Tagging method for text mining

Duc Vu Trung

May 10, 2018

Contents

Preface **2**

1 General method **3**

2 Cleaning **6**
2.1 General cleaning . . . 7
2.1.1 Use `mgsub()` for cleaning . . . 7
2.1.2 Replace contraction . . . 7
2.1.3 More on abbreviation . . . 8
2.1.4 Some unnecessary cleanings: . . . 8
2.1.5 Should we change the case? . . . 9
2.2 Some specific case of raw-text cleaning . . . 10
2.2.1 Spell checking . . . 10
2.2.2 Deal with compound words, phrasal verb and emoticon . . . 12

3 Tokenization and tagging **15**
3.1 Sentence tokenization: . . . 15
3.2 Word tokenization . . . 19
3.3 Tagging . . . 20
3.4 The advantages of tagging method: . . . 22
3.5 Deal with negations . . . 22
3.6 Deal with phrasal verbs/ compound words . . . 24
3.7 Deal with emoticon . . . 24
3.8 Can we add bigram to the word-level dataframe? . . . 25
3.9 Steming . . . 26
3.10 Part of speech (POS) . . . 27
3.11 Get names (POS continued. . .) . . . 29

4 Analyzing text **31**
4.1 Some useful functions to explore data . . . 31
4.1.1 View the data . . . 31
4.1.2 See the categorical columns . . . 32
4.2 Analyze text from word-level dataframe . . . 34
4.2.1 Example: . . . 34
4.2.2 Analyze without coding . . . 36
4.2.3 The most specific terms: tf-idf: . . . 37
4.3 Analyze pair of words . . . 39

Last words **45**

Reference **46**

Preface

This article is written based on the ideas of `Tidy Text Mining with R book`. Specifically thanks to `Hadley Wickham` (tidyverse), `Julia Silge`, `David Robinson` (tidytext, widyr) and `Tyler Rinker` (qdap,sentimentr) and other R authors for their wonderful packages.

The article focuses on dealing with some specific problems in text mining like: **Spell checking**, **negation**, **phrasal verb** and the usage of **tagging** in sentiment analysis.It discusses pros and cons of multiple tools in text analysis. It broadens the ideas of sentiment analysis to (hopefully) make it more effective and meaningful. Visualization methods in this articles are **minimal**, as it is perfectly covered in `Tidy Text Mining with R book`

Throughout the article, the following packages are used (There might be other packages for some specific problems):

```r
library(tidyverse)
library(zoo)
library(tidytext)
library(qdap)
library(qdapDictionaries)
library(sentimentr)
library(hunspell)
library(SnowballC)
library(widyr)
library(openNLP)
library(openNLPdata)
library(openNLPmodels.en)
library(NLP)
options(stringsAsFactors = F)
```

An example is worth a thousand words - Anonymous

My article is flooded with very simple and sometimes silly examples. I believe that in order to fully understand a method, it is better if we can have the whole pictures of what the method actually does. Small examples lead to small results which we can easily navigate ourselves. In part `4.1 Some useful function to explore data`, I will introduce some of my personal function that can help you understand any table(with or without text) effectively. It should be noted that dealing with words are never easy and most of the methods, simple or compilcated, can never guarantee a 100% accuracy. In each methods, I will discuss the pros/cons and point out the situation in which the method is not accurate. The analysis is not fixed, but must go through a `trial-and-error` process in order to make it useful.

1 General method

Text mining process will be divided into 4 steps:

- **Cleaning** the raw text.
- **Tokenization** into different levels: paragraph, sentences, bigram, words.
- **Taggings/ join** : It will tags each levels with different ID, sentiments, etc for further analysis. With tagging, the number of rows after tokenization will remain (eg: use left_join) while with joining, the original dataframe will be reduced.
- **Analyze**: Use statistical tools to extract helpful information.

All of the steps will revolve around the workhorse function `unnest_tokens()` in `tidytext` library. Tokenization is a process which enables the users to split text into **tokens** (sentence,word). The whole idea in this article is to divide original text into different levels and **tag** them with different types of values, which enable easy further analysis.

We begin with a basic examples:

```
x <- "This isn't a good day. I feel     terrible, Mr. John :( .
Hope that I will get over this bad situation"

# cleaning
text_cleaned <- x %>%
  replace_abbreviation() %>%
  replace_contraction() %>%
  str_replace_all(' +',' ')
text_cleaned
```

[1] "This is not a good day. I feel terrible, Mister John :(. Hope that I will get over this bad situation"

```
# tokenization
word_token <- text_cleaned %>% as.tibble() %>%
  unnest_tokens(word,value)
word_token
```

```
## # A tibble: 20 x 1
##    word
##    <chr>
##  1 this
##  2 is
##  3 not
##  4 a
##  5 good
##  6 day
##  7 i
##  8 feel
##  9 terrible
## 10 mister
## 11 john
## 12 hope
## 13 that
## 14 i
## 15 will
## 16 get
## 17 over
## 18 this
## 19 bad
## 20 situation
```

```r
# Join method
word_join <- word_token %>%
  inner_join(get_sentiments('bing'))
word_join
```

```
## # A tibble: 3 x 2
##   word     sentiment
##   <chr>    <chr>
## 1 good     positive
## 2 terrible negative
## 3 bad      negative
```

```r
# Tagging method:
word_tag <- word_token %>%
  left_join(get_sentiments('bing')) %>%
  mutate(bing = case_when(sentiment =='positive' ~1,
                  sentiment =='negative' ~ -1,
                  is.na(sentiment) ~0))
word_tag
```

```
## # A tibble: 20 x 3
##    word      sentiment  bing
##    <chr>     <chr>     <dbl>
##  1 this      <NA>         0.
##  2 is        <NA>         0.
##  3 not       <NA>         0.
##  4 a         <NA>         0.
##  5 good      positive     1.
##  6 day       <NA>         0.
##  7 i         <NA>         0.
##  8 feel      <NA>         0.
##  9 terrible  negative    -1.
## 10 mister    <NA>         0.
## 11 john      <NA>         0.
## 12 hope      <NA>         0.
## 13 that      <NA>         0.
## 14 i         <NA>         0.
## 15 will      <NA>         0.
## 16 get       <NA>         0.
## 17 over      <NA>         0.
## 18 this      <NA>         0.
## 19 bad       negative    -1.
## 20 situation <NA>         0.
```

```r
# Analyze
word_join %>%
  count(sentiment) %>%
  spread(sentiment,n) %>%
  mutate(overall_sentiment = positive- negative)
```

```
## # A tibble: 1 x 3
##   negative positive overall_sentiment
##      <int>    <int>             <int>
## 1        2        1                -1
```

```r
word_tag %>%
  summarize(sum(bing))
```

```
## # A tibble: 1 x 1
##   `sum(bing)`
##         <dbl>
## 1         -1.
```

As the overall sentiment score is -1, then the sentiment of the above text is negative. It is true, isn't it?

2 method are basically the same, except for the fact that, with `join` method, we will reduce the dataframe by `inner_join()`, while with `tagging` method, we will keep the number of rows. Also, one of the common technique used in `tagging` method is to convert scores to `-1,0 and 1` which indicate the positive, neutral and negative. Not only for `bing` score, sentiment scores used by other method, eg: afinn or qdap_score (using `sentimentr` library), we will convert to `1,0,-1` too.

You can see some factors which are totally ignored in the analysis. Firstly, the 1st sentence is in fact, a negative one, so `"good"` should be tagged with **negative sentiments**, not **positive**. Secondly, the emoticon `<:(>` is removed in the tokenization process and never be taken into consideration. Lastly, the word `"get over"` is a **positive phrasal verb** but it is ignored, too. In this article, I will discuss the way to deal with all of them! '

2 Cleaning

Clean or not clean, that is the question - Anonymous

We start with an example:

```r
# Load the library specified in preface
dat1 <- as.tibble("I saw Ms.Julia yesterday. She is quite gorgeous")
dat1 %>% unnest_tokens(sentence,value,token = 'sentences')
```

```
## # A tibble: 3 x 1
##   sentence
##   <chr>
## 1 i saw ms.
## 2 julia yesterday.
## 3 she is quite gorgeous
```

As `unnest_tokens()` use `<.>` as the seperator to split, the sentence tokenization above is incorrect. A little cleaning would help:

```r
# Load the library specified in preface
dat1 <- tibble("I saw Ms.Julia yesterday. She is quite gorgeous")
dat1 %>% replace_abbreviation() %>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 2 x 1
##   sentence
##   <chr>
## 1 i saw miss julia yesterday.
## 2 she is quite gorgeous
```

It looks better after the cleaning!
However, cleaning sometimes can make garbages! Let's see other examples:

```r
# Load the library specified in preface
# Rekless cleaning:
dat1 <- tibble("I live in rooms.104 .It is very well-decorated")
dat1 %>% replace_abbreviation() %>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 2 x 1
##   sentence
##   <chr>
## 1 i live in roo miss 104 .
## 2 it is very well-decorated
```

```r
# Cleaning with a litle care:
dat1 <- tibble("I live in rooms.104 .It is very well-decorated")
dat1 %>% replace_abbreviation(ignore.case = F) %>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 2 x 1
##   sentence
##   <chr>
## 1 i live in rooms.104 .
## 2 it is very well-decorated
```

We can see from above example that sometimes, cleaning will produce **unexpected results** and It is unlikely to check all results(It is against the rule of programming!). Remember that, after cleaning, we will have 2 other processes before any analysis (tokenization and tagging). Hence, we must answer following question before any thoughts of cleaning:

- Will the results of cleaning **distort** or **help** the **tokenization** and **analysis**(eg: sentiment analysis)?
- Will the results of cleaning can be _**overlapped/easily dealt with** later(eg: tokenization will remove leading and trailing spaces so there' s no need to clean leading and trailing spaces).
  We will go into detail to see some of the common cleaning and its pros/cons.

2.1 General cleaning

2.1.1 Use mgsub() for cleaning

qdap has many wonderful technique for cleaing, but it can not offer an 1-for-all solutions.Therefore, We should(must) personalize our cleaing to meet certain needs. We will continue the abbreviation cleaning above:

```
# qdap abbreviation list:
qdapDictionaries::abbreviations
```

```
##        abv     rep
## 1      Mr.  Mister
## 2     Mrs.  Misses
## 3      Ms.    Miss
## 4     .com dot com
## 5     www. www dot
## 6     i.e.      ie
## 7     A.D.      AD
## 8     B.C.      BC
## 9     A.M.      AM
## 10    P.M.      PM
## 11 et al.   et al
## 12     Jr.  Junior
## 13     Dr.  Doctor
## 14     Sr.  Senior
```

```
# We want to add other words,eg : A.S.A.P ( as soon as possible):
# Create a modified table of abbr
abbr_mdf <- qdapDictionaries::abbreviations %>% as.tibble() %>%
  bind_rows(data.frame(abv = c('A.S.A.P'),rep =c('as soon as possible')))
# Use mgsub()
" I will see you A.S.A.P,before 7 P.M.  , Ms. Julia " %>%
  mgsub(abbr_mdf$abv,abbr_mdf$rep,text.var = .)
```

```
## [1] "I will see you as soon as possible,before 7 PM , Miss Julia"
```

General steps for cleaning are:

- Start with a pre-defined tables (qdapDictionaries library is a good source!)
- Add (using rbind() with other dataframe) or remove (using filter())
- Use mgsub() for multiple replacement!

