American University - DATA 312

Instructions for Homework 2-C = Project 2

Unit 2 – Text Mining
Part C - Word frequency analysis

Pre-requisite work:
Please complete Homework 2-B before starting this assignment. (No new libraries are required.)

Work Process:
This assignment largely follows Chapter 3 of *Text mining with R*

https://www.tidytextmining.com/tfidf.html

and then has you circle back through the HW2-A and HW2-B to perform a careful analysis of a set of texts written by a single author of your choosing. This particular assignment focuses on two ideas that are related: Zipf's law and tf-idf analysis. Both of these are somewhat more refined examinations of word frequencies that we encountered in HW2-A. Specifically, the use of a stop word lexicon is a bit crude. While it may capture the main words that we don't care too much about, if the language in the text differs substantially from the stop word lexicon, you'll get poor results. A case in point is the modern English stop word lexicon included in the tidytext library versus the **early** modern English used by Shakespeare.

The idea of Zipf's law is that if you sort words from most frequent to least, then the actual frequency of a word is inversely proportional to its position (rank) in that sorted list. Very frequent words evidently appear earlier in the list, but Zipf's law **predicts** that frequency by asserting a formula for it. As it happens, Zipf's law tends to work well for words that are not too common nor too infrequent. Usually, people like to plot frequency of a word versus its position in the list on a log-log plot, which is to say that you plot log(frequency) versus log(rank). When you do this, Zipf says you should see a line with slope -1.

The other topic, tf-idf analysis, is a bit more heuristic. While we found that word probability which people usually call "tf" = term frequency,

tf = (number of times a word appears in a document / total words in document),

works nicely **after** you remove stop words, it would be nice if you didn't have to remove stop words. A trick that usually works is to down-weight words that are common across all documents, because this emphasizes words that are both frequent and specific to a given document. One way to do this is with the *inverse document frequency*, given by the formula

idf = log(total number of documents / number of documents containing word).

Multiply them together, to get the quantity we care about

tf_idf = tf * idf,

If you sort words in descending order of their tf_idf value, this works pretty well at picking out the words you want without finding any stop words.

OK... let's try these!

1. Load your libraries:

```
library(tidyverse)
library(gutenbergr)
library(tidytext)
```

2. As in HW2-B, load the "Shakespeare corpus", most of Shakespeare's works...

```
shakespeare_corpus<-read_csv('shakespeare_gutenberg.csv')
mirror<-'http://gutenberg.readingroo.ms/'
book <- gutenberg_download(shakespeare_corpus$`Gutenberg ID`[1], mirror=mirror)
for(i in 2:nrow(shakespeare_corpus)){
  book<-book %>% add_row(gutenberg_download(shakespeare_corpus$`Gutenberg ID`[i],
mirror=mirror))
}
```

Note that I've put the mirror url in a variable called `mirror`, and that this doesn't cause any problems in the gutenberg_download. If you find that the mirror I used is down for some reason, feel free to try another!

Now you have a data frame `book` with most of Shakespeare loaded into it, assuming this all went well. Adjust mirrors as needed.

3. Let's collect up all word frequencies again. But unlike what we've done previously, we're not going to do any stop word removal... Stop words are part of the analysis this time!

```
shakespeare_words <- book %>%
  unnest_tokens(word,text) %>%
  count(gutenberg_id,word,sort=TRUE) %>%
  inner_join(shakespeare_corpus,by=c(gutenberg_id="Gutenberg ID"))
```

At this point, it's helpful to recognize that the order of the last two commands matters! If instead you switch the count and the inner_join, you'll end up with a very different result. Figure out what happens!

4. If you look at what we're doing in the introduction of this assignment, you'll notice that we usually care about word frequencies normalized by the number of words in the document. This means we ought to add a column to our shakespeare_words dataframe with that information. You can do this in a bunch of different ways, but I like to take two steps. Compute the total number of words:

```
total_words <- shakespeare_words %>%
  group_by(Name) %>%
  summarize(total=sum(n))
```

(Aside: since tidyverse was written by a Brit living in the US, you can usually use British or American English spelling... That is, summarize = summarise... colour = color, and so on.)

