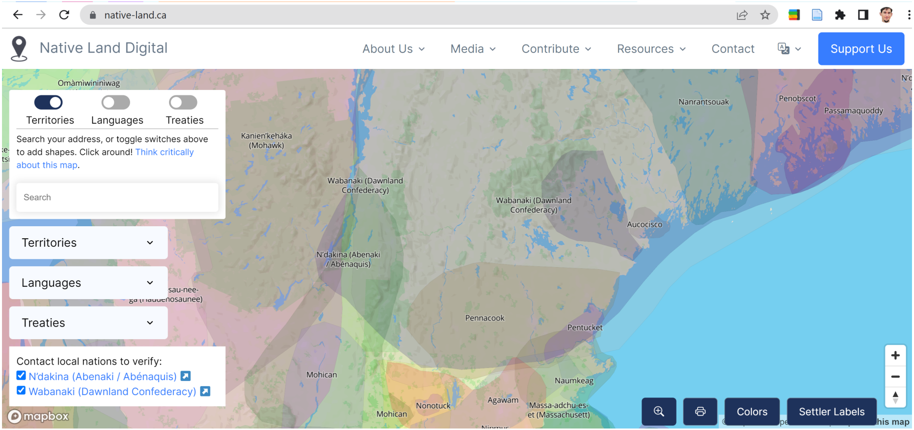
QSS20: Modern Statistical Computing

Unit 09: Text as data

Goals for today

- Pset logistics
- Recap of fuzzy matching
- Lecture & code walkthrough: text as data!

Land acknowledgment



Happy Indigenous People's Day!

Goals for today

- **Pset logistics**
- Recap of fuzzy matching
- Lecture & code walkthrough: text as data!

Pset logistics

- 6 of 8 groups have submitted pset 2
- Pset 2 solutions file and blank pset 3 will be uploaded this week
- Confused by a module/method? Then own it! Consider contributing to GitHub Wiki function dictionary
- Reminders:
  - To use a late day, let Prof & TA know via private Piazza message
  - Files to upload: '.ipynb' and '.html'
  - Questions? Come to office hours or group tutoring, or ask class via Piazza!
- Questions?

Goals for today

- Pset logistics
- **Recap of fuzzy matching**
- Lecture & code walkthrough: text as data!

Recap of fuzzy matching

What do you remember?

Recap of fuzzy matching: Tips

- "Fuzzy matching" tries to match between datasets when you don't have a shared unique identifier between them. You make a best guess (i.e., a "fuzzy" guess) as to what row from dataset A matches what row from dataset B (if any)
- Fuzzy matching starts from precise information you DO have: Columns shared between the two datasets that are reliable, e.g. name or (especially) address information like zip code.
- Using a reliable exact match (e.g., zip code), you first "block" out your guesses by trying to match a given row from main dataset to ONLY those secondary rows that match this column (e.g., businesses in same zip code).
- You then construct a similarity metric from which to make a "fuzzy guess". Some metrics are based on "edit distance": how many chars need be added, removed, or swapped to make string A into string B? Others use "jaccard distance": thinking of each string as a vector, what's the intersection of the two–shared characters–vs. the union–total number of chars?
- After blocking and computing distances, you look for the candidate match (within the block) lowest distance to a given main dataset row.

Recap of fuzzy matching: Useful commands

```
df.colname.astype(str).str[:2] # for slicing first two chars of
  string, e.g. identifiers
df.colname.apply(lambda id: str(id)[:2]) # another way to slice

nltk.edit_distance(stringA, stringB) # get edit distance
nltk.jaccard_distance(set(stringA), set(stringB) # get
  jaccard distance

my_recordmatcher = recordlinkage.Index(); my_recordmatcher.
  block(blockvar) # how to block with recordinlinkage package
candidate_links = my_recordmatcher.index(main, aux) # create
  candidate links
compare = recordlinkage.Compare(); compare_vectors =
  compare.compute(candidate_links, main, aux) # compute
  comparisons using candidate links

kmeans = recordlinkage.KMeansClassifier() # initialize bad
  classifier
```

General workflow for probabilistic matching, regardless of package

1. **Preprocess the relevant fields in the data:** none of these algorithms are magic bullets; each can have significant gains from basic string preprocessing of the relevant fields (e.g., should we remove `LLC`?; how are street addresses formulated)
2. **Decide if/what to "block" or exact match on:** when creating the candidate pairs, what's a *must have* field where if they don't match exactly, you rule out as a candidate pair?
   - **How do you decide this:** fields that are more reliably formatted (e.g., two-digit state)
   - **Main advantages:** potentially reduces false positives; reduces runtime/computational load
3. **If blocking, creating candidate pairs based on blocking variables:** if we blocked on state, for instance, this would leave the two IL businesses as candidate pairs for our focal business
4. **Decide on what fields to match "fuzzily":** these are things like name, address, etc. that might have typos/different spellings. The two components are:
   - How to define similarity: string distance functions
   - What threshold counts as similar enough
5. **Within candidate pairs, look at those fuzzy fields**
6. **Aggregate across fields to decide on "likely match" or "likely not"**