2.1.2 Replace contraction

Contraction is very important in raw text cleaning as there are many negations like "didn't","don't". Ignoring it will affect the sentiment analysis later. Replace word like "don't" with "do not" will help tokenization and analysis later.

```r
"I don't like her. He doesn't like here,too" %>%
  replace_contraction() %>% as.tibble() %>%
  unnest_tokens(word,value)
```

```
## # A tibble: 11 x 1
##    word
##    <chr>
##  1 i
##  2 do
##  3 not
##  4 like
##  5 her
##  6 he
##  7 does
##  8 not
##  9 like
## 10 here
## 11 too
```

We can see that both `"don't"` and `"doesn't"` will return only `"not"`, hence it is easier for sentiment analysis later. (I will discuss how to **shift the sentiment** of word from **positive to negative** in the next chapter)

2.1.3 More on abbreviation

As we see in the above example, abbreviation with the `<.>` will distort the sentence tokenization. Therefore, it should be replaced.
However,in order to lower the chance of unexpected behavior, (remember the `<rooms. 101>` above?), we should only replace words with upper case by specify `ignore.case = F`
As `mgsub()` doesn't have the args: `ignore.case`,if we want to add more words, we simply `replace_abbreviation()` one time, then create an other `abbr_mdf` dataset later and do the substitution again. See the examples below:

```r
# Load the library specified in preface
" I will see you A.S.A.P,before 7 p.m. " %>%
  replace_abbreviation() %>%
  mgsub(abbr_mdf$abv,abbr_mdf$rep,.)
```

```
## [1] "I will see you as soon as possible,before 7 PM."
```

2.1.4 Some unnecessary cleanings:

- **Remove number** is unnecesary, eg: `"123"` will turn to `"one two three"` which, after tokenization will become `"one"`,`"two"`,`"three"` and very distorting. If we find the number is totally unnecessary, then simply use `filter()`
- **Word stemming** sounds really fancy, but it seems to do more harm than good. Let's see an example:

```r
"he is the only ones who work on Sunday " %>%
  as.tibble() %>%
  unnest_tokens(word,value) %>%
  mutate(word = SnowballC::wordStem(word) )
```

```
## # A tibble: 9 x 1
##   word
##   <chr>
## 1 he
## 2 i
## 3 the
## 4 onli
## 5 on
## 6 who
## 7 work
## 8 on
## 9 sundai
```

You can see how disturbing the stemming is!
But the question is, if we do not stem the words, will it affect sentiment analysis ? You can check `get_sentiment('bing')` for the some words, eg: `"love"`:

```
get_sentiments('bing') %>%
  filter(str_detect(word,'love'))
```

```
## # A tibble: 9 x 2
##   word       sentiment
##   <chr>      <chr>
## 1 beloved    positive
## 2 love       positive
## 3 loved      positive
## 4 loveless   negative
## 5 loveliness positive
## 6 lovelorn   negative
## 7 lovely     positive
## 8 lover      positive
## 9 loves      positive
```

It seems that `bing` (the most common sentiment dataset) provide **different tenses and forms** of a word that we don't need to do the word stemming.

- **Space** and **escaped character**: **trimming and leading** space will be **erased** through the unnest tokens, so it will be necessary, however for the **space-in-the-middle**, in order to provide for the phrasal verb (which I will discuss later), we should remove the middle space. Escaped character will be removed in tokeniztion, too. Below are the codes to remove extra space:

```
# Remove extra space
x <- "It is a lot of      fun. I really       love   it"
x %>% str_replace_all(regex(' +'),' ')
```

```
## [1] "It is a lot of fun. I really love it"
```

- **Punctuation** : Never think of removing punctuation in raw text, otherwise you can never tokenize into sentences!

2.1.5 Should we change the case?

- **Case of words** is really important to identify the entity name, and, more importantly, the **tagger** for tokenization. See the example below:

```
"the weekend is a good time to relax. Sunday is the best" %>%
  str_to_lower() %>%
  as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 1 x 1
##   sentence
##   <chr>
## 1 the weekend is a good time to relax. sunday is the best
```

```
"the weekend is a good time to relax. Sunday is the best" %>%
  #str_to_lower() %>%
  as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 2 x 1
##   sentence
##   <chr>
## 1 the weekend is a good time to relax.
## 2 sunday is the best
```

We can see how changing the case can affect the tokenization process!

- However, in the tagging/joining step, it is really important to convert all words to `lower case` (or `upper case`) to enable join to a `look-up` table (We might need to convert the case of the look-up table, too!)

2.2 Some specific case of raw-text cleaning

2.2.1 Spell checking

Some words in English is complicated and vunerable to mistake, like `"embarrassing"`, have different ways of use, like `"advisor"` vs `"adviser"`, or simply the contraction/abbreviation, like `"mins"` for `"minutes"` and `"didnt"` vs `"didn't"`.

2.2.1.1 Detecting misspelled words

Misspelled words will have a great impacts if they are sentiment words.

Consider the following example:

```
library(sentimentr)
c("Your method is not helpfull at all.
  It is so embarassing to use it.") %>%
  sentiment()
```

```
##    element_id sentence_id word_count sentiment
## 1:          1           1          7         0
## 2:          1           2          7         0
```

```
# After spell check
c("Your method is not helpful at all.
  It is so embarrassing to use it.") %>%
  sentiment()
```

```
##    element_id sentence_id word_count  sentiment
## 1:          1           1          7 -0.2834734
## 2:          1           2          7 -0.2834734
```

`sentiment()` in `sentimentr` library is quite an advanced method, but it **fails** to recognized any sentiments here since the words that indicate sentiments are **misspelled** (It recognize the sentiments correctly after the spell-checking, though)

Let's go checking the spell in a (first) big example:

```r
# load package in preface
# Use hotel_reviews table above
library(sentimentr)
names_modified <- qdapDictionaries::NAMES %>% as_tibble() %>%
  mutate(word = str_to_lower(name)) # list of name in English
# Get the words which are misspelled
word_mispelled <- hotel_reviews %>%
  as.tibble() %>%
  unnest_tokens(word,text,to_lower = F) %>%
  filter(hunspell_check(word) == F) %>%
  mutate(word = str_to_lower(word)) %>%
  anti_join(parts_of_speech) %>%
  anti_join(names_modified) %>%
  count(word,sort = T)
```

```r
word_mispelled
```

```
## # A tibble: 6,582 x 2
##    word            n
##    <chr>       <int>
##  1 mins          168
##  2 bellagio      138
##  3 dont          117
##  4 didnt         111
##  5 wifi          109
##  6 advisor        87
##  7 definately     83
##  8 westin         80
##  9 tripadvisor    79
## 10 jacuzzi        74
## # ... with 6,572 more rows
```

In the example above, first we check the spell by `hunspell()` function. However, this function is quite rigorous on case-sensitivity so it might **overlook** some words, eg `"WiFi"` is recognized, but not `"wifi"` . Then we convert word to upper case and lookup to the word in English dictionary in `part_of_speech`. Finally, we remove some English names.

2.2.1.2 Correct misspelled words

a) **hunspell_suggest()** provide a very nice way to deal with misspelled words. For examples

```r
x <- "tihs si ym nmae. It isnt a niec nema. Quite an embarasing one. "
y <- "Hey, Jesie, Michaell, wifi doesnt work. It is really bad"

check_spell <- function(x){
  as.tibble(x) %>%
    unnest_tokens(word,value) %>%
    filter(hunspell_check(word)==F) %>%
    mutate(correct =  hunspell_suggest(word) %>% str_extract('\\b\\w{2,}\\b'))
}

# Note that we can never misspell a word with a single character,
# so I only extract words with 2 characters and more
check_spell(x)
```

```
## # A tibble: 8 x 2
##   word       correct
##   <chr>      <chr>
## 1 tihs       this
## 2 si         chi
## 3 ym         my
## 4 nmae       name
## 5 isnt       inst
## 6 niec       nice
## 7 nema       name
## 8 embarasing embarrassing
```

```
check_spell(y)
```

```
## # A tibble: 4 x 2
##   word     correct
##   <chr>    <chr>
## 1 jesie    Jessie
## 2 michaell Michael
## 3 wifi     wife
## 4 doesnt   doesn
```

We can see from the example, it fails to recognize some common mistake like `"wifi"`, `"realy"` and contraction `"doesnt"`.However, we already dealt with contraction previously, so it no longer pose any problems. Btw, you can use my `check_spell()` function to automatically check words! However, it should be noted that some non-English name work are extremely hard to check, making the process very slow. I did the same check on the `word_mispelled` table above and it takes more than 10 minutes! Here is some note on using `hunspell_suggest()` function:

- Consider contractions (must be cleanned earlier).
- Consider any abbreviations (eg: `"mins"`).
- Long words are better than short words as the **string distance** of correct and incorrect long words are lower.
- **Non-English** words are very **slow** to check spell. Be careful!

b) **Manually check spell:**

Check spelling is never an easy job, and if the machine can not deal with it, let the human do! However, we **don't need** to check **every word** for spelling, just choose the words with **high frequency** or words that we believe that can change the **sentiment analysis!**

Manually checking spell doesn't mean that you will repeat every time. Just form a common table of common misspelled words, and use `mgsub()` (see **2.1.1** for more details).For examples:

```
y <- "Hey, Jesie, Michaell, wifi doesnt work. It is really bad"
misspelled_correct <- data_frame(word = c('wifi','realy'),
                                 correction = c('WiFi','really'))
mgsub(misspelled_correct$word,misspelled_correct$correction,y)
```

```
## [1] "Hey, Jesie, Michaell, WiFi doesnt work. It is really bad"
```

Days after days, your **mispelled correction table** will become **big enough** so that **no manual check** is necessary!

2.2.2 Deal with compound words, phrasal verb and emoticon

An emoticon is worth than thousand words - Anonymous

I put 3 types of problems in the same category as in **raw-text cleaning**, we treat them the same way! (They will be quite different in the analysis process, though).

Let's start with an example:

```r
# Phrasal verbs 1
"He seems to get over" %>%  sentiment()
```

```
##    element_id sentence_id word_count sentiment
## 1:          1           1          5         0
```

```r
# Phrasal verbs 2
"I throw up when I see her" %>%  sentiment()
```

```
##    element_id sentence_id word_count sentiment
## 1:          1           1          7         0
```

```r
# Compound words 1
"She is narrow-minded "%>%  sentiment()
```

```
##    element_id sentence_id word_count sentiment
## 1:          1           1          4         0
```

```r
# Compound words 2
"It is a record-breaking performance" %>% sentiment()
```

```
##    element_id sentence_id word_count   sentiment
## 1:          1           1          6 -0.08164966
```

```r
# Emoticon 1
"It is the saddest moment I had in my life :))" %>%  sentiment()
```

```
##    element_id sentence_id word_count  sentiment
## 1:          1           1         10 -0.3162278
```

```r
# Emoticon 2
"I was so happy those day :(" %>%  sentiment()
```

```
##    element_id sentence_id word_count sentiment
## 1:          1           1          6 0.3061862
```

We can clearly see in case of **phrasal verb** and **compound words**, the single words here are *neutral*, or *negative* (`record` in `record-breaking`), but the **compounded** one has very clear sentiment so that the sentiments calculated is not correct!

In case of **emoticon**, if the users use the emo **correctly**, then the emoticon **alone** can decide the sentiment of the **whole sentence**!. The *first* one will become a **happy**, and the *second* one will become a **sad** statement, despite the word components **doesn't show** anythings like that!

The idea to deal with those problems are: turning them into **unigram** (single word). Of course, for **compound words/ phrasal verbs**, we can examine the **bigrams**. However, it means that we need to make **other columns** and go through **many steps** to **shift the sentiments** (which I will discuss later) and it is very clunky to do so. Working with unigram will make our analysis **universal**:

- **Deal with compound words/phrasal verbs:**

```r
library(stringr)
compound_word <- tibble(
  word =c('get over','throw up','narrow-minded','record-breaking'),
  replacement = c('get_over','throw_up','narrow_minded','record_breaking'),
  sentiment = c(1,-1,-1,1)
)
```

```r
# function to clean
compound_sentiment <- function(x){
    x  %>%  str_trim %>%
    mgsub(compound_word$word,compound_word$replacement,.) %>%
    as_tibble() %>%
    unnest_tokens(word,value) %>%
    inner_join(compound_word,by = c('word'='replacement'))
}
# test the function
compound_sentiment('he seems to get over')
compound_sentiment("It is a record-breaking performance")
```

```
## # A tibble: 1 x 3
##   word     word.y   sentiment
##   <chr>    <chr>        <dbl>
## 1 get_over get over        1.
## # A tibble: 1 x 3
##   word            word.y          sentiment
##   <chr>           <chr>               <dbl>
## 1 record_breaking record-breaking        1.
```

It seems to be much **better** now. It should be noted that phrasal verb use preposition words like `"up"`,`"down"` to form the verb, and they are all stop words in `stop_words` table. Therefore, cleaning the raw-text before any transformation is a good way to preserve the meaning.

