This document will walk you through the process of doing topic modeling with a corpus of tweets associated with the April 2016 General Conference of the LDS (Mormon) Church. If you have any questions about this process, feel free to email me!

1) Prep Work

Before we get into the good stuff, there are a few housekeeping items we have to take care of.

1.1) Install R

Unless you already have R installed, you'll need to install it from here.

1.2) Install RStudio

Likewise, unless you have R Studio installed, you'll need to install it from here.

1.3) Create a folder for this project

To make things easier down the road, you'll want to create a folder where you can put some of the materials for this project. I would recommend putting it somewhere pretty permanent so that you don't have to change any filepaths that you'll be putting in your code.

1.4) Download files

If you're planning on following along line by line, you'll need the same data that I used when putting this document together. You can find the data file I used here. To access the file, you'll need to sign in with a Google Account and request permission to access the file. Please just include a brief note mentioning that you'll be using this for the topic modeling walkthrough.

You'll also want a *stopword list*—more on this later. You can download the one I used from here.

Go ahead and put these files in the folder that you created in the previous step.

1.5) Download reference book (optional)

Finally, while it's not necessary for this tutorial, you should know that I've been relying heavily on Jockers's *Text Analysis with R for Students of Literature* for developing this code. This link will allow you to download your own copy of the book through the MSU Library.

2) Setting up RStudio

Now that we have everything ready, we'll need to start setting things up in RStudio.

2.1) Create a new R Script

To do this, open RStudio, click on *File*, go to *New File*, and click on *R Script*. I would recommend saving the file into the folder you created right away.

2.2) Get the lay of the land in RStudio

Now that you've created a new R Script in RStudio, there should be several "panes" in your screen. You can learn about all of the panes here, but the most important for us right now are the pane that represents the file you've just created and the *Console* pane. We'll be entering and executing code in the file pane and seeing the results in the Console pane.

2.2) Learn how to enter code into the R Script

In the following steps of this walkthrough, you'll see snippets of code that I used to topic model this data. To follow along, you'll want to copy that code into the new script you've created. You also have the ability to copy it into the *Console*, but I would recommend against that. Any code you enter there won't be saved, and you'll have to either repeatedly copy it or re-write it if you ever need it to run again.

2.3) Learn how to execute code in RStudio

RStudio isn't the only way to use R, but one of the big advantages associated with it is that you can run just parts of a program. That is, you don't have to run the whole thing every single time.

To run code in RStudio, make sure that your cursor is on the line of code that you'd like to run and press Ctrl+Enter (Windows) or Cmd+Enter (Mac). If you want to run several lines of code at once, use your mouse to highlight everything you'd like to run before hitting Ctrl+Enter / Cmd+Enter. You'll see the results of the code you've executed (including any possible errors) in the *Console* pane of RStudio.

2.4) Set up a workspace

We have a couple of files that we need to work with throughout this process, so we need to tell R where to look for those files (and where to output any files that we create here). We can do that by setting up a *workspace*.

```
## ------------------------------------------
## set up workspace
## ------------------------------------------
```

This first chunk right here actually isn't executable code—in R, we can use the `#` symbol to "comment out" anything that we'd like to write for our own reference but that we don't want to be executed. I find this helpful for distinguishing different parts of a script, which makes it easier to find what part I need to be paying attention to. I'll occasionally include comments and comment headers where I would want to put them, but don't feel like you need to copy them exactly.

So, with that header in place, here's the code I actually need to execute to set up my workspace. It boils down to passing a particular filepath to the `setwd()` function.

```
setwd("/Users/spgreenhalgh/Google Drive/5) Researcher/#ldsconf/DH Workshop")
```

Obviously, you won't want to use the same filepath; instead, you'll want to direct R to the folder on your computer that you set up in step 1.3.

If you're on a Mac, you can get this path pretty easily by right-clicking on **a file within** the folder you have in mind, clicking on *Get Info*, and copying everything in the *Where* field. That stuff doesn't look like a filepath, but if you paste it into R, it will automagically become one.

If you're on a PC, you should be able to right-click on **the folder** you have in mind, click *Copy as path*, and paste it into R.

2.5) Install packages

The power of R comes not from what's already baked into it but from the ease of adding new functions and extending its ability by installing and loading *packages*. Before getting any further in this process, let's install and load all of the packages that we'll be using.

```
## ------------------------------------------
## installing and loading packages
## ------------------------------------------
```

First, as above, I'm going to include a comment to set this code apart.

```
install.packages("mallet")
install.packages("wordcloud")
```

So, we're installing two packages here. The first one is *mallet*, which is an R implementation of the MALLET

Java tool. The mallet package isn't the only R package that will do topic modeling, but since MALLET is widely used in DH communities, it's not a bad place to start.

