Introduction to Social Media Analytics (Lec 4)

Hao PENG
Department of Data Science
City University of Hong Kong
https://haoopeng.github.io/

Layers of text processing

1. Normalization (e.g. case conversion)
2. Tokenization / word-breaking ("what is a word?")
3. Spelling correction (if from interactive input)
4. Stopwords ("what is a useful word?")
5. Part-of-speech tagging (nouns, verbs, adjectives, adverbs, etc.)
6. Stemming ("what is an underlying word root?")
7. Named-entity recognition (“Lingfei WU studied at CityUHK.")

Text Mining:

1. Pattern detection (e.g., detect quotes in news articles)
2. Text classification (e.g., sentiment analysis, emotion, etc.)
3. Identify themes in text corpus (topic modeling; clustering)

Normalization

- you probably know about this step:
  - don't forget to do this!
  - `str.lower()`
- Remove social media specific elements
  - e.g., hashtags, mentions, URLs, emojis

Words and Tokens

The term *word* can be used in two different ways:

1. To refer to each occurrence of a word
2. To refer to a unique vocabulary item

For example, the sentence *"my dog likes his dog"* contains <u>five</u> occurrences of words, but <u>four</u> vocabulary words/items.

To avoid confusion, use more precise terminology:

1. ***Word token:*** a specific occurrence of a word
2. ***Word type:*** a vocabulary item

Tokenization

- The simplest way to represent a **text** is with a single string.
- Difficult to process large amounts of text in this format.
- Often, it is more convenient to work with a list of tokens.
- The task of converting a text from a single string to a list of tokens is known as *tokenization*.

What's the (dis)advantage of tokenization?

Consider this piece of text: "The big dwarf only jumps high and lands low, while the small dwarf jumps low and lands high."

How many word types? How many times does each word show up?

Tokenization

Sentence tokenization:

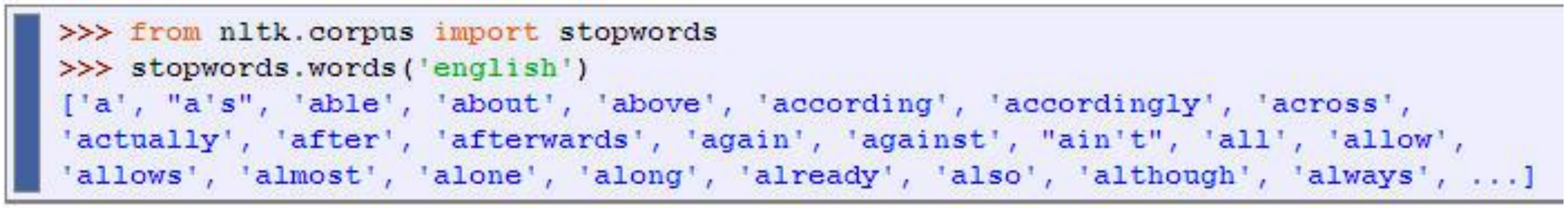
```
import nltk

text = "Hello, world. How are you, world?"

sent_text = nltk.sent_tokenize(text)

sent_text

Output: ['Hello, world.', 'How are you, world?']
```

Word tokenization:

```
sentence = "The quick brown fox jumped over the lazy dog!"

nltk.word_tokenize(sentence)


Output: ['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog', '!']
```

Lexical resources in NLTK

- NLTK Package includes some corpora that are nothing more than **wordlists** (https://www.nltk.org/howto/corpus.html)
- There is also a corpus of **stopwords**, that is, high-frequency words such as *the*, *to, also* that we often want to filter out before further processing a text/document.



```
>>> from nltk.corpus import stopwords
>>> stopwords.words('english')
['a', "a's", 'able', 'about', 'above', 'according', 'accordingly', 'across',
'actually', 'after', 'afterwards', 'again', 'against', "ain't", 'all', 'allow',
'allows', 'almost', 'alone', 'along', 'already', 'also', 'although', 'always', ...]
```

