repository.*

With the `tweets` object in your environment, you could perform normal exploratory data analysis (EDA) but this chapter is focused on NLP principles so the code jumps into string manipulation, organization and feature extraction as part of step 3.

Since the aim of our analysis is not to explore frequently shared links the code below uses both

`rm\_url(text) ` and `rm\_twitter\_url(text)` from the `qdapRegex` regular expression library. The code applies both because twitter sometimes uses short URLs which means the pattern matching may change. Further the higher level `qdap` package usually is loaded instead but doing so may be difficult because installing `qdap` on non-Windows computers is a hassle due to Java issues. However, the `qdapRegex` library should install on Ubuntu, Apple and Windows machines with ease.

tweets$text <- rm\_url(tweets$text)

tweets$text <- rm\_twitter\_url(tweets$text)

Next, to streamline string clean up among multiple data files, it is often helpful to create a custom function wrapping the string manipulations you intend to perform rather than repeat code over and over. The next code chunk creates a function call `customClean`. The function accepts a corpus object, then another parameter called “stops” which is a vector of “stop words” with a default value “SMART.” This vector of stop words will be removed from the text.

The default “SMART” stop word vector has 571 terms such as “the” and “why.” The list was first documented in an information retrieval system made at MIT. There are multiple basic stop word lists and you should adjust them according to your analysis. In fact, adjusting the stop word list is one reason NLP is an iterative process.

Stop words: a vector of terms that are frequent yet provide little to no information such as “and”, “the” and “is.” Stop words are customized according to the channel and messenger.

Now that you understand the inputs to the function, let’s move to the functions that will clean the text. The `tm` library works with a new object class called “corpus.” This corpus has functions mapped to it, hence the `tm\_map(corpus)` function calls below. Further, the `tm` library has additional string manipulation functions built-in such as “removeWords” and “removePunctuation.” However, the additional use of “content\_transformer” allows a function outside of the `tm` library to be applied to the object class “corpus.” As a result, a corpus enters the function, is made lowercase using the base R function along with `content\_transformer` before moving to tm’s `removeWords` function which also needs the terms to remove in the `stops` vector. This process continues throughout the function before returning the modified corpus with all the string modified accordingly.

customClean <- function(corpus, stops = stopwords('SMART')){

corpus <- tm\_map(corpus, content\_transformer(tolower))

corpus <- tm\_map(corpus, removePunctuation) #chk tomorrow

corpus <- tm\_map(corpus, removeWords, stops)

corpus <- tm\_map(corpus, removePunctuation)

corpus <- tm\_map(corpus, removeNumbers)

corpus <- tm\_map(corpus, content\_transformer(trimws))

return(corpus)

}

A common mistake among new data scientists, is not paying attention to the order of the internal functions in the `customClean` code. If you were to use `trimws` then begin to remove punctuation and numbers more white space is created which could impact results. Within the custom function, special care must be given to the order and specific preprocessing steps depending on the type of analysis.

To apply the `customClean` function the `tm` package requires the source to be defined. Depending on your analysis documents may be in a single vector, a data frame with meta-data columns like author or a file directory. In this example the documents are in a data frame so you must declare the `DataframeSource(data frame of text & metadata)` function. In the next code chunk, this is nested inside the `VCorpus(a declared source object)` function. A *volatile* corpus is a collection of documents held in active memory. It is volatile because if your system shuts down or you close R without saving your environment the corpus is lost. Thus, any change you make only affects the R object not the original documents on disk. This is the most popular method since you do not risk forever altering your original document collection by mistake. Another common mistake when using `VCorpus` with `DataframeSource` is that the first column must be named `doc\_id` and the second `text`. All other columns in the data frame will be considered meta data relevant to the individual document.

textCorp <- VCorpus(DataframeSource(tweets))

Now when you call the `textCorp` object you no longer have a data frame. Instead it is a corpus object class, which is actually a list of the text and meta data. Calling the object in console shows you have meta-data and content.