Where we are

- Pset logistics
- Recap of fuzzy matching
- **Lecture & code walkthrough: text as data!**

Outline of text as data

- Text as data: where can we find?
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

Outline of text as data

- **Text as data: where can we find?**
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

Where can you find text to use as data?

- **General guide**: just as an ethnographer thinks carefully about a field site, begin with your substantive interests–e.g., how do police treat residents of different races? How do college students share knowledge about Dartmouth's hidden curriculum–and think about text generated as things unfold in that area
- Two broad types:
  1. **One-way text outputs:** official documents (e.g., legislation; news articles; court cases); informal broadcasts (tweets, Yelp reviews, 311 complaints, and other social media data); informal notes professionals write about clients (e.g., caseworker notes; free text fields in medical records)
  2. **Two-way dialogues/interactions (may involve transforming video data $\implies$ audio data $\implies$ text)**: transcripts from body camera data (Voigt et al. 2017); transcripts from physician-patient conversations (Hagiwara et al. 2017); message board data (Dimaggio et al., 2019); Slack data

Where can you find *openly-available* text to use as data?

- Usually combined with web scraping or using an API to acquire efficiently. Examples with clickable links:
- Kaggle text data: DOJ press releases; IMDB movie reviews data
  - Example: "If you like original gut wrenching laughter you will like this movie. If you are young or old then you will love this movie, hell even my mom liked it."
- Restaurant reviews dataset
- NYC housing code violations data
  - Example: "Abate the nuisance consisting of roaches in the entire apartment"
- Congressional bills data
- Tutorial on scraping Craigslist, which can be used to study things like how people describe gentrifying neighborhoods
- Job addendums: "Workers may be subject to disciplinary action for failing to obtain employer's permission for a personal long-distance call or to repay the cost of such call within a reasonable time."

Outline of text as data

- Text as data: where can we find
- **Text mining/"supervised" analyses: know what we're looking for in advance**
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

What do I mean by "supervised"?

1. *Text mining:* look for a pre-specified concept or category. Methods:
   - **Pattern matching**: look for a match for a specific sequence of characters
   - **Dictionary**: we have a list of words we think represent a category or concept (e.g., if we want to classify a review as negative, we might have a list of words or phrases we think represent the category like *boring*; *terrible*; *awful*; *literally the worst*)
2. *Supervised machine learning for classification:* pre-specify a category at the document level and learn how text predicts that category
   - Inputs, training data:
     - **Text**: movie review; legislative bill; message board chain; admissions essay
     - **Label for that text**: negative or not; Repub. sponsor or not; contentious or not; accepted or not
   - **Method**: often binary classification
   - **Output**: classifier that one can use for unlabeled data

Example of combining text mining with supervised ML

- Had human raters code snippets of transcripts to generate labels of whether the interaction was "respectful" or not in a smaller sample of documents
- Generated features from the text using dictionary-based methods, e.g.
  - Informal titles: ["dude*", "bro*", "boss", "bud", "buddy", "champ", "man", "guy*", "guy", "brotha", "sista", "son", "sonny", "chief"]
  - Time minimizing: "(minu-te—min—second—sec—moment)s?—this[^ .,?!]+quick—right back"
- Built model to predict respect ratings using those features

Language from police body camera footage shows racial disparities in officer respect

Rob Voigt[a,1], Nicholas P. Camp[b], Vinodkumar Prabhakaran[c], William L. Hamilton[c], Rebecca C. Hetey[b], Camilla M. Griffiths[b], David Jurgens[c], Dan Jurafsky[a,c], and Jennifer L. Eberhardt[b,1]

[a]Department of Linguistics, Stanford University, Stanford, CA 94305; [b]Department of Psychology, Stanford University, Stanford, CA 94305; and [c]D of Computer Science, Stanford University, Stanford, CA 94305

Contributed by Jennifer L. Eberhardt, March 26, 2017 (sent for review February 14, 2017; reviewed by James Pennebaker and Tom Tyler)

Using footage from body-worn cameras, we analyze the respectfulness of police officer language toward white and black community members during routine traffic stops. We develop computational linguistic methods that extract levels of respect automatically from transcripts, informed by a thin-slicing study of participant ratings of officer utterances. We find that officers speak with consistently less respect toward black versus white community members, even after controlling for the race of the officer, the severity of the infraction, the location of the stop, and the outcome of the stop. Such disparities in common, everyday interactions between police and the communities they serve have important implications for procedural justice and the building of police–community trust.

racial disparities | natural language processing | procedural justice | traffic stops | policing

some have argued that racial disparities in perceived during routine encounters help fuel the mistrust of the controversial officer-involved shootings that have such great attention. However, do officers treat white nity members with a greater degree of respect than th to blacks?