Notice that the total_words is a data frame that still has most of the same columns as shakespeare_words, so it's easy to join that back into the original data frame:

```
shakespeare_words <- left_join(shakespeare_words,total_words)
```

A word about joins. We've thus far been using two different kinds: left_join (in HW1A) and inner_join (in HW2A). What's the difference? Both work on two data frames, let's call them d1 and d2. We specify one column as the "key" column, whose values we'll use for matching. If d1 and d2 have the same keys, then the two joins are the same. But if d1 has some keys that d2 doesn't, inner_join only keeps the rows that match those keys. On the other hand left_join(d1,d2) always keeps all of d1's rows (and all its keys) -- dropping in NAs when there's missing data. As you might imagine, there's a right_join(d1,d2) that keeps all of d2's rows. Finally, there's a full_join(d1,d2) that uses the union of both sets of rows.

As an example, suppose that you are a teacher assembling your gradebook. You have two tables:

- `attendance`, which has columns for student ID, student name, and each class date
- `homework`, which has columns for student ID and each homework

The problem is that some of the students who attended initially dropped the class (and so don't appear in the homework), and a few students are auditing, and so only do the homework. Then...

- left_join(attendance,homework) lists all the students who ever attended in person, including those students who dropped,
- right_join(attendance,homework) lists all the students who submitted any homework, but ignores students who dropped,
- inner_join(attendance,homework) lists only the students who show up to class **and** submit homework, and
- full_join(attendance,homework) lists everyone.

5. OK, let's draw a word frequency histogram. We first did this in HW2A, and that kind of thing will work just fine for what it's worth. But let's try to make a more informative plot. First of all, Shakespeare wrote many things, and plots that have a separate subplot (a "facet" in tidyverse-language) for each work tend to get cluttered. It's therefore smart to pick out just some of them to examine. That's what the filter(Type=="Tragedy") does below ... it picks out just the tragedies. Let's start simple:

```
shakespeare_words %>%
  filter(Type=='Tragedy') %>%
  ggplot(aes(n/total,fill=Name)) + # this implies that the y axis is y=n/total
  geom_histogram() # The bars are word frequencies, normalized to be between 0 and 1
```

OK that works, but it's kind of ugly. Notice that the "fill" above is being interpreted as color, where you should recall that in the shakespeare_words data frame, `Name` refers to the title of the work.

6. Actually, the plot in Step 5 is **really** ugly and not very useful. Maybe if we split it out with a separate subplot (facet) for each book it'll look better? That's easy. Just add

```
+ facet_wrap(~Name,ncol=2)
```

to the above and try again. The "~" usually means "versus", which in this case is reasonable. The "ncol" makes that many columns of plots. One thing you might notice is that the scales on each plot are the same; sometimes that's good, and sometimes that's not. Check out the "scales=" option on facet_wrap and see what the other choices do!

Also, you can say geom_histogram(show.legend=FALSE) to save space...

7. **And**, my version of RStudio reminds me that

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

You're probably thinking "**I never called** `stat_bin()`... so why is it complaining?" Well, the answer is that geom_histogram() did call stat_bin(). Actually, the different geoms and the different stats go together by default. You can change them, but if you don't say anything, there are defaults.

In any case, you can silence the error by saying geom_histogram(bins=30) or some other choice, like geom_histogram(binwidth=1), because that's what it's suggesting to you. Experiment.

8. One of the main points of this assignment is to explore Zipf's law, so let's do that. As a preliminary, we need to rank order words by their frequencies in each work. Because group_by() is magic, it changes the behavior of a number of useful tidyverse commands, like the sum() we ran across in previous assignments. The function row_number() is another such function that pays attention to grouping, which means that if you make a new column filled with row_number(), the numbers **do not** go 1 to the end if you have any groups. They start at 1 **in each group**, and count up **within groups**. That is super handy!

We can get the ranking we want pretty easily by calling mutate to make a new column:

```
words_by_rank <- shakespeare_words %>%
  group_by(Name) %>%
  mutate(rank=row_number())
```

Check for yourself with View that this did indeed make the new column... in which the rank does not go up by 1.

