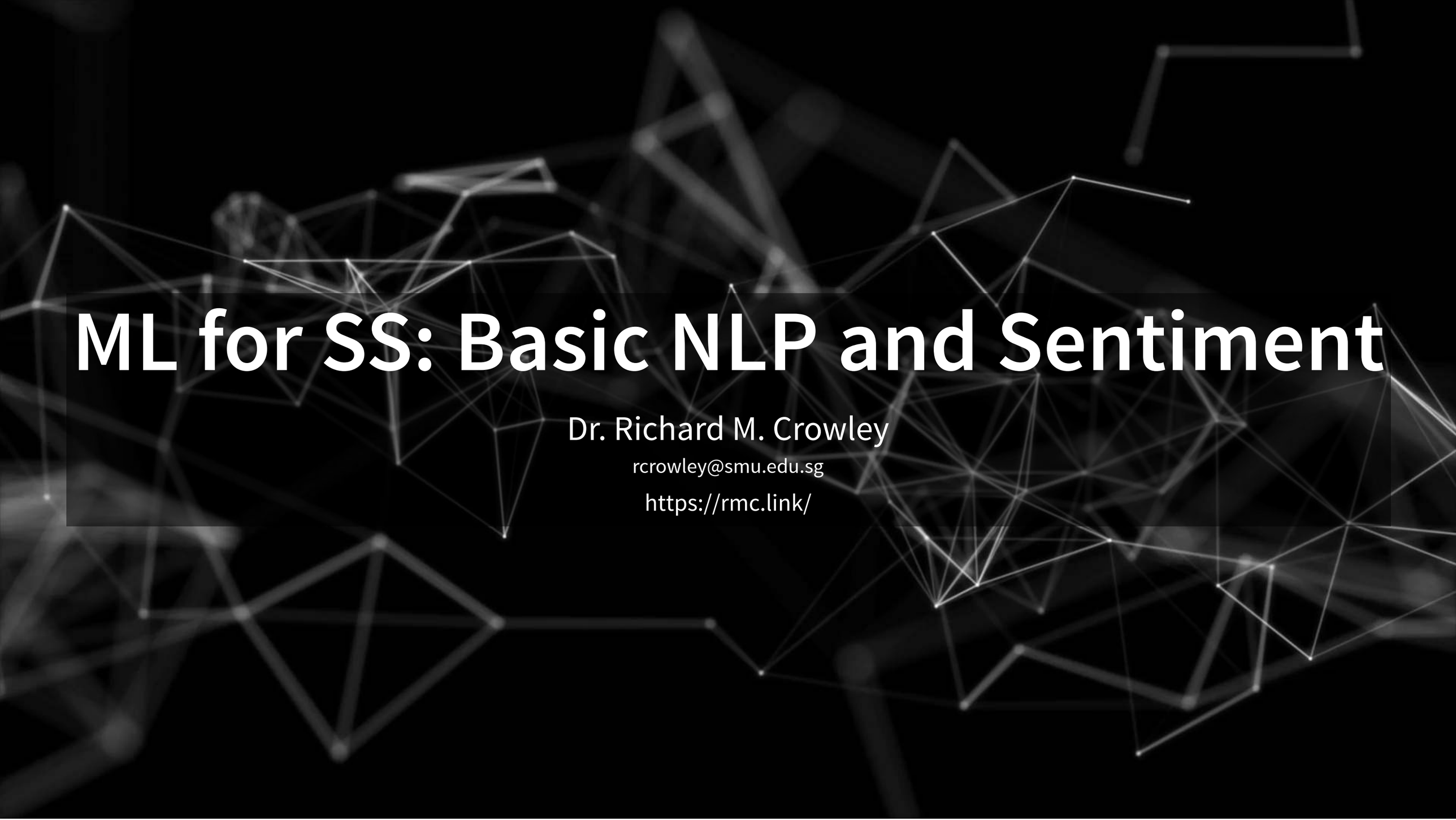

ML for SS: Basic NLP and Sentiment

Dr. Richard M. Crowley

rcrowley@smu.edu.sg

https://rmc.link/

Overview

Papers

> Paper 1: Loughran and McDonald (2011 JF)

- One of the most influential papers on sentiment in finance, accounting, and business
- A rigorous implementation of dictionary methods

> Paper 2: Antweiler and Frank (2005 JF)

- A seminal paper demonstrating traditional supervised sentiment classification

> Paper 3: Hassan, Hollander, Van Lent and Tahoun (2019 QJE)

- A supervised text classification approach using textbooks and news articles to supervise

Technical Discussion: Basic text analytics

Python

- Working with text
- Pattern matching (regex)
- Parsing text
- Dictionary sentiment
- Hassan et al. (2019 QJE) measure

R

- Use tidytext for most things
  - Introduces a dplyr approach to text
- Use stringr for pattern matching
- Use quanteda for simple measures
  - Sentiment
  - Readability

> Python is generally a bit stronger for these topics, unless your data is clean and fairly small.