- **Deal with emoticons:**

The idea is quite similar, turn emoji into **unigram**. However, in order to recognize it as emoji, we should add the prefix `emo_` . The `qdapDictionaries::emoticon` is a very nice dataset to deal with emoji

```r
x <- "It is the saddest moment I had in my life :)"
y <- "I was so happy those day :("
# Create a function to give proper explanation for emoji:
make_emo <- function(x){
str_split(x,pattern = ' ') %>%
  .[[1]] %>%
  str_c(collapse = '_') %>%
  str_c('emo_',.)
}
# Apply make_emo function to qdap table of emoticon.
# If you  want more emoticon, just add others!
emo_table <- qdapDictionaries::emoticon %>%
  as.tibble() %>%
  mutate(new_meaning = sapply(.$meaning,make_emo))
# Replace emoticon with the right unigrams
mgsub(emo_table$emoticon,emo_table$new_meaning,text.var = x)
```

```
## [1] "It is the saddest moment I had in my life emo_Smile"
```

```r
mgsub(emo_table$emoticon,emo_table$new_meaning,text.var = y)
```

```
## [1] "I was so happy those day emo_Sad"
```

Now **all** of the emos become **unigrams** and are very distinguisable!

3 Tokenization and tagging

3.1 Sentence tokenization:

I have used the `unnest_tokens()` function in throughout the article, but in this chapter, we will have a closer look to it.

There are 3 punctuations that indicate sentence ending, that is `<.>`, `<?>` and `<!>`. The rules for unnest_tokens() to split text into sentences are:

- <.> must followed by a capital characters. Or we can has the following pattern:
  $< . > +(optional space) + capital character.$

- <!> or <?> : always split, no matter what the case of the character that followe right after is

But it poses some problems:

- What if <.> are followed by a non-capital characters? Many times, the users forget to use the right case, so it is a very common problem.
- What if <.> is not the ending of sentence, but just part of an expression, eg: 'Ms. Julia'
- What if <!> is not the ending of sentence, but just end of expression in the <()>.

I will illustrate by the following examples:

```
#writer use non capital character ( number):
"This hotel is small. 20 rooms is not enough" %>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 1 x 1
##   sentence
##   <chr>
## 1 this hotel is small. 20 rooms is not enough
```

```
#writer use non capital character (lower case):
"This hotel is small. other one is better " %>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 1 x 1
##   sentence
##   <chr>
## 1 this hotel is small. other one is better
```

```
#writer use non capital character (symbol):
"This hotel is nice. @miss Julia: Do you love it?"%>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 1 x 1
##   sentence
##   <chr>
## 1 this hotel is nice. @miss julia: do you love it?
```

```
# Capital character is part of an expression, not sentence ending
"I want to meet Ms. Julia. Where is she?"%>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 3 x 1
##   sentence
##   <chr>
## 1 i want to meet ms.
## 2 julia.
## 3 where is she?
```

```
# <!)> and <?)> is not sentence ending,but end of expression in bracket <()>
"The hotel is beautiful (I love it!) and very big" %>% as.tibble() %>%
   unnest_tokens(sentence,value,'sentences')
```

```
## # A tibble: 2 x 1
##   sentence
##   <chr>
## 1 the hotel is beautiful (i love it!)
## 2 and very big
```

The problems with `<.>` are very commmon so we should not ignore it. The function `get_sentences()` can deal with most of the problems above, though!

The method of sentence splitting with `get_sentences()` are quite complicated. I will try to summarize it :

- <.> and <. . . > We need to consider the folloing pattern:
  $1stchar + < . > / < ... > + (optionalspace) + 2ndchar$
  It will split if they do not consider the above pattern as an expression, eg"Ms. Julia" is an expression, therefore it will not split
- <!)> only recognize it as end of expression in bracket,there are still need other sentence ending marks for sentence splitting

Let's see the following examples:

```
# It wil consider small. 20 rooms as a non-expression--> will split
"This hotel is small. 20 rooms is not enough" %>% as.tibble() %>%
  get_sentences()
```

```
##                      value element_id sentence_id
## 1:   This hotel is small.          1           1
## 2: 20 rooms is not enough          1           2
```

```
# It is the same for other cases. Just try it:
"This hotel is small. other one is better " %>% as.tibble() %>%
  get_sentences()
```

```
##                    value element_id sentence_id
## 1: This hotel is small.          1           1
## 2:  other one is better          1           2
```

```
# Another test with symbol:
"This hotel is nice. @miss Julia: Do you love it?"%>% as.tibble() %>%
  get_sentences()
```

```
##                            value element_id sentence_id
## 1:          This hotel is nice.          1           1
## 2: @miss Julia: Do you love it?          1           2
```

```
# However, it will split everytime it detect the <...>, though:


# It will not split at <!)> or <?)> pattern:
"The hotel is beautiful (I love it!) and very big" %>% as.tibble() %>%
   get_sentences()
```

```
##                                               value element_id sentence_id
## 1: The hotel is beautiful (I love it!) and very big          1           1
```

```
# If there is space between, eg : <! )> then it fails to recognize, though
"The hotel is beautiful (I love it! ) and very big" %>% as.tibble() %>%
   get_sentences()
```

```
##                                  value element_id sentence_id
## 1: The hotel is beautiful (I love it!          1           1
## 2:                      ) and very big          1           2
```

How can `get_sentences()` know that the pattern is an expression or non-expression. I did not read the source code, though, but from my observation (trial-and-error), it will check if the `1st char` in the formula is an abbreviation.

- If the `1st char` is an abbreviation, then the pattern is an expression and it will not split.
- If the `1st char` is not an abbreviation (most of the time), then the pattern is an expression and it will not split. We can check with all of the abbreviation words:

```
## see all abbreviations available
qdapDictionaries::abbreviations
```

```
##       abv     rep
## 1     Mr.  Mister
## 2    Mrs.  Misses
## 3     Ms.    Miss
## 4    .com dot com
## 5    www. www dot
## 6    i.e.      ie
## 7    A.D.      AD
## 8    B.C.      BC
## 9    A.M.      AM
## 10   P.M.      PM
## 11 et al.   et al
## 12    Jr.  Junior
## 13    Dr.  Doctor
## 14    Sr.  Senior
```

```
# except for <www.> , all other abbreviation will form an expression-->
# --> not split. It is not case-sensitive, though
"I want to meet dr. John tommorrow afternoon?" %>% as.tibble() %>%
  get_sentences
```

```
##                                            value element_id sentence_id
## 1: I want to meet dr. John tommorrow afternoon?          1           1
```

```
# <www.> behave very different, though
"go to my website: www.myspace and you will see something" %>% as.tibble() %>%
  get_sentences()
```

```
##                                  value element_id sentence_id
## 1:              go to my website: www.          1           1
## 2: myspace and you will see something          1           2
```

Now we can see that `get_sentences` has it owns problems:

- It always recognize `<...>` as sentence ending marks. We can deal with it with a litter bit cleaning!

```
# x has 1 sentence and y has 2 sentences:
x <- "This hotel has only 5 rooms...and it is... a small hotel "
y <- "This hotel has only 5 rooms... However, I love it"
# regular expression to detect <..> + (optional space) + capital char
```

```r
regex1 <- '[.]{3,}[\\s]*(?=[a-z])'
# replace and split x. No split--> correct
str_replace_all(x,regex1,' ') %>%
  get_sentences()
```

```
## [[1]]
## [1] "This hotel has only 5 rooms and it is a small hotel"
##
## attr(,"class")
## [1] "get_sentences"           "get_sentences_character"
## [3] "list"
```

```r
# replace and split y. Split --> correct
str_replace_all(y,regex1,' ') %>%
  get_sentences()
```

```
## [[1]]
## [1] "This hotel has only 5 rooms..." "However, I love it"
##
## attr(,"class")
## [1] "get_sentences"           "get_sentences_character"
## [3] "list"
```

- It fails to recognize the expression which is the name of website. Remember the "compound word" section, we will apply the same rules here: Transform website names into unigrams!. My solution use a regex that can detect websites which comes from: https://stackoverflow.com/questions/42618872/regex-for-website-or-url-validation

```r
# remember that in R we need to double escape with \\
# You might want to use a more complicated regex for unconventional website
url_regex <- str_c("\\b((https?|ftp|smtp):\\/\\/)?(www.)?[a-z0-9]+",
                   "\\.[a-z]+(\\/[a-zA-Z0-9#]+\\/?)*\\b")

x <- "Want more about Dr.john, go to www.john.com  or johnson.us for more detail"

# Get the website name
website <- x %>% str_extract_all(url_regex) %>%.[[1]] %>% as.tibble() %>%
  mutate(first_word = str_extract(value,'\\w+(?=\\.)') %>% str_to_lower()) %>%
  filter(!first_word %in% c('mr','mrs','ms','jr','dr','dr')) %>%
  mutate(web_name_mdf = str_replace_all(value,'[.]','_'))
website
```

```
## # A tibble: 2 x 3
##   value        first_word web_name_mdf
##   <chr>        <chr>      <chr>
## 1 www.john.com www        www_john_com
## 2 johnson.us   johnson    johnson_us
```

```r
# change the website name to unigram
x_cleanned <- x %>% mgsub(website$value,website$web_name_mdf,.)
# Before cleaning
x %>% get_sentences()
```

```
## [[1]]
## [1] "Want more about Dr.john, go to www.john."
## [2] "com  or johnson."
## [3] "us for more detail"
```

```
##
## attr(,"class")
## [1] "get_sentences"           "get_sentences_character"
## [3] "list"
```

```
# After cleaning
x_cleanned %>% get_sentences()
```

```
## [[1]]
## [1] "Want more about Dr.john, go to www_john_com or johnson_us for more detail"
##
## attr(,"class")
## [1] "get_sentences"           "get_sentences_character"
## [3] "list"
```

`get_sentences()` is slower than `unnest_tokens()`, though. In this articles, I prefer using `get_sentences()` as it is more accurate and it also work better with the *tagging* part that I will discuss later.

3.2 Word tokenization

Word tokenization is way easier to sentences. Only one things to recall: For some words/ pair of words, we can turn to unigrams by using `<_>` for easier analysis later (see previous chapters for more details!)

Also, word tokenization need to consider some symbols like `<#>` or `<$>` which are used a lot nowsaday (eg: twitters). I use 1 regex coming from the book `Tidy Text Mining with R` book.

```
# Word splitting complication comes from <.>, too:
"My name is Duc.Nice to meet you" %>% as.tibble() %>%
  unnest_tokens(word,value)
```

```
## # A tibble: 7 x 1
##   word
##   <chr>
## 1 my
## 2 name
## 3 is
## 4 duc.nice
## 5 to
## 6 meet
## 7 you
```

```
# However, we can do sentence split before word split
"My name is Duc.Nice to meet you" %>% as.tibble() %>%
  get_sentences()
```

```
##                 value element_id sentence_id
## 1:   My name is Duc.          1           1
## 2: Nice to meet you           1           2
```

```
# Also we might want to use regex for the symbols, too!
unnest_reg <- "([^A-Za-z_\\d#@']|'(?![A-Za-z_\\d#@]))"
"  : #analysis: @david: I will see you tomorrow" %>% as.tibble() %>%
  unnest_tokens(word,value,token = 'regex',pattern = unnest_reg)
```

```
## # A tibble: 7 x 1
##   word
##   <chr>
## 1 #analysis
## 2 @david
## 3 i
## 4 will
## 5 see
## 6 you
## 7 tomorrow
```

There are other tokens but they are not very important and very straightforward, too!
We completed tokenizing orginal text into 2 levels: sentences and words. Next, we will **tag** them with values!