We address this question by analyzing officers' during vehicle stops of white and black community Although many factors may shape these interactions, a words are undoubtedly critical: Through them, the o communicate respect and understanding of a citizen's tive, or contempt and disregard for their voice. Furt the language of those in positions of institutional pow officers, judges, work superiors) has greater influence course of the interaction than the language used by t less power (12–16). Measuring officer language thus a quantitative lens on one key aspect of the quality o police–community interactions, and offers new opport

Our working example: NYC airbnb listings

name	neighbourhood_group	price
Nice and cozy little apt available	Bronx	75
Cozy and privat studio near Times Sq	Manhattan	140
NYCT02-3: Private Sunny Rm, NYU, Baruch, SOHO	Manhattan	75
Prime area in Chinatown and Little Italy	Manhattan	160
Midtown Manhattan Penthouse	Manhattan	100
2BR Comfy Apt - 15min from MIDTOWN	Queens	150
FourTwin bunkbeds- 5 minutes from JFK	Queens	90
Pvt Room in Quiet Home JFK 6mi LGA 10mi	Queens	38

Key variables: `name`: descriptive listing; `neighbourhood_group`: borough; `price`: price of listing
Where you can find: QSS20_public/public_data/airbnb_text.zip

What are some interesting text features that might be correlated with price?

Positive words/euphemisms: nice; cozy; privat/private/pvt; comfy
Proximity to landmarks: Little Italy; Chinatown; NYU; SOHO; Times Sq
Proximity to airports: JFK; LGA

name	neighbourhood_group	price
Nice and cozy little apt available	Bronx	75
Cozy and privat studio near Times Sq	Manhattan	140
NYCT02-3: Private Sunny Rm, NYU, Baruch, SOHO	Manhattan	75
Prime area in Chinatown and Little Italy	Manhattan	160
Midtown Manhattan Penthouse	Manhattan	100
2BR Comfy Apt - 15min from MIDTOWN	Queens	150
FourTwin bunkbeds- 5 minutes from JFK	Queens	90
Pvt Room in Quiet Home JFK 6mi LGA	Queens	38

How might we go about creating indicators for whether the listing contains those attributes?

Code to follow along

https://github.com/jhaber-zz/QSS20_public/blob/main/activities/07_textasdata_partI_textmining.ipynb

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - **Manual lookup of words or counting words from a dictionary**
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

Manual approach 1: looking for single word

```python
## using the `name_upper` var, look at where reviews mention cozy
ab['is_cozy'] = np.where(ab.name_upper.str.contains("COZY"),
                    True, False)

## find the mean price by neighborhood and whether mentions cozy
mp = pd.DataFrame(ab.groupby(['is_cozy',
                    'neighbourhood_group'])['price'].mean())

## reshape to wide format so that each borough is row
## and one col is the mean price for listings that describe
## the place as cozy; other col is mean price for listings
## without that word
mp_wide = pd.pivot_table(mp, index = ['neighbourhood_group'],
                         columns = ['is_cozy'])

mp_wide.columns = ['no_mention_cozy', 'mention_cozy']
```

Result: within the same borough, higher prices in Airbnbs that don't describe the listing as cozy

neighbourhood_group	no_mention_cozy	mention_cozy
Bronx	89.231088	74.214286
Brooklyn	128.175441	91.130224
Manhattan	204.109775	129.917140
Queens	102.596682	80.344388
Staten Island	120.650307	74.319149

Manual approach 2: create dictionary of words summarizing concept and score each listing

```

## construct dictionary
space_indicators = {'small': ['COZY', 'COMFY', 'LITTLE', 'SMALL'],
                    'large': ['SPACIOUS', 'LARGE', 'HUGE', 'GIANT']}


## for each listing, find the number of occurrences
## of words in each key

### first, let's test with one listing
practice_listing = "NICE AND COZY LITTLE APT AVAILABLE"
```