> textCorp

<<VCorpus>>

Metadata: corpus specific: 0, document level (indexed): 2

Content: documents: 453875

Although the top-level object is vague, the data is still present. Calling `meta(a document)` on the object with a single bracket index will return the other columns now stored as meta information.

> meta(textCorp[1])

created team

1 2019-10-01 21:19:16 Atlanta Hawks

Similarly calling `content(a document)` with either single or double bracket indexing will provide the content information. Single brackets will return high level information calculated by the function call while using double brackets returns the text itself. This code examines the content of the first tweet.

> content(textCorp[1])

[[1]]

<<PlainTextDocument>>

Metadata: 7

Content: chars: 86

> content(textCorp[[1]])

[1] "Ole Miss: 15 Braves: 0 Falcons: 0 Hawks: 0 I enjoy suffering with my Atlanta friends "

Before passing the corpus into the custom string processing function you should always adjust your stop words to fit your analysis. In this analysis it is likely the documents contain the team names and cities. Thus, it may be a good idea to remove them which is done using some string manipulation and the original meta data column in `tweets$team` . The `uniqueTerms` object is made by first identifying the unique terms in the column from the original tweet CSV. This is nested in `tolower` because the custom function makes text lowercase *then* removes stop words. Finally, this is passed to both `strsplit` and `unlist(a list object)` to get a clean vector of individual terms to be appended to the original stop words. This is because our string manipulation at this point will be working with single unique words as opposed to two or more word combinations. Additionally words such as “game” and “basketball” are appended to the `uniqueTerms` object. Finally in the `stops` object, the original stop words and the customized tokens are combined as shown in the `tail(R Object)` function call.

uniqueTerms <- tolower(unique(tweets$team))

uniqueTerms <- unlist(strsplit(uniqueTerms, ' '))

uniqueTerms <- c(uniqueTerms, 'nba', 'game', 'basketball', 'team','amp', 'preseason', 'extension', 'season')

stops <- c(stopwords('SMART'), uniqueTerms)

tail(stops, 100)

Now that the object is an official corpus you pass it into the custom function to perform the string manipulations along with the newly improved stop words vector. The result of the function is still a corpus but the text has been altered significantly. When dealing with a large amount of text, this step may take some time or even crash computers with smaller amounts of RAM. There are two remedies other than getting another computer or server with more RAM. First you can run this code with a smaller number of documents to ensure it functions and second run this code on chunks of documents, such as in sections by a single date. However, once complete it is a good idea to save this object as an RDS object to avoid having to repeat the mundane cleaning task. The commented code below shows both how to save and read in RDS file types which is an optimized R file extension.

textCorp <- customClean(textCorp, stops)

#saveRDS(textCorp, #'~/Documents/nbaTweets/uniqueTweets/clean\_A\_Oct2019.rds')

#textCorp <- readRDS( #'~/Documents/nbaTweets/uniqueTweets/clean\_A\_Oct2019.rds')

Table 2.0 reviews the first tweet after it has been “cleaned.” Overall you will see the terms “Atlanta” and “Hawks” have been removed. These were included with the stop words along with “I”, “with” and “my.” Do not worry if you want to perform an analysis such as sentiment while still keeping the team associated with the individual document (tweet) because the meta-data has been retained. It is merely removed from the text itself. In the table you should note the overall important information has been retained while the uninformative terms have been removed. Sometimes additional string manipulation may be needed based on the text Encoding but for this chapter’s purposes the text has been manipulated and now should be organized.

content(textCorp[[1]])

|  |  |
| --- | --- |
| Original | Preprocessed |
| "Ole Miss: 15 Braves: 0 Falcons: 0 Hawks: 0 I enjoy suffering with my Atlanta friends " | "ole miss braves falcons enjoy suffering friends" |

Table 2.0 Comparing the effect of ` customClean` on a single tweet.

