

# QSS20: Modern Statistical Computing

## Unit 08: Text as data

# Outline

- ▶ 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

- ▶ **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  $\Rightarrow$  audio data  $\Rightarrow$  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 interacting with 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

- ▶ 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

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

Rob Voigt<sup>1</sup>, Nicholas R. Camp<sup>2</sup>, Vinodkumar Prabhakaran<sup>1</sup>, William L. Hamilton<sup>1</sup>, Rebecca C. Hetey<sup>3</sup>, Camilla M. Griffiths<sup>1</sup>, David Jurgens<sup>1</sup>, Dan Jurafsky<sup>1,2</sup>, and Jennifer L. Eberhardt<sup>1</sup>

<sup>1</sup>Department of Linguistics, Stanford University, Stanford, CA 94305; <sup>2</sup>Department of Psychology, Stanford University, Stanford, CA 94305; and <sup>3</sup>Department 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.

Some have argued that racial disparities in perceived respect during routine encounters help fuel the mistrust of the controversial officer-involved shootings that have such great attention. However, do officers treat white and black community members with a greater degree of respect than it is 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 word is undeniably critical. Through them, the officer communicates respect and understanding of a citizen's life, or contempt and disregard for their voice. Past the language of those in positions of institutional power (officers, judges, work superiors) has greater influence on the course of the interaction than the language used by less powerful (12–16). Measuring officer language thus a quantitative lens on one key aspect of the quality of police-community interactions, and offers new insights.

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

- ▶ 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: “(minute—min—second—sec—moment)s?—this[ ^ .,?! ]+quick—right back”
- ▶ Built model to predict respect ratings using those features



## 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_slides_activities/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/rebeccajohnson88/qss20\_slides\_  
activities/blob/main/activities/w22\_activities/solutions/  
06\_textasdata\_partI\_textmining\_examplecode.ipynb
```

# Outline

- ▶ 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

```
1 ## using the `name_upper` var, look at where reviews mention cozy
2 ab['is_cozy'] = np.where(ab.name_upper.str.contains("COZY"),
3                           True, False)
4
5 ## find the mean price by neighborhood and whether mentions cozy
6 mp = pd.DataFrame(ab.groupby(['is_cozy',
7                               'neighbourhood_group'])['price'].mean())
8
9 ## reshape to wide format so that each borough is row
10 ## and one col is the mean price for listings that describe
11 ## the place as cozy; other col is mean price for listings
12 ## without that word
13 mp_wide = pd.pivot_table(mp, index = ['neighbourhood_group'],
14                             columns = ['is_cozy'])
15
16 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

```
1
2 ## construct dictionary
3 space_indicators = {'small': ['COZY', 'COMFY', 'LITTLE', 'SMALL'],
4                       'large': ['SPACIOUS', 'LARGE', 'HUGE', 'GIANT']}
5
6
7 ## for each listing, find the number of occurrences
8 ## of words in each key
9
10 ### first, let's test with one listing
11 practice_listing = "NICE AND COZY LITTLE APT AVAILABLE"
```



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

```
1 ### example string
2 practice_listing = "NICE AND COZY LITTLE APT AVAILABLE"
3
4 ### splitting at space and looking at overlap with each key in the
   dictionary
5 words_overlap_small = [word for word in practice_listing.split(" ")
6                         if word in space_indicators['small']]
7
8 words_overlap_large = [word for word in practice_listing.split(" ")
9                        if word in space_indicators['large']]
10
11 ### could then take length as a fraction of all words
12 len(words_overlap_small)/len(practice_listing.split(" "))
13 len(words_overlap_large)/len(practice_listing.split(" "))
```

# Outline

- ▶ 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

```
1 ## specify example
2 example_for_tag = "This is a chill apt next to the subway in LES
   Chinatown"
3
4 ## try part of speech tagging using nltk
5 tokens = word_tokenize(example_for_tag) # Generate tokens
6 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)	
VCN	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)

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

**Output:**

```
all_prop_noun
```

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

# Outline

- ▶ 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 [here](#) 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
  - ▶ DATE: e.g., May 4th, 2021
- ▶ To execute, we switch from `nltk` package to `spaCy` package



## Example tweet to find named entities in

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

```
1 spacy_dtweet = nlp(d_tweet)
2 for one_tok in spacy_dtweet.ents:
3     print( " Entity: " + one_tok.text +
4           "; 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

- ▶ 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

```
1 ## initialize a scorer
2 sent_obj = SentimentIntensityAnalyzer()
3
4 ## score one listing
5 practice_listing = "NICE AND COZY LITTLE APT AVAILABLE"
6 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

```
1 ## create a dictionary with
2 ## negative scores for pests
3 pest_words = {
4     'rat': -1.9, 'rats': -1.9,
5     'mice': -1.9, 'mouse': -1.9,
6     'roach': -1.9, 'cockroach': -1.9
7 }
8 ## update the lexicon with that
9 ## dictionary (created new sentiment object
10 ## to avoid writing over older one)
11 new_si = SentimentIntensityAnalyzer()
12 new_si.lexicon.update(pest_words)
13 ## use that updated scorer
14 new_si.polarity_scores(practice_listing_3)
```

### Output (went from 0 negative to 0.228 negative):