But wait! How did just listing the row number make the ranking correct? What if the rows were in the wrong order?!?!? Good question indeed. Actually, look back to step 3... we asked count to sort our rows :-)

9. We need word frequencies too because Zipf's law compares rank and frequency. Easily done, just extend the mutate command in Step 8 to add that as a new column. Also, let's ungroup once we're done... don't want that to confuse us later!

```
words_by_rank <- shakespeare_words %>%
  group_by(Name) %>%
  mutate(rank=row_number(), frequency=n/total) %>%
  ungroup()
```

10. The classic Zipf law plot is easy now! We just plot words_by_rank$rank against words_by_rank$frequency. On a log-log plot. Right. ggplot is nice that way, and it suggests that you do all your data wrangling first, then worry about plotting. So:

```
words_by_rank %>%
  ggplot(aes(rank,frequency,color=Name))+
  geom_line(show.legend=FALSE) + # When you have many plots, turn off the legend!
  scale_x_log10() +
  scale_y_log10()
```

basically does just what we want! Note that we set up our data (first line), figure out what to plot (second line), figure out how we want it plotted (third line), and then tweak the plot options (the last two lines). I make no claims that this plot is truly amazing... so you should spend a little time making it better!

11. Now that you have a wonderful plot, look carefully at it. Especially the middle part of it looks to be pretty flat. Maybe it's a straight line? If you recall STAT-202, you should remember linear regression! R makes linear regression a snap:

```
lm(log10(frequency) ~ log10(rank), data = words_by_rank)
```

Don't you wish you had that before? Let's unpack it a little. The data are obviously the data we're using. (Though, annoyingly, since lm is base R and not tidyverse, data is not the first input. You can't %>% your data into lm like one really should be able to. :-(Oh well.) The "~" is "versus", and notice that you can posit a formula for the linear regression including some basic algebraic functions, like log10(). So for instance, you don't need to have a log regression function!

Look closely at the slope that the above command reports (the "log10(frequency)"); you'll see it's about -1. Zipf's law claims that this ought to be exactly -1 for natural language, so that's not too far off. (Zipf's law is an **empirical** law, and while it has some justification in information theory, it's not the end of the story...)

12. If you call the linear regression as above, it prints out some answers for you. But if you want those answers to be added to our plot (as I'm sure you very much want to have happen), we need to store the regression output in a variable:

```
lfit<-lm(log10(frequency) ~ log10(rank), data = words_by_rank)
```

Dig a little bit into this lfit variable! It's got a column labeled `coefficients` that's most useful. The `coefficients` column has two *labeled rows* (remember the column_to_rownames() command in HW2B? Yeah, that thing) called somewhat awkwardly "(Intercept)" and "log10(frequency)". As you might imagine, the latter of these depends exactly on the linear fit you asked for. Awkward indeed. Beware.

In algebra class, you probably remember that the equation of a line is y = mx + b, right? So to find a line you need the b and the m. What the linear regression contains are these two things:

b is stored in lfit$coefficients["(Intercept)"] and you can also get it using coef(lfit)[1]

and

```
m is stored in lfit$coeffcients["log10(rank)"] and you can also get it using coef(lfit)[2]
```

Gross, but it works, and it's what we need.

13. To add our linear fit to the Zipf's law plot, we need to add another geom. For this, geom_abline() is the right thing, and it actually is a bit strange, because it almost ignores the scale_x_log10() and scale_y_log10() commands that we used in Step 10, but not completely. A line plotted on a log-log plot is close to (but actually usually not) a line when plotted on a linear-linear plot because the intercept messes things up. But geom_abline() has two inputs for slope (m) and intercept (b) that override this kind of behavior, and it just works. It's a tidyverse design choice: log scaling applies to the data, not to the geom_abline... (It's funny; I coded it up, it worked... and then later came back confused because mathematically it **ought not** work. Whatever.)

```
words_by_rank %>%
  ggplot(aes(rank,frequency,color=Name))+
  geom_line(show.legend=FALSE) +
  scale_x_log10() +
  scale_y_log10() +
  geom_abline(intercept=lfit$coefficients["(Intercept)"],
          slope=lfit$coefficients["log10(rank)"])
```

You might want to tweak some of the parameters of the geom_abline(), for instance, its color or linetype.