> There is a fully worked out solution for using python, data and dictionaries are on eLearn.

Main application: Analyzing Wall Street Journal articles

- On eLearn you will find a full issue of the WSJ in text format

> We will not do any specific text-based analyses today, but we will do some simple measure construction

1. Regular expressions
2. Loughran and McDonald 2011 JF sentiment
3. Hassan et al. 2019 QJE supervised classification


Getting started with text

Special characters in python

- `\t` is tab
- `\r` is newline (files from Macs)
- `\r\n` is newline (files from Windows)
- `\n` is newline (files from *nix-based systems)
  - This is the usual convention used in data sets
- `\'` is an explicit single quote – it always works
  - E.g., `'\'Single\''` works, though so would `"'Single'"`
- `\"` is an explicit double quote – it always works
  - E.g., `"\"Double\""` works, though so would `'"Double"'`
- `\\` is a backslash
  - As `\` is used to denote special characters, a single backslash would ambiguous

> In some contexts, the following are special: `. ^ $ * + ? {} [] | ()`

Defining a string

- Use single quotes

```
print('This is a string')
```

```
This is a string
```

- Use double quotes

```
print("This is also a string")
```

```
This is also a string
```

Defining a string

- Multi-line strings: Triple quoting with either `'''` or `"""`

```
print("""This is a multi-line
string since it has triple quotes""")
```

```
This is a multi-line
string since it has triple quotes
```

- Multi-line strings: use a `\n` instead

```
print('This is also two lines\nsince it has a newline')
```

```
This is also two lines
since it has a newline
```

Common conventions

- The `"""` convention is mostly used for documentation
- The `\n` convention is used programmatically

Commonly used text functions

- Extract text using square brackets

```
# WSJ "About Us" description from: https://www.wsj.com/about-us
text = "The Wall Street Journal was founded in July 1889. Ever since, the Journal has led the way in chroni
print(text[0:100])
```

```
The Wall Street Journal was founded in July 1889. Ever since, the Journal has led the way in chronic
```

Commonly used text functions

- Convert anything to a string with `str( )`

```
x = 72
x_string = str(x)
x
```

```
72
```

```
x_string
```

```
'72'
```

- Combining text with `+`

```
'Hello' + ' ' + 'world'
```

```
'Hello world'
```

Commonly used text functions

- Casing text with `.lower()`, `.upper()`, and `.title()`

```
print('soon TO be UPPERCASE'.upper())
```

```
SOON TO BE UPPERCASE
```

```
print('SOON tO be lowercase'.lower())
```

```
soon to be lowercase
```

```
print('soon to be titlecase'.title())
```

```
Soon To Be Titlecase
```

Commonly used text functions

- Checking if text contains something particular

```
x = 'What is in this string?'

[x.startswith('What'), x.startswith('this')]
```

```
[True, False]
```

```
[x.endswith('string?'), x.endswith('string')]
```

```
[True, False]
```

```
['this' in x, 'ing' in x, 'zzz' in x]
```

```
[True, True, False]
```

`in` is an operator

In python, `in` is an operator much like `>` or `<`. It indicates if the LHS is contained in the RHS, working on strings or lists!