The second package is *wordcloud*, which will come in handy later.

2.6) Install Java (as needed)

When you install the *mallet* package, you will likely be prompted to install a certain build of Java on your computer. Please follow the instructions to do so.

If you do not get a prompt, please try the following - install the latest version of Java from the official Java website - install the *Java Development Kit* (JKD) from the official Java website

2.7) Load R packages

```
library(mallet)
library(wordcloud)
```

Now that we've installed both packages, we need to load them, using the *library* function. Notice that when we install packages, we put them between quotation marks, but we don't do have to do that when we load them.

At this point, I would actually suggest deleting the earlier snippet of code about installing packages; once they've been installed on your machine, you'll only need to load them when you want to use them. There's usually no reason to re-install packages.

2.8) Import data

Earlier, we told R where to look for any files that we'd be importing, but we haven't actually imported them yet. Let's do that now:

```
## -----------------------------------------
## import data
## -----------------------------------------
```

The code below is going to import a *CSV* (a *comma-separated values* file, which can be thought of as a simple kind of spreadsheet) and assign it to an R object named "tweetData." In R, we kind of represent that backwards. First, we identify the name of the object we're assigning something to, then we include an *arrow* ("<-"), and finally we list what we're assigning to that object. That "what" can be a number, a string, or (most frequently in this code) the result of a *function*.

If you adapt this for your own data, keep in mind that you'll definitely need to change the name of the file that you're importing (make sure to download it and move it to your workspace folder first!). You may also wish to (but don't have to) change the name of the object that you're creating. You probably won't need to change anything else (the stringsAsFactors argument is telling R to treat text as text rather than a series of text-based categorical variables—not doing this would cause us big problems later).

```
tweetData <- read.csv("ldsconfData.csv", stringsAsFactors = FALSE)
```

2.9) Examine data

If you opened this CSV in Excel (or another program) prior to importing it, you'll already have an idea of what your data looks like. If you didn't, though, we can use the *str* function to get a quick idea of what's in here:

```
str(tweetData)
```

First, we can see that the object created from this CSV is a *data.frame*, which is one of the basic classes of objects in R. A data.frame is similar to a spreadsheet in that it's composed of columns and rows—R users typically spend a **lot** of time working with data.frames.

Furthermore, we can see that this data.frame contains over 24,000 different observations (in this case, tweets), each of which is described by two variables: *text*, which gives us the content of the tweet, and *status_url*, which is the link to the tweet on Twitter and can serve as a kind of identifier for us (it may also help provide context for any single tweet).

3) Clean text

```
## ----------------------------------------
## clean text
## ----------------------------------------
```

Computers can be very literal-minded: Whereas a human might understand "cheese", "Cheese", and "cheese!" to be the same word, a computer program typically won't. Plus, we might be interested in removing particular chunks of text if they aren't useful to us. That's what we'll be doing in this section. There are four specific changes that we'll be making. After each step, we'll display the first twenty tweets to show how they're changing over time. Let's take a baseline look at them now. Notice how I use brackets to define a subset of what I'm looking at.

```
tweetData$text[1:20]
```

3.1) Make text lower case

First, let's take all text and make it lower case. One of the functions we'll use later will actually also do this for us, but this is a handy function to know, so redundancy can't hurt for our purposes. This will ensure that our program treats "cheese" and "Cheese" as the same word for topic modeling purposes.

Notice the way that we're accessing this data. tweetData is the name of our data.frame, but we're only interested in messing with one of the columns, so we specify the column as well (using *$* to separate the two names).

```
tweetData$text <- tolower(tweetData$text)
tweetData$text[1:20]
```

3.2) Remove "#ldsconf"

Second, let's remove references to "#ldsconf". Since this appears in every single tweet (it's how the tweets were singled out for collection), it's more likely to get in the way than to reveal anything amazing. The *gsub* function is used for pattern matching and replacement. First, we tell it a pattern to look for, then we tell it a second pattern to replace the first one with. Finally, we tell it where to be looking for that pattern.

```
tweetData$text <- gsub("#ldsconf", "", tweetData$text)
tweetData$text[1:20]
```

3.3) Remove links

Third, URLs are not going to be our friend for this project, so let's get rid of them. We can't possibly specify all the URLs that are in our data, so we're going to use *regular expressions* (ways of representing *patterns* of text rather than literal sequences of text) to search for things that look like URLs. For example, the regular expression below looks for a sequence of text that starts with this pattern: either an "f" or an "ht", "tp", maybe an "s", "://" ... If that looks familiar, it's because that's how URLs start. The rest of the expression covers how URLs end, too, so searching for this regular expression will find all URLs in our text. Then, we replace all of those URLs with a blank string (i.e., nothing; we're essentially deleting them). If you want to learn more about regular expressions, check out this website.

```
tweetData$text <- gsub(" ?(f|ht)(tp)(s?)(://)(.*)[.|/](.*)", "", tweetData$text)
tweetData$text[1:20]
```