```
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

listing_id	avg_stars	terms in listing					...
		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

- ▶ 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

# Preprocess to “boost” signal to noise ratio

- ▶ **Compound words:** analyst may want to combine tokens into a single term:  
clean[quiet apt] [home] [by] [the] [park]
- ▶ **Stopword removal:** remove words not related to what we’re studying:  
clean[quiet apt] [home] [by] [the] [park]  
clean[quiet apt] [home] [park]
- ▶ **Stemming:** takes the end off conjugated verbs or plural nouns, just leaving the “stem”:  
clean[quiet apt] [home] [park]  
clean[quiet apt] [hom] [park]
- ▶ **Lemmatization:** similar in goal to stemming but, rather than just stripping from end, truncates with regards to context (e.g., might treat park differently depending on whether it seemed like it was the verb park like “park a car” or the noun park like “a beautiful park”)

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: [06\\_textasdata\\_partII\\_topicmodeling...](#)



## Step one: create stopwords 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)

```
1 ## call the specific module
2 from nltk.corpus import stopwords
3
4 ## call a specific set of stopwords from package
5 list_stopwords = stopwords.words('english')
6
7 ## augment with your own
8 list_stopwords = stopwords.words("english")
9
10 custom_words_toadd = ['apartment', 'new york', 'nyc',
11                       'bronx', 'brooklyn',
12                       'manhattan', 'queens',
13                       'staten island']
14
15 list_stopwords_new = list_stopwords + custom_words_toadd
```

## Step two: convert text to lowercase and filter out stopwords

```
16 ## convert to lowercase and a list
17 corpus_lower = ab.name.str.lower().to_list()
18
19 ## use wordpunct tokenize and filter out with one
20 example_listing = corpus_lower[0]
21 nostop_listing = [word for word in wordpunct_tokenize(
22     example_listing
23     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

```
24 ## initialize stemmer
25 porter = PorterStemmer()
26
27 ## apply to one tokenized text by iterating
28 ## over the tokens in the text
29 example_listing_preprocess = [porter.stem(token)
30                               for token in nostop_listing
31                               if token.isalpha() and
32                               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`  $\Rightarrow$  `entir`; `cozy`  $\Rightarrow$  `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 stopwords 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

```
33 def create_dtm(list_of_strings, metadata):
34     ## init tokenizer and apply to list of documents
35     vectorizer = CountVectorizer(lowercase = True)
36     dtm_sparse = vectorizer.fit_transform(list_of_strings)
37     ## convert to (1) dense mat; (2) dataframe and (3) add metadata
38     dtm_dense_named = pd.DataFrame(dtm_sparse.todense(),
39                                     columns=vectorizer.get_feature_names())
40     dtm_dense_named_withid = pd.concat([metadata.reset_index(),
41                                         dtm_dense_named], axis = 1)
42     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

- ▶ 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/fixed number of topics ex ante.

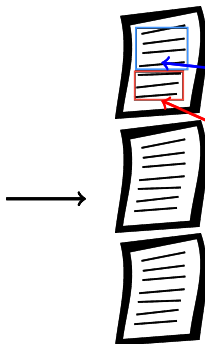
This and next slide with visualization from: Stewart, LDA

# What this means in pictures

Say you have  
a lot of people.



Each writes  
some texts



That discuss a few  
different topics

**T1: Politics**

congress, nations,  
power, votes, agree-  
ment, bargaining

**T2: Statistics**

estimator, data, ana-  
lysis, variance, model,  
inference

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 <sub>1</sub>	Word <sub>2</sub>	...	Word <sub>J</sub>
Doc <sub>1</sub>	0	1	...	0
Doc <sub>2</sub>	2	0	...	3
⋮	⋮	⋮	⋱	⋮
Doc <sub>N</sub>	0	1	...	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

- ▶ 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):** stopwords 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

```
1 ## Step 1: tokenize documents and store in list
2 text_raw_tokens = [wordpunct_tokenize(one_text)
3                     for one_text in ab_small.name_lower]
4 ## Step 2: use gensim create dictionary – gets all unique
   words across documents
5 text_raw_dict = corpora.Dictionary(text_raw_tokens)
6 ## Step 3: filter out very rare and very common words
   from dictionary; feeding it n docs as lower and upper
   bounds
7 text_raw_dict.filter_extremes(no_below = lower_bound,
8                               no_above = upper_bound)
9 ## Step 5: map words in dictionary to words in each
   document
10 ## in the corpus
11 corpus_fromdict = [text_raw_dict.doc2bow(one_text)
12                    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>

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

Break for second part of activity in  
06\_textasdata\_partII\_topicmodeling\_examplecode