An Intro to Text Analysis for Social Scientists

Patrick van Kessel, Senior Data Scientist
Pew Research Center

12/12/19 AAPOR Webinar

Agenda

- Basic principles: how to convert text into quantitative data
- Overview of common methods: a map of useful analysis tools
- Demo: text analysis in action

The role of text in social research

The role of text in social research
Why text?

- Free of assumptions
- Potential for richer insights relative to closed-format responses
- If organic, then data collection costs are often negligible

The role of text in social research
Where do I find it?

- Open-ended surveys / focus groups / transcripts / interviews
- Social media data (tweets, FB posts, etc.)
- Long-form content (articles, notes, logs, etc.)

The role of text in social research
What makes it challenging?

- Messy
    - "Data spaghetti" with little or no structure
- Sparse
    - Low information-to-data ratio (lots of hay, few needles)
- Often organic (rather than designed)
    - Can be naturally generated by people and processes
    - Often without a research use in mind

Data selection and preparation

Data selection and preparation

- Know your objective and subject matter (if needed find subject matter expert)
- Get familiar with the data
- Don’t make assumptions - **know your data, quirks and all**

Data selection and preparation
Text Acquisition and Preprepation

Select relevant data (text corpus)
- Content
- Metadata

Prepare the input file
- Determine unit of analysis
- Process text to get one document per unit of analysis

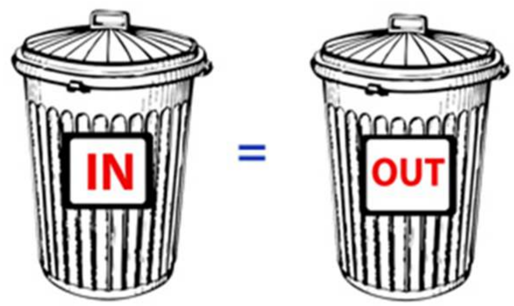


Image credit: http://www.nickmilton.com/2016/12/garbage-lessons-in-garbage-knowledge-out.html

(Pre-)Processing
Turning text into data

Turning text into data

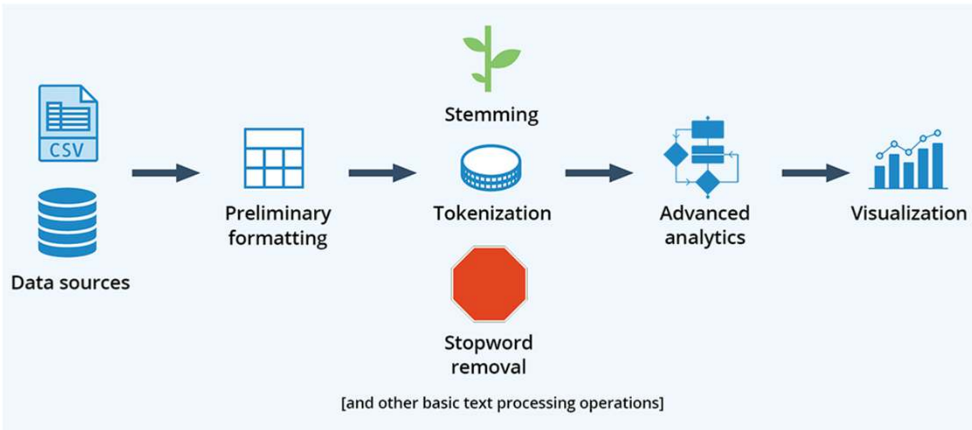


Image credit: https://www.softwareadvice.com/resources/what-is-text-analytics/

Turning text into data

- How do we sift through text and produce insight?
- Might first try searching for keywords
- How many times is “analysis” mentioned?

	Raw Documents	
1	Text **analysis** is fun	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data

- How do we sift through text and produce insight?
- Might first try searching for keywords
- How many times is “analysis” mentioned?

We missed this one

And this one too

	Raw Documents	
1	Text **analysis** is fu	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data

- Variations of words can have the same meaning but look completely different to a computer

	Raw Documents	
1	Text **analysis** is fun	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data
Regular Expressions

- A more sophisticated solution: regular expressions
- Syntax for defining string (text) patterns

	Raw Documents	
1	Text analysis is fun	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data
Regular Expressions

- Can use to search text or extract specific chunks
- Example use cases:
    - Extracting dates
    - Finding URLs
    - Identifying names/entities
- https://regex101.com/
- http://www.regexlib.com/

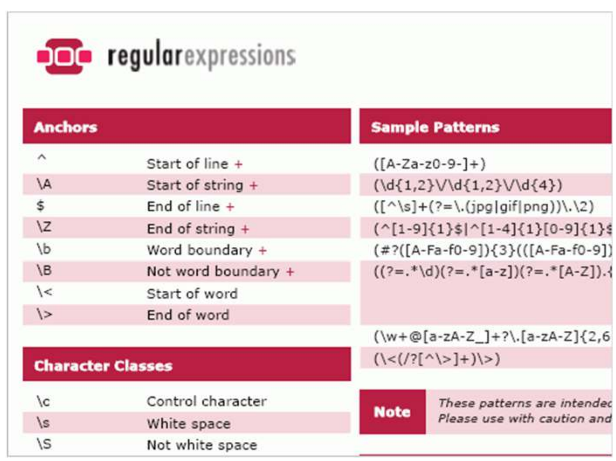


Image credit: https://www.smashingmagazine.com/2009/06/essential-guide-to-regular-expressions-tools-tutorials-and-resources/

Turning text into data
Regular Expressions

\banaly[a-z]+\b

	Raw Documents	
1	Text **analysis** is fun	
2	I enjoy **analyzing** text data	
3	Data science often involves text **analytics**	

Turning text into data
Regular Expressions

Regular expressions can be extremely powerful…

...and terrifyingly complex:

URLS: ((https?:\/\/(www\.)?)?[-a-zA-Z0-9@:%._\+~#=]{2,4096}\.[a-z]{2,6}\b([-a-zA-Z0-9@:%_\+.~#?&//=]*))
DOMAINS: (?:http[s]?\:\/\/)?(?:www(?:s?)\.)?([\w\.\-]+)(?:[\\\/](?:.+))?
MONEY: \$([0-9]{1,3}(?:(?:\,[0-9]{3})+)?(?:\.[0-9]{1,2})?)\s

Turning text into data
Pre-processing

- Great, but we can't write patterns for everything
- Words are messy and have a lot of variation
- We need to collapse semantically
- We need to *clean / pre-process*

	Raw Documents	
1	Text analysis is fun	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data
Pre-processing

- Common first steps:
  - Spell check / correct
  - Remove punctuation / expand contractions

can't -> cannot
they're -> they_are
doesn't -> does_not

	Raw Documents	Processed Documents
1	Text analysis is fun	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data
Pre-processing

- Now to collapse words with the same meaning
- We do this with *stemming* or *lemmatization*
- Break words down to their roots

	Raw Documents	Processed Documents
1	Text analysis is fun	
2	I enjoy analyzing text data	
3	Data science often involves text analytics	

Turning text into data
Pre-processing

- Stemming is more conservative
- There are many different stemmers
- Here's the Porter stemmer (1979)

	Raw Documents	Processed Documents
1	Text analysis is fun	Text analysi is fun
2	I enjoy analyzing text data	I enjoy analyz text data
3	Data science often involves text analytics	Data scienc often involv text analyt

Turning text into data
Pre-processing

- Stemming is more conservative
- There are many different stemmers
- Here's the Porter stemmer (1979)

	Raw Documents	Processed Documents
1	Text analysis is fun	Text **analysi** is fun
2	I enjoy analyzing text data	I enjoy **analyz** text data
3	Data science often involves text analytics	Data scienc often involv text **analyt**

Turning text into data
Pre-processing

- The Lancaster stemmer (1990) is newer and more aggressive
- Truncates words a LOT

	Raw Documents	Processed Documents
1	Text analysis is fun	text **analys** is fun
2	I enjoy analyzing text data	I enjoy **analys** text dat
3	Data science often involves text analytics	dat sci oft involv text **analys**

Turning text into data
Pre-processing

- Lemmatization uses linguistic relationships and parts of speech to collapse words down to their root form - so you get actual words ("lemma"), not stems
- WordNet Lemmatizer

	Raw Documents	Processed Documents
1	Text analysis is fun	text analysis is fun
2	I enjoy analyzing text data	I enjoy analyze text data
3	Data science often involves text analytics	data science often involve text analytics

Turning text into data
Pre-processing

- Picking the right method depends on how much you want to preserve nuance or collapse meaning
- We'll stick with Lancaster

	Raw Documents	Processed Documents
1	Text analysis is fun	text analys is fun
2	I enjoy analyzing text data	I enjoy analys text dat
3	Data science often involves text analytics	dat sci oft involv text analys

Turning text into data
Pre-processing

- Finally, we need to remove words that don't hold meaning themselves
- These are called "stopwords"
- Can expand standard stopword lists with custom words

	Raw Documents	Processed Documents
1	Text analysis is fun	text analys fun
2	I enjoy analyzing text data	enjoy analys text dat
3	Data science often involves text analytics	dat sci oft involv text analys