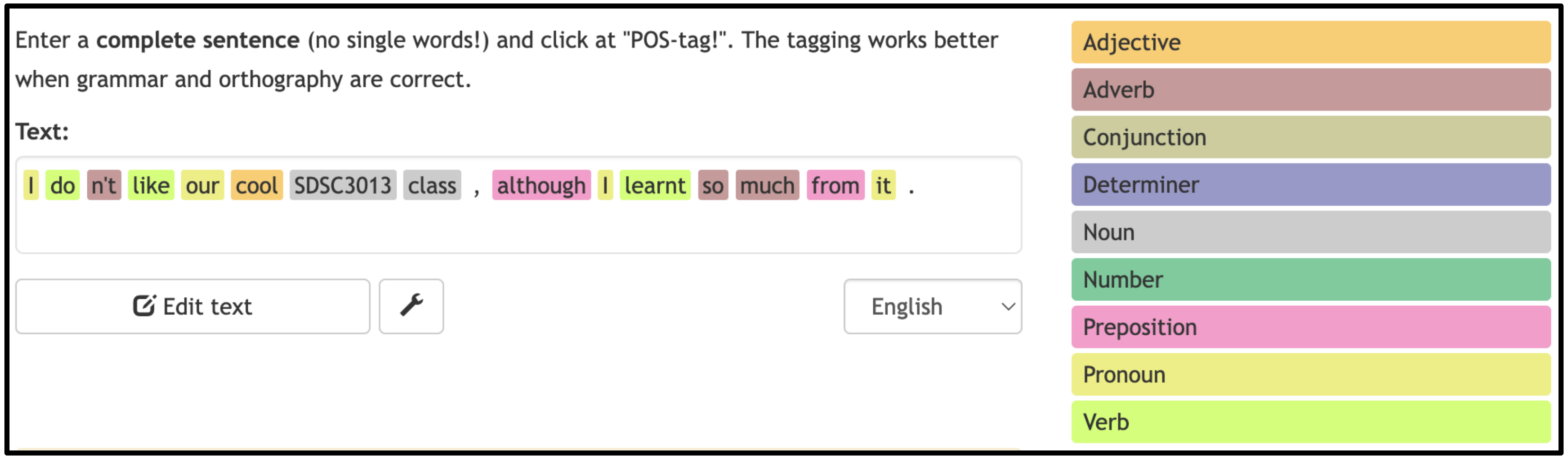
Remove stopwords:

```
text = "I don't want to go to SDSC3013 today."
clean = ' '.join([w for w in text.lower().split() if w not
in stopwords])
```

Part-of-speech tagging

- Part of speech (POS) tagging is the automated process of assigning a grammatical category, or "part of speech," to each word in a text, such as a noun, verb, or adjective
- There are various parts of speech categories, each with its own function in a sentence.

https://parts-of-speech.info/



Enter a **complete sentence** (no single words!) and click at "POS-tag!". The tagging works better when grammar and orthography are correct.

Text:

I do n't like our cool SDSC3013 class , although I learnt so much from it .

Part-of-speech tagging

POS tagging in NLTK:

```
>>> sentence = "The quick brown fox jumped over the lazy dog!"
>>> tokens = nltk.word_tokenize(sentence)
>>> tags = nltk.pos_tag(tokens)
>>> tags
[('The', 'DT'),
 ('quick', 'JJ'),
 ('brown', 'NN'),
 ('fox', 'NN'),
 ('jumped', 'VBD'),
 ('over', 'IN'),
 ('the', 'DT'),
 ('lazy', 'JJ'),
 ('dog', 'NN'),
 ('!', '.')]
```