In any case, the takeaway is a pretty close linear fit! Shakespeare does a pretty good job satisfying Zipf's law. The plot falls away from the linear fit at either end a bit. This is typical.

14. Let's pivot to tf-idf analysis! While this sounds pretty sophisticated, it's actually really easy because tidytext provides the tools out-of-the-box:

```
shakespeare_tf_idf <- shakespeare_words %>%
  bind_tf_idf(word,Name,n)
```

Go ahead and

```
View(shakespeare_tf_idf)
```

to see how it looks!
But I'm sure you'd like your words sorted by tf_idf, since that was the whole point, right? Yes, I would like that! Easily done:

```
shakespeare_tf_idf <- shakespeare_words %>%
  bind_tf_idf(word,Name,n) %>%
  arrange(desc(tf_idf))
```

When you View(shakespeare_tf_idf), you'll see the most important words at the top of the list -- not just the most common, but the most common words that are specific to those particular works.

Common words that are common to all documents don't usually make the top of the list! It's pretty slick that stop words percolated downwards in the list without us having to get rid of them.

15. Now if you want a smaller data frame that just lists the top 10 words in each work, that's easy with some more group_by magic:

```
shakespeare_tf_idf %>%
  group_by(Name) %>%
  slice_max(tf_idf, n=10)
```

16. Let's finish with a plot of the top few words from each of Shakespeare's comedies:

```
shakespeare_tf_idf %>%
  filter(Type=='Comedy') %>%
  group_by(Name) %>%
  slice_max(tf_idf, n=10) %>%
  ggplot(aes(tf_idf,word,fill=Name)) +
  geom_col(show.legend=FALSE)+
  facet_wrap(~Name)
```

You can tweak this quite a bit! One thing that makes for a nicer plot is if you turn off the default that the facets have the same scales… Replace the last line with facet_wrap(~Name,scales="free").

Someone along the line also screwed up the word order, which you can repair with the oddly-named fct_reorder function, replacing the ggplot line with

```
ggplot(aes(tf_idf,fct_reorder(word,tf_idf),fill=Name))
```

Finally, you might consider turning off legends in geom_col(), since that information is already present in the title of each facet. Hint: we've done that before in this assignment!

Assignment

Work the answers to the following questions into an Rmarkdown document and knit that into a PDF. Please submit both your .Rmd and .pdf file on Canvas. If you've forgotten how to do this, please review HW1-C for instructions to create an Rmarkdown document, or consult your instructor. Remember that your goal is to **communicate data effectively**, so here are a few pointers:

- Make sure that your formatting looks good: include your name, project title, etc.,
- Use complete sentences,
- I'm more interested in having a correct, detailed, document from which I can follow all your steps than I am interested in beautiful graphics, so
- Provide careful explanations of what analyses you are doing,
- Explain why you've chosen those specific analyses, and
- Explain how to interpret the results.

Here are the questions to be addressed for this particular Project:

1. Select an author who has written at least 4 fictional works available on Project Gutenberg. Please do not select William Shakespeare or Jane Austen!

2. List all the Gutenberg IDs for that author's works that you plan to use. Use a loop in R to download all the files into a tidy data frame (tibble). You might find it helpful to do a bit of cleaning before proceeding.

3. Using the sentiment analyses from HW2B, classify the author's works according to their sentiment. How does this align with the genre of the works? Can you use a statistical test (chi-squared, ANOVA, or linear regression) to justify your claim? Hint: the *standard residuals* discussed in Step 15 of HW2B might be your best shot if the author's works don't fall neatly into "types" like tragedy or comedy. In that case, just group by work rather than type.

4. Produce at least one word cloud for the most common words used by your author. Explain why these words play an important role. It might be useful to compare word clouds across individual works, rather than all together.

5. Does your author's word usage patterns follow Zipf's law? Produce a word frequency versus rank plot, and explore some of the outliers.

6. Use the tf-idf analysis discussed in the instructions above to identify the key words in each of your author's works. Be sure to explain what the implications of these analyses are. Discuss any words with unusually high tf-idf, any commonalities across works, or any other things that seem unusual.