Counting the number of appearances in one listing (double counts if appears twice)

```
### example string
practice_listing = "NICE AND COZY LITTLE APT AVAILABLE"

### splitting at space and looking at overlap with each key in the
    dictionary
words_overlap_small = [word for word in practice_listing.split(" ")
                        if word in space_indicators['small']]

words_overlap_large = [word for word in practice_listing.split(" ")
                       if word in space_indicators['large']]

### could then take length as a fraction of all words
len(words_overlap_small)/len(practice_listing.split(" "))
len(words_overlap_large)/len(practice_listing.split(" "))
```

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - **Automated process 1: part-of-speech tagging**
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

Intro to part-of-speech tagging (POS) and named entity recognition (NER)

- Previous approach was very manual — we needed to read some reviews and manually construct a dictionary summarizing adjectives we thought were related to a concept
- We also didn't yet capture other price-relevant attributes of the review, or what we might call `named entities`
  1. **Places:** e.g., Chinatown, Little Italy, Times Square
  2. **Infrastructure** e.g., LGA; JFK

Part of speech tagging with example

```
## specify example
example_for_tag = "This is a chill apt next to the subway in LES
    Chinatown"

## try part of speech tagging using nltk
tokens = word_tokenize(example_for_tag) # Generate tokens
tokens_pos = pos_tag(tokens) # generate part of speech tags for
    those tokens
```

Output: a list of tuples where the first element in the tuple is the original word; second element in the tuple is the part of speech

```
for one_tok in tokens_pos:
    print(one_tok)
```

```
('This', 'DT')
('is', 'VBZ')
('a', 'DT')
('chill', 'NN')
('apt', 'JJ')
('next', 'JJ')
('to', 'TO')
('the', 'DT')
('subway', 'NN')
('in', 'IN')
('LES', 'NNP')
('Chinatown', 'NNP')
```

What do these mean? Common parts of speech

"This is a chill apt next to the subway in LES Chinatown"

tag	meaning	in our example
CC	coordinating conjunction	
CD	cardinal digit	
DT	determiner	This; the; a
JJ	adjective	apt; next
JJR	adjective (comparative; e.g., bigger)	
NN	noun (singular; e.g., desk)	chill; subway
NNS	noun (plural; e..g, desks)	
NNP	proper noun (singular; e.g., Harrison)	LES; Chinatown
NNPS	proper noun (plural; e.g., Americans)	
TO	to	
UH	interjection	
VB	verb (base form; e.g., take)	
VBD	verb (past form; e.g., took)	
VBG	verb (gerund/present; e.g., taking)	
VBN	verb (past participle; e.g., taken)	
VBZ	verb (3rd person singular present; e.g. takes)	

What if, after tagging, we want to extract the words from our text containing a specific part of speech?

Example: in our example string, extract the proper nouns (LES and Chinatown)

```
## use list iteration to extract proper nouns
## i'm first checking if the second element in the tuple is equal to NNP
## if so, i'm returning the first element of the tuple (the
## actual word)
all_prop_noun = [one_tok[0] for one_tok in tokens_pos
                 if one_tok[1] == "NNP"]
all_prop_noun
```

Output:

```
all_prop_noun

['LES', 'Chinatown']
```

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - **Automated process 2: named entity recognition**
  - Automated process 3: sentiment analysis using a scoring dictionary
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

Named entity recognition

- Previous was useful for broad categories — e.g., LES and Chinatown both tagged as proper nouns
- With named entity recognition, we want to be able to classify proper nouns into more granular subtypes. See spaCy label schemes or this blog for a longer list of types; some relevant ones:
  - `PERSON`: e.g., Professor Xavier
  - `FAC`: building; highway; bridges - e.g., Boston Logan International Airport
  - `GPE`: countries; cities; states- e.g., Hanover, NH
  - `ORG`: organizations; e.g., Dartmouth College
  - `DATE`: e.g., October 10th, 2022
- To execute, we switch from `nltk` package to `spaCy` package

Example tweet to search for named entities

We'll be hosting on-campus COVID-19 booster clinics at Dartmouth College in New Hampshire from 9 a.m. to 6 p.m. on Monday, Jan. 10, and Tuesday, Jan. 11, at Alumni Hall in the Hopkins Center. For information on how to register and additional winter updates, head to

Which words do we think will be tagged as named entities?