Next the processed text will be organized into a document term matrix, DTM, for analysis. There are many parameters for the `DocumentTermMatrix(corpus)` function but the next code chunk accepts defaults. Once again this can be time consuming so saving the result is helpful.

textDTM <- DocumentTermMatrix(textCorp)

dim(textDTM)

#saveRDS(textDTM, #'~/Documents/nbaTweets/uniqueTweets/clean\_A\_Oct2019\_DTM.rds')

#textDTM <-#readRDS('~/Documents/nbaTweets/uniqueTweets/clean\_A\_Oct2019\_DTM.rds')

The result of this function is a sparse matrix. Each row is represented by a document and each column is the count of unique terms in the tweet. The matrix represents word usage among the thousands of tweets. Is it usually “sparse” because the entirety of the matrix is the lexical diversity of the entire collection yet a single document is only a subset. For example, the first tweet is made up of 7 terms after processing. However, among the 450,000+ tweets, 99000+ unique words are used. Thus, row one will have “1” listed in seven columns and “0” in the rest of the 99000! Some NLP examples use a “Term Document Matrix” instead. Rest assured this is the same information but the transposition of the DTM created here. A TDM has rows of terms and a column for each document. Table 3.0 shows a small portion of a document term matrix as an example. The table is merely a small section of 3 tweets among thousands and 3 terms among thousands.

|  |  |  |  |
| --- | --- | --- | --- |
|  | kare | kawhi | kcjhoop |
| 1179024230517805056 | 0 | 0 | 0 |
| 1179022022690496512 | 0 | 1 | 0 |
| 1179021874535243776 | 0 | 0 | 0 |

Table 3.0 A portion of the document term matrix showing the term kawhi occurred once in tweet ending in 512.

Before entering the next step the code below will create one more DTM for analysis. The previous DTM used columns of single terms, known in NLP as “unigram tokens.” This new DTM will tokenize to “bigrams” or two-word combinations for use later in the analysis. While bigram tokens can be more insightful than unigrams because phrases are more easily understood, the effect of increased tokenization is an ever-sparser matrix. This can make a bigram DTM harder to process computationally. While the previous example tweet was made of seven single word tokens the combinations of two words means the bigram version is longer. Multiply this by the many thousands of tweets and you will notice the dimensionality grows.

To perform bigram tokenization the following additional custom code is needed when constructing a DTM. This function accepts the text and using the `ngrams(words to be tokenized and splitting integer)` function will break up the columns into two word pairs. This function is then passed to the `DocumentTermMatrix` function as an extra parameter not present previously. The resulting bigram DTM has the same number of documents but now has 536000 two word combinations as columns.

bigramTokens <-function(x){

unlist(lapply(NLP::ngrams(words(x), 2), paste, collapse = " "),

use.names = FALSE)

}

bigramDTM <- DocumentTermMatrix(textCorp,

control=list(tokenize=bigramTokens))

dim(bigramDTM)

In most instances this amount of sparse information is not needed. Unless doing anomaly detection it is more likely your analysis would need focus on the most frequent terms. The `removeSparseTerms(DTM or DTM, decimal number)` function from the `tm` package can be applied to any DTM or TDM, including the unigram version made earlier. The numeric input is a percentage of cells in the matrix which are 0, or sparse, which are to be removed. This means 0.99 entails all terms (columns) are removed if 99% of the cells are zero. A column is retained if 1% of the cells contain a non-zero number. After running the code below, the number of frequent two-word pairs is less than 1000 so the matrix is now resized to 450000 rows by 850 columns making analysis much easier.

bigramDTM <- removeSparseTerms(bigramDTM, 0.999)

dim(bigramDTM)

## Word Frequency

Moving to step 4, the code transitions from these basic matrices to extract information and build visualizations. To begin let’s create a *document length matrix*, helping us understand average length of each tweet. While the output of the `DocumentTermMatrix ` function is a matrix, it is actually a lightweight sparse matrix class known as a “simple triplet matrix” so that the information with zeros is not actively held in memory. The `row\_sums(sparse matrix)` function is similar to the base R `rowSums` function but made to work on the DTM object class. The row sums are then organized alongside the document id and team meta data from the original objects.