3.3 Tagging

- **Steps of tagging:**
  - Tokenize text data to paragraph, sentence, and word(bigram) level
  - For each level, there will be ID number: para_ID,sentence_ID, word_ID–> **position tagging**
  - Add characteristics column,eg: **sentiment** (different method) at para_ID, sentence_ID, word_ID, **part of speech**, **negation**, **emoticon** without reducing the word- level dataframe —> **characteristics tagging**

We will start with a basic example of tagging:

```
# Load library in preface
# Use hotel_reviews dataset from sentimentr
# Use View() to see the results in Rstudio as it really big
hotel_sentence<- hotel_reviews %>%  head(1000) %>%
  mutate(text = replace_contraction(text)) %>%
  select(text) %>%
  get_sentences() %>%
  mutate(snt_sentiment = sentiment(.$text)$sentiment,
         sentence_id = row_number()) %>%
  select(element_id,sentence_id,sentence = text,snt_sentiment)
```

It should be noted that, in the example above, I use `get_sentences()` from `qdap`, not `unnest_tokens()` because the function `sentiment()` only work with sentences split by `qdap` function. The results of sentence tokenization are very similar between 2 methods, though.

```
# Qdap_score is at sentence level, bing and afinn is at word level
# For convenience in computation, sentiment score will be converted to 1 (positive)
# 0(no sentiment) and -1 (negative).
library(zoo)
hotel_word <- hotel_sentence %>%
  unnest_tokens(word,sentence) %>%
  mutate(word_id = row_number()) %>%
  left_join(get_sentiments('bing')) %>%
  mutate(sentiment = case_when(sentiment=='positive'~1,sentiment=='negative'~ (-1),
                               is.na(sentiment)~0)) %>%
  # left_join(get_sentiments('afinn')) %>%
  # mutate(score = na.fill(score,0)) %>%
  mutate(qdap_score= case_when(snt_sentiment >0 ~1,snt_sentiment < 0 ~ (-1), T ~ 0),
         bing = case_when(sentiment >0 ~ 1,sentiment <0 ~ (-1), T ~0)) %>%
  select(element_id,sentence_id,word_id,word,qdap_score,bing)
head(hotel_word,15)
```

```
##    element_id sentence_id word_id     word qdap_score bing
## 1           1           1       1     this          1    0
## 2           1           1       2    hotel          1    0
## 3           1           1       3       is          1    0
## 4           1           1       4       in          1    0
## 5           1           1       5      the          1    0
## 6           1           1       6  perfect          1    1
## 7           1           1       7 location          1    0
## 8           1           2       8       it          0    0
## 9           1           2       9       is          0    0
## 10          1           2      10   within          0    0
## 11          1           2      11  walking          0    0
## 12          1           2      12 distance          0    0
## 13          1           2      13       to          0    0
## 14          1           2      14     many          0    0
## 15          1           2      15    sites          0    0
```

Now we can calculate sentiment score based on each method

```
hotel_sentiment <- hotel_word %>%
  group_by(sentence_id) %>%
  summarise(qdap_score = mean(qdap_score), bing = sum(bing))
head(hotel_sentiment,10)
```

```
## # A tibble: 10 x 3
##    sentence_id qdap_score  bing
##          <int>      <dbl> <dbl>
##  1           1         1.    1.
##  2           2         0.    0.
##  3           3         1.    1.
##  4           4         1.    2.
##  5           5         1.    2.
##  6           6         1.    1.
##  7           7        -1.   -1.
##  8           8         1.    1.
##  9           9         0.    0.
## 10          10         1.    3.
```

The method using `spread()` in tidyr will return the same results:

```
hotel_sentiment2 <- hotel_sentence %>%
  unnest_tokens(word,sentence) %>%
  inner_join(get_sentiments('bing')) %>%
  count(sentence_id,word,sentiment) %>%
  spread(sentiment,n,fill =0) %>%
  group_by(sentence_id) %>%
  summarise(bing = sum(positive- negative))

# Use setequal() to compare 2 dataframe
# table hotel_sentiment2 already exclude sentence with sentiment = 0
hotel_sentiment2 %>%  setequal(hotel_sentiment %>%
                        semi_join(hotel_sentiment2,by = c('sentence_id'='sentence_id')) %>%
                        select(sentence_id,bing))
```

```
## TRUE
```

Now we can check which sentence in which **qdap_score** (complicated algorithm) is different from **bing score** (simple method):

```
# We must do the score conversion again so there are only 3 values: 1,0,-1
# Text data is big, so use View() to see the results
qdap_vs_bing <- hotel_sentiment %>%
     mutate(bing = case_when(bing >0~1, bing <0 ~ (-1), T ~ 0)) %>%
     filter(qdap_score != bing)

score_diff_sentence <- hotel_sentence %>%
  inner_join(qdap_vs_bing)
```

Out of 11174, there are 2883 sentences in which **qdap score** is diffrent from **bing score**. You can manually check each sentences to see the the differences! In most situation, `qdap score` way is more **accurate!**

3.4 The advantages of tagging method:

- **Cons:**
  - The size of table is bigger compared to the inner join("reduced" method). Normally, there are 3 position tagging columns(paragraph, sentence, word level) and 4-12 charactersitics tagging columns (it depends!). If the number of words are around **20 millions**. then the size of **word-level table** will be around **3GB!**
  - It is very *SQL-like* or *Pivottable-like* which seems to be **old-fashioned**.

- **Pros:**
  - **Position-tagging** allow us to **trace back** easily. From the example above, we can easily trace back to see which sentences that the `qdap_score` is different from `bing_score`.
  - It allow many **combinations** (I will discuss it in chaper 3) . For example, combine `sentence_id`, `qdap_score` can answer the question : *Which sentences is negative ?*. After that, we can combine with the original bing_score (at word level) to answer: *Which is negative words in negative sentences?* It is just 1 combination. 1-for-all tables will easily stimulate combination for the analyst. It depends on their ideas!
  - **1-for-all table** (word level) will also make **interactive graphics** a piece of cake! Each tagging column will become the filtered column for the graphics, hence making the process easier. In the scope of this articles, I will not go discussing the interactive graphics here

3.5 Deal with negations

We go back to our beloved **negations**! For **simplification**, I will use a short examples:

```
# The paragraph below is full of negation
x <-" I live without any happiness. It is not a very happy one. I couldn't be succesful.
There 's no things lucky happen in my life. Nobody loves me. No ones appreciate me.
Anyway, it is not really a disaster "
```

Firstly, we create the personal negation words and stop words:

```
negation_words <- c(negation.words,'without')
# We need to remove negation words from stop_words-->
# --> anti_join later will not filter out negations.
# Some sentiments words in bing is stopwords--> need to remove.
stop_words_mdf <- stop_words %>%
  filter(!word %in% negation_words) %>%
  filter(!word %in% get_sentiments('bing')$word) %>%
  bind_rows(data_frame(word = qdapDictionaries::amplification.words))
```

Then we create a word-level with position tagging:

```r
word_level <- x %>% replace_abbreviation() %>%
  replace_contraction() %>%
  as.tibble() %>%
  unnest_tokens(sentence,value,'sentences') %>%
  mutate(sentence_id = row_number()) %>%
  unnest_tokens(word,sentence) %>%
  mutate(word_id = row_number())
word_level
```

```
## # A tibble: 40 x 3
##    sentence_id word      word_id
##          <int> <chr>       <int>
##  1           1 i               1
##  2           1 live            2
##  3           1 without         3
##  4           1 any             4
##  5           1 happiness       5
##  6           2 it              6
##  7           2 is              7
##  8           2 not             8
##  9           2 a               9
## 10           2 very           10
## # ... with 30 more rows
```

Make the list of negation words:

```r
#Most of word between negation words and sentiment words are stop words.-->
#--> Therefore, it will be filtered out
negation <- word_level %>% anti_join(stop_words_mdf) %>%
  group_by(sentence_id) %>%
  mutate(negation = lag(word)) %>%
  filter(negation %in% negation_words) %>%
  inner_join(get_sentiments('bing'))
negation
```

```
## # A tibble: 6 x 5
## # Groups:   sentence_id [?]
##   sentence_id word       word_id negation sentiment
##         <int> <chr>        <int> <chr>    <chr>
## 1           1 happiness        5 without  positive
## 2           2 happy           11 not      positive
## 3           4 lucky           22 no       positive
## 4           5 loves           28 nobody   positive
## 5           6 appreciate      32 no       positive
## 6           7 disaster        40 not      negative
```

Now we will shift the word sentiments!

```r
# As we tag word_id, it is easy to perform the join
word_level_negate <- word_level %>%
  left_join(negation) %>%
  mutate(negation = if_else(is.na(negation),0,-1),
         sentiment = case_when(sentiment=='positive'~ 1, sentiment =='negative'~ -1,
                               is.na(sentiment)~0),
         sentiment_mdf = sentiment *negation)
```

```
word_level_negate %>% filter(sentiment != 0)
```

```
## # A tibble: 6 x 6
##   sentence_id word       word_id negation sentiment sentiment_mdf
##         <int> <chr>        <int>    <dbl>     <dbl>         <dbl>
## 1           1 happiness        5      -1.        1.           -1.
## 2           2 happy           11      -1.        1.           -1.
## 3           4 lucky           22      -1.        1.           -1.
## 4           5 loves           28      -1.        1.           -1.
## 5           6 appreciate      32      -1.        1.           -1.
## 6           7 disaster        40      -1.       -1.            1.
```

All the happy words become sad one, and sad becomes happy!

The **complication** of negation arise from the fact that the negation and sentiment words doesn's not comes **hand in hand!**. Eg: `"not really happy"` or `"without any happiness"`, There is 1 word to seperate them. Luckily, most of the words stand between them are either amplification words or stop words, so we can easily remove and use the `lag()` function.

Also we need to pay attention to deamplification words, too!

```
qdapDictionaries::deamplification.words
```

```
##  [1] "barely"       "faintly"      "few"          "hardly"
##  [5] "little"       "only"         "rarely"       "seldom"
##  [9] "slightly"     "sparsely"     "sporadically" "very few"
## [13] "very little"
```

In my opinion, most of deamplification words, eg: `"hardly"`, `"barely"` can be considered as **negations** and add to **negation word table**. For example, `"I can barely win"` is similar to `"I can not win"`. Of course, there are many **other possibilities** that I can not list down in this article (Well, I am not a native English speaker so my linguistic skills are quite restricted), but I believe that with my methods, people can **fully customize** their analysis!

Let's see how to apply my ideas into **phrasal verb/compound words** situation!