3.4) Remove punctuation

Finally, we'll use another regular expression to remove most punctuation. Technically, what we're doing is looking for anything that isn't an alphanumeric character (i.e., a-z, 0-9), a space, an apostrophe, a hash symbol, or an at symbol and replacing it with a space. The result is removing most punctuation, though we are keeping a few specific kinds. Apostrophes are useful for distinguishing "well" from "we'll", and #s and @s play a big role in Twitter, so that will help us keep some context. Also, you may notice that we're replacing these with spaces rather than with blank strings (which effectively delete them outright). This is tricky, because we might want to consider words on either side of a hyphen to be a single word (by deleting the space between them)... but we don't want that to be the case for words on either side of a dash or an ellipsis.

```
tweetData$text <- gsub("[^[:alnum:][:space:]'|#|@]", " ", tweetData$text)
tweetData$text[1:20]
```

4) Set up the topic model

```
## ----------------------------------------
## topic model setup
## ----------------------------------------
```

Now that we've done all of this prep work, it's time to start the actual topic modeling process.

4.1) Import Java-friendly data

As mentioned earlier, the package we're using for topic modeling uses Java rather than R, so the first thing we need to do is to import all of the data in a way that the package understands. We'll use the mallet.import function, and we need to feed it five arguments:

- the name of a list of identifiers for each text
- the name of a list containing each text
- the name of a file containing a *stoplist*
- whether we want to preserve upper case and lower case distinctions
- a regular expression that essentially defines what a word is for the purposes of this topic modeling

So, we already have a handy data.frame that contains a list of identifiers and a list of texts, so we just need to indicate the data.frame and the appropriate column (separated by a $) for each one of those.

There's also a stoplist that you can download from the workshop folder. First, though, what's a stoplist? Do you remember how I removed the #ldsconf hashtag from this data set because it showed up so often? Well, there are plenty of regular English words that are going to show up in regular English text ("a", "an", and "the" come to mind, though there are **plenty** of others, too). A stoplist is a list of *stopwords*, these common words that are more likely to muck up our analysis than contribute any powerful insights. Using a stoplist essentially filters these words out of our text, helping us concentrate on the most important stuff. Our stoplist is a combination of an English stoplist provided by Jockers and—because I know there's some Spanish in my data set—a Spanish stoplist from the Python Natural Language Toolkit, another powerful tool for text analysis.

Our stoplist is called *stoplist.csv*, so feeding that information to our function is as simple as typing that name in the right place. You'll notice that we provide the name of the stoplist in quotes, but we're giving the names of the lists of identifiers and texts without quotes. That's because the lists of identifiers and texts that we're providing are already R objects that we've imported into the *R Environment* we've been working in. That's not the case for the stopword list, so we have to treat it as a snippet of text rather than an object name that R will recognize. The function knows, though, that this snippet of text corresponds to a file in the R workspace.

As far as upper and lower case distinctions go, we've already made everything lower case, so it doesn't really matter.

Finally, to "define a word," we'll include a regular expression that tells the function to look for any combination of letters but to preserve apostrophes, at symbols, and hashtags.

```
ldsconfInstances <- mallet.import(tweetData$status_url, tweetData$text, "stoplist.csv
", preserve.case = FALSE, token.regexp = "[\\p{L}'|@|#]+")
```

4.2) Create trainer object

Once we've imported all of our data in, we can create a trainer object, which we can think of as an empty mold for our final product. That is, we're defining the final *shape*, but it hasn't been filled in yet. Since we're defining the final shape, though, it's important that we define how many topics we expect to see at the end of this process—we do so using the *num.topics* argument. The process of identiying the correct number of topics to look for is complicated enough that we won't get into it in this walkthrough; instead, we'll use the semi-scientific method of relying on our personal understanding of the subject matter, our experience with the data, plenty of trial and error, and our *gut* to come up with a correct number of topics.