Code to get named entities from that tweet

```
spacy_dtweet = nlp(d_tweet)
for one_tok in spacy_dtweet.ents:
    print( "Entity: " + one_tok.text +
           "; NER tag: " + one_tok.label_)
```

Breaking this down:

- `nlp`: black-boxy function within spacy that adds tags to a string (not only named entities)
- `spacy_dtweet.ents`: extracting all named entities from the spacy object
- `one_tok`: arbitrary placeholder for one entity
- `one_tok.text`: original string
- `one_tok.label_`: named entity label for that string

Output of named entities in tweet

We'll be hosting on-campus COVID-19 booster clinics at Dartmouth College in New Hampshire from 9 a.m. to 6 p.m. on Monday, Jan. 10, and Tuesday, Jan. 11, at Alumni Hall in the Hopkins Center. For information on how to register and additional winter updates, head to

```
Entity: Dartmouth College; NER tag: ORG
Entity: New Hampshire; NER tag: GPE
Entity: 9 a.m. to 6 p.m.; NER tag: TIME
Entity: Monday, Jan. 10; NER tag: DATE
Entity: Tuesday, Jan. 11; NER tag: DATE
Entity: Alumni Hall; NER tag: WORK_OF_ART
Entity: the Hopkins Center; NER tag: FAC
```

Coding break

Play around with different variations of the Dartmouth tweet and look at the results. E.g.:

- What happens if you abbreviate New Hampshire to NH?
- What happens if you add the word Pfizer before COVID-19?
- What entities seem misclassified?

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - **Automated process 3: sentiment analysis using a scoring dictionary**
- Unsupervised analyses: how can we more inductively discover themes/patterns in text?
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - Topic model: implementation

Sentiment analysis: dictionary-based approach

- Operates similarly to our manual dictionary but, in this case, keys are words in a "lexicon"; values are the sentiment score
- In basic form, a dictionary of two types of words (often non-exhaustive, where others treated as neutral):
  1. Positive
  2. Negative

Code for VADER sentiment scoring: calc. sentiment

```
## initialize a scorer
sent_obj = SentimentIntensityAnalyzer()

## score one listing
practice_listing = "NICE AND COZY LITTLE APT AVAILABLE"
sentiment_example = sent_obj.polarity_scores(practice_listing)
```

Breaking this down:

- `sent_obj = SentimentIntensityAnalyzer()`: initializing a scorer
- `sent_obj.polarity_scores(practice_listing)`: from that initialized scorer, apply the polarity scores function to the single string we're feeding it
  - Score is aggregated to the level of the text you feed it; e.g., here we're scoring a sentence; might score a paragraph or document

Code for VADER sentiment scoring: what the output is

Dictionary with four keys: neg, neu, pos, compound (summary of pos, neg, neutral; standardized to be between -1 = most negative to +1 = most positive)

```
print("String: " + practice_listing + " scored as:\n" + str(sentiment_example))
print("String: " + practice_listing_2 + " scored as:\n" + str(sentiment_example_2))
print("String: " + practice_listing_3 + " scored as:\n" + str(sentiment_example_3))
```

```
String: NICE AND COZY LITTLE APT AVAILABLE scored as:
{'neg': 0.0, 'neu': 0.641, 'pos': 0.359, 'compound': 0.4215}
String: NICE AND COZY LITTLE APT AVAILABLE. REALLY TERRIBLE VIEW. scored as:
{'neg': 0.257, 'neu': 0.531, 'pos': 0.212, 'compound': -0.1513}
String: NICE AND COZY LITTLE APT AVAILABLE. HAS RATS THOUGH. scored as:
{'neg': 0.0, 'neu': 0.741, 'pos': 0.259, 'compound': 0.4215}
```

Issues:

- Many words classified as neutral
- Appropriately added Terrible to negative score, but didn't know the context-specific rats should be scored negatively

One way to improve: augment the default VADER dictionary

```python
## create a dictionary with
## negative scores for pests
pest_words = {
    'rat': -1.9, 'rats': -1.9,
    'mice': -1.9, 'mouse': -1.9,
    'roach': -1.9, 'cockroach': -1.9
}
## update the lexicon with that
## dictionary (created new sentiment object
## to avoid writing over older one)
new_si = SentimentIntensityAnalyzer()
new_si.lexicon.update(pest_words)
## use that updated scorer
new_si.polarity_scores(practice_listing_3)
```

Output (went from 0 negative to 0.228 negative):

```python
print("After lexicon update: " + practice_listing_3 + " scored as:\n" + \
      str(new_si.polarity_scores(practice_listing_3)))
```

```
After lexicon update: NICE AND COZY LITTLE APT AVAILABLE. HAS RATS THOUGH. scored as:
{'neg': 0.228, 'neu': 0.551, 'pos': 0.22, 'compound': -0.0258}
```

Better way to improve: build a custom classifier

		terms in listing					
listing_id	avg_stars	cozy	rat	spacious	cable	marble	...
1	2.4	1	1	0	0	0	
2	3.8	0	0	1	0	1	
3	4.9	1	0	1	1	0	
⋮							