3.6 Deal with phrasal verbs/ compound words

I discuss the phrasal verbs problems in detail in the `1.2.2`, therefore, in this section, I will only dicuss further problems:

- Most of the time, **phrasal verbs/ compound words** go **hand in hand** (eg: **bigram** or **trigram**). There are no **noise words** between them so things are quite easy!
- Remember to add _ between words, so that it will turn bigram/trigram into unigram.The rest are simple sentiment analysis as word position are tagged with word_id
- Some combination should be paid attention to, eg `let sb down` .
- Remember to add tense to the list of phrasal verb sentiments. eg: `come` will have `came`,`comes`, so that we dont overlook some words

3.7 Deal with emoticon

Emoticon seems to be easily dealth with very easily (see `1.2.2` for more details), however, there are some **complications** that should be noted:

- Emoticons, if existed, and used correctly, can decide the sentiment of the **sentence** alone, no matter what the content is!. Eg `I was so sad :))` and `I was so happy :))` are both happy statements,if the users use emoticon correctly. There is **no sentiment shift**, unlike negations.
- However, the likelihood of users being **sarcastic** or **using emoji incorrectly** are very high, compare to word usages. From my observation, some one even use `<:(>` or `<:((>` just **for fun**(me,too!), especially in **chat**. But the **longer** the comment is, then the more **serious** the user is, the chance of emoji being used **correctly** will be **higher**.
- **Emoji spammer** should be simply ignored. It is **unlikely** to analyze it, and the content is very **doubtful!**

  Consider the following examples:

```
s1 <- " I am so sad, :( :( :( :( :( . I hope u can help me :P :P "
s2 <- " The hotel service is quite good, and I will definitely recommend it to others :)"
```

We can clearly see that the 1st comment is not very serious, and we can never know the intention of the comments without the whole context. While the 2nd comment is very likely that the users use the right emoji to express a happy experience. A simple counting of emoticon would help, and it depends a lot on the context of analysis.

3.8 Can we add bigram to the word-level dataframe?

Most of the problems that we need **bigram** to deal with (**negation, compound words**) are handled by my methods above. However, some specific words like `"Miss Julia"`, `"Captain BlackJack"` should not be overlooked!

However, the **difference** in length between **bigram** and **unigram** poses a problems. Eg: `"I lover her"` has 2 bigrams, but 3 unigrams. In any situation, the difference in length is 1!

Let's deal with it:

```
x <- "Miss Julia is so gorgeous. I really admire her so much"
# Make a bigram and seperate it into words
bigram <- x %>% as.tibble() %>%
  unnest_tokens(sentence,value,'sentences') %>%
  mutate(sentence_id = row_number()) %>%
  unnest_tokens(bigram,sentence,'ngrams',n =2) %>%
  group_by(sentence_id) %>%
  mutate(bigram_id = row_number()) %>%
  separate(bigram,c('word1','word2'),remove = F)
# In each group, only get the first word1, tag the position of 0
add_unigram <- bigram %>% #mutate(word2 =first(word1)) %>%
  filter(bigram_id ==1) %>%
  mutate(word2 =word1, word1 = NA_character_, bigram_id =0,bigram = NA_character_)
# bind rows and arrange
word_level <- bind_rows(bigram,add_unigram) %>%
  arrange(sentence_id,bigram_id) %>%
  ungroup() %>%
  mutate(word_id = row_number()) %>%
  select(sentence_id, word_id, bigram, word = word2)
word_level
```

```
## # A tibble: 11 x 4
##    sentence_id word_id bigram         word
##          <int>   <int> <chr>          <chr>
##   1            1       1 <NA>          miss
##   2            1       2 miss julia    julia
##   3            1       3 julia is      is
##   4            1       4 is so         so
##   5            1       5 so gorgeous   gorgeous
##   6            2       6 <NA>          i
##   7            2       7 i really      really
##   8            2       8 really admire admire
##   9            2       9 admire her    her
##  10            2      10 her so        so
##  11            2      11 so much       much
```

Our word_level dataframe is nearly complete!

3.9 Steming

As my discussion in 1.1.4, use **stemming** for cleaning is disturbing. However, after tokenizing to **word-level dataframe**, it can give some useful insights to the users! We can simply have another tagging columns without any harms!

There are 2 libraries that are most prominent in wordstemming: `SnowballC` and `hunspell`:

```r
x <- 'My employers employ
a method so that employee can provide a good provision'
# use SnowballC
x %>%  as.tibble() %>%
  unnest_tokens(word,value) %>%
  mutate(word_stm = wordStem(word))
```

```
## # A tibble: 13 x 2
##    word      word_stm
##    <chr>     <chr>
##  1 my        my
##  2 employers employ
##  3 employ    emploi
##  4 a         a
##  5 method    method
##  6 so        so
##  7 that      that
##  8 employee  employe
##  9 can       can
## 10 provide   provid
## 11 a         a
## 12 good      good
## 13 provision provis
```

```r
#hunspell_stem() returns a list, therefore a litte work might be needed
x %>%  as.tibble() %>%
  unnest_tokens(word,value) %>%
  mutate(word_stm = hunspell_stem(word) %>% str_extract('\\b\\w+\\b'))
```

```
## # A tibble: 13 x 2
##    word      word_stm
##    <chr>     <chr>
##  1 my        my
##  2 employers employer
##   3 employ    employ
##   4 a         a
##   5 method    method
##   6 so        so
##   7 that      that
##   8 employee  employee
##   9 can       can
##  10 provide   provide
##  11 a         a
##  12 good      good
##  13 provision vision
```

We can see from the results, `SnowballC::wordStem` seems to **reduce** words to its **roots**, and most of the time, in my opinion, it distorts the meaning so that we can not understand what the word really means. On the other hand, words with `hunspell_stem()` reduce the word **only** to the **word-levels**. Personally, I prefer the `hunspell()` function, but it is up to personal preference!

3.10 Part of speech (POS)

We begin with an examples:

```
# Load library specified in preface
# This example is to extract POS from text, return results in tidy text form
# Train the model for the fist time:
sent_token_annotator <- Maxent_Sent_Token_Annotator()
word_token_annotator <- Maxent_Word_Token_Annotator()
ent_person <- Maxent_Entity_Annotator('en','person')
ent_location <- Maxent_Entity_Annotator('en','location')
ent_org <- Maxent_Entity_Annotator('en','organization')
pos_tag_annotator <- Maxent_POS_Tag_Annotator()
# Extract POS and return in tidy text form
x <- "His name is Barrack Obama. Michelle Obama is his wife. His idol is Micheal Jackson.
He used to travel to Canada and now he works in London"
get_pos <- function(x){
  s <- as.String(x)
  annotate(s, list(sent_token_annotator,word_token_annotator,pos_tag_annotator, ent_person,
                          ent_location,ent_org)) %>%
  as_tibble() %>%
  #extract feature name from a list
  mutate(type2 = str_extract(features,'(?<=\").*(?=\")') ,
         name = str_sub(s,start,end)) %>%
  # Filter to keep only the words
  filter(str_detect(type2,'\\b[A-Z$]+\\b'))
}
a<- get_pos(x)
head(a,10) # You can use View() to check yourself the full results
```

```
## # A tibble: 10 x 7
##        id type  start   end features   type2 name
##     <int> <chr> <int> <int> <list>     <chr> <chr>
##  1      5 word      1     3 <list [1]> PRP$  His
##  2      6 word      5     8 <list [1]> NN    name
##  3      7 word     10    11 <list [1]> VBZ   is
##  4      8 word     13    19 <list [1]> NNP   Barrack
##  5     9 word     21    25 <list [1]> NNP   Obama
##  6    11 word     28    35 <list [1]> NNP   Michelle
##  7    12 word     37    41 <list [1]> NNP   Obama
##  8    13 word     43    44 <list [1]> VBZ   is
##  9    14 word     46    48 <list [1]> PRP$  his
## 10    15 word     50    53 <list [1]> NN    wife
```

You can go to this website: https://cs.nyu.edu/grishman/jet/guide/PennPOS.html to check the meaning of POS abbreviation,eg: `JJ` is for `adjective`.

The above process can be considered as `work token–> POS tagging`. However, the algorithm of `unnest_token()` is a little bit different, hence produce different results

```
x<- hotel_reviews$text[1]
# Use NLP
get_pos(x)
```

```
## # A tibble: 98 x 7
##       id type  start   end features   type2 name
##    <int> <chr> <int> <int> <list>     <chr> <chr>
##  1     8 word      1     4 <list [1]> DT    This
##  2     9 word      6    10 <list [1]> NN    hotel
##  3    10 word     12    13 <list [1]> VBZ   is
##  4    11 word     15    16 <list [1]> IN    in
##  5    12 word     18    20 <list [1]> DT    THE
##  6    13 word     22    28 <list [1]> JJ    perfect
##  7    14 word     30    37 <list [1]> NN    location
##  8    16 word     40    41 <list [1]> PRP   It
##  9    17 word     42    43 <list [1]> VBZ   's
## 10    18 word     45    50 <list [1]> IN    within
## # ... with 88 more rows
```

```
# Use unnest_token() in tidy text
x %>% as.tibble %>%
  unnest_tokens(word,value)
```

```
## # A tibble: 95 x 1
##    word
##    <chr>
##  1 this
##  2 hotel
##  3 is
##  4 in
##  5 the
##  6 perfect
##  7 location
##  8 it's
##  9 within
## 10 walking
## # ... with 85 more rows
```

We can see that the former method produce a tibble with 98 rows, while with the former, the number of rows are 95 rows. Words like `it's` is the reason for the difference here.We can cleaning (read part 1 for more details) but there's no guarantee that our cleaning is perfect! THerefore, a simple join wordID-to-wordID to get the word POS is unreliable (only accurate if 2 results have the same number of rows).

As `unnest_tokens()` is the workhorse function of the whole articles and is the foundation for the other method, we should keep it (POS tagging is just a part of tagging, hence it is less important!). Therefore, the idea is join by `sentence_ID` combined with `word`.It can not deal with the situations when 1 sentence have 2 words with different POS, eg: "I long for a long meeting". Let's try this

```r
# Sentence token
hotel <- hotel_reviews %>%  head(10) %>%
  select(text) %>%
  get_sentences %>%
  mutate(sentence_id = row_number())
# Loop through each sentence to get the pos in each sentence
dt <- vector('list',length(hotel$sentence_id))
for ( i in seq_along(hotel$sentence_id)){
  dt[[i]] <- get_pos(hotel$text[i]) %>%
    mutate(sentence_id = i)
}
# Bind all results into one
sentence_pos <- do.call(rbind,dt) %>%
  select(sentence_id,word = name, pos = type2)
# Join with sentence_id and word.
# Note that they use the same get_sentence() for sentence token.
# Therefore, the sentence id will match
word_pos <- hotel %>% unnest_tokens(word,text,to_lower = F) %>%
  left_join(sentence_pos, by = c('word'='word','sentence_id'='sentence_id'))
head(word_pos,15)
```

```
##    element_id sentence_id     word  pos
## 1           1           1     This   DT
## 2           1           1    hotel   NN
## 3           1           1       is  VBZ
## 4           1           1       in   IN
## 5           1           1      THE   DT
## 6           1           1  perfect   JJ
## 7           1           1 location   NN
## 8           1           2     It's <NA>
## 9           1           2   within   IN
## 10          1           2  walking   NN
## 11          1           2 distance   NN
## 12          1           2       to   TO
## 13          1           2     many   JJ
## 14          1           2    sites  NNS
## 15          1           2      and   CC
```

It is really accurate!

3.11 Get names (POS continued…)

Although the medthod can extract the name through `NNP`, however, the results are quite fragmented.For example, the word `Barrack Obama` will be separated into `Barrack` and `Obama`. I propose a way to deal with the situation, similar to how we deal with compound words:

- Extract the name from the text via POS
- Create a lookup tables with modified name, eg: `Barrack Obama` turns to `Barrack_Obama`
- Use mgsub() to substitute a compound name with a unigram names in raw text.
- Do tokenization again and join with the lookup table above

```
# Text is not cleaned
# Strip middle space is not important
x <- "His name is Barrack     Obama. Michelle Obama is his wife. His idol is Micheal Jackson.
He used to travel to Canada and now he works in London"
# x <- str_replace_all(x,regex(' +'),' ')   # if want to remove extra white space

s <- as.String(x)
name_table <- annotate(s, list(sent_token_annotator,word_token_annotator,pos_tag_annotator, ent_person,
                       ent_location,ent_org)) %>%
  as_tibble() %>%
  #filter(type =='entity') %>%
  filter( type =='entity')  %>%
  mutate(name = str_sub(s,start,end),
         name_type = str_extract(features,'(?<=\").*(?=\")') ,
         name_mdf = str_replace_all(name,regex(' +'),'_'))

x2 <- mgsub(name_table$name,name_table$name_mdf,text.var = x)

word_levelwithname <- x2%>% as.tibble() %>%
  unnest_tokens(word,value,to_lower = F) %>%
  left_join(name_table ,by = c('word'= 'name_mdf')) %>%
  select(word,name_type)
word_levelwithname %>% filter(!is.na(name_type))
```

```
## # A tibble: 5 x 2
##   word            name_type
##   <chr>           <chr>
## 1 Barrack_Obama   person
## 2 Michelle_Obama  person
## 3 Micheal_Jackson person
## 4 Canada          location
## 5 London          location
```

If you even want a better accuracy, consider add the `sentence_id` or `comment/element_id` and use it as additional keys to join. It is quite the same but you might need a litte extra work.