So, while we're currently listing 50 different topics for #ldsconf, that's the result of a lot of fiddling. I initially tried 10 topics, tried to see if I could get any lower, made my way back up to 10, and eventually went even higher because I felt that some of the 10 were actually combinations of separate topics. As you adapt this code for the data that you're looking at, you'll want to fiddle with and re-run the code from here to the end until you find a result that you like.

```
ldsconfModel <- MalletLDA(num.topics = 50)
```

4.3 Associate trainer object with data

Then, we need to start "filling the mold," by associating this object with the data we imported into mallet earlier.

```
ldsconfModel$loadDocuments(ldsconfInstances)
```

4.4) Create the topic model

```
## -----------------------------------------
## creating topic model
## -----------------------------------------
```

Once we've done all of this, we can start the process of topic modeling. One thing that we need to do is to specify how many times we'd like our model to go through the process of determining word associations. 400 is a good number of iterations, but you can try different numbers to see how different topics result.

```
ldsconfModel$train(400)
```

5) Look at results

```
## -----------------------------------------
## looking at results
## -----------------------------------------
```

So, now we have a model! That's great, but we need to be able to see what's included in this model so that we can interpret each of our topics (and see if we need to change the number of topics).

5.1 Create data frame

The first step we need to take isn't going to be immediately helpful to us. First, we're going to create an enormous data.frame with a single column for each of the individual words and a single row for each of the individual topics. Each cell then represents the percentage each word represents of all of the words in that topic. It's easy to get lost in the weeds here, so for the time being, just concentrate on making sure you replace the names of my objects with the names of your objects.

```
ldsconfWords <- mallet.topic.words(ldsconfModel, smoothed=TRUE, normalized=TRUE)
```

5.2 Identify top words for topics

To make this a little more helpful, the *mallet.top.words* function will allow us to identify the top words for any given row in the data.frame we just created. Since each row represents a topic, this function essentially allows us to find the top words for a given topic, which is going to be critical for interpreting that topic. Let's make a data.frame for each of the topics we've come up with that shows us the top words (and top weights) for each topic. However, rather than do that individually, let's use the code below, which will create a data.frame called "topic_n" (with n representing a number) with the 100 highest-weighted words for each topic. This is a loop (which repeats a snippet of code several times), and it's set to pick up on the number of topics that exist in our previous data.frame, so as long as we're pointing it to all of the right objects, we won't have to change this loop code even if we change the number of topics—we'll just need to run it again.

```
for(i in 1:length(ldsconfWords[,1])){
  assign(paste0("topic_",i),mallet.top.words(ldsconfModel, ldsconfWords[i,], 100))
}
```

Let's take a look at one of the resulting data.frames, too.

```
topic_1
```

Based off of this alone, I'm starting to get a good feel for this topic. What do you think?

5.3 Explore wordclouds

To be honest, though, there are some easier ways of trying to interpret these topics. Let's take a look at a couple of them.

First, a wordcloud. The code below will create a wordcloud of the words in each of these data.frames and will even adjust the size of each word according to its weight within that topic. Using the wordcloud, we can look at the image to get a somewhat clearer idea of what this topic might be getting at.

```
wordcloud(topic_1$words, topic_1$weights, c(4,.8), rot.per=0, random.order=F) ##
```

5.4 Associate topics with original data

The other thing that we can do is to reassociate these topics with the original data. You can think of this first topic as "reverse-engineering" the topics we've just created. The result is a data.frame with a row for each of the original tweets and a column for each of the topics. Each cell represents the percentage of that tweet dedicated to that topic.

```
ldsconfTopics <- mallet.doc.topics(ldsconfModel, smoothed=T, normalized=T)
```

5.5 Associate topics and original data with tweet text

Unfortunately, though, the only way we have of distinguishing between those tweets at this point is by row numbers, which I don't find very intuitive. So, let's join our original data.frame to this one—that will allow us to see the tweets alongside the topics that they're associated with.

```
ldsconfTopics <- cbind(tweetData,ldsconfTopics)
```

5.6 Examine top tweets for a topic

Having done that, we can do a little finagling to ask R to show us the top ten tweets for a given topic, so that we can see these words in context. Notice that in the first line of code, I'm asking for the order of the third column (which is the first topic). If you want to change this code to grab a particular topic, you'll want to add two to the number of the topic to make sure you're grabbing the correct column.

```
topTopicTweets <- order(ldsconfTopics[,3], decreasing = TRUE)
topTopicTweets <- as.numeric(topTopicTweets[1:10])
ldsconfTopics$text[topTopicTweets]
```

So, does this give you a better idea of what the topic is about? You can go ahead and produce the wordclouds and tweet lists for other topics to see if you can interpret them. The point here isn't to make everyone a Mormon Twitter researcher; rather, I'm hoping to give you the experience of seeing different kinds of topics being produced by this process.

If you're feeling like an eager beaver, though, this page will give you some context on the different talks that took place at that conference; it can be helpful if you're wondering why a certain person's name shows up a lot. Also, you can check out a blog post that I've written about my fiddling with this data to see how I've interpreted the topics that I've found running this code.