Turning text into data
Pre-processing

- **A word of caution: there aren't any universal rules for making pre-processing decisions**
- Do what makes sense for your data - but be cautious of the researcher degrees of freedom involved
- See:
  - Denny and Spirling, 2016. Assessing the Consequences of Text Pre-processing Decisions"
  - Denny and Spirling, 2018. "Text Preprocessing for Unsupervised Learning: Why It Matters, When It Misleads, and What to Do About It"

Turning text into data
Tokenization

- Now we need to tokenize
- Break words apart according to certain rules
- Usually breaks on whitespace and punctuation
- What's left are called "tokens"
- Single tokens or pairs of two or more tokens are called "ngrams"

Turning text into data
Tokenization

- We can express the presence of each "ngram" as a column
- This is often called a "term frequency matrix"
- Here are unigrams

text	analys	fun	enjoy	dat	sci	oft	involv
1	1	1					
1	1		1	1			
1	1			1	1	1	1

Turning text into data
Tokenization

- We can express the presence of each “ngram” as a column
- This is often called a “term frequency matrix”
- And here are bigrams

text analys	analys fun	enjoy analys	analys text	text dat	dat sci	sci oft	oft involv
1	1						
		1	1	1			
1					1	1	1

Turning text into data
Tokenization

- If we want to characterize the whole corpus, we can just look at the most frequent words
- Here’s the “term frequency matrix”:

text	analys	fun	enjoy	dat	sci	oft	involv
1	1	1					
1	1		1	1			
1	1			1	1	1	1
3	**3**	**1**	**1**	**2**	**1**	**1**	**1**

Turning text into data
TF-IDF

- But what if we want to distinguish documents from each other?
- We know these documents are about text analysis
- What makes them unique?

Image credit: https://medium.com/@imamun/creating-a-tf-idf-in-python-e43f05e4d424

Turning text into data
TF-IDF

- Divide word frequencies by the number of documents they appear in
- Down-weight words that are common; log-scale emphasizes unique words
- Several variants that add smoothing

$$\text{tf-idf} = \text{tf} \times \text{idf} \quad (1)$$

$$\text{idf}(t) = \log \frac{n+1}{\text{df}(d,t)+1} + 1 \quad (2)$$

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Image credit: https://sites.temple.edu/tudsc/2017/03/30/measuring-similarity-between-texts-in-python/tfidf-equations/
Image credit: https://medium.com/@imamun/creating-a-tf-idf-in-python-e43f05e4d424

Turning text into data
TF-IDF

- The overall distribution of words is still largely preserved
- But now we're emphasizing what makes each document unique

text	analys	fun	enjoy	dat	sci	oft	involv
1	1	2.1					
1	1		2.1	1.4			
1	1			1.4	2.1	2.1	2.1
3	**3**	**2.1**	**2.1**	**2.8**	**2.1**	**2.1**	**2.1**

Turning text into data
TF-IDF

- The overall distribution of words is still largely preserved
- But now we're emphasizing what makes each document unique
- Within each document, we're now highlighting distinctive terms

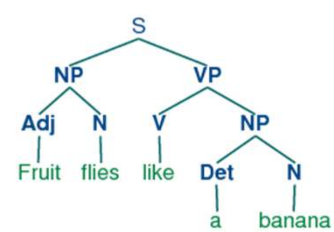
text	analys	fun	enjoy	dat	sci	oft	involv
1	1	2.1					
1	1		2.1	1.4			
1	1			1.4	2.1	2.1	2.1
3	**3**	**2.1**	**2.1**	**2.8**	**2.1**	**2.1**	**2.1**

Turning text into data
TF-IDF

- TF-IDF is an extremely common and useful way to convert text into useful quantitative features
- It's often all you need
- But there are other, more complex ways to quantify text

Turning words into numbers
Part-of-Speech Tagging

- Sometimes you care about how a word is used
- Can use pre-trained **part-of-speech (POS) taggers**
- Can also help with things like negation
  - "Happy" vs. "NOT happy"




```
>>> text = word_tokenize("And now for something completely different")
>>> nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'),
('completely', 'RB'), ('different', 'JJ')]
```

Image credit: http://nltk.sourceforge.net/doc/en/ch03.html
Image credit: https://www.nltk.org/book_1ed/ch05.html

Turning words into numbers
Named Entity Extraction

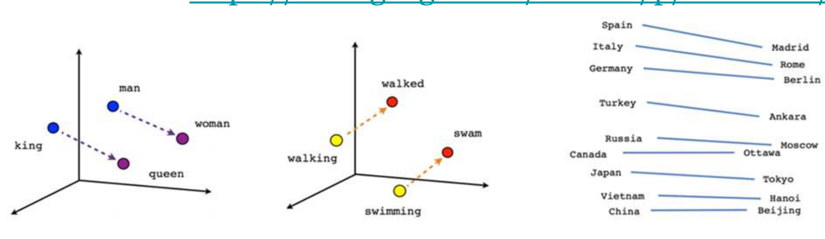
- Might also be interested in people, places, organizations, etc.
- Like POS taggers, **named entity extractors** use trained models

Figure 1: An example of NER application on an example text

Image credit: http://inspiratron.org/blog/2019/04/15/building-named-entity-recognizer-ner-using-conditional-random-fields-crf/

Turning words into numbers
Word Embeddings

- Other methods can quantify words not by frequency, but by their relation
- **Word2vec** uses a sliding window to read words and learn their relationships; each word gets a vector in N-dimensional space
- Pretrained model: https://code.google.com/archive/p/word2vec/



Male-Female Verb tense Country-Capital

Image credit: https://www.tensorflow.org/tutorials/text/word_embeddings

Analysis
Finding patterns in text data

Finding patterns in text data

Two types of approaches:

- Unsupervised NLP: automated, extracts structure from the data
  - Clustering
  - Topic modeling
  - Mutual information
- Supervised NLP: requires training data, learns to predict labels and classes
  - Classification
  - Regression

Finding patterns in text data
Unsupervised methods

Collocation / phrase detection

- Simple way to get a quick look at common themes
- Bigrams are a form of "collocation" – a more general term for words that occur together

```
In [12]: import nltk, re
         from nltk.collocations import *
         bigram_measures = nltk.collocations.BigramAssocMeasures()
         tokens = nltk.corpus.genesis.words('english-web.txt')
         tokens = [re.sub(r'\W', '', t) for t in tokens]
         finder = BigramCollocationFinder.from_words(tokens)
         ignored_words = nltk.corpus.stopwords.words('english')
         finder.apply_word_filter(lambda w: len(w) < 3 or w.lower() in
         ignored_words)
         scored = finder.score_ngrams(bigram_measures.raw_freq)
         sorted(bigram for bigram, score in scored)[:10]

Out[12]: [(u'Abel', u'Mizraim'),
          (u'Abel', u'also'),
          (u'Abimelech', u'called'),
          (u'Abimelech', u'charged'),
          (u'Abimelech', u'king'),
          (u'Abimelech', u'rose'),
          (u'Abimelech', u'said'),
          (u'Abimelech', u'took'),
          (u'Abimelech', u'went'),
          (u'Abraham', u'answered')]
```

Code modified from: https://www.nltk.org/howto/collocations.html

Finding patterns in text data
Unsupervised methods

Co-occurrence matrices

- We can also find words that occur in the same documents together (not just next to each other)

	able	absolutely	acid	actual	actually	add
able	0	12	4	3	34	25
absolutely	12	0	9	6	26	21
acid	4	9	0	1	28	23
actual	3	6	1	0	16	11
actually	34	26	28	16	0	53

5 rows × 1026 columns

```
from sklearn.feature_extraction.text import CountVectorizer

count_vectorizer = CountVectorizer(max_df=1.0, min_df=50)
counts = count_vectorizer.fit_transform(sample['Text'])
ngrams = count_vectorizer.get_feature_names()
cooccurs = (counts.T * counts)
cooccurs.setdiag(0)

rows, scanned = [], []
for index, row in pd.DataFrame(cooccurs.todense()).iterrows():
  for j, val in enumerate(row):
    if ngrams[j] not in scanned and val >= 10:
      rows.append({
          "pair": (ngrams[index], ngrams[j]), "count": val
      })
  scanned.append(ngrams[index])
pd.DataFrame(rows).sort_values("count", ascending=False)[:10]
```

	pair	count
36253	(cat, food)	3307
43697	(coffee, cup)	3236
62660	(dog, food)	3093
85935	(good, taste)	2321
44291	(coffee, taste)	2256
78780	(flavor, taste)	2103
43802	(coffee, flavor)	2094
63126	(dog, treat)	2031
100054	(just, taste)	2003
137548	(taste, tea)	1984

Finding patterns in text data
Unsupervised methods

- Might want to compare documents (or words) to one another
- Possible applications
  - Spelling correction
  - Document deduplication
  - Measure similarity of language
    - Politicians' speeches
    - Movie reviews
    - Product descriptions

Figure 1. Deduplication reduces the amount of stored data.