wordsPerTweet <- row\_sums(textDTM, na.rm = T)

dlm <- data.frame(doc\_id = rownames(textDTM),

words\_per\_tweet = wordsPerTweet,

team = tweets$team)

head(dlm)

With the document length matrix made above, you can confirm the length of the first tweet is seven terms. More interestingly you can see the average length of tweets by teams. Average tweet length can be indicative of noteworthy information, like an active roster or news about the team but in large corpora average length of document often indicatives the author’s effort. An analog is an online review. The longer a review the more likely the author of the review is passionate about the service or product. In order to get the average terms by team the following code uses `aggregate(y ~x, data object, function)` passing in the formula “words\_per\_tweet” by “team”, within the “dlm” object then apply the “mean” function. The same function can be used to get a total count of tweets by team as shown in the second code line with different inputs. Finally the two tables are joined by team so that each row represents a team with average word and number of tweets.

fanEffort <- aggregate(words\_per\_tweet ~team, dlm, mean)

fanCount <- aggregate(doc\_id ~team, dlm, length)

fanEffort <- left\_join(fanEffort, fanCount, by ="team")

After constructing the `fanEffort` object the following plot is constructed with `ggplot`. The grammar of graphics library works by giving you control of the layers within a visualization. Here the fan effort data is passed in declaring the X and Y values. Then another layer of geometric points is added to construct the scatter plot. The size aesthetic is declared based on the count of the `doc\_id` variable calculated in the `fanCount` object. The last layers improve aesthetics but are optional. The resulting figure is shown in figure 1.0. Of course this data could be normalized by the total population in the local market but is outside the scope of the chapter.

ggplot(fanEffort, aes(x = reorder(team, - words\_per\_tweet), y = words\_per\_tweet)) +

geom\_point(aes(size=doc\_id)) + ggtitle("Words per Tweet") +

theme\_hc() +

theme(axis.title.x=element\_blank(), axis.title.y=element\_blank(),

axis.text.x = element\_text(angle = 90, vjust = 0.5, hjust=1))

A flock of birds

Description automatically generated

Figure 1.0 showing the difference in Oct 2019 fan tweet length & total number of tweets in the corpus.

Overall, it may be interesting to explore frequent terms among the entire corpus. In doing so, you may identify league wide themes of interest in the October 2019 corpus. Instead of row sums, now the code uses `col\_sums(sparse matrix)` to obtain the overall term frequency. This is organized into a *word frequency matrix*. Again the information is organized into a concise data frame. After applying column sums, the result is reordered with `order(vector of numbers)` with the parameter `decreasing=T`.

# Word Frequency Matrix

termUseInCorpus <- col\_sums(textDTM, na.rm = T)

wfm <- data.frame(term = names(termUseInCorpus),

frequency = termUseInCorpus)

wfm <- wfm[order(wfm$frequency, decreasing = T),]

head(wfm)

Reviewing the top 6 terms using `head(R object)` it is clear that the term “win” is important in the corpus. Interestingly, and maybe unexpectedly, the nation “China” appears frequently in the overall corpus. Alternatively, the same frequency review can be done at the team level. To do so, the previous steps would need to be repeated after “subset-ing” the original tweets data frame by the team column. In essence the steps are the same but repeated 30 times, once per team including cleaning and working with 30 DTMs.