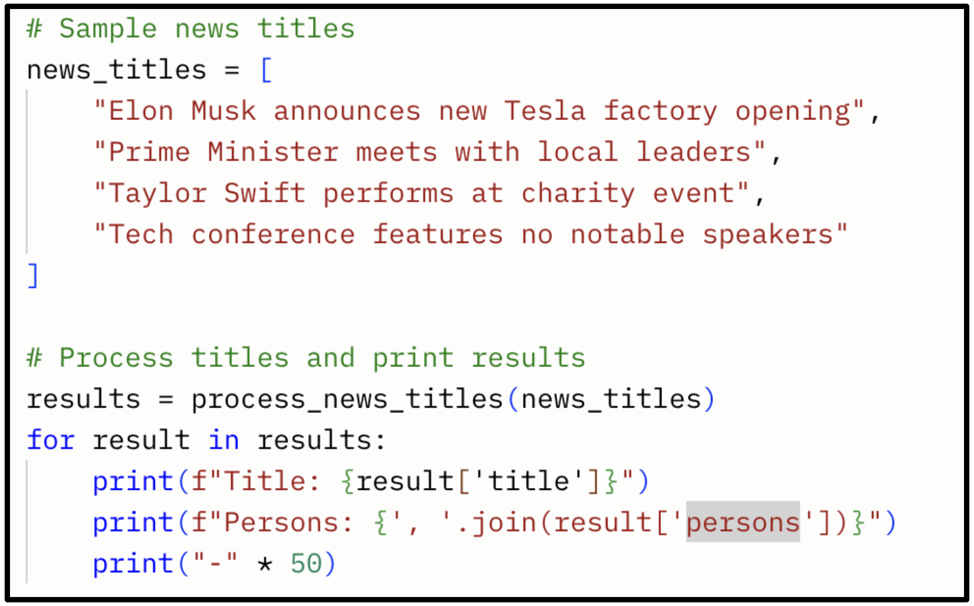
Named Entity Recognition: for detecting people, places, things …

NER in Python NLTK:

```
>>> sentence = "The quick brown fox spoke to Abraham Lincoln."
>>> tokens = nltk.word_tokenize(sentence)
>>> tagged_tokens = nltk.pos_tag(tokens)


>>> entities = nltk.chunk.ne_chunk(tagged_tokens)

>>> entities

Tree('S', [('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox',
'NNS'), ('spoke', 'VBD'), ('to', 'TO'),

Tree('PERSON', [('Abraham', 'NNP'), ('Lincoln', 'NNP')]), ('.', '.')])
```

Q: How to identify famous people based on their news ~~net~~ worth?

- Use NER to identify person entity in news titles.
- Demo using Grok to help us write Python code.
- Prompt: "Write python code to identify people in news titles."



```python
# Sample news titles
news_titles = [
    "Elon Musk announces new Tesla factory opening",
    "Prime Minister meets with local leaders",
    "Taylor Swift performs at charity event",
    "Tech conference features no notable speakers"
]

# Process titles and print results
results = process_news_titles(news_titles)
for result in results:
    print(f"Title: {result['title']}")
    print(f"Persons: {', '.join(result['persons'])}")
    print("-" * 50)
```

Stemming: merge diff. word forms

thinks → think

thinking → think
thinker → think

argue → argu
argument → argu

arguing → argu

argus → argu

Porter Stemmer: fast but inaccurate

```
>>> stemmer = PorterStemmer()
>>> plurals = ['caresses', 'flies', 'dies', 'mules', 'denied',
...            'died', 'agreed', 'owned', 'humbled', 'sized',
...            'meeting', 'stating', 'siezing', 'itemization',
...            'sensational', 'traditional', 'reference', 'colonizer',
...            'plotted']
>>> singles = [stemmer.stem(plural) for plural in plurals]
>>> print(' '.join(singles))
caress fli die mule deni die agre own humbl size meet
state siez item sensat tradit refer colon plot
```

WordNet Lemmatization: slower, more precise and conservative

```
>>> from nltk.stem.wordnet import WordNetLemmatizer
>>> lmtzr = WordNetLemmatizer()
>>> lmtzr.lemmatize('cars')
'car'
>>> lmtzr.lemmatize('feet')
'foot'
>>> lmtzr.lemmatize('people')
'people'
>>> lmtzr.lemmatize('fantasized','v')
'fantasize'
```

WordNet Lemmatization: slower, more precise and conservative

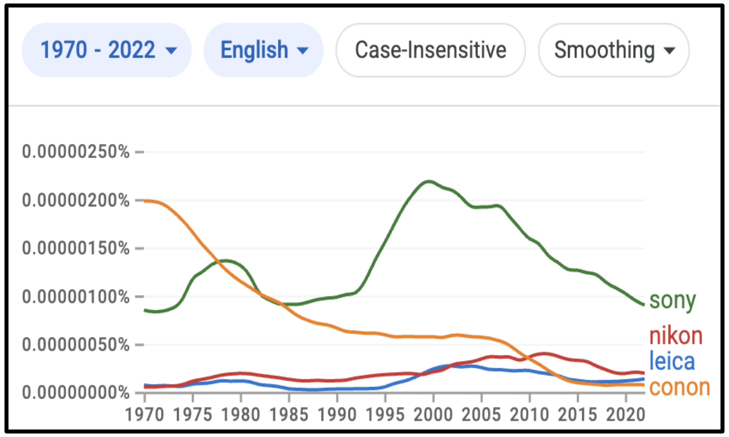
```
plurals = ['caresses', 'flies', 'dies', 'mules', 'denied',
...             'died', 'agreed', 'owned', 'humbled', 'sized’, 'meeting',
...             'stating', 'siezing', 'itemization', 'sensational',
...             'traditional', 'reference', 'colonizer', 'plotted']

>>> singles = [lmtzr.lemmatize(plural) for plural in plurals]
>>> print (' '.join(singles))
```