Commonly used text functions

- Finding *where* the content is
  - '.find()' returns `-1` if your query isn't found
  - '.index()' works the same as `.find()`, except it gives an error if your query isn't found

```
x = 'What is in this string?'
[x.find('this'), x.find('ing'), x.find('zzz')]
```

```
[11, 19, -1]
```

```
for y in ['this', 'ing', 'zzz']:
    try:
        print(x.index(y))
    except:
        print('Error!')
```

```
11
19
Error!
```

Commonly used text functions

- Counting the number of occurrences of a word or phrase
  - Can only check 1 phrase at a time
  - There are more efficient ways to check this for a list of words

```
print(text.count('Journal'))
```

```
3
```

Commonly used text functions

- Splitting strings

```
x = '1,2,3,4,5'.split(',')
print(x)
```

```
['1', '2', '3', '4', '5']
```

- Joining strings together

```
print(' & '.join(x))
```

```
1 & 2 & 3 & 4 & 5
```

> Joining strings is very useful when working with a list of data

Commonly used text functions

- Replacing string content

```
x = 'I like mee goreng with mutton and mee goreng with chicken'
print(x.replace('mee', 'nasi'))
```

```
I like nasi goreng with mutton and nasi goreng with chicken
```

```
print(x.replace('mee', 'nasi', 1))
```

```
I like nasi goreng with mutton and mee goreng with chicken
```

> `.replace()` has two required arguments (what to replace, replacement), and an optional argument (how many times to replace, default: infinite)

Commonly used text functions

- Removing blank content
  - Nice functions for keeping text clean

```
x = '   this is awkwardly padded        '
print([x.strip(), x.lstrip(), x.rstrip()])
```

```
['this is awkwardly padded', 'this is awkwardly padded        ', '   this is awkwardly padded']
```

- Padding strings
  - This is particularly useful when working with databases that zero-pad keys

```
gvkey = 1024
gvkey = str(gvkey).zfill(6)
print(gvkey)
```

```
001024
```

Commonly used text functions

12. Checking if strings are a certain type

```
output = '\t'.join(['input', 'alnum', 'alpha', 'decimal', 'digit', 'numeric', 'ascii'])
for x in ['ABC123', 'AAABBB', '12345', '12345²', '12345½', '123.1', '£12.0']:
  output += '\n' + '\t'.join(map(str,[x, x.isalnum(), x.isalpha(), x.isdecimal(),
                                      x.isdigit(), x.isnumeric(), x.isascii()]))
print(output)
```

```
input   alnum   alpha   decimal digit   numeric ascii
ABC123  True    False   False   False   False   True
AAABBB  True    True    False   False   False   True
12345   True    False   True    True    True    True
12345²  True    False   False   True    True    False
12345½  True    False   False   False   True    False
123.1   False   False   False   False   False   True
£12.0   False   False   False   False   False   False
```

- If you want a match on an explicit set of characters, using a regular expression is likely more intuitive

Importing a single text file

```
with open('../../Data/S4_WSJ_2013.09.09.txt', 'rt') as f:
    text = f.read()
```

- Guarantees the file gets closed properly
- A bit more readable than other approaches
- This is the preferred approach when possible

Cleaning text

- The text we have imported is not so clean
- We can use a for loop and conditional statements to clean the files
  - Code is in the jupyter notebook
- Also helpful is to import the file line-by-line rather than as 1 string

```
with open('../../Data/S4_WSJ_2013.09.09.txt', 'rt') as f:
    text2 = f.readlines()
```

Addendum: Using R

- The `read_file()` function from tidyverse's readr package works well.

```r
library(tidyverse)
# Read text from a .txt file using read_file()
doc <- read_file("../../Data/S4_WSJ_2013.09.09.txt")
# str_wrap is from stringr from tidyverse
cat(str_wrap(substring(doc,1,500), 80))
```

```
Document 1 of 119 Business and Finance Author: Anonymous Abstract: A U.S.
appeals court will hear oral arguments today in a suit by Verizon challenging
FCC "net-neutrality" rules. --- Full text: Hedge funds are cutting their
standard fees of 2% of assets under management and 20% of profits amid pressure
from investors. --- A team of Ares and CPP Investment Board are in the final
stages of talks to buy luxury retailer Neiman Marcus for around $6 billion. ---
Federal regulators pla
```

- For string manipulation, I recommend using the stringr library
  - The functions have more readable syntax and are dplyr-friendly