4 Analyzing text

After creating a word-level dataframe, now it is time for us to do analysis! This part is the easiest in terms of programming as all we need are mostly **group by** and **filter** (most of information are already prepared in the word-level dataframe!). However, in this part, **imagination** and a good **common sense** are really important!

4.1 Some useful functions to explore data

This part is quite off-track as it doesn't focus on text, but help you to work more effectively with data. I strongly believe that, in data analysis field, the idea is the most important, especially in R, where most of the tools are contributed by our beloved, enthusiastic authors.

How to have ideas? In my opinion, we need to understand the data. R has some wonderful functions to do it,namely `head()/tail()` , `View()`, `str()`,`summary()` and Rstudio is simply the best IDE ever. However, as a programming language, it can never be as interactive as Excel. Excel is our friends afterall and we should never abandon it!

I just want to add some small functions which I believe that can make the process of understanding data less painful. My functions will:

- Help people interactively work with Excel.
- Help people to easily understand the categorical data.

4.1.1 View the data

The idea is: copy the work from excel to R and vice verse via clipboards:

```
# writing from R to Excel
ex_wr <- function(x){write.table(x, "clipboard-16384",sep = '\t',row.names = F )}
# read from Excel to R
ex_re <-function(){
  e1 <- read.table('clipboard-16384',sep = '\t',header = T,comment.char = '') %>%
    as.tibble()
  show(head(e1,5))
  return(e1)
                 }
```

For example, you have a datagrame which you want to navigate in Excel

```
mpg %>% ex_wr() # then go to an Excel and Ctr+V
```

And you have a table in excel that you want to read back in R right away

```
# choose the table, ctr+C
your_data <- ex_re()
```

Now you can switch between Excel and R without a sweat!

The following (very simple) function can help you a little bit typing time

```
# View() 30 first line of table, if nrow <100, return all
hview <- function(x,y =30, z = 100){
  x_name <- deparse(substitute(x))
  if(nrow(x)< z){
  x  %>% View(title = x_name)
  }
  else{
  x %>%  head(y) %>% View(title = x_name)
  }
}
# View() 30 first line of table, if nrow <100, return all
tview <- function(x,y = 30, z = 100){
  x_name <- deparse(substitute(x))
  if(nrow(x)< z){
    x  %>% View(title = x_name)
  }
  else{
    x %>%  tail(y) %>% View(title = x_name)
  }
}
```

For example, after a lot of works, you want to see your results very quickly

```
# Run your own result!
your_result %>% hview()
```

It can save a lot of time. Sometimes, I click on the variables in Environment (Rstudio) just to see the dataframe without knowing that dataframe is too big and it takes me forever!

4.1.2 See the categorical columns

```
unq_val <- function(x,t = 3,sample_size = 1000){
  # create a data for test if column is categorical or not
  dt_test <- list('vector',length(colnames(x)))
  # if number of rows of x < sample size (1000)--> take nrow of x as sample size
  sample_size <- min(nrow(x),sample_size)
  for (i in seq_along(colnames(x))){
    y <- sample_n(x,sample_size) # make a sample of x
    dt_test[[i]]<- y[[i]] %>%  unique()# get unique value for each column of x
  }
    for ( i in seq_along(dt_test)){
    if(length(dt_test[[i]])< sample_size/t){
      # if number of unique value < sample_size/t (t =3)--> categorical
      dt_test[[i]]<- dt_test[[i]]
    }
    else{
      # non categorical data--> don't need to get unique value-->
      # --> assign them zero length --> filer out later
      dt_test[[i]] <- vector()
    }
  }
  names(dt_test) <- names(x)# give colname for the data test
  # Get the names of columns which are categorical
  col_categorical <- names(dt_test[lapply(dt_test,length)>0])
  dt <- vector('list',length(col_categorical))
  for (i in seq_along(col_categorical)){
    # get unique value for each categorical column
    dt[[i]] <- unique(x[[col_categorical[i]]]) %>% sort(method = 'quick')
  }
  # Give the same vector length to each unique column--> can combine later
  max_length <- sapply(dt,length) %>%  max()
  for ( i in seq_along(dt)){
    length(dt[[i]])<- max_length
    dt[[i]]<- as.tibble(dt[[i]])
  }
  # combine all column to one
  dt_last <- bind_cols(dt)
  names(dt_last) <- col_categorical
  return(dt_last)
}
```

It is really useful to examine categorical data

```
unq_val(mpg)
```

```
## # A tibble: 38 x 11
##    manufacturer model      displ  year   cyl trans  drv     cty   hwy fl
##    <chr>        <chr>      <dbl> <int> <int> <chr>  <chr> <int> <int> <chr>
##  1 audi         4runner ~   1.60  1999     4 auto(~ 4         9    12 c
##  2 chevrolet    a4          1.80  2008     5 auto(~ f        11    14 d
##  3 dodge        a4 quatt~   1.90    NA     6 auto(~ r        12    15 e
##  4 ford         a6 quatt~   2.00    NA     8 auto(~ <NA>     13    16 p
##  5 honda        altima      2.20    NA    NA auto(~ <NA>     14    17 r
##  6 hyundai      c1500 su~   2.40    NA    NA auto(~ <NA>     15    18 <NA>
##  7 jeep         camry       2.50    NA    NA auto(~ <NA>     16    19 <NA>
##  8 land rover   camry so~   2.70    NA    NA auto(~ <NA>     17    20 <NA>
##  9 lincoln      caravan ~   2.80    NA    NA manua~ <NA>     18    21 <NA>
## 10 mercury      civic       3.00    NA    NA manua~ <NA>     19    22 <NA>
## # ... with 28 more rows, and 1 more variable: class <chr>
```

```
unq_val(sentiments)
```

```
## # A tibble: 14 x 3
##    sentiment    lexicon  score
##    <chr>        <chr>    <int>
##  1 anger        AFINN       -5
##  2 anticipation bing        -4
##  3 constraining loughran    -3
##  4 disgust      nrc         -2
##  5 fear         <NA>        -1
##  6 joy          <NA>         0
##  7 litigious    <NA>         1
##  8 negative     <NA>         2
##  9 positive     <NA>         3
## 10 sadness      <NA>         4
## 11 superfluous  <NA>         5
## 12 surprise     <NA>        NA
## 13 trust        <NA>        NA
## 14 uncertainty  <NA>        NA
```

It will check a sample of 1000 rows to identify what column are catergorical (has only a small number of values), which columns are not (has many values). Then it returns all of categorical columns with their unique values.

4.2 Analyze text from word-level dataframe

4.2.1 Example:

- Let's answer an interesting question: What are the **positive words** in **negative sentence** ?

```
word_negtv_snt_negtv <- hotel_word %>%
  filter(qdap_score <0,bing >0) %>%
  count(word,sort = T)
word_negtv_snt_negtv
```

```
## # A tibble: 200 x 2
##    word       n
##    <chr>  <int>
##  1 like      81
##  2 great     47
##  3 nice      46
##  4 good      45
##  5 enough    41
##  6 work      36
##  7 clean     32
##  8 hot       27
##  9 right     27
## 10 well      26
## # ... with 190 more rows
```

We can see that there are some interesting words like `"great"`,`"clean"`. Let's see why the word setiment contradict the sentence sentiment (based on qdap):

```
snt_negtv_filter <- hotel_sentence %>%
  inner_join(hotel_word %>% filter(qdap_score <0 )) %>%
  filter(word == 'clean')
# see the data yourself!
```

Remember my `hview()` or `ex_wr()` functions in `3.1.1`. It will be very useful here!

We can see that most negative sentence which including `clean` words because:
* Has negation words: it is `"not clean"` at all!
* Hotel is clean, but other factors are just bad.

Now we can check which hotel is not clean

```
# see part 3.5 for dealing with negation
negation_clean <- hotel_word %>%
# Use the stop_words_mdf which remove the negation words
# dataframe stop_words_mdf is at part 3.5
  anti_join(stop_words_mdf) %>%
# most word between the negation and the sentiments are stop words-->
# It will be removed from anti join
  mutate(negation = lag(word)) %>%
  filter(negation %in% negation_words) %>%
  inner_join(get_sentiments('bing')) %>%
  filter(word == 'clean')

negation_clean
```

```
##   element_id sentence_id word_id  word qdap_score bing negation sentiment
## 1         53         578    9305 clean         -1    1      not  positive
## 2         104       1125   18245 clean         -1     1       not  positive
## 3         105       1138   18494 clean          1     1       not  positive
## 4         150       1669   27561 clean         -1     1       not  positive
## 5         220       2376   38407 clean         -1     1       not  positive
## 6         266       2872   46128 clean         -1     1       not  positive
## 7         288       3100   49655 clean         -1     1       not  positive
## 8         330       3537   56740 clean          1     1       not  positive
## 9         461       4964   78985 clean         -1     1       not  positive
## 10        479       5231   83522 clean         -1     1        no  positive
## 11        580       6454  101929 clean         -1     1       not  positive
## 12        611       6878  109104 clean          1     1       not  positive
## 13        644       7192  113862 clean         -1     1       not  positive
## 14        660       7371  116610 clean         -1     1       not  positive
## 15        787       8852  141207 clean         -1     1       not  positive
## 16        879       9880  158083 clean         -1     1       not  positive
## 17        890       9989  160087 clean         -1     1       not  positive
## 18        901      10095  161662 clean         -1     1       not  positive
## 19        985      10953  175163 clean          1     1       not  positive
```

Well, imagine that we have a tagging column that has hotel name, we can easily tell which hotel is not clean!

The method above looks quite good, but there are some sentence that it recognize incorrectly (use `View()` or my `hview()` function to explore data!)

```r
s1<- "The handicapped room (which we did not ask for) was clean"
s2 <-  "The room was not that big, but it was clean."
# function to see which words remain after removing stop words
negation_test <- function(x){
  x %>% as.tibble() %>%
    unnest_tokens(word,value) %>%
    anti_join(stop_words_mdf)
}
negation_test(s1)
```

```
## # A tibble: 3 x 1
##   word
##   <chr>
## 1 handicapped
## 2 not
## 3 clean
```

```r
negation_test(s2)
```

```
## # A tibble: 2 x 1
##   word
##   <chr>
## 1 not
## 2 clean
```

It seems that the problems lies on the `stop_words_mdf` table which remove so many words between `"not"` and `"clean"`, therefore it make positive expression become a negative one, eg: it remove all of words between `"not"` and `"clean"` from the `" not that big, but it was clean"`

```r
stop_words_mdf %>% filter(str_detect(word,('(big)|(ask)')))
```

```
## # A tibble: 7 x 2
##   word   lexicon
##   <chr>  <chr>
## 1 ask    SMART
## 2 asking SMART
## 3 ask    onix
## 4 asked  onix
## 5 asking onix
## 6 asks   onix
## 7 big    onix
```

Words like `"big"`, `"ask"` should not be stopwords, but they are included in our `stop_words` table that consequently become `stop_words_mdf`. We can mannually check the `stop_words` and remove words that we believe that are **not stopwords**.