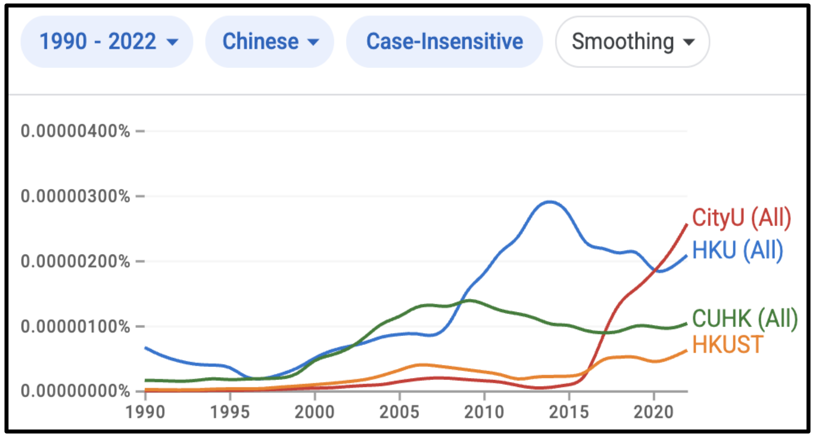
```
caress fly die mule denied died agreed owned humbled sized
meeting stating siezing itemization sensational traditional
reference colonizer plotted
```

Google books: N-gram viewer

Check it out: https://books.google.com/ngrams

- Popularity trends of camera brands / HK universities.
- How does this compare to mentions on social media?


1970 - 2022 English Case-Insensitive Smoothing


1990 - 2022 Chinese Case-Insensitive Smoothing

Regular Expressions (or 'regex')

Concise, flexible **pattern matching** expressions

Match substrings of text (words or complex patterns)

Appear in **many tools**

- grep (command line), text editors, ….

Therefore, useful to learn!

The Python re module

Search/match operations:

`re.search()`	finds **first match anywhere** in string	(`match` object)
`re.match()`	finds **first match at beginning** of string	(`match` object)
`re.findall()`	finds **all matches anywhere** in a string	(`list` of `strings`)
`re.finditer()`	finds **all matches anywhere** in a string	(`iterator` of `match` objects)

Regex-enabled string operations:

`re.split()`	splits strings at all occurrences of a pattern
`re.sub()`	replaces (substitutes) all occurrences of a pattern

re.search

`search(pattern, string, flags=0)`

Scan through **string** looking for the first match to **pattern**

Return:

- If something is found: a `Match` object
- If nothing is found: None

re.match

`match(pattern, string, flags=0)`

Check if **pattern** matches at the **beginning** of **string**

Return:

- If something is found: a `Match` object
- If nothing is found: None

Most methods return match objects

```
import re

lazy = 'The lazy dog went to sleep'

result = re.search('dog', lazy)

if result is not None:
    print('found', result.group() )
else:
    print('did not find')
```

```
found dog
```

match objects:

group()	Contents of the match
start()	Start position of match
end()	End position of match

Basic matching patterns

Ordinary characters just match themselves

```
result = re.search('dog', 'The lazy dog went to sleep’)
```

Many reserved **special characters**, including:

```
. ^ $ * + ? { }[ ] | ( ) \
```

And escape sequences that begin with “\”:

`\t`	tab
`\n`	newline
`\r`	return

https://developers.google.com/edu/python/regular-expressions

Symbols for matching single char

- `.` Any char (except newline = `\n`)

```
'F..m:'
'Farm:'  -> Yes
'Foom:'  -> Yes
'Firm!'  -> No
```

