Natural Language Processing

http://datascience.tntlab.org

Module 13

Today's Agenda

- Overview of NLP

- Process
  1. Data Wrangling
     - Already done this
  2. Pre-Processing
     - Example Code
     - General Guidelines
  3. Dataset Generation
     - Tokenization
  4. Analysis
     - Visualization
     - How to Ready for Statistical Analysis

Overview of Natural Language Processing

- NLP refers to a huge family of approaches allowing computing systems to create meaning from text
  - One you are probably very familiar with is spam detection

- Natural language processing > text analytics > text mining

- For our purposes, NLP can be defined as any set of algorithms employed in meaningful sequences to draw conclusions about text
  - Many different sequences possible, but generally follows four broad steps when working with real world data:
    1. Data wrangling/munging to create a *corpus*
    2. Text pre-processing to clean it up
    3. Dataset generation from pre-processed text (e.g., tokenization) to serve as the basis for decision-making and analysis
    4. Analysis to draw insights

Step 1: Data Wrangling/Munging

- (unstructured text) → corpus

- For the most part, utilizes the programming techniques we've been talking about all semester
  - The text needs to be converted such that it is ultimately represented as either a vector (text only) or a data frame (text plus meta-data)

- Common Sources
  - Qualitative survey data
  - Web scraping/API requests
  - Audio or video files (with some extra steps)

- Actually creating the corpus is easy with *tm:*
  - **myCorpus <- Corpus(VectorSource(*textvector*))** # or use DataframeSource()

Step 2: Pre-Processing

- Pre-processing involves converting your raw corpus into something that can be converted into a meaningful dataset for analysis

- A decision needs to be made at this point: why are you doing this?
  - If your goal is to predict something else from texts, the dataset this ultimately generates needs to have a series of meaningful IVs
    - If your sample size is *huge*, that may be enough.
    - If it isn't, you may need to simplify the data with a lexicon-based approach, a machine-learning approach, or simply cutting low-variance variables.
  - If your goal is to find patterns in the data, to identify latent classes, the dataset this ultimately generates needs the words or word patterns to be meaningful.
  - If you want to visualize anything along the way, you need to be sure that the pre-processing you do still results in meaningful variable names.
  - Between all of these, you need to determine what's linguistically meaningful for your particular application: word meaning alone, or word meaning plus something else?

General Text Modeling Approaches

- **Bag of words modeling**
  - Convert individual words or word groups into "linguistically meaningful" variables
- **Semantic modeling**
  - Derive the meanings of the phrases, clauses, sentences, etc. (i.e., semantics)
- In between: **part of speech tagging**

- Ideally, semantic modeling would involve the computer developing a human-like understanding of each sentence and then making predictions on the basis of this holistic understanding.
  - This is currently impossible, so we take shortcuts.
  - Systems like IBM Watson, Facebook DeepText, Google Cloud NLP, Microsoft Azure LUIS, and Amazon Alexa NLP already make predictions about semantics.
  - These systems vary dramatically in their capabilities.

Pre-Processing for Bag of Words

- The next step will convert whatever is in your corpus into individual variables, so you need to modify the corpus so that it can be most meaningfully converted

- **Example**
  - He likes boats.
  - He liked his boat.

  - Within a bag of words representation, are these different for your ultimate modeling purposes?

Pre-Processing for Bag of Words

- Common pre-processing steps using *qdap* and *tm*:
  - **myCorpus <- tm_map(myCorpus, content_transformer(str_to_lower))**
  - **myCorpus <- tm_map(myCorpus, content_transformer(replace_abbreviation))**
  - **myCorpus <- tm_map(myCorpus, content_transformer(replace_contraction))**
  - **myCorpus <- tm_map(myCorpus, removeNumbers)**
  - **myCorpus <- tm_map(myCorpus, removePunctuation)**
  - **myCorpus <- tm_map(myCorpus, removeWords, stopwords("en"))**
  - **myCorpus <- tm_map(myCorpus, stripWhitespace)**
  - **myCorpus <- tm_map(myCorpus, stemDocument, language="english")**

- In each case, **tm_map()** (from *tm*) systematically applies some function to myCorpus. If the function is from *tm*, you can just type it directly. If the function is from somewhere else (like *qdap* or base-R), you need **content_transformer()**

Remember Tidyverse and Utilize as Convenient

- **myCorpus <- myCorpus %>%**
  tm_map(content_transformer(str_to_lower)) %>%
  tm_map(content_transformer(replace_abbreviation)) %>%
  tm_map(content_transformer(replace_contraction)) %>%
  tm_map(removeNumbers) %>%
  tm_map(removePunctuation) %>%
  tm_map(removeWords, stopwords("en")) %>%
  tm_map(stripWhitespace) %>%
  tm_map(stemDocument, language="english")

- You might not want to do this if:
  - Some pre-processing steps take a while and others are fast.
  - You need to diagnose why your pre-processing isn't resulting in a final corpus the way you intended.