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- **Unsupervised analyses: how can we more inductively discover themes/patterns in text?**
  - **Bag of words representation of text/preprocessing**
  - Topic model: concepts
  - Topic model: implementation in python

Text mining of Airbnb listings versus topic modeling

- Suppose we were interested in looking at relationship between (1) what words people use to describe their airbnb listing and (2) neighborhood change (e.g., rapid demographic change, as measured through changes in ethnicity/income of those residing in the neighborhood)
- **Text mining approach**: build a dictionary of words or phrases we think signal gentrifying (*cute*; *safe*; *near cold brew*) and look at correlation with neighborhood change
- Drawbacks:
  - Might be difficult to know in advance which words to include
  - Lack of surprise: what if there's a pattern in the listings correlated with demographic change, but that we didn't anticipate?
- Therefore, rather than search for specific words or phrases, begin with *full text* of the document

Tokenize/represent document as a bag of words

- Represent each document as a "bag of words", where order doesn't matter
- Examples:

```
['clean', '&', 'quiet', 'apt', 'home', 'by', 'the', 'park']
['skylit', 'midtown', 'castle']
['the', 'village', 'of', 'harlem', '....', 'new', 'york', '!']
['cozy', 'entire', 'floor', 'of', 'brownstone']
```

- Notice that it contains a lot of extraneous information

Repeat with each document and then represent as document-term matrix

doc	1br	apartment	apt	area	backyard	bdrm …
1	0	0	1	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	1	0	0	0
⋮						

How do we do this in Python?

Switch to: 07_textasdata_partII_topicmodeling...

Step one: create stopword list to filter out

Why do this early? Especially if you want to create your own list of stopwords for your context, it's easier to do that before additional preprocessing that alters the words (e.g., might abbreviate apartment to apart)

```
## call the specific module
from nltk.corpus import stopwords

## call a specific set of stopwords from package
list_stopwords = stopwords.words('english')

## augment with your own
list_stopwords = stopwords.words("english")

custom_words_toadd = ['apartment', 'new york', 'nyc',
                       'bronx', 'brooklyn',
                      'manhattan', 'queens',
                      'staten island']

list_stopwords_new = list_stopwords + custom_words_toadd
```

Step two: convert text to lowercase and filter out stopwords

```python
## convert to lowercase and a list
corpus_lower = ab.name.str.lower().to_list()

## use wordpunct tokenize and filter out with one
example_listing = corpus_lower[0]
nostop_listing = [word for word in wordpunct_tokenize(
    example_listing)
                  if word not in list_stopwords_new])
nostop_listing
```

Before:

```
['cozy', 'entire', 'floor', 'of', 'brownstone']
```

After (removes by and the):

```
['cozy', 'entire', 'floor', 'brownstone']
```

Step three: stem and additional preprocessing

```
## initialize stemmer
porter = PorterStemmer()

## apply to one tokenized text by iterating
## over the tokens in the text
example_listing_preprocess = [porter.stem(token)
                              for token in nostop_listing
                              if token.isalpha() and
                              len(token) > 2]
```

Output:

```
['cozi', 'entir', 'floor', 'brownston']
```

Breaking it down:

- `if token.isalpha()`: only retaining token if it's words (so would strip out things like 1 from 1 bedroom); might skip depending on context
- `len(token) > 2`: requires that a token is 2 or more characters; e.g., removes br
- `porter.stem(token)`: use the porter stemmer i've initialized to stem the words; e.g., entire $\Longrightarrow$ entir; cozy $\Longrightarrow$ cozi

Next up...

- With that preprocessed text, we'll learn how to create a "document-term" matrix where each row is one text (in this case, one Airbnb listing); each col is a term; values are 0, 1 for presence or absence of that term
- Concepts and mechanisms of using topic modeling to cluster that matrix

Small group code break: embed the preprocessing code in 1-2 functions and apply to all the airbnb listings

The previous code used list comprehension to iterate over each word in a single airbnb listing.

To apply to all listings, and to avoid a nested for loop, we want to:

1. Create a function(s) that applies those preprocessing steps (could have one function for stopword removal; another for stemming; or one combined)
2. Iterate over listings and preprocess. Output could either be a list where each list element is a string of a list (e.g., ‘cozy brownstone apt‘), or a list of lists where each element is a tokenized string (e.g., [‘cozy’, ’brownstone’, ’apt’])

Output is flexible (could be a list of lists containing tokenized/stemmed text or a list of strings)

Repeat over all documents, and combine into a document-term matrix

- **More manual way**: basically, need to find union of all words; can do it by (1) creating an empty dictionary; (2) looping over the documents; (3) when a document contains a new term, it gets added to dictionary as a key; (4) when a document contains a term already in the dictionary, we start counting how many times the term appears in the doc
- **More automatic way**: uses `sklearn` function