Regular expressions

A motivating example

> Suppose you want to find all quotes in a document

- Quotes follow a pattern: a double quote, some text, and another double quote

```
x=re.findall('(?m)\".+?\"', articles[1])
print(x)
```

```
['"Anna Karenina"', '"All happy families are alike."', '"Why two more now, and in the same year? I have no
idea,"', '"The Decameron,"', '"The Iliad"', '"The Odyssey"', '"I thought I could do better,"', '"The
Decameron"', '"naked from the waist down."', '"naked from the waist up."', '"I would be happy to address this
question if you allow me to go over Wayne\'s edition and find some mistakes that he can address,"', '"Each new
translation profits from those that went before,"', '"I am sure that Wayne took a look at our version,
especially since we tried to take a nonarchaic, non-British approach to Boccaccio\'s great and very clear
vernacular Italian."', '"Anna Karenina,"', '"perfectionist,"', '"spats,"', '"an example of where she was off
the mark."', '"brogues,"', '"smart shoes with perforations."', '"light peasant moccasins."', '"moccasin"',
'"It\'s a loaded word, particularly in the U.S.,"', '"Most disagreements over words ignore the context, which
is all important,"', '"porshni,"', '"is obsolete in Russian,"', '"primitive peasant shoes made from raw
leather."', '"rather close to the first meaning of brogues in the Oxford English Dictionary: \'rough shoes of
untanned hide.\' "', '"will be a long time before I get those."', '"we want to have a 21st-century translation
with a critically up-to-date introduction and notes."', '"The Death of Ivan Ilyich & Confession."', '"He
consciously chose to spend the last year of his life translating this book,"', '"The Tale of Genji,"',
'"There\'s always room for another excellent translation,"']
```

Breaking down the example

- `(?m)` allows output to span multiple lines
- `\"` is a literal double quote
- `.` represents any text
- `+` is used to indicate that we want *at least* 1 of the pattern immediately preceding the `+`
  - Regular expressions are *greedy* by default, meaning they will choose the longest matching text
- `?` forces the preceding command to *not be greedy*, preferring the shortest match that works for the full pattern
- `\"` is another literal double quote

Calling regexes

- 3 most useful functions to call regexes
  1. `re.findall()`
     - Finds all occurrences of your pattern and provides them back in a list
     - If you just want the count, apply `len()` to the list
  2. `re.sub()`
     - Use this for complex substitutions that are too much for `.replace()`
  3. `re.split()`
     - Use this for complex splits that are too much for `.split()`

Useful components

- `.` matches anything
- `\w` matches all characters that could be in a word
  - Except `-` and including `_`
- `\S` matches any non-whitespace characters
- `\s` matches any whitespace characters
- `\b` matches the start or end of a word
  - It is the boundary between `\S` and `\s`
  - Useful for matching whole words
- `\B` matches anything except the end of a word
- `^` or `\A` match the beginning of a string
  - Note: in *multiline* mode, `^` becomes the beginning of a line
- `$` or `\Z` match the end of a string
  - Note: in *multiline* mode, `$` becomes the end of a line

Useful patterns

- `[ ]` matches anything inside of it, like an “or” for regex
- `[^ ]` matches anything *except* for what is inside it
- Quantity specification (they always try to get the most text possible)
  - `x?` looks for 0 or 1 of `x`
  - `x*` looks for 0 or more of `x`
  - `x+` looks for 1 or more of `x`
  - `x{n}` looks for `n` (a number) of `x`
  - `x{n, }` looks for at least `n` of `x`
  - `x{n,m}` looks for at least `n` and at most `m` of `x`
  - To make any of the above non-greedy, append a `?` to them
    - E.g., `x+?` means 1 or more of `x`, but the least needed to fulfill the pattern

Complex patterns: Groups

- `( )` can be used to make groups
  - You can call for explicit matches of groups using a slash number:
    - `([0-9]).+\\1` Will match a number, followed by anything up until it hits that number again
  - By default, groups are capturing, meaning that the regex will only return the group text
  - There are two solutions:
    1. Put a group around the whole regex
    2. If you don't need to reference the group, use a non-capturing group with `(?: )`

```
re.findall('([0-9]).+\\1', '12asda2asd')
['2']
re.findall('(([0-9]).+\\2)', '12asda2asd')
[('2asda2', '2')]
re.findall('(?:12|sd)a', '12asda2asd')
['12a', 'sda']
```

Complex patterns: Looking assertions

- Sometimes you want text that was preceded or followed by something, but don't want that something in the output
- `(?=...)` provides a lookahead where the `...` must be next in the string, but won't output
- `(?!...)` provides a negative lookahead; if the `...` is next in the string, the match won't count
- `(?<=...)` provides a lookbehind, while `(?<!...)` provides a negative lookbehind

```
re.findall('(?<=\.)[0-9]+', '1 2.3 4. 5 6.78')
```

```
['3', '78']
```