Check Pre-Processing as You Go

- Remember to build your code one line at a time, testing at each step to see what the corpus looks like, preferably with a random sample:
  - **myCorpus[[1]]$content** # within item 1
  - **myCorpus[[sample(1:length(myCorpus),1)]]$content** # within random item

- You might even build a function to do this more quickly
  - **randText <- function(x) { x[[sample(1:length(x),1)]]$content }**
  - **randText(myCorpus)**

Stemming

- Stemming in *tm* uses Porter's stemming algorithm to create stems
  - It's pretty stupid, by design.
  - Gets rid of common endings (-s, -e, -ed, -ing, -ion, etc.) but doesn't do much else.
  - Compare stems of **became** and **become**

- Still serves as a useful stem processor when you don't have any better option (e.g., due to coding ability or time).

- Will mess up your word clouds, which might or might not matter.
- Will change what sentiment analysis examines the sentiment of.

It's Easy to Over-process or Process Incorrectly

- Remember that the goal is to create a final corpus with linguistically meaningful words and word groups. Different orders of these algorithms will create different final corpora. This matters most with natural text.

- **Example**
  - "I like bread!! Accept no other fake breads!"
  - **Pre-Processing Approach 1**
    - removePunctuation: I like bread Accept no other fake breads
    - stemDocument: I like bread Accept no other fake bread
    - tolower: i like bread accept no other fake bread
    - removeWords, stopwords("en"): like bread accept fake bread
  - **Pre-Processing Approach 2**
    - stemDocument: I like bread!! Accept no other fake breads!
    - removeWords: I like bread!! Accept fake breads!
    - tolower: i like bread!! accept fake breads!
    - removePunctuation: i like bread accept fake breads

A Final Warning

- Remember your goal in pre-processing is to get reasonably consistent representations of linguistically meaningful units.

- This goal is comparable to the psychometric concept of *reliability*.

- **It does not need to be perfect, but the closer to perfect it is, the more valid your predictions will be out-of-sample**.

Step 3: Create a Dataset

- In bag of words, your dataset will be the result of *n-gram tokenization,* which refers to the conversion of words or word groups into variables.
  - An **n-gram** is a word or group of words converted into a variable.
  - A **unigram** is when you've done this with one word.
  - A **bigram** or **trigram** is when you've done this with 2-word pairs or 3-word triplets.

- Bigram and higher are usually not automatic; you specify a cutoff in terms of sparsity.

- **Example**
  - "One of the draws of human factors psychology is that human factors psychology can solve real-world problems."

  - Lots of unigrams, maybe one trigram.

Tokenizing to a Document-Term Matrix

- If you only want unigrams, it's easy:
  - **DTM <- DocumentTermMatrix(myCorpus)**

- If you want bigrams+, it's a little harder and requires *RWeka*:
  - **myTokenizer <- function(x) { NGramTokenizer(x, Weka_control(2, 3)) }**
  - **DTM <- DocumentTermMatrix(myCorpus,**
    control = list(
    tokenize = myTokenizer
)
)

- The resulting DTM will be a list, but you can coerce it to other data types.
  - **DTM.matrix <- as.matrix(DTM)**
  - **DTM_tbl <- as_tibble(as.matrix(DTM))**

Step 4: Analysis (and visualization)

- Wordclouds with *wordcloud*
  - **wordCounts <- colSums(DTM_df)**
  - **wordNames <- names(DTM_df)**
  - **wordcloud(wordNames, wordCounts, max.words=50)**

  - Use colors analytically by using **?brewer.pal** to find palette names, then:
    - **colorset <- brewer.pal(10, "Blues")[4:10]**
    - **wordCloud(wordNames, wordCounts, max.words=50, colors=colorset)**

- Use ggplot2 to visualize top words directly
  - **tibble(wordNames, wordCounts) %>%**
    arrange(desc(wordCounts)) %>%
    top_n(20) %>%
    mutate(wordNames = reorder(wordNames, wordCounts)) %>%
    ggplot(aes(x=wordNames,y=wordCounts)) + geom_col() + coord_flip()

Analytic Options You Might Want at This Point

1. Interpret this dataset directly
   - Use an unsupervised classification algorithm to create document groupings and interpret those groupings

2. Join this DF into another dataset to apply machine learning
   - Look for prediction of variables of interest from the raw word counts

3. Simplify this dataset before joining it elsewhere
   - Look for clusters of variables and create factor scores (but don't use factor analysis)
   - Using an existing lexicon, convert word frequencies into some other score of interest (e.g., sentiment)

Sentiment Analysis

- Used to predict/identify box office performance, presidential elections, natural disasters, etc.

- To get the sentiment of each text, follow a few steps:
  1. Convert your text into a corpus and do pre-processing
  2. Convert your corpus into a TDM (not a DTM)
  3. For each row, use a join to get the sentiment of that word from an existing lexicon (i.e., create a lookup table such as by using **get_sentiment()** from *tidytext*)
  4. For each row of your original DTM, convert raw word counts into word frequencies by document (so that each number means, for example: "14% of this text was the word "happy")
  5. Depending on your lexicon:
     1. If continuous, multiply lexical weights by sentiment weights by word
     2. If discrete, recode lexical words into a meaningful number and do the same. For example, positive sentiment might become +1 and negative sentiment, -1.
  6. Calculate the weighted mean by document.

For Next Time

- Week 13 Project!
  - Practice these skills in an applied project
  - Submit your project on Blackboard