Drilling down a level, publicists and public relations firms measure “share of voice” for brands and celebrities. This metric is tracked among others over time. The goal of “share of voice” is to understand how much of the overall dialog in a channel is dedicated to the product, brand or celebrity of interest. For this example, define the players of interest first.

players <- c('kobe bryant', 'lebron james', 'giannis antetokounmpo','kawhi leonard')

Don’t be intimidated by the next section of code with a “for loop.” A loop merely states to perform an action repeatedly, “for this number of times, perform these actions.” In our case the loop will perform tasks for Kobe Bryant, then Lebron James and so on. Within the loop the `grepl` function is searching for the pattern, a player’s name, within the original text. It also reformats the date to a more legible format. To begin the loop has a variable, “i” which is set to 1. This means `players[1]` is represented a `players[i]`. So `grepl` will search for `player[1]` which corresponds to “Kobe Bryant”. The next time through the loop “i” becomes 2 and now corresponds to “Lebron James.”

playerShare <- list()

for(i in 1:length(players)){

print(paste('working on', players[i]))

x <- grepl(players[i], tweets$text, ignore.case = T)

y <- as.POSIXct(tweets$created, format = "%Y-%m-%d")

dailyShare <- data.frame(player = players[i], doc\_id = tweets$doc\_id,

playerMention = x, date = y)

nam <- make.names(players[i])

playerShare[[nam]] <- dailyShare

}

Examining the resulting list for Kobe Bryant is simple with `head` and shown in table 4.0

head(playerShare$kobe.bryant)

|  |  |  |  |
| --- | --- | --- | --- |
| Player | Doc\_id | playerMention | date |
| “kobe bryant” | 1179143769574297600 | FALSE | 2019-10-01 |
| “kobe bryant” | 1179143120375762944 | FALSE | 2019-10-01 |
| “kobe bryant” | 1179142532191723520 | FALSE | 2019-10-01 |
| “kobe bryant” | 1179141961753792512 | FALSE | 2019-10-01 |
| “kobe bryant” | 1179141924218920960 | FALSE | 2019-10-01 |
| “kobe bryant” | 1179141590612353024 | FALSE | 2019-10-01 |

Table 4.0 the result of calling `head` showing that among the first six tweets on October 1, Kobe Bryant was not mentioned.

The result of the loop is a list with four elements one for each of the players. The following code applies `rbind(data vectors)` within `do.call(function to construct, list)` to the list. As a result, the list elements are row bound to unify the results into a single data frame. Once again, the data is aggregated. Specifically, the `playerMention` data is grouped by player and by date. Once grouped the True or False values are summed. As a Boolean, R will recognize the True as 1 and False as 0. In the end, each player will have a sum of mentions for each day in the data frame which makes a compelling visual to understand the share of voice among the 4 players.

mentionsByDate <- do.call(rbind, playerShare)

mentionsByDate <- aggregate(playerMention~player+date,

mentionsByDate, sum)

The timeline results of mentions are illustrated in figure 2.0. There is a clear spike in mentions of Lebron James compared to the other players in the study. The simple `ggplot` call sets up the X as `date`, the Y as `playerMention` value. The `linetype` is declared by player to differentiate the lines within a colorless context. However, the lines could be changed by color and even declared using a color blind safe palette using the `viridis` library.

ggplot(mentionsByDate, aes(x = date,

y = playerMention,

linetype = player)) +

geom\_line() + theme\_hc()

.Notice they are bigram tokens first and last name. As a result, the share of voice measures will rely on the bigram DTM created earlier.

## Word Associations & Networks

## Pyramid Plot

# Word Clouds

## Simple

## Commonality

## Comparison Cloud

# Sentiment Analysis

## Lexicons

## Polarity

## Tidy Joins

# Sources of Text

## API’s

## Webscraping

# Dashboarding

## Flexdashboard

# Exercises

1. Given some tweet, use grep, grepl, make lower case and gsub
2. Count how often the team New York Knicks appear in a corpus of 500 tweets. NY Knicks? Knicks? Use gsub then calculate or an “or” statement in grepl.
3. Define the NLP term Corpus
4. Define a “stopword” and why you would need to adjust them in an anlaysis?
5. In a document term matrix that do rows of the matrix represent? In most DTM or TDM the matrix is very sparse, why would that be?