Pros and cons of regexes

Positives

- Very flexible, can match almost any pattern
  - E.g., finding the MD&A of a 10-K
- Allows us to find text directly rather than just indices
- Built in to python already

Negatives

- Regexes can be quite slow to run
- Complex regexes are hard to read

Extra info

- Regexes can run in other modes rather than just the default
  - These can be passed using the `re` `flags` parameter, or by using shorthand in your regex itself
- Ignore case with `re.IGNORECASE` or `(?i)`
- Convert UTF to ASCII for matching with `re.ASCII` or `(?a)`
- Run regexes across multiple lines using `re.MULTILINE` or `(?m)`
- Make `.` match newlines using `re.DOTALL` or `(?s)`
- Write better documented regular expressions using `re.VERBOSE` or `(?x)`

> Full documentation here

Addendum: Using R

- The same stringr library from earlier handles these well as well
- Note that while the overall pattern structure is the same in R…
  - The special characters are often different
- There's a nice cheat sheet here
  - More detailed documentation here


Sentiment using dictionaries

Dictionaries

- Dictionaries in computing are just lists of words
- Usually these are words that are used to match to some concept
  - E.g., the Loughran and McDonald 2011 dictionaries are for Annual report text

```
with open('../../Data/S4_LM_Neg.csv', 'rt') as f:
    LM_neg = [x.strip().lower() for x in f.readlines()]
print(LM_neg[0:5])
```

```
['abandon', 'abandoned', 'abandoning', 'abandonment', 'abandonments']
```

```
with open('../../Data/S4_LM_Pos.csv', 'rt') as f:
    LM_pos = [x.strip().lower() for x in f.readlines()]
print(LM_pos[0:5])
```

```
['able', 'abundance', 'abundant', 'acclaimed', 'accomplish']
```

Applying a dictionary

- To apply a dictionary, we need to *tokenize* our text
  - Tokenize: Split into discrete chunks; words in this case
- 3 approaches:
  1. Use regular expressions to extract words
  2. Use NLTK's tokenizer
  3. Use SpaCy's NLP pipeline, which includes tokenizing in it

> For this example, the code uses NLTK

```
article_tokens = [nltk.tokenize.word_tokenize(article) for article in articles]
```

Stopwords

- Stopwords: words we remove because they have little content
  - the, a, an, and, …
- NLTK has word lists for many languages
- For our uses, we need to add some extra entries (punctuation) and remove negation terms

```
# If you get an error that you are missing 'stopwords', run: nltk.download('stopwords')
stop_words = set(nltk.corpus.stopwords.words("english"))
stop_words.remove('no')
stop_words.remove('not')
punct = {'.', ',', ';', '"', '\'', '-', '--', '---', '``', '\'\'', '%', '\'s'}
stop_words = stop_words | punct
```

Counting text – BoW

```
filtered_tokens = []
for tokens in article_tokens:
    filtered_tokens.append([t.lower() for t in tokens if t.lower() not in stop_words])

filtered_counts = [Counter(tokens) for tokens in filtered_tokens]

neg = []
for counts in filtered_counts:
    temp = 0
    for w in LM_neg:
        temp += counts[w]
    neg.append(temp)
```

> The `Counter()` is from `collections` (built in to Python)

Counting text – non-BoW

- To handle negation, we need to iterate through every word, or we would need to switch to n-grams for tokens

```
pos = []
for tokens in filtered_tokens:
    prior_token = ''
    temp = 0
    for token in tokens:
        if token in LM_pos and prior_token != ['no', 'not']:
            temp += 1
        prior_token = token
    pos.append(temp)
```

- We will also need to count the total number of words per article for our sentiment measure

```
words = [sum(counts.values()) for counts in filtered_counts]
```

Calculating sentiment

$$Sentiment = \frac{\#Positive - \#Negative}{\#Words}$$

```
df = pd.DataFrame(zip(words, pos, neg), columns=['words', 'pos', 'neg'])
df['sentiment'] = (df.pos - df.neg) / df.words
df
```

	Unnamed: 0	words	pos	neg	sentiment
0	0	132	2	3	-0.007576
1	1	608	8	14	-0.009868
2	2	1841	21	137	-0.063009
3	3	736	10	15	-0.006793
4	4	141	2	5	-0.021277
..	...	...	...	...	...
113	113	239	6	3	0.012552
114	114	493	9	6	0.006085
115	115	414	5	16	-0.026570
116	116	366	3	3	0.000000
117	117	571	9	11	-0.003503