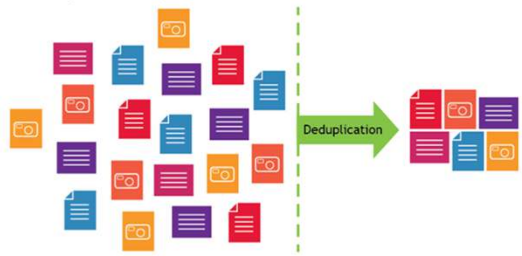


Image credit: https://pibytes.wordpress.com/2013/02/02/deduplication-internals-part-1/

Finding patterns in text data
Unsupervised methods

Levenshtein distance

- Compute number of steps needed to turn a word/document into another
- Can express as a ratio (percent of word/document that needs to change) to measure similarity

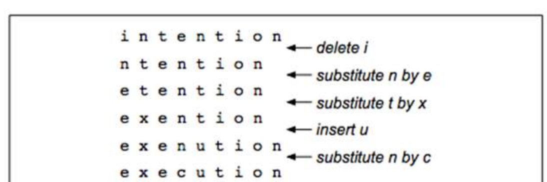


Figure 2.16 Path from *intention* to *execution*.

Image credit: http://web.stanford.edu/~jurafsky/slp3/2.pdf

Finding patterns in text data
Unsupervised methods

Cosine similarity

- Compute the “angle” between two word vectors
- TF-IDF: axes are the weighted frequencies for each word
- Word2Vec: axes are the learned dimensions from the model

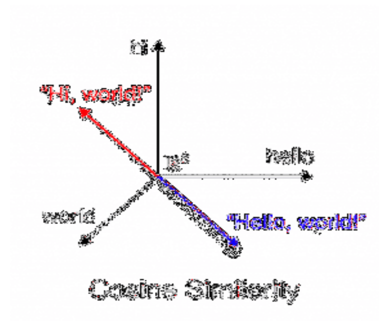


Image credit: https://www.machinelearningplus.com/nlp/cosine-similarity/

Finding patterns in text data
Unsupervised methods

Clustering

- Algorithms that use word vectors (TF-IDF, Word2Vec, etc.) to identify structural groupings between observations (words, documents)
- K-Means is a very commonly used one

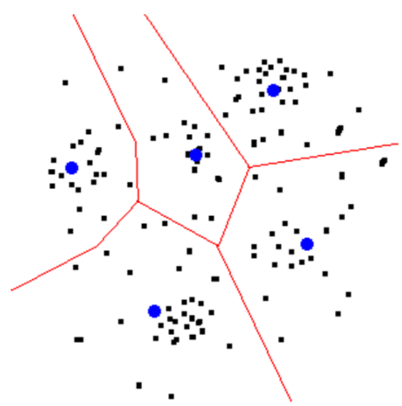
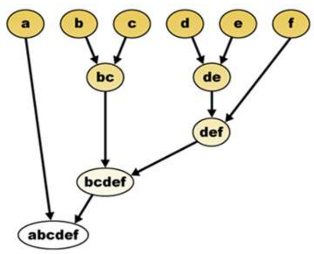


Image credit: http://mnemstudio.org/clustering-k-means-introduction.htm

Finding patterns in text data
Unsupervised methods

Hierarchical/agglomerative clustering

- Start with all observations, and use a rule to pair them up, and repeat until there's only one group left



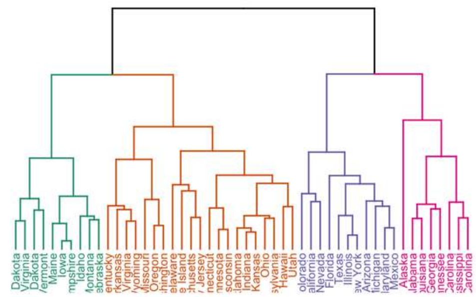


Image credit: https://scrnaseq-course.cog.sanger.ac.uk/website/biological-analysis.html
Image credit: https://rpkgs.datanovia.com/factoextra/reference/fviz_dend.html

Finding patterns in text data
Unsupervised methods

Network analysis

- Can also get creative
- After all, we're just working with columns and numbers
- Example: link words together by their strongest correlations

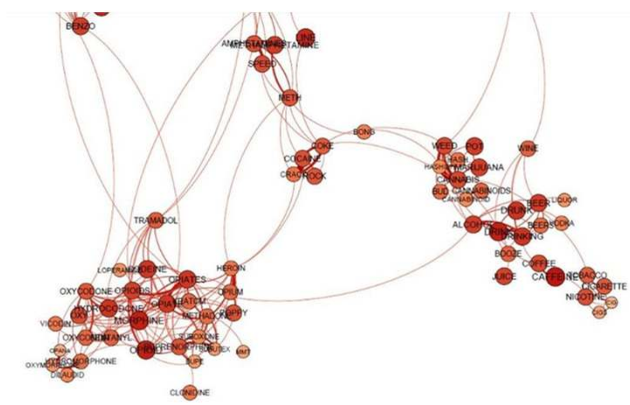


Image credit: https://www.linkedin.com/in/patrick-van-kessel

Finding patterns in text data
Unsupervised methods

Pointwise mutual information

- Based on information theory
- Compares conditional and joint probabilities to measure the likelihood of a word occurring with a category/outcome, beyond random chance

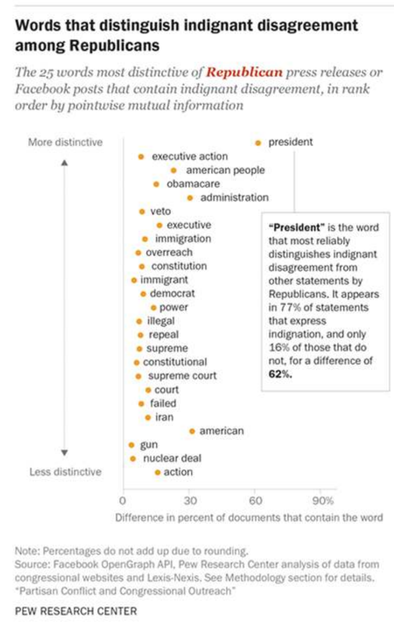


Image credit: https://www.people-press.org/2017/02/23/partisan-language-in-congressional-outreach/

Finding patterns in text data
Unsupervised methods

Topic modeling

- Algorithms that characterize documents in terms of topics (groups of words)
- Find topics that best fit the data

"Arts"	"Budgets"	"Children"	"Education"
NEW	MILLION	CHILDREN	SCHOOL
FILM	TAX	WOMEN	STUDENTS
SHOW	PROGRAM	PEOPLE	SCHOOLS
MUSIC	BUDGET	CHILD	EDUCATION
MOVIE	BILLION	YEARS	TEACHERS
PLAY	FEDERAL	FAMILIES	HIGH
MUSICAL	YEAR	WORK	PUBLIC
BEST	SPENDING	PARENTS	TEACHER
ACTOR	NEW	SAYS	BENNETT
FIRST	STATE	FAMILY	MANIGAT
YORK	PLAN	WELFARE	NAMPHY
OPERA	MONEY	MEN	STATE
THEATER	PROGRAMS	PERCENT	PRESIDENT
ACTRESS	GOVERNMENT	CARE	ELEMENTARY
LOVE	CONGRESS	LIFE	HAITI

The William Randolph Hearst Foundation will give $1.25 million to Lincoln Center, Metropolitan Opera Co., New York Philharmonic and Juilliard School. "Our board felt that we had a real opportunity to make a mark on the future of the performing arts with these grants an act every bit as important as our traditional areas of support in health, medical research, education and the social services," Hearst Foundation President Randolph A. Hearst said Monday in announcing the grants. Lincoln Center's share will be $200,000 for its new building, which will house young artists and provide new public facilities. The Metropolitan Opera Co. and New York Philharmonic will receive $400,000 each. The Juilliard School, where music and the performing arts are taught, will get $250,000. The Hearst Foundation, a leading supporter of the Lincoln Center Consolidated Corporate Fund, will make its usual annual $100,000 donation, too.

Image credit: http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

Finding patterns in text data
Supervised methods

- Often we want to categorize documents
- Unsupervised methods can help
- But often we need to read and label them ourselves
- Classification models can take labeled data and learn to **make predictions**

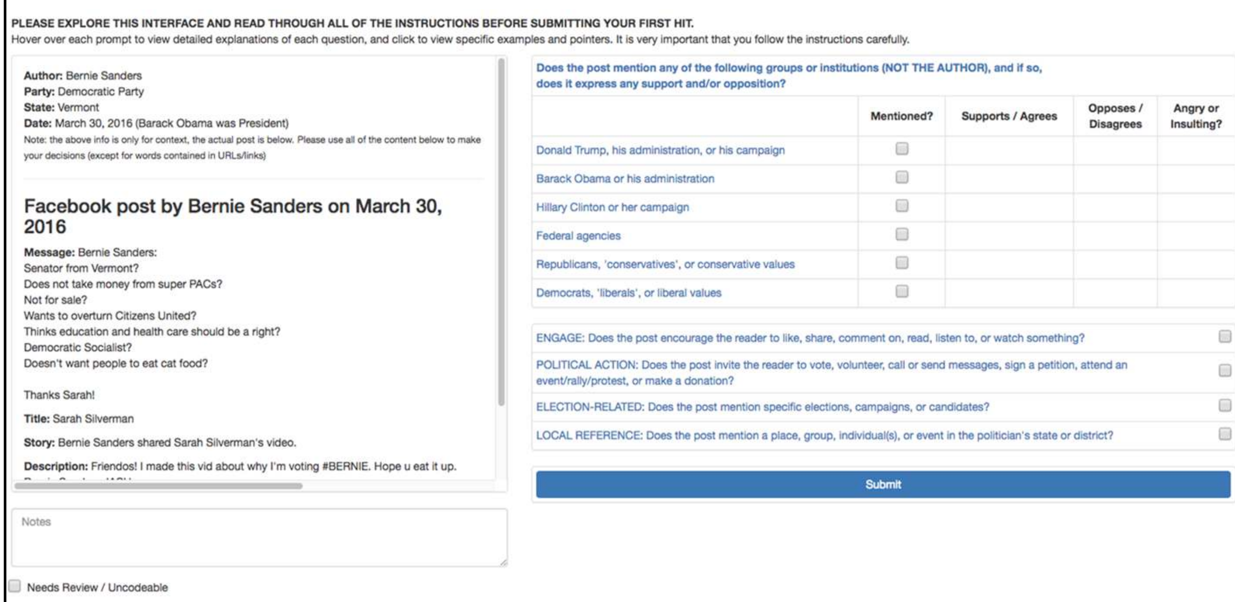
Finding patterns in text data
Supervised methods