We can easily see that with my ideas of word-level dataframe, we can **trace back** to see what really happens and also modify the method accordingly. Also, we should not blindly use any lookup tables that are provided. Most of the look-up tables are not very big, so we can check and **modify** it according to the **personal needs**. Text analysis are not static, but works of **trial-and-errors**!

4.2.2 Analyze without coding

Be careful! This section is a **wall of text** and **without any codes**! However, I believe that it is the most important part of this articles :)

We come back to our beloved **word-level** dataframe and see what are available here:

- Position tagging: **paragrapth/comment** ID, **sentence** ID, **bigram/ word** ID (phrasal verb and compound words are at word level). Depending on different situations, we can add **time** tag or **some specific tags** (eg: `hotel id`). Remember in the example above, I suggest tag the position with hotel so that we can find out which hotel is not clean!

- Sentiment tagging: **sentence-level** qdap(you can also measure the sentiment at the paragraph level with qdap-sentimentr combo), **word-level quantitative** bing,afinn (we convert bing from qualitative to quantitative, eg: `positive` to `1`), **qualitative** : nrc, **negation**(which shift sentiment), **emoticon** (very important or nothing)
- Characteristic tagging: **stemming**, **part-of-speech**(noun,verb), **name** tag, **phrasal_verb** and **compound_words** .

Now let's **combine all the taggings!**

- In the example above, I combine the `sentence_ID`, `qdap_score`, `bing_score` and with some `negation` help, we can answer the question: Which are **positive words** in **negative sentence** and which are the sentence position ? Let's paraphrase it:
  - Positive words (bing_score)–> negative sentence(qdap_score)–> sentence_position (sentence_id)
- Also, in part `3.3: Taging`, with `sentence_ID`, `qdap_score`, `bing_score`, we also can answer the following question?
  - At which sentence(`sentence_ID`) that sentence-level sentiment (`qdap_score`) are different from word-level method of sentiment (`bing_score`)

It is time for imagination (note that I put the tagging you need to use in `<()>`)

- Add `nrc` and remove `qdap_score` , we can deal with the following question:

  - Which positive words (`bing_score`) contribute the joy sentiment (`afinn`) at which sentence(`sentence id`)?

- Add `afinn` : Which are the **most extreme** (`afinn` as it has 4,5,-4,-5 score) positive/ negative? words (`bing`)

- Add `part of speech`: Which noun/adjective (`Part-of-speech`) are used the most in positive sentence (`qdap score` and `sentence id`). Are they positive or negative (`bing` or `afinn`)?

- Add `name` (We already extracted location as well as people's name: see `3.11` for more detail) : At which place (`name`) people often have most positive experience (`qdap score` or `bing score`), at which sentence (`sentence id`)?
- Add other postition,eg: hotel id/name: which hotel (`hotel id`) has the most negative comments (`qdap` at comments level), negative comment content(`sentence level qdap`, `bing score`), and which characteristics(`words` or `part-of-speech: noun`) are hated the most?
- Add `timeline tagging`: we can examine: how sentiments (`qdap`,`bing`...) change over time (`time tagging`) in 1 hotel (`hotel id`). Is it better?
- Add `emoticon` and we may have the most exciting question/answer: Which hotel(`hotel id`), place(`name tagging`) has the most young people (`emoticon tagging/phrasal verb tagging`). We can just count the number of **emoticon** and as **young people** has tendency to use emoticon (or maybe phrasal verb), so we just take the group with the greatest counts (`n()`) of emoticon.
- Add `negation` and we may have other questions/anwers pairs: Which hotel (`hotel id`) has 1 good characteristics (bing) but other charactericstics are just bad. In **4.2.1**. We find out the negation of good sentiments (remember the `"clean"` word?)–> therefore, good sentiments(`bing`) without negation(! %in% `negation`) but in negative sentence(`qdap`, `sentence id`) will answer this question!

The number of combination is not even exhausting and I believe that once the analyst get exposed to different situation, they can come up with more and more wonderful combination!

4.2.3 The most specific terms: tf-idf:

Consider our following examples:

```
# I deliberately make 5 comments with the same beginning part for easier comparison
s1 <- "The service here is worderful, food is good, staff is very nice.
However, room is small,extremely small"
s2 <- "The service here is worderful, food is good, staff is very nice.
Atmosphere is a little bit cold, not to say very cold though"
s3 <- "The service here is worderful, food is good, staff is very nice.
Is is quite expensive"
s4 <- "The service here is worderful, food is good, staff is very nice.
Very expensive,though"
s5 <- "The service here is worderful, food is good, staff is very nice.
The location is quite inconvenient"

hotel_comment <- c(s1,s2,s3,s4,s5) %>% as.tibble() %>%
  mutate(comment_id = row_number()) %>%
  unnest_tokens(word,value) %>%
  anti_join(stop_words)

#simple take the word frequency
word_count <- hotel_comment %>% count(word,sort = T)
word_count
```

```
## # A tibble: 12 x 2
##    word            n
##    <chr>       <int>
##  1 food            5
##  2 nice            5
##  3 service         5
##  4 staff           5
##  5 worderful       5
##   6 cold             2
##   7 expensive        2
##   8 atmosphere       1
##   9 bit              1
##  10 extremely        1
##  11 inconvenient     1
##  12 location         1
```

```
# use tf-idf method
tf_idf <- hotel_comment %>%
  count(comment_id,word) %>%
  bind_tf_idf(word,comment_id,n) %>%
  arrange(desc(tf_idf))
tf_idf
```

```
## # A tibble: 33 x 6
##    comment_id word              n    tf   idf tf_idf
##         <int> <chr>         <int> <dbl> <dbl>  <dbl>
##  1          2 cold              2 0.222 1.61   0.358
##  2          1 extremely         1 0.167 1.61   0.268
##  3          5 inconvenient      1 0.143 1.61   0.230
##  4          5 location          1 0.143 1.61   0.230
##  5          2 atmosphere        1 0.111 1.61   0.179
##  6          2 bit               1 0.111 1.61   0.179
##  7          3 expensive         1 0.167 0.916  0.153
##  8          4 expensive         1 0.167 0.916  0.153
##  9          1 food              1 0.167 0.     0.
## 10          1 nice              1 0.167 0.     0.
## # ... with 23 more rows
```

Our review comments all start with the **same compliments**. The difference are the **critical comments** at the end, and they are specific to each comments. If we only focus on word frequency, it means that we missed out important information like: `"inconvenient"`, `"atmosphere"`, `"cold"`, `"expensive"`.

Words like `"food"`, `"service"` have the **idf = 0** because they appears in all of comments, therefore, they are **not specific enough**. So, what should we use? **word count** or **tf_idf**?

The word-count method that we used before this chapter is to answer the questions: "Which are the **most frequent words** in **the whole text** that has certain characteristics?"

The tf-idf method is to answer the question: "Which are the most **specific words** in **each unit** that has certain characteristics?"

Also, an other important question to answer is : "Which are the most **frequent words** in **each unit** that has certain characteristics?"

I believe that the combination between 2 methods will give us a more comprehensive picture:

```
important_word <- bind_rows(head(word_count,5) %>% select(word),
                            head(tf_idf,5) %>% select(word))
```

We also need to consider other factors:

- Should we remove the stopwords? Let's do a small mathematic:
  $tf_idf = tf * idf = (wordcount/numberofwords) * idf$
  The idf and word_count will not change regardless of we remove stop words or not. Consider following example:

```r
"As far as I know, it is bad and ugly" %>% as_tibble() %>%
  unnest_tokens(word,value) %>%
  anti_join(stop_words_mdf)
```

```
## # A tibble: 2 x 1
##   word
##   <chr>
## 1 bad
## 2 ugly
```

```r
"Is it bad and ugly" %>% as_tibble() %>%
  unnest_tokens(word,value) %>%
  anti_join(stop_words_mdf)
```

```
## # A tibble: 2 x 1
##   word
##   <chr>
## 1 bad
## 2 ugly
```

We can see that 2 comments has basically same meaning. If we remove stopwords, then the tf of the word `bad` are both **1/2** for 2 comments, while is it **1/10** for comment1 and **1/5** for comment2 (without removing the stop words). For comparison, it is clear that removing stopwords are better and gives at a fairer view. In addition, some extremely frequent words like `the`,`a`,`an` also have a low **idf** (as it appears in most of the documents!) so if we look at the tf_idf, it will be filtered out with or without removing stop words.

Anyway, the ideas of stopwords are: words which **doesn't affect** the contents, so removing them are always better.

- How about words with **high tf**. Are we already covered by the word_count and tf_idf method?
  - high tf, low tf_idf: Low tf_idf means that words will appear in most of the documents, therefore with high tf –> **most frequent words** in **the whole text**. We already dealt with it.
  - high tf, high tf_idf: the most **specific words** in **each unit**. We already covered it
  - high tf, medium tf_idf: not yet covered–> need to be considered.

Therefore, our most important_words will be:
$importantwords = highwordcount + hightf + hightfidf$
As there are some overlap between high word_count and high tf , it will become:
$important_words = union(highword_count + hightf) + hightf_idf$

We already covered all of the most important **single words** in the **whole text**. Next part, we will cover most important **pair of words** with the package `widyr`

4.3 Analyze pair of words

Let's start with an example (It looks quite silly, though!)

```r
s1 <- "price is good. Staff is bad"
s2 <- "Price is good. Staff is bad. Decoration is good"
s3 <- "Decoration is bad. Atmosphere is good. Staff is bad"
s4 <- "Price is bad. Decoration is good. Staff is good"
s5 <- "Atmosphere is good. Price is bad. Decoration is good"
# single word- method
c(s1,s2,s3,s4,s5) %>% as.tibble() %>%
  unnest_tokens(word,value) %>%
  anti_join(stop_words_mdf) %>%
  count(word)
```

```
## # A tibble: 6 x 2
##   word           n
##   <chr>      <int>
## 1 atmosphere     2
## 2 bad            6
## 3 decoration     4
## 4 good           8
## 5 price          4
## 6 staff          4
```

```r
# pair of word-method
 c(s1,s2,s3,s4,s5) %>% as.tibble() %>%
  unnest_tokens(sentence,value,"sentences") %>%
  mutate(sentence_id = row_number()) %>%
  unnest_tokens(word,sentence) %>%
   anti_join(stop_words_mdf) %>%
  #count(sentence_id,word) %>%
  pairwise_count(word,sentence_id) %>%
  inner_join(get_sentiments('bing'),by = c('item1'='word')) %>%
  anti_join(get_sentiments('bing'),by = c('item2'='word'))
```

```
## # A tibble: 7 x 4
##   item1 item2          n sentiment
##   <chr> <chr>      <dbl> <chr>
## 1 good  price         2. positive
## 2 bad   price         3. negative
## 3 good  staff         2. positive
## 4 bad   staff         3. negative
## 5 good  decoration    3. positive
## 6 bad   decoration    1. negative
## 7 good  atmosphere    2. positive
```

With the single word method, we can easily see all of important words, but we can never know whether the `atmosphere` is `good` or `bad` (We might use sentiment at the sentence level using qdap or bing, but there will be a lot of noise in a complicated sentences!). With the **pair of word** method, however, we can clearly see which sentiments go with which characteristics. It is clear that the comments indicate a place with good decorations and atmosphere!.