[118 rows x 5 columns]


Supervised text classification

The Hassan et al. (2019 QJE) approach

- Just like how we can used data about a phenomenon to supervise algorithm construction with numeric data (i.e., regression), Hassan et al. (2019 QJE) suggests a similar idea based on using text to supervise text.
- The methodology requires 3 sets of textual information:
  1. Data that you want to analyze
  2. Data that represents the information you want to quantify the extent of
  3. Data that represents the rest of the information, e.g., what you **don't** want to quantify

> There is a simple requirement here: what you want and what the baseline text in your file is must be sufficiently different

- The method is mentioned in the computer science literature in Song and Wu (2008) and Schütze et al. (2008)

The study

> Goal: measure political risk

- Data:
  1. Conference call transcripts from 2002 to 2016
  2. Political text: American Politics Today (Bianco and Canon); articles from NYT, USA Today, WSJ, Washington Post on “domestic politics”
  3. Nonpolitical text: Financial Accounting (Libby, Libby and Short); articles from NYT, USA Today, WSJ, Washington Post on “performance,” “ownership changes,” and “corporate actions;” the Santa Barbara Corpus of Spoken American English (excluding politics-related episodes)

> A lot of baseline data is needed! But why?

Other work needed

1. Cleaning up the data
    - Removing a lot of bigrams based on part-of-speech tags that are unlikely to be relevant
    - Removing bigrams with: i, ve, youve, weve, im, youre, were, id, youd, wed, thats
    - Removing “princeton university”
2. Removing 3 synonyms for risk due to contextual differences: questions, question, venture

What do they do with the data?

- They construct a list of *bigrams* (2 word phrases) such that
  - Each bigram appears in the political baseline
  - Each bigram *never* appears in the nonpolitical baseline
- They will weight words accordingly
- They will measure risk by using these weights paired with phrases where a synonym for risk is nearby.

> Benefits of the method

1. More complete than a dictionary approach
2. Very clean approach given that political discussion should be fairly different from other discussion in annual reports
3. Generally applicable for any easy to pick out discussion
   - So long as you can find training data

What do we need to know to implement it?

1. How to chunk text into bigrams
2. How to tokenize text
   - Use NLTK ✓
   - Or use Spacy (we'll see this next session)
     - Spacy is more accurate, so the code will use this
3. How to count words or phrases
   - Use a `Counter()` ✓

> Optional advanced stuff: You can vectorize most of the calculation and just use matrix algebra with `numpy`

Workflow

- Set up blacklists

```
word_blacklist = "i i've you've we've i'm you're we're i'd you'd we'd that's".split(' ')
pattern_blacklist = ["PRP|PRP", "IN|IN", "RB|RB", "WRB|RB", "IN|RB", "RB|IN",
                     "IN|WRB", "WRB|IN", "DT|IN", "IN|DT", "RB|WRB", "RB|DT",
                     "DT|RB", "WRB|DT", "DT|WRB", "SYM|SYM"]
gram_blacklist = 'princeton|university'
```

Workflow

- Define the main function for cleaning

```
def grammer(doc, n, processed_patterns, word_blacklist, gram_blacklist, lower=True, stopword=True):
    if not stopword:
        grams = textacy.extract.ngrams(doc, n=n, filter_stops=False, filter_nums=True)
    else:
        grams = textacy.extract.ngrams(doc, n=n, filter_stops=True, filter_nums=True)
    ngrams = Counter()
    for gram in grams:
        pos = '|'.join([word.tag_ for word in gram])
        if not lower:
            text = '|'.join([word.text for word in gram])
        else:
            text = '|'.join([word.text for word in gram]).lower()
        if pos not in processed_patterns:
            if not np.any([word.text in word_blacklist for word in gram]):
                if text not in gram_blacklist:
                    ngrams[text] += 1
    return ngrams
```

Process a document

- We'll use the data from earlier

```
# install the spacy language model with `python -m spacy download en_core_web_sm`
nlp = spacy.load('en_core_web_sm', disable=['parser', 'ner'])
nlp.max_length = 10000000

documents = list(nlp.pipe(articles))

grams = [grammer(document, n=2, processed_patterns=pattern_blacklist,
                 word_blacklist=word_blacklist,
                 gram_blacklist=gram_blacklist) for document in documents]

# Intermediary measures
gram_counts = [sum(gram.values()) for gram in grams]
gram_sets = [set(gram) for gram in grams]
```

What is this `set()`?