Steps:

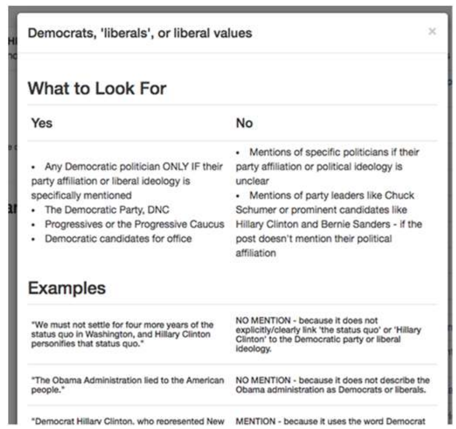
- Label a sample of documents
- Break your sample into two sets: a **training sample** and a **test sample**
- **Train** a model on the training sample
- **Evaluate** it on the test sample
- **Apply** it to the full set of documents to make **predictions**

Finding patterns in text data
Supervised methods

- First you need to develop a codebook
- Codebook: set of rules for labeling and categorizing documents
- The best codebooks have clear rules for hard cases, and lots of examples
- Categories should be MECE: mutually exclusive and collectively exhaustive



Finding patterns in text data
Supervised methods



Finding patterns in text data
Supervised methods

- Need to validate the codebook by measuring **interrater reliability**
- Makes sure your measures are consistent, objective, and reproducible
- Multiple people code the same document

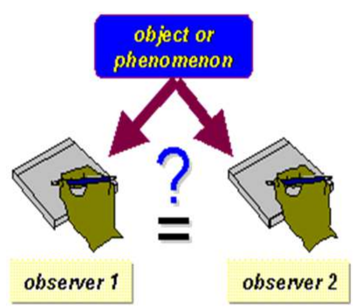


Image credit: https://socialresearchmethods.net/kb/reltypes.php

Finding patterns in text data
Supervised methods

- Various metrics to test whether their agreement is high enough
  - Krippendorf's alpha
  - Cohen's kappa
- Can also compare coders against a gold standard, if available

Values	Interpretation
Smaller than 0.00	Poor Agreement
0.00 to 0.20	Slight Agreement
0.21 to 0.40	Fair Agreement
0.41 to 0.60	Moderate Agreement
0.61 to 0.80	Substantial Agreement
0.81 to 1.00	Almost Perfect Agreement

Image credit: https://www.researchgate.net/figure/Interpretation-of-Cohens-Kappa-Values_tbl2_302869046

Finding patterns in text data
Supervised methods

- Mechanical Turk can be a great way to code a lot of documents
- Have 5+ Turkers code a large sample of documents
- Collapse them together with a rule
- Code a subset in-house, and compute reliability

amazonmechanical turk
Artificial Artificial Intelligence

Image credit: https://machmachines.com/make-some-extra-cash-with-amazon-mechanical-turk060515/

Finding patterns in text data
Supervised methods

- After coding, split your sample into two sets (~80/20)
  - One for training, one for testing
- We do this to check for (and avoid) **overfitting**

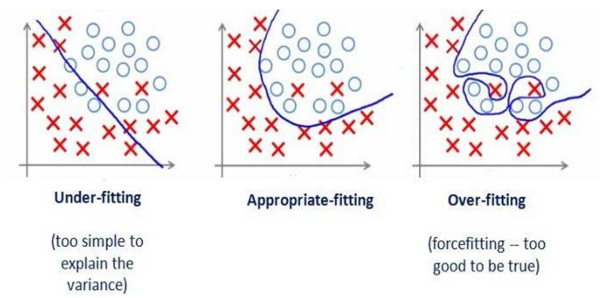


Image credit: https://medium.com/ml-research-lab/under-fitting-over-fitting-and-its-solution-dc6191e34250

Finding patterns in text data
Supervised methods

- Next step is called **feature extraction** or **feature selection**
- Need to extract "features" from the text
  - TF-IDF
  - Word2Vec vectors
- Can also utilize metadata, if potentially useful

Finding patterns in text data
Supervised methods

- Select a classification algorithm
- Common choice for text data are **support vector machines (SVMs)**
- Similar to regression, SVMs find the line that best separates two or more groups
- Can also use non-linear “kernels” for better fits (radial basis function, etc.)
- **XGBoost** is a newer and very promising algorithm

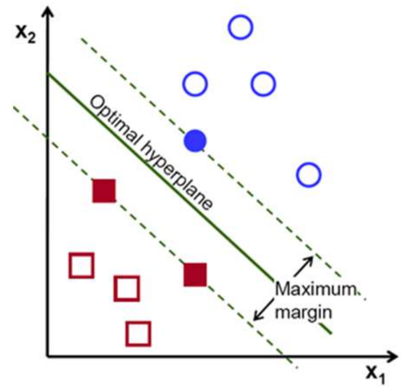


Image credit: https://towardsdatascience.com/support-vector-machine-vs-logistic-regression-94cc2975433f

Finding patterns in text data
Supervised methods

- Time to evaluate performance
- Lots of different metrics, depending on what you care about
- Often we care about precision/recall
  - Precision: did you pick out mostly needles or mostly hay?
  - Recall: how many needles did you miss?
- Other metrics:
  - Matthew’s correlation coefficient
  - Brier score
  - Overall accuracy

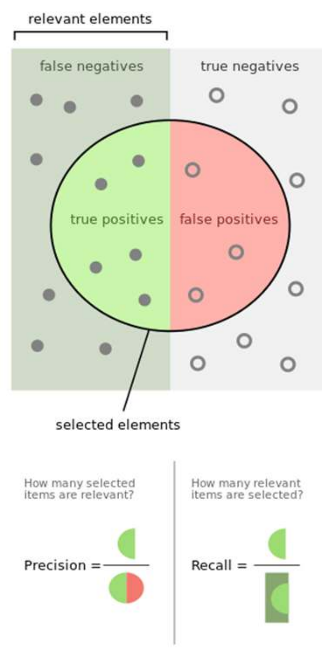


Image credit: https://en.wikipedia.org/wiki/Precision_and_recall

Finding patterns in text data
Supervised methods

- Doing just one split leaves a lot up to chance
- To bootstrap a better estimate of the model's performance, it's best to use **K-fold cross-validation**
- Splits your data into train/test sets **multiple times** and averages the performance metrics
- Ensures that you didn't just get lucky (or unlucky)

Finding patterns in text data
Supervised methods

- Model not working well?
- You probably need to **tune your parameters**
- You can use a **grid search** to test out different combinations of model parameters and feature extraction methods
- Many software packages can automatically help you pick the best combination to maximize your model's performance

Finding patterns in text data
Supervised methods

- Suggested design:
  - Large training sample, coded by Turkers
  - Small evaluation sample, coded by Turkers and in-house experts
  - Compute IRR between Turk and experts
  - Train model on training sample, use 5-fold cross-validation
  - Apply model to evaluation sample, compare results against in-house coders and Turkers

Finding patterns in text data
Supervised methods

- Some (but not all) models produce probabilities along with their classifications
- Ideally you fit the model using your preferred scoring metric/function
- But you can also use post-hoc probability thresholds to adjust your model's predictions

Tools and Resources

Open-source tools

stackoverflow

- Python
  - NLTK, scikit-learn, pandas, numpy, scipy, gensim, spacy, etctw
- R
  - https://cran.r-project.org/web/views/NaturalLanguageProcessing.html
- Java
  - Stanford Core NLP + many other useful libraries https://nlp.stanford.edu/software/

Image credit: https://stackoverflow.com/

Commercial tools

- Cloud-based NLP
  - Amazon Comprehend
  - Google Cloud Natural Language
  - IBM Watson NLU
- Software
  - SPSS Text Modeler
  - Provalis WordStat

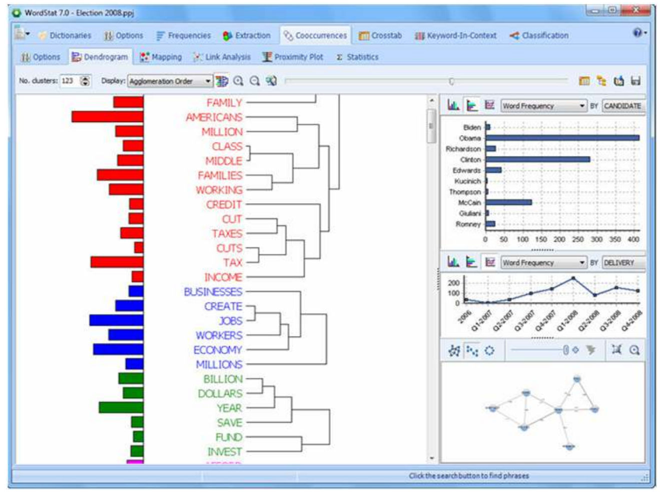


Image credit: https://provalisresearch.com/products/content-analysis-software/

Time for a demo!