We can also use `pairwise_cor()` function which not only take into consideration how word pairs appear together, but also how word pairs not appear together.

```r
c(s1,s2,s3,s4,s5) %>% as.tibble() %>%
   unnest_tokens(sentence,value,"sentences") %>%
   mutate(sentence_id = row_number()) %>%
   unnest_tokens(word,sentence) %>%
   anti_join(stop_words_mdf) %>%
   count(sentence_id,word) %>%
   pairwise_cor(word,sentence_id,n) %>%
   inner_join(get_sentiments('bing'),by = c('item1'='word')) %>%
   anti_join(get_sentiments('bing'),by = c('item2'='word')) %>%
   arrange(desc(correlation))
```

```
## # A tibble: 8 x 4
##   item1 item2      correlation sentiment
##   <chr> <chr>            <dbl> <chr>
## 1 good  atmosphere      0.583  positive
## 2 bad   price           0.492  negative
## 3 bad   staff           0.492  negative
## 4 good  decoration      0.201  positive
## 5 good  staff          -0.0680 positive
## 6 bad   decoration     -0.0909 negative
## 7 good  price          -0.408  positive
## 8 bad   atmosphere     -0.452  negative
```

It gives the same answer to `pairwise_count` above! (Eg: we have 3 `bad staff` and 2 `good staff`, hence staff is **positively correlated with bad**, and **negatively correlated with good**)

However, `pairwise_cor()` should be used with great care. Take the following examples:

```
'price is good. Atmosphere is good. Staff is good.
 Decoration is bad. Price is expensive . Overally good'  %>% as.tibble() %>%
  unnest_tokens(sentence,value,"sentences") %>%
  mutate(sentence_id = row_number()) %>%
  unnest_tokens(word,sentence) %>%
  anti_join(stop_words_mdf) %>%
  pairwise_cor(word,sentence_id) %>%
  inner_join(get_sentiments('bing'),by = c('item1'='word')) %>%
  anti_join(get_sentiments('bing'),by = c('item2'='word')) %>%
  arrange(desc(correlation))
```

```
## # A tibble: 15 x 4
##    item1     item2      correlation sentiment
##    <chr>     <chr>            <dbl> <chr>
##  1 bad       decoration       1.00  negative
##  2 expensive price            0.632 negative
##  3 good      atmosphere       0.316 positive
##  4 good      staff            0.316 positive
##  5 good      overally         0.316 positive
##  6 bad       atmosphere      -0.200 negative
##  7 expensive atmosphere      -0.200 negative
##  8 bad       staff           -0.200 negative
##  9 expensive staff           -0.200 negative
## 10 expensive decoration      -0.200 negative
## 11 bad       overally        -0.200 negative
## 12 expensive overally        -0.200 negative
## 13 good      price           -0.250 positive
## 14 bad       price           -0.316 negative
## 15 good      decoration      -0.632 positive
```

We can see from above example that, although the result is still correct, but as the `good` is link to many characteristics, then the correlation is simply diluted–> smaller correlation–> will be filtered out if we `select top`. In this example, all of pair of words has the same weights, so counting is better!

We also need to consider the **noise**:

```
# We can never know price is good or bad, since good and bad at the same sentence
"The price is good, but staff is bad"
```

```
## [1] "The price is good, but staff is bad"
```

It should be noted that, we try to reduce the **"noise"** as much as possible by analyzing at the **smallest group** available (sentence_id). If we use a bigger group, eg: `comment` then the **noise** will be greater. Once again, you can see how position tagging with sentence_id can be useful!

Count method and correlation methods are both useful and if we use it at the sentence level, it is very likely that the results are correct. However, both of them are not perfect and can supplement each other.Therefore, we should combine 2 methods to show which are the most important pair of words:

$importantwordpair = mostfrequentpair + mostcorrelatedpair$ Combination is always good!

We will do a small example to deal with word pairs: Which word pairs are most common in negative sentence(`qdapscore`, `sentence_id`)?

```
# The data frame hotel words are word-level dataframe. See part 3.3 for detail
word_cnt_snt_negtv <- hotel_word %>%
  filter(qdap_score <0) %>%
  anti_join(stop_words_mdf) %>%
  pairwise_count(word,sentence_id) %>%
  arrange(desc(n)) %>%
  inner_join(get_sentiments('bing'),by = c('item1'='word')) %>%
# there are so many item2 with negations--> need to filter out!
  filter(!item2 %in% negation_words)
```

We can see that there are some positive sentiment pairs here which contradict the negative sentiments of the sentence. We will try to analyze them:

```
hotel_sentence %>%
  filter(snt_sentiment <0) %>%
# use my hview() function to see top 30 rows in R studio
# View the result by youtself
  filter(str_detect(sentence,'hot'),str_detect(sentence,'water')) %>% hview()
## Do other filter by yourself
```

Here are some conclusion from the analysis and our traceback!
* The word `"like"` is more likely to use as a connection word rather than a sentiments. * It seems that some hotels has the problems with **water** not being **hot** enough!
* Some hotels are not **good** (negation) and some are **good but not enough**! so that it still receive negative comments!
* Some hotels are `"nice"` but **not enough**. And we can see some of `noise` context here,eg:" I was worried that the actual hotel would not be as nice as photos but I was wrong“. It is definitely a **positive sentence** but the **sentiment is twisted** as there are so **many negations** here (It is negation of negation). * We can check some negative pairs: `expensive hotel`, `noise street`, `bad hotel`, `rude staff`. It is quite clear that those pair truly represent what it means!

Next we test the word correlations:

```
hotel_word %>%
  filter(qdap_score <0) %>%
  anti_join(stop_words_mdf) %>% #hview()
  group_by(word) %>%
  filter(n() >=8) %>%
  pairwise_cor(word,sentence_id) %>%
  inner_join(get_sentiments('bing'),by = c('item1'='word')) %>%
  arrange(desc(correlation))
```

```
## # A tibble: 30,959 x 4
##    item1      item2    correlation sentiment
##    <chr>      <chr>          <dbl> <chr>
##  1 outrageous prices         0.406 negative
##  2 terrible   customer       0.349 negative
##  3 lost       found          0.333 negative
##  4 smoke      smoking        0.329 negative
##   5 stains     sheets         0.306 negative
##   6 free       lobby          0.303 positive
##   7 free       day            0.294 positive
##   8 free       internet       0.293 positive
##   9 smoke      casino         0.287 negative
##  10 hot        water          0.274 positive
## # ... with 30,949 more rows
```

It should be noted that every time you **change** the value in `filter(n() >=...)`, the results will be **different** as we only focus on relatively **high frequent words**. However, you can change the values and `bind` all dataframe for each values so that we can have a better views!

A little bit loop might help!

```
dt <- list()
n <- 1:10
for (i in 1:10){
dt[[i]] <- hotel_word %>%
    filter(qdap_score <0) %>%
    anti_join(stop_words_mdf) %>% #hview()
    group_by(word) %>%
# only filter for words which has the freqency >=8 .... to 17
    filter(n() >= (i +7)) %>%
    pairwise_cor(word,sentence_id) %>%
    inner_join(get_sentiments('bing'),by = c('item1'='word')) %>%
    arrange(desc(correlation)) %>%
    head(5)
}

word_cor_snt_negtv <- bind_rows(dt) %>%
  distinct(item1,item2,correlation,sentiment)
```

As we only filter out single words with low frequency, it doesn't guarantee the words pair are highly frequent. Remember our friends the `word pair frequency` dataframe? It is time to meet him.

```
# only filter for the word pairs which has frequency greater than 4
word_cor_snt_negtv_high_freq <- word_cor_snt_negtv %>%
  semi_join(word_cnt_snt_negtv %>%  filter(n >=4))

word_cor_snt_negtv_high_freq
```

```
## # A tibble: 26 x 4
##    item1      item2    correlation sentiment
##    <chr>      <chr>          <dbl> <chr>
##   1 outrageous prices         0.406 negative
##   2 smoke      smoking        0.329 negative
##   3 smoke      smoking        0.329 negative
##   4 smoke      smoking        0.329 negative
##   5 free       internet       0.293 positive
##   6 smoke      smoking        0.329 negative
##   7 free       internet       0.293 positive
##   8 smoke      casino         0.286 negative
##   9 smoke      smoking        0.329 negative
##  10 free       internet       0.293 positive
## # ... with 16 more rows
```

As we apply different `n filtering`, then there are some item pairs having different correlation, just use `distinct()` function to remove it!

```
word_cor_snt_negtv_high_freq %>%  distinct(item1,item2)
```

```
## # A tibble: 4 x 2
##   item1      item2
##   <chr>      <chr>
## 1 outrageous prices
## 2 smoke      smoking
## 3 free       internet
## 4 smoke      casino
```

Just a litte check and we can see some other problems: `"outrageour prices"`, `"smoking"` , especially in `"casino"` and there are no `"free internet"`. Very informative, isn't it?

We can see that `word count` and `word correlation` methods can supplement each others very well. We can modify our previous formulas:

$Importantwordpair = wordcount + join(highwordcount, wordcor)$

Apply the same method in **3.2.2** for raising ideas and we might get stuck in a never-ending interesting analysis worlds!

Here are some notes/summary for working with `word pairs`:
* For the analysis being more useful, we should filter 1 item with **sentiments**. For the remaining item, we can filter out some *"not interesting words"* (eg: filter for negations word) .Words will always go with pairs, eg :`a-b, b-a`–> apply 1 kind of filter for each column is sufficient!
* To avoid **noise**, pair of words should be in the **same sentence** (`sentence id`). I've discussed it already.
* We normally apply 2 filter for `word correlation`, 1 filter is to remove low frequency **single words**, the other is to remove low frequency **word pairs**. They are very different!
* We use `group by` for `word correlation` but it is just for filtering out word with low frequency(count each word–> filter for those with low frequency). Never use `summarisation/count()` after that (I mistakently do it for my first time just as a habit of using `summarisation` after `group by`). It stills give the results but the results are incorrect!

Last words

We are nearly to the end of the (very long) articles! I strongly believe that, although my method is very simple with (almost) no modelling, it is not necessarily a primitive methods:

- It use different tools and combine them flexibly. No tools are perfect and comprehensive enough.
- It use the tidy dataframe methods which are very prevalent in R! It is very similar to Excel and SQL which I beleve that many data analyst are used to.
- As it is so simple, it avoid the situation of modelling when analyst just put everythings in blackbox and hope something happen. With my method, we can have full control over what we are doing!
- It allows maximum and never-ending combination. We will never get stuck in one analysis. It is not restricted by technique, but ideas. It highly correlate with data analysis. The most important factor is idea, not technique. Since many data analyst comes from non-programming occupation, a method focusing on ideas, not technique might be more suitable!

I have to admit that, despite being discussed in this articles, I find that the `part-of-speech` and `stemming` parts are very unsatisfactory and highly vunerable to the inaccuracy. The `emoticon` part is objectively unsatisfactory, too, due to the possibility of users using incorrect,sarcastic emoji is really high. I hope that there will be suggestion to improve them, alongside other parts.

This articles is mostly based on the ideas in the book `Tidy Text Mining with R` with supplemental ideas coming from `qdap` and `sentimentr` library. For more details, go to the original sources.

Good luck!

Reference

Text mining with R, Julia Silge and David Robinson https://www.tidytextmining.com/.

Qdap vignette, Tyler Rinker http://trinker.github.io/qdap/vignettes/qdap_vignette.html.

Sentimentr vignettes, Tyler Rinker https://github.com/trinker/sentimentr.

Widyr packages, David Robinson https://cran.r-project.org/web/packages/widyr/widyr.pdf.

R for data science, Hadley Wickham http://r4ds.had.co.nz/.

Hunspell vignette, Jeroen Ooms https://cran.r-project.org/web/packages/hunspell/vignettes/intro.html.

RDRPOSTagger, Dat Quoc Nguyen, Dai Quoc Nguyen, Dang Duc Pham and Son Bao Pham, https://github.com/bnosac/RDRPOSTagger.

Arnold, Taylor B. 2016. cleanNLP: A Tidy Data Model for Natural Language Processing. https://cran.r-project.org/package=cleanNLP.

Arnold, Taylor, and Lauren Tilton. 2016. coreNLP: Wrappers Around Stanford CoreNLP Tools. https://cran.r-project.org/package=coreNLP.