- Sets in python are an interesting and rather useful structure
- Like lists, they contain a bunch of objects, such as text in our case
- Unlike lists, they do not have an order and cannot contain duplicates
- Also unlike lists, they are very fast to query
  - E.g., if you ask if something is in a very large set, the response is quick
- We can apply set functions to them!
  - `set1 & set2` represents the intersection of the two sets
    - Much faster than `[i for i in list1 if i in list2]`
  - `set1 | set2` represents the union

Applying a hypothetical dictionary

- The hypothetical weighted dictionary:

```
weights = {'earnings|foreign':0.5, 'currency|foreign':0.4, 'foreign|currencies':0.35, 'foreign|subsidiary':
           'foreign|currency':0.25, 'foreign|investment':0.2, 'foreign|holdings':0.2, 'foreign|borrowing':0
           'overseas|sales':0.1, 'foreign|investors':0.05}
weight_set = set(weights)
```

- Use set intersection to quickly get the overlap
- Determine the aggregate weight of the overlapping text

```
foreign_weight = []
for i in range(0, len(grams)):
    shared_keys = list(gram_sets[i] & weight_set)
    ns = len(shared_keys)
    v_weights = np.empty(ns)
    v_counts = np.empty(ns)
    c = 0
    for key in shared_keys:
        v_weights[c] = weights[key]
        v_counts[c] = grams[i][key]
        c += 1
    spec_weight = np.dot(v_weights, v_counts)
    measure = spec_weight / gram_counts[i] if gram_counts[i] > 0 else 0
    foreign_weight.append(measure)
```

Finalize the measure

```
df['foreign'] = foreign_weight
df.sort_values(by='foreign', ascending=False)
```

```
     Unnamed: 0  words  pos  neg  sentiment   foreign
9             9    507    4   25  -0.041420  0.002941
17           17    158    9    4   0.031646  0.001695
93           93    202    6    2   0.019802  0.000617
89           89    656   10   21  -0.016768  0.000209
87           87    223   11   14  -0.013453  0.000000
..          ...    ...  ...  ...        ...       ...
36           36    131    2    3  -0.007634  0.000000
35           35    179    1   21  -0.111732  0.000000
34           34     57    2    1   0.017544  0.000000
33           33     68    1    3  -0.029412  0.000000
117         117    571    9   11  -0.003503  0.000000

[118 rows x 6 columns]
```

- Note that this exercise shows you how to calculate a simpler score, not the risk score from Hassan et al. (2019 QJE)
  - For the risk score, you need to replace the counts by a count of times the bigram was within 10 words of a risk word
  - You also need to filter bigrams against a baseline document or documents

Conclusion

Wrap-up

> Text is pretty easy to work with in python

- Can approach it as a big string
- Can approach it line-by-line
- Can work with it word-by-word or gram-by-gram
- It’s all very flexible

> Many measures are pretty easy to calculate

> Some upcoming things

1. A survey to see what your thoughts are on the delivery method of the course
   - Helps with adjusting the course over the next sessions
2. Assignment 1

Packages used for these slides

Python

- `nltk`
- `numpy`
- `pandas`
- `spacy`
- `textacy`

R

- downlit
- kableExtra
- knitr
- quarto
- reticulate
- revealjs

References

- Antweiler, Werner, and Murray Z. Frank. “Is all that talk just noise? The information content of internet stock message boards.” The Journal of finance 59, no. 3 (2004): 1259-1294.
- Hassan, Tarek A., Stephan Hollander, Laurence Van Lent, and Ahmed Tahoun. “Firm-level political risk: Measurement and effects.” The Quarterly Journal of Economics 134, no. 4 (2019): 2135-2202.
- Loughran, T. and McDonald, B., 2011. When is a liability not a liability? Textual analysis, dictionaries, and 10-Ks.The Journal of Finance, 66(1), pp.35-65.
- Schütze, Hinrich, Christopher D. Manning, and Prabhakar Raghavan. Introduction to information retrieval. Vol. 39. Cambridge: Cambridge University Press, 2008.
- Song, Min, and Yi-Fang Brook Wu, eds. Handbook of research on text and web mining technologies. IGI global, 2008.