https://bit.ly/2rlCOUG

Full link: https://colab.research.google.com/github/patrickvankessel/AAPOR-Text-Analysis-2019/blob/master/Tutorial.ipynb
GitHub repo: https://github.com/patrickvankessel/AAPOR-Text-Analysis-2019

Feel free to reach out:
pvankessel@pewresearch.org
patrickvankessel@gmail.com

Special thanks to Michael Jugovich for help putting these materials together for previous workshops

2019 AAPOR Text Analytics Tutorial

Patrick van Kessel

Senior Data Scientist, Pew Research Center

These materials are adapted from workshops I did in 2018 and 2019 for NYAAPOR, the World Bank, and IBM, with a lot of help from an old colleague of mine, Michael Jugovich (now at IBM). You can access a GitHub repository containing this notebook and the data sample here: https://github.com/patrickvankessel/AAPOR-Text-Analysis-2019

Loading in the data

We'll use a sample from the Kaggle Amazon Fine Food Reviews dataset. The full dataset can be found here: https://www.kaggle.com/snap/amazon-fine-food-reviews

```
[ ] import pandas as pd
```

```
[ ] sample = pd.read_csv("https://raw.githubusercontent.com/patrickvanke
```

```
[ ] print(len(sample))
```

```
10000
```

Examine the data

Run the cell below a few times, let's take a look at our text and see what it looks like. Always take a look at your raw data.

```
[ ] sample.sample(10)['Text'].values
```

```
ì roasted varieties provide some sort of "extra" punch of flavor for h
ıbout how awful it tastes and how much it's an acquired taste.  I boug
 item again!',
 isn't. So whenever I have soup or whatever, I give him one of these.
: spicy enough to satisfy my husband, and just right for me.',
:otein to help curb hunger! Perfect go-to snack, and I\'ve also had th
ìays when we work overtime. I have to say they do work, some days we s
: Also available in several sizes. This one is perfect for pocket or p
.ke some dog treats for small dogs!<br />I love how you can see the oa
ɔrder.'],
```

Preprocess the text (clean it up!)

I don't know about you, but I noticed some junk in our data - HTML and URLs. Let's clear that out first. We'll also take this opportunity to lemmatize the words - to do that, we'll install NLTK's WordNet library.

```
[ ] import nltk
    nltk.download('wordnet')
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
True
```

```python
import re
from nltk.stem import WordNetLemmatizer

# Initialize a lemmatizer
lemmatizer = WordNetLemmatizer()

def clean_text(text):
    # First we'll use regular expressions to strip out links and HTM
    text = re.sub(r'http[a-zA-Z0-9\&\?\=\?\/\:\.]+\b', ' ', text)
    text = re.sub(r'\<[^\<\>]+\>', ' ', text)
    # Next, let's clear out all punctuation and replace it with whit
    text = re.sub(r'\W+', ' ', text)
    # And clear out numbers
    text = re.sub(r'[0-9]+', ' ', text)
    # And then lowercase
    text = text.lower()
    # This isn't going to be perfect - ideally we expand contractior
    # And also deal with spelling corrections
    # But this will work well enough for now

    # Next, let's split on whitespace and then lemmatize each token
    tokens = text.split()
    tokens = [lemmatizer.lemmatize(x) for x in tokens]
    text = " ".join(tokens)

    return text

sample['Text'] = sample['Text'].map(clean_text)
```

Let's see what our data look like now that we've processed the text

```python
[ ]  sample.sample(10)['Text'].values
```

```
array(['i bought the whirley pop stovetop popcorn popper several mont
       'i called upon a panel of two expert feline food tester assist
       'so far my three male kitty have loved all the wellness wet fc
       'moderate size packaging and not too sweet like other small ju
       'they are by far the best flavor they really do taste like a h
       'i m a big fan of the carbs fat and protein balance of these c
       'i have been making almond milk for about month and it ha take
       'marley one love organic coffee pod are targeted at hip people
       'so i have a dietitian now who s helping me lose another pounc
       'i place order on june but this snack expire on sep i order ak
      dtype=object)
```

TF-IDF Vectorization (Feature Extraction)

Just to be safe, let's add some additional words to a standard list of English stop words.

```
[ ] from sklearn.feature_extraction import stop_words as sklearn_stop_wc
    # Grab standard English stopwords
    stop_words = set(sklearn_stop_words.ENGLISH_STOP_WORDS)
    # And add in some of our own ("like" is really common and doesn't te
    stop_words = stop_words.union(set([
        "www", "http", "https", "br", "amazon", "href", "wa", "ha",
        "like", "just",
    ]))
```

Okay, now let's tokenize our text and turn it into numbers

```
[ ] from sklearn.feature_extraction.text import TfidfVectorizer, CountVe

    tfidf_vectorizer = TfidfVectorizer(
        max_df=0.9, # Remove any words that appear in more than 90% of c
        min_df=5, # Remove words that appear in fewer than 5 document
        ngram_range=(1, 1), # Only extract unigrams
        stop_words=stop_words, # Remove stopwords
        max_features=2500 # Grab the 2500 most common words (based on ak
    )
    tfidf = tfidf_vectorizer.fit_transform(sample['Text'])
    ngrams = tfidf_vectorizer.get_feature_names()
```

```
[ ] tfidf
```

```
<10000x2500 sparse matrix of type '<class 'numpy.float64'>'
        with 245835 stored elements in Compressed Sparse Row format>
```

Because words are really big, by default we work with sparse matrices. We can expand the sparse matrix with `.todense()` and compute sums like a normal dataframe. Let's check out the top 20 words.

```
[ ] ngram_df = pd.DataFrame(tfidf.todense(), columns=ngrams)
    ngram_df.sum().sort_values(ascending=False)[:20]
```

```
coffee     316.980063
good       308.366277
taste      306.032427
great      296.119454
tea        286.896549
love       283.353920
product    281.596502
flavor     278.327891
food       213.044387
dog        205.873217
really     178.868613
price      175.593576
time       165.726609
make       165.491621
cup        165.306249
buy        163.445115
best       162.054263
bag        154.881833
ve         151.040409
don        145.061858
dtype: float64
```

We can also explore word co-occurrences - the words that most frequently appear together in the same documents

```
[ ] count_vectorizer = CountVectorizer(
        max_df=.9,
        min_df=50,
        stop_words=stop_words
    )
    counts = count_vectorizer.fit_transform(sample['Text'])
    ngrams = count_vectorizer.get_feature_names()
    cooccurs = (counts.T * counts)
    cooccurs.setdiag(0)
    cooccurs = pd.DataFrame(cooccurs.todense(), index=ngrams, columns=nç
    cooccurs.head()
```

	able	absolutely	acid	actual	actually	add	added	addict
able	0	12	4	3	34	25	18	
absolutely	12	0	9	6	26	21	14	
acid	4	9	0	1	28	23	10	
actual	3	6	1	0	16	11	9	
actually	34	26	28	16	0	53	50	

5 rows × 1026 columns

```
[ ] rows, scanned = [], []
    for word1, row in cooccurs.iterrows():
      for word2 in row.keys():
        if word2 not in scanned and row[word2] >= 100:
          rows.append({
              "pair": (word1, word2), "count": row[word2]
          })
      scanned.append(word1)
```

```
[ ] sorted(rows, key=lambda x: x["count"], reverse=True)[:25]
```

```
[{'count': 3307, 'pair': ('cat', 'food')},
 {'count': 3236, 'pair': ('coffee', 'cup')},
 {'count': 3093, 'pair': ('dog', 'food')},
 {'count': 2321, 'pair': ('good', 'taste')},
 {'count': 2256, 'pair': ('coffee', 'taste')},
 {'count': 2103, 'pair': ('flavor', 'taste')},
 {'count': 2094, 'pair': ('coffee', 'flavor')},
 {'count': 2031, 'pair': ('dog', 'treat')},
 {'count': 1984, 'pair': ('taste', 'tea')},
 {'count': 1962, 'pair': ('coffee', 'good')},
 {'count': 1859, 'pair': ('flavor', 'good')},
 {'count': 1659, 'pair': ('eat', 'food')},
 {'count': 1619, 'pair': ('flavor', 'tea')},
 {'count': 1592, 'pair': ('food', 'product')},
 {'count': 1580, 'pair': ('product', 'taste')},
 {'count': 1553, 'pair': ('coffee', 'pod')},
 {'count': 1553, 'pair': ('green', 'tea')},
```

Classification

Let's go back to the TF-IDF matrix and use it to do some classification

```
[ ] from sklearn.feature_extraction.text import TfidfVectorizer, CountVe

    tfidf_vectorizer = TfidfVectorizer(
        max_df=0.9, # Remove any words that appear in more than 90% of c
        min_df=5, # Remove words that appear in fewer than 5 document
        ngram_range=(1, 1), # Only extract unigrams
        stop_words=stop_words, # Remove stopwords
        max_features=2500 # Grab the 2500 most common words (based on ab
    )
    tfidf = tfidf_vectorizer.fit_transform(sample['Text'])
    ngrams = tfidf_vectorizer.get_feature_names()
```