Code for more automatic document-term matrix creation

```
def create_dtm(list_of_strings, metadata):
    ## init tokenizer and apply to list of documents
    vectorizer = CountVectorizer(lowercase = True)
    dtm_sparse = vectorizer.fit_transform(list_of_strings)
    ## convert to (1) dense mat; (2) dataframe and (3) add metadata
    dtm_dense_named = pd.DataFrame(dtm_sparse.todense(),
                          columns=vectorizer.get_feature_names())
    dtm_dense_named_withid =pd.concat([metadata.reset_index(),
                                dtm_dense_named], axis = 1)
    return(dtm_dense_named_withid)
```

Breaking things down:

- `CountVectorizer`: initializes a sklearn tokenizer; this helps us tokenize the preprocessed string
- `vectorizer.fit_transform`: we feed this a list of documents (each document can be many strings). The output is a sparse matrix where each row is a document; each column is a term; 0 if term t is not in doc d, 1 if term t is in doc d (sparse representation saves memory given prevalence of zeroes)
- `dtm_sparse.to_dense()`: if we want to treat the sparse matrix as a normal pandas dataframe, we need to switch it from the sparse representation to the normal dense representation
- Remainder of code just (1) converts the dense matrix to a pandas dataframe to work with; (2) adds back document-level covariates (what I'm calling metadata)

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- **Unsupervised analyses: how can we more inductively discover themes/patterns in text?**
  - Bag of words representation of text/preprocessing
  - **Topic model: concepts**
  - Topic model: implementation in python

Latent Dirichlet Allocation

- Idea: documents exhibit each topic in some proportion. This is an admixture.
- Each document is a mixture over topics. Each topic is a mixture over words.

- Latent Dirichlet Allocation estimates:
  - The distribution over words for each topic.
  - The proportion of a document in each topic, for each document.

Maintained assumptions: Bag of words/fix number of topics ex ante.

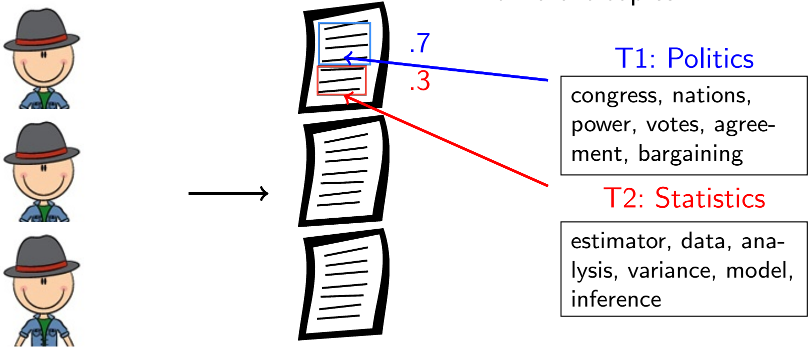
This and next slide with visualization from: Stewart, LDA

What this means in pictures

Say you have a lot of people.

Each writes some texts

That discuss a few different topics



The Latent Dirichlet Allocation estimates:

1. The topics- each is a distribution over words
2. The proportion of each document in each topic

Why does this work $\rightsquigarrow$ Co-occurrence

Where's the information for each word's topic?

Reconsider document-term matrix

	Word_1	Word_2	$\ldots$	Word_J
Doc_1	0	1	$\ldots$	0
Doc_2	2	0	$\ldots$	3
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
Doc_N	0	1	$\ldots$	1

We are learning the pattern of what words occur together.

The model wants a topic to contain as few words as possible, but a document to contain as few topics as possible. This tension is what makes the model work.

From: Stewart, LDA

Outline of text as data

- Text as data: where can we find
- Text mining/"supervised" analyses: know what we're looking for in advance
  - Manual lookup of words or counting words from a dictionary
  - Automated process 1: part-of-speech tagging
  - Automated process 2: named entity recognition
  - Automated process 3: sentiment analysis using a scoring dictionary
- **Unsupervised analyses: how can we more inductively discover themes/patterns in text?**
  - Bag of words representation of text/preprocessing
  - Topic model: concepts
  - **Topic model: implementation in python**

Two routes to topic modeling

1. Create the document-term matrix yourself and then it's compatible with a variety of clustering methods
2. Use built-in functions in `gensim` to start with a list of preprocessed documents and end in estimating a topic model that returns (1) topics and high-probability words, (2) for each document, a k length vector of topic probabilities, where k is the number of topics