Let's make an outcome variable. How about we try to predict 5-star reviews, and then maybe helpfulness?

```
[ ] sample['good_score'] = sample['Score'].map(lambda x: 1 if x == 5 els
    sample['was_helpful'] = ((sample['HelpfulnessNumerator'] / sample['H
```

```
[ ] column_to_predict = 'good_score'
```

```
[ ] from sklearn.model_selection import StratifiedKFold
    from sklearn import svm
    from sklearn import metrics

    results = []
    kfolds = StratifiedKFold(n_splits=5)
```

We just created an object that'll split the data into fifths, and then iterate over it five times, holding out one-fifth each time for testing. Let's do that now. Each "fold" contains an index for training rows, and one for testing rows. For each fold, we'll train a basic linear Support Vector Machine, and evaluate its performance.

```
for i, fold in enumerate(kfolds.split(tfidf, sample[column_to_predic

    train, test = fold
    print("Running new fold, {} training cases, {} testing cases".fc

    clf = svm.LinearSVC(
        max_iter=1000,
        penalty='l2',
        class_weight='balanced',
        loss='squared_hinge'
    )
    # We picked some decent starting parameters, but you can try out
    # http://scikit-learn.org/stable/modules/generated/sklearn.svm.I
    # If you're ambitious - check out the Scikit-Learn documentatior
    # http://scikit-learn.org/stable/supervised_learning.html
    # XGBoost is one of my favorites, and there's an Scikit-Learn wr
    # https://machinelearningmastery.com/develop-first-xgboost-model

    training_text = tfidf[train]
    training_outcomes = sample[column_to_predict].loc[train]
    clf.fit(training_text, training_outcomes) # Train the classifier
```

```
test_text = tfidf[test]
test_outcomes = sample[column_to_predict].loc[test]
predictions = clf.predict(test_text) # Get predictions for the t

precision, recall, fscore, support = metrics.precision_recall_fs
    test_outcomes, # Compare the predictions against the true ou
    predictions
)

results.append({
    "fold": i,
    "outcome": 0,
    "precision": precision[0],
    "recall": recall[0],
    "fscore": fscore[0],
    "support": support[0]
})

results.append({
    "fold": i,
    "outcome": 1,
    "precision": precision[1],
    "recall": recall[1],
    "fscore": fscore[1],
    "support": support[1]
})
```

```
results = pd.DataFrame(results)
```

```
Running new fold, 8000 training cases, 2000 testing cases
Running new fold, 8000 training cases, 2000 testing cases
Running new fold, 8000 training cases, 2000 testing cases
Running new fold, 8000 training cases, 2000 testing cases
Running new fold, 8000 training cases, 2000 testing cases
```

How'd we do?

```
[ ] print(results.groupby("outcome").mean()[['precision', 'recall']])
    print(results.groupby("outcome").std()[['precision', 'recall']])
```

```
         precision    recall
outcome
0         0.641014  0.698769
1         0.817090  0.774626
         precision    recall
outcome
0         0.008326  0.020460
1         0.009987  0.005897
```

Now we know that our model is pretty stable and reasonably performant, we can fit and transform the full dataset.

```
[ ] clf.fit(tfidf, sample[column_to_predict])
    print(metrics.classification_report(sample[column_to_predict].loc[te
    print(metrics.confusion_matrix(sample[column_to_predict].loc[test],
```

```
              precision    recall  f1-score   support

           0       0.64      0.72      0.68       731
           1       0.83      0.77      0.80      1269

    accuracy                           0.75      2000
   macro avg       0.74      0.75      0.74      2000
weighted avg       0.76      0.75      0.75      2000

[[528 203]
 [293 976]]
```

And now we can see what the most predictive features are.

```
[ ] import numpy as np

    ngram_coefs = sorted(zip(ngrams, clf.coef_[0]), key=lambda x: x[1],
    ngram_coefs[:10]
```

```
[('highly', 3.1013089738143287),
 ('best', 2.444740644553053),
 ('love', 2.306617070446386),
 ('perfect', 2.2929056338458307),
 ('favorite', 2.1200027087198525),
 ('wonderful', 2.006322948279272),
 ('cancer', 1.9431169727387974),
 ('fabulous', 1.8950394468562675),
 ('satisfied', 1.8690450683854933),
 ('addicted', 1.8094483299768616)]
```

What happens if you change the outcome column to "was_helpful" and re-run it again? Can you think of ways to improve this? Add more stopwords? Include bigrams in addition to unigrams?

Topic Modeling

```
[ ]  from sklearn.decomposition import NMF, LatentDirichletAllocation
```

```
[ ]  def print_top_words(model, feature_names, n_top_words):
         for topic_idx, topic in enumerate(model.components_):
             print("Topic #{}: {}".format(
                 topic_idx,
                 ", ".join([feature_names[i] for i in topic.argsort()[:-r
             ))
```

Let's find some topics. We'll check out non-negative matrix factorization (NMF) first.

```
[ ] nmf = NMF(n_components=10, random_state=42, alpha=.1, l1_ratio=.5).f
    # Try out different numbers of topics (change n_components)
    # Documentation: http://scikit-learn.org/stable/modules/generated/sk
    print("\nTopics in NMF model:")
    print_top_words(nmf, ngrams, 10)
```

```
Topics in NMF model:
Topic #0: flavor, taste, sugar, ve, really, make, water, tried, don,
Topic #1: coffee, cup, strong, roast, bold, flavor, blend, keurig, da
Topic #2: tea, green, bag, drink, cup, iced, stash, black, taste, ear
Topic #3: dog, treat, love, food, chew, bone, small, size, teeth, toy
Topic #4: cat, food, eat, dry, wellness, canned, chicken, ingredient,
Topic #5: product, store, price, order, buy, local, grocery, shipping
Topic #6: great, love, snack, price, deal, taste, healthy, recommend,
Topic #7: chocolate, bar, dark, snack, nut, peanut, candy, protein, s
Topic #8: chip, bag, salt, potato, kettle, snack, vinegar, salty, fla
Topic #9: good, really, price, taste, pretty, quality, tasting, quite
```

LDA is an other popular topic modeling technique

```
[ ] lda = LatentDirichletAllocation(n_components=10, random_state=42).fi
    # Documentation: http://scikit-learn.org/stable/modules/generated/sk
    # doc_topic_prior (alpha) - lower alpha means documents will be comp
    # topic_word_prior (beta) - lower beta means topics will be composed
    print("\nTopics in LDA model:")
    print_top_words(lda, ngrams, 10)
```

```
Topics in LDA model:
Topic #0: coffee, cup, flavor, taste, drink, good, strong, great, roa
Topic #1: taste, bar, sugar, good, great, flavor, chocolate, product,
Topic #2: sauce, chip, salt, great, pasta, flavor, soup, good, cheese
Topic #3: chocolate, great, love, good, cereal, snack, box, cider, cu
Topic #4: tea, popcorn, taste, flavor, good, bag, drink, green, chai,
Topic #5: dog, treat, love, chew, teeth, toy, bone, size, training, g
Topic #6: product, price, arrived, gift, order, great, store, good, i
Topic #7: sleep, product, night, help, container, calm, great, link,
Topic #8: food, cat, dog, product, love, eat, good, year, bag, time
Topic #9: store, baby, love, product, great, price, time, buy, year,
```

We can use the topic models the same way we did our classifier - everything in Scikit-Learn follows the same fit/transform paradigm. So, let's get the topics for our documents.

```
[ ] doc_topics = pd.DataFrame(lda.transform(tfidf))
```

```
[ ] doc_topics.head()
```

	0	1	2	3	4	5	6	
0	0.024099	0.024098	0.783103	0.024103	0.024096	0.024099	0.024098	0.024
1	0.750107	0.027767	0.027766	0.027764	0.027766	0.027765	0.027768	0.027
2	0.016939	0.016949	0.016949	0.016937	0.815528	0.016946	0.016941	0.016
3	0.023150	0.023149	0.023147	0.023146	0.791670	0.023147	0.023147	0.023
4	0.021649	0.021662	0.021655	0.021648	0.021648	0.021651	0.021647	0.021

Next we use Pandas to join the topics with the original sample dataframe

```
[ ] sample_with_topics = pd.concat([sample, doc_topics], axis=1)
```

Let's look for patterns by running some means and correlations

```
[ ] topic_columns = [col for col in sample_with_topics.columns if col.st
    sample_with_topics.groupby("good_score").mean()[topic_columns]
```

	topic_0	topic_1	topic_2	topic_3	topic_4	topic_5	topi
good_score							
0	0.183903	0.226292	0.065056	0.041901	0.076930	0.058601	0.07
1	0.142316	0.226311	0.087173	0.047796	0.090577	0.068197	0.09

```
[ ] for topic in topic_column_names:
        print("{}: {}".format(topic, sample_with_topics[topic].corr(samp
```

```
topic_0: -0.025292235465098012
topic_1: 0.012046757693743665
topic_2: 0.06959235818628244
topic_3: 0.023446401876468924
topic_4: 0.03599689524129321
topic_5: 0.03409134227342527
topic_6: 0.04691101261547623
topic_7: -0.001684816652652065
topic_8: -0.13859039591384575
topic_9: 0.019007709950536102
```

Here's an example of a linear regression

```
[ ] from sklearn import datasets, linear_model
    from sklearn.metrics import mean_squared_error, r2_score

    training_data = sample_with_topics[topic_column_names[:-1]]
    # We're leaving a column out to avoid multicollinearity

    regression = linear_model.LinearRegression()

    # Train the model using the training sets
    regression.fit(training_data, sample_with_topics['Score'])
    coefficients = regression.coef_
    for topic, coef in zip(topic_column_names[:-1], coefficients):
      print("{}: {}".format(topic, coef))
```

```
topic_0: -0.21269382796846162
topic_1: -0.0843114794912539
topic_2: 0.3907048768291855
topic_3: 0.10179168229468046
topic_4: 0.10070443473499552
topic_5: 0.1935251748886604
topic_6: 0.1836870262014428
topic_7: -0.21131708369325589
topic_8: -0.7277794219956407
```

Sadly Scikit-Learn doesn't make it easy to get p-values or a regression report like you'd normally expect of something like R or Stata. Scikit-Learn is more about prediction than statistical analysis; for the latter, we can use Statsmodels.

```
[35] import statsmodels.api as sm

     training_data = sm.add_constant(training_data)
     regression = sm.OLS(sample_with_topics['Score'], training_data)
     results = regression.fit()
     print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Score   R-squared:
Model:                            OLS   Adj. R-squared:
Method:                 Least Squares   F-statistic:
Date:                Mon, 09 Dec 2019   Prob (F-statistic):
Time:                        17:42:27   Log-Likelihood:
No. Observations:               10000   AIC:
Df Residuals:                    9990   BIC:
Df Model:                           9
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025
------------------------------------------------------------------------------
const          4.2597      0.071     59.682      0.000       4.120
topic_0       -0.2127      0.085     -2.506      0.012      -0.379
topic_1       -0.0843      0.081     -1.035      0.301      -0.244
topic_2        0.3907      0.105      3.723      0.000       0.185
topic_3        0.1018      0.134      0.758      0.448      -0.161
topic_4        0.1007      0.098      1.025      0.306      -0.092
topic_5        0.1935      0.109      1.768      0.077      -0.021
topic_6        0.1837      0.102      1.802      0.072      -0.016
topic_7       -0.2113      0.175     -1.209      0.227      -0.554
topic_8       -0.7278      0.087     -8.371      0.000      -0.898
==============================================================================
Omnibus:                     1944.496   Durbin-Watson:
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
Skew:                          -1.368   Prob(JB):
Kurtosis:                       3.620   Cond. No.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors i
/usr/local/lib/python3.6/dist-packages/numpy/core/fromnumeric.py:2495
  return ptp(axis=axis, out=out, **kwargs)
```

K-Means Clustering

We can also check out other unsupervised methods like clustering. I borrowed/modified some of this code from http://brandonrose.org/clustering

```
[ ] from sklearn.cluster import KMeans

    kmeans = KMeans(n_clusters=10, max_iter=50, tol=.01, n_jobs=-1)
    # http://scikit-learn.org/stable/modules/generated/sklearn.cluster.K
    kmeans.fit(tfidf)
    clusters = kmeans.labels_.tolist() # You can merge these back into t
```

```
[ ] centroids = kmeans.cluster_centers_.argsort()[:, ::-1]
    for i, closest_ngrams in enumerate(centroids):
        print("Cluster #{}: {}".format(i, ", ".join(np.array(ngrams)[clo
```

```
Cluster #0: chip, potato, bag, flavor, kettle, snack, great, salt
Cluster #1: dog, treat, food, love, chew, product, good, bone
Cluster #2: tea, taste, green, bag, drink, flavor, good, cup
Cluster #3: coffee, cup, flavor, strong, good, taste, roast, bold
Cluster #4: chocolate, dark, cooky, taste, hot, good, flavor, milk
Cluster #5: cat, food, eat, love, treat, wellness, chicken, dry
Cluster #6: product, great, price, good, store, love, taste, time
Cluster #7: bar, chocolate, snack, taste, nut, protein, good, sweet
Cluster #8: great, love, good, flavor, price, buy, time, store
Cluster #9: taste, good, sugar, great, flavor, drink, water, free
```

Agglomerative/Hierarchical Clustering

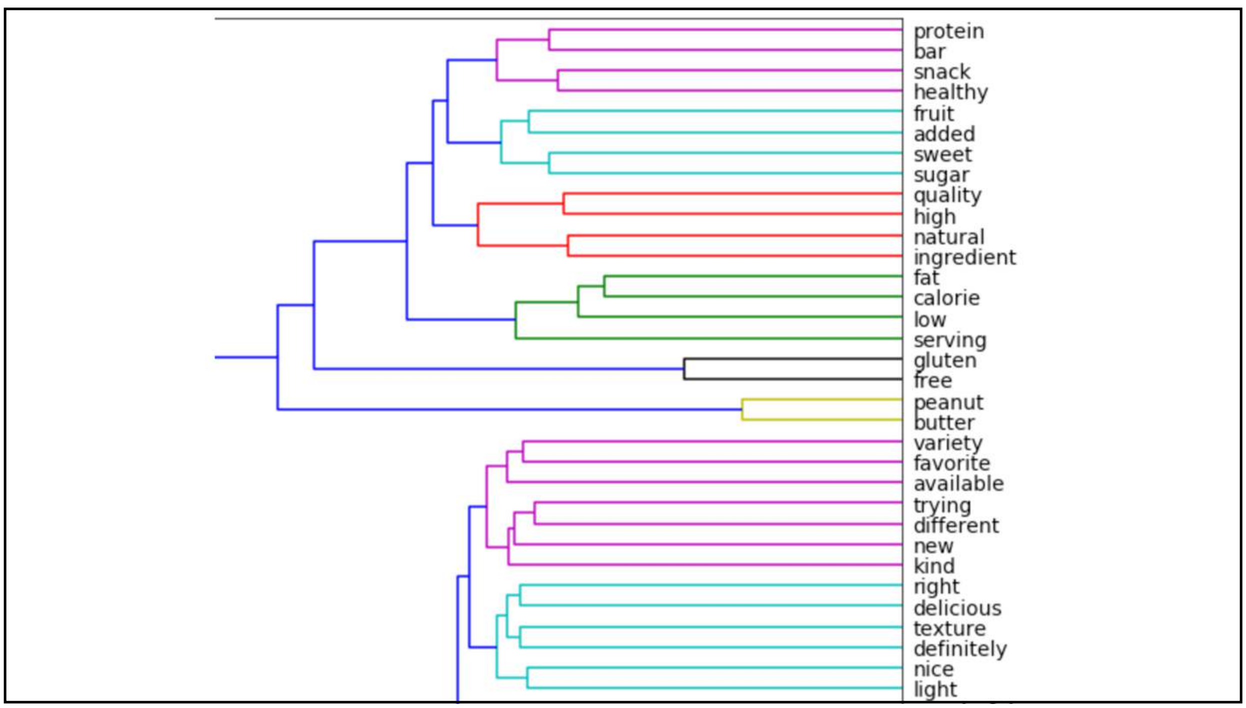
Instead of specifying the number of clusters upfront, now we're going to use hierarchical clustering to characterize how similar words are to each other, again based on their co-occurrence within documents. To keep things manageable, we'll use a smaller set of 500 words.

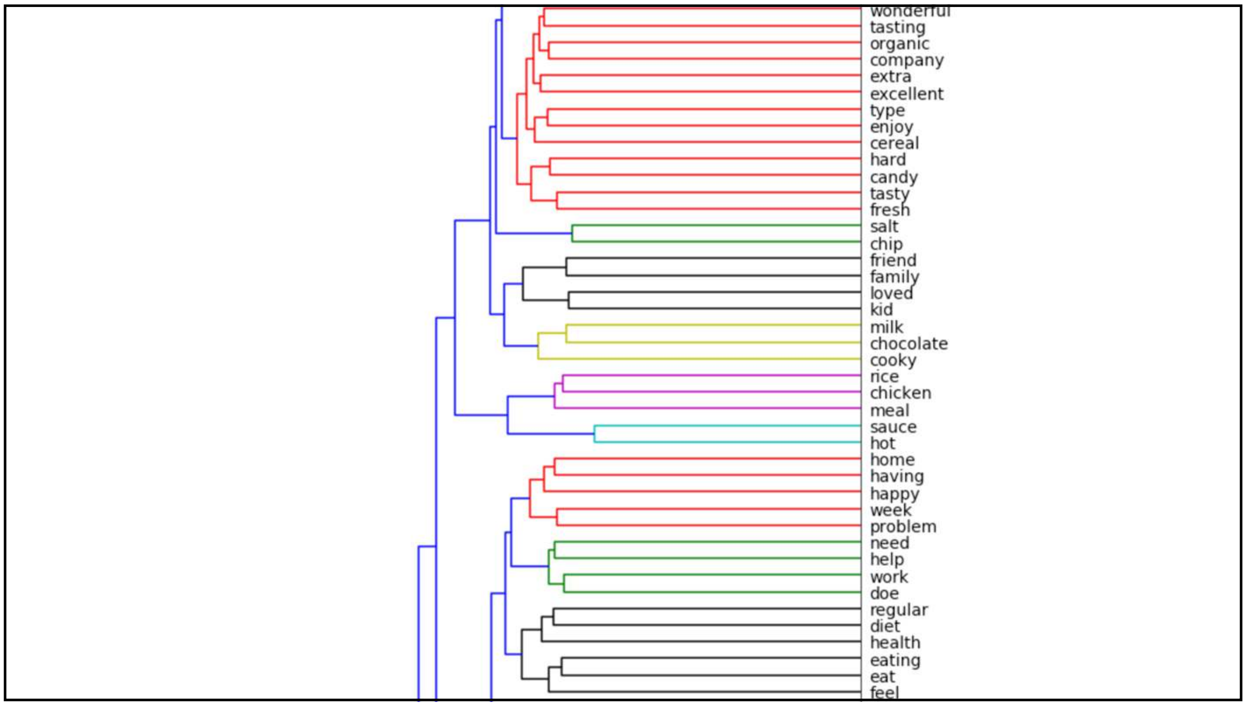
```
[ ] # This Python library lets us produce graphics
    %matplotlib inline
    import matplotlib.pyplot as plt
```