Steps for topic modeling using `gensim`: in words

- **Create a dictionary**: this is the union of all stemmed/preprocessed words in the corpus (collection of documents); it's fed tokenized text; results in dictionary where keys are a "term id"; value is word itself
- **Filter out words from the dictionary that appear in either a very low proportion of documents (lower bound) or a very high proportion of documents (upper bound)**: stopword removal should have gotten rid of most of the latter; former is since we need words to co-occur in multiple documents for the themes to be meaningful
- **Apply the dictionary to the tokenized text to create a crosswalk between: (1) each word in the text and (2) that word in the filtered dictionary**: this is a final preprocessing that helps get rid of words in the original texts that were filtered out of the corpus dictionary
- **Estimate the topic model**: use LDA model within gensim

Steps for topic modeling using gensim: preprocessing

```
## Step 1: tokenize documents and store in list
text_raw_tokens = [wordpunct_tokenize(one_text)
                   for one_text in ab_small.name_lower]
## Step 2: use gensim create dictionary - gets all unique
    words across documents
text_raw_dict = corpora.Dictionary(text_raw_tokens)
## Step 3: filter out very rare and very common words
   from dictionary; feeding it n docs as lower and upper
    bounds
text_raw_dict.filter_extremes(no_below = lower_bound,
                                  no_above = upper_bound)
## Step 5: map words in dictionary to words in each
   document
## in the corpus
corpus_fromdict = [text_raw_dict.doc2bow(one_text)
                      for one_text in text_raw_tokens]
```

Steps for topic modeling using gensim: estimation

See documentation for many parameters you can vary!:
https://radimrehurek.com/gensim/models/ldamodel.html

```
## Step 6: estimate a model by feeding it: (1) the corpus
## mapped to the dictionary, (2) the dictionary itself,
## (3) number of topics (often k if you're reading
   notation),
## and assorted other arguments
ldamod = gensim.models.ldamodel.LdaModel(corpus_fromdict,
                                 num_topics = 5,
                                 id2word=text_raw_dict,
                                 passes=6, alpha = 'auto',
                                 per_word_topics = True)
```

Returns a trained ldamodel class with various methods/attributes

Interacting with the model output

Notebook contains a couple different post-model summaries:

- **Top words for each topic:** by default, these are the highest-probability words; but they also may just reflect frequently-occuring words in corpus; `pyldavis` has ways to introduce penalties to find words more "unique to" a topic
- **For each document, a k-length vector of topic probabilities**

Post-modeling diagnostics: how model fit varies as function of number of topics

- **Concept:** tradeoff between two metrics:
  1. **Within-topic coherence:** increases when the "top words" for a topic (highest-probability words) tend to co-occur in the same document; tends to be higher if you have a few topics dominated by frequently-occurring words
  2. **Between-topic exclusivity:** increases when words are "exclusive" to a topic, or only have a high-probability of appearing in a few topics; tends to be higher as you increase the number of topics, since each is more granular
- **See here for some code snippets within gensim:** https://datascienceplus.com/evaluation-of-topic-modeling-topic-coherence/

Many different types of topic models

- **Structural topic model**: allow (1) topic prevalence, (2) topic content to vary as a function of document-level covariates (e.g., how do topics vary over time or documents produced in 1990 talk about something differently than documents produced in 2020?); implemented in `stm` in R (Roberts, Stewart, Tingley, Benoit)
- **Correlated topic model:** way to explore between-topic relationships (Blei and Lafferty, 2017); implemented in `topicmodels` in R; possibly somewhere in Python as well!
- **Keyword-assisted topic model:** seed topic model with keywords to try to increase the face validity of topics to what you're trying to measure; implemented in `keyATM` in R (Eshima, Imai, Sasaki, 2019)
- And more...

Extensions thus far to material we're not covering

- **N-grams**: similar setup but modify `create_dtm` to use bigrams.
- **Word embeddings**: bag of words treats all text in document as an unordered collection. But we might think the context of the word (e.g., cozy brownstone versus spacious brownstone) matters, and we want to have more flexible windows than n-grams. Basic idea behind embeddings is to (1) predict a focal word (e.g., "cozy"); (2) predict its context within some window (e.g., "brownstone", "tiny", "cramped"). Each word is then represented as a vector, and vectors can be related to each other—e.g., "doctor:nurse"; "female:male." Good intro here.
  - For description: can look at relationships between words over time
  - Classification: if using words to predict some outcome (e.g., respectful rating), alternate representation to a document-term matrix

Coding break

Second part of activity in
https://github.com/jhaber-zz/QSS20_public/blob/main/activities/07_textasdata_partII_topicmodeling.ipynb