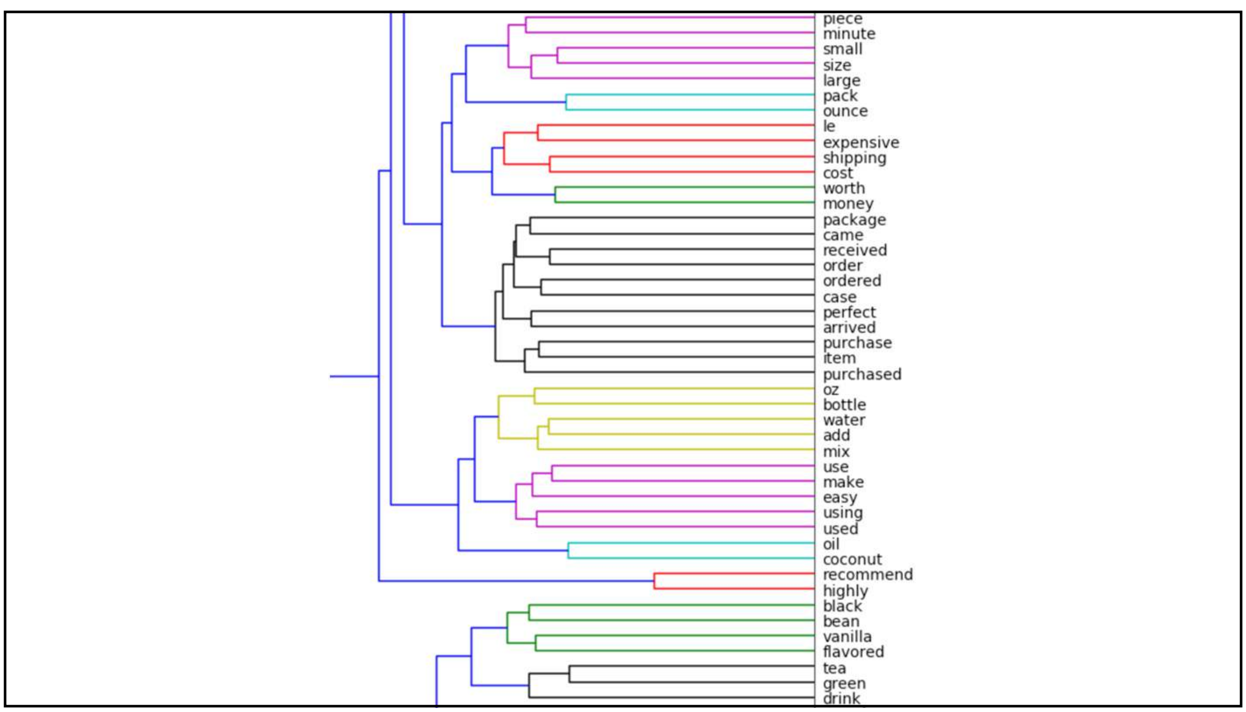
```
[ ] tfidf_vectorizer = TfidfVectorizer(
        max_df=0.25, # Focus on less common, more unique words
        min_df=5,
        ngram_range=(1, 1),
        stop_words=stop_words,
        max_features=200 # <- smaller set of words
    )
    tfidf = tfidf_vectorizer.fit_transform(sample['Text'])
    ngrams = tfidf_vectorizer.get_feature_names()
```

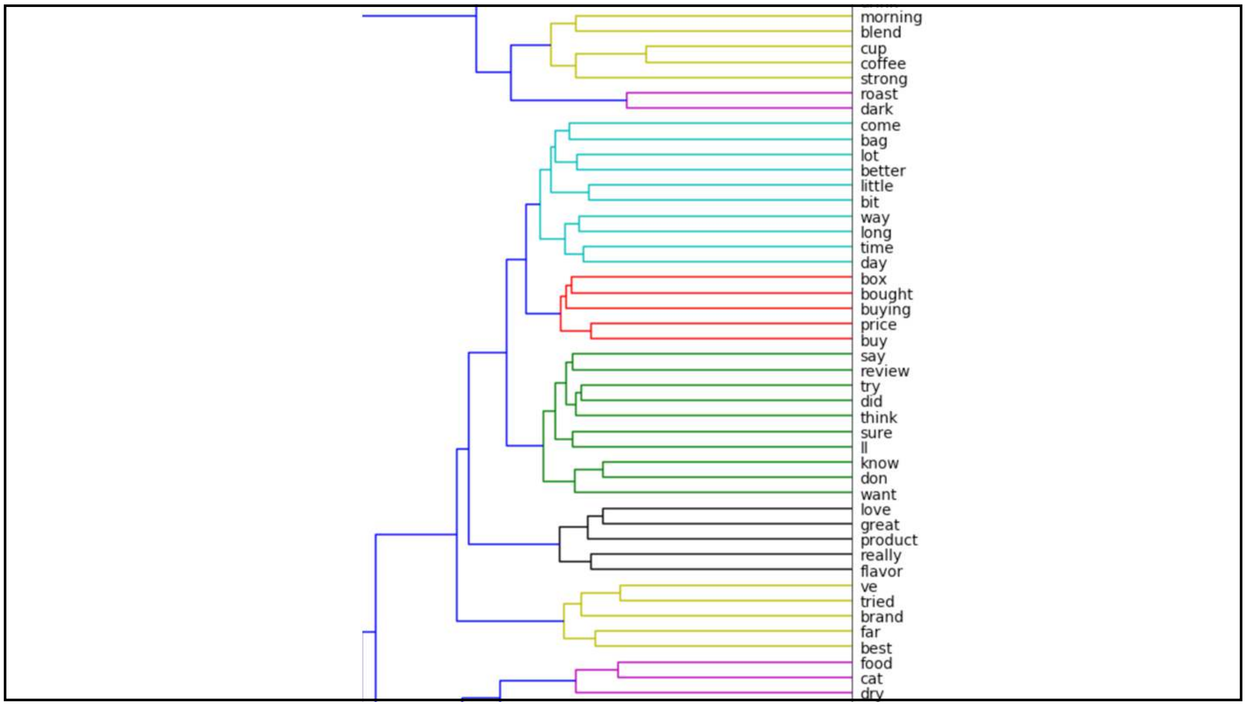
```
[ ] from scipy.cluster.hierarchy import linkage, dendrogram
    from sklearn.metrics.pairwise import cosine_similarity

    # We'll use cosine similarity to get word similarities based on docu
    # This produces a matrix of every word compared to every other word
    # With a value of 0 - 1, indicating how often they occur together ir
    # To get document similarities in terms of word overlap, just drop t
    similarities = cosine_similarity(tfidf.transpose())
    distances = 1 - similarities # Converts to distances
    clusters = linkage(distances, method='ward') # Run hierarchical clus
```

```
[ ] fig, ax = plt.subplots(figsize=(10, 40))
    ax = dendrogram(
        clusters,
        labels=ngrams,
        orientation="left",
        leaf_font_size=14,
        color_threshold=1.5
    )
    plt.tight_layout()
```




wonderful
tasting
organic
company
extra
excellent
type
enjoy
cereal
hard
candy
tasty
fresh
salt
chip
friend
family
loved
kid
milk
chocolate
cooky
rice
chicken
meal
sauce
hot
home
having
happy
week
problem
need
help
work
doe
regular
diet
health
eating
eat
feel


piece
minute
small
size
large
pack
ounce
le
expensive
shipping
cost
worth
money
package
came
received
order
ordered
case
perfect
arrived
purchase
item
purchased
oz
bottle
water
add
mix
use
make
easy
using
used
oil
coconut
recommend
highly
black
bean
vanilla
flavored
tea
green
drink



Thank you!

https://bit.ly/2rlCOUG

Full link: https://colab.research.google.com/github/patrickvankessel/AAPOR-Text-Analysis-2019/blob/master/Tutorial.ipynb
GitHub repo: https://github.com/patrickvankessel/AAPOR-Text-Analysis-2019

Feel free to reach out:
pvankessel@pewresearch.org
patrickvankessel@gmail.com

Special thanks to Michael Jugovich for help putting these materials together for previous workshops