Symbols for matching single char

\s Whitespace

```
'Pine\sapple'
'Pine apple'   -> Yes
'Pine\tapple'  -> Yes
'Pineapple'    -> No
```

\S Non-whitespace

```
'Pine\Spple'
'Pineapple'   -> Yes
'Pine pple'   -> No
```

\d Digit, 0-9

```
'Apple \d\d\d'
'Apple 123'   -> Yes
'Apple ABC'   -> No
'Apple 12C'   -> No
```

\D Non-digit

```
'Apple \D\D\D'
'Apple ABC'   -> Yes
'Apple 123'   -> No
```

Symbols for matching single char

^	Beginning of a line

```
'^From:'
'From: Matt’         -> Yes
'It said, From: …' -> No
```

$	End of a line

```
'Michigan$'
'Michigan, USA'   -> No
'Michigan\nUSA'   -> Yes
```

Symbols for matching single char

`\w` "word" or "regular" character
(letter, digit, underscore: [a-zA-Z0-9_]

```
'cat\w'
'cats'   -> Yes
'cat3'   -> Yes
'cat!'   -> No
```

`\W` Non-"word" character

```
'cat\W'
'cats'   -> No
'cat!'   -> Yes
```

`\b` word boundary
(beginning or end of word)

```
'\bcat\b'
'cats'   -> No
'cat3'   -> No
'cat!'   -> Yes
```

`\B` Not beginning/end of word

```
'\Bcat\B'
'cats'       -> No
'meercats'   -> Yes
```

Why use \B instead of \w?

```
re.search(r'\wcat\w', 'meercats')
<_sre.SRE_Match object; span=(4, 7), match='rcats'>

re.search(r'\Bcat\B', 'meercats')
<_sre.SRE_Match object; span=(4, 7), match='cat'>
```

Escape special symbols with \

What if we really want to look for `'$'` or `'.'` or `'\'` ?

Use an **escape sequence**: **backslash** + **special char**

e.g. `'\$'`

Examples:

`'\$19\.99’`	will match `'$19.99'`
`'\\folder'`	will match `'\folder'`

Raw string notation: r'text'

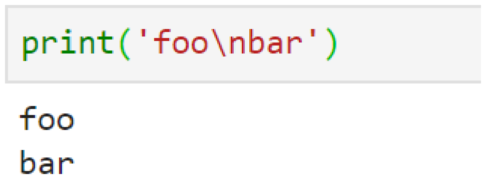
(Python uses “\” to escape special characters)

What if we really want to look for the backslash char?

Solution: prefix string with r

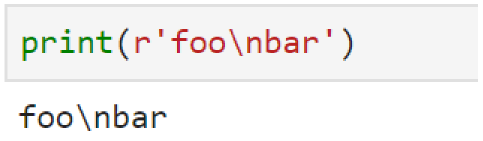
`'\n'` one-char string
containing just **newline** character

`r'\n'` two-char string
containing \ and **n** characters



```
print('foo\nbar')
```

```
foo
bar
```



```
print(r'foo\nbar')
```

```
foo\nbar
```

Repeating: match multiple chars

`*`	**zero or more** of the previous thing
`+`	**one or more** of the previous thing
`?`	**zero or one** of the previous thing
`{3}`	**exactly** 3 of the previous thing
`{3,6}`	**between** 3 **and** 6 of the previous thing
`{3,}`	3 **or more** of the previous thing

Repeating pattern examples

`ab*` will match

- `'a'` (must have)
- followed by zero or more `'b'`s

`ab+` will match

- `'a'` (must have)
- followed by one or more `'b'`s (**will not** match just **`'a'`**)

`ab?` will match

- `'a'` (must have)
- followed by zero or one `'b'`s

A set of characters: [ ]

- `[aeiou]` A **single character** in the set {a, e, i, o, u}
- `[^aeiou]`A **single character** **not** in set {a, e, i, o, u}

Example:
What substrings does `[aeiou]{2,}` match in this text?

```
The eerie wind said "Oooo" and "Rrr".
```

```
The eerie wind said "Oooo" and "Rrr".
```

Define a range with dash “-”

Valid:

`[A-Z]`	Upper Case Roman Alphabet
`[a-z]`	Lower Case Roman Alphabet
`[A-Za-z]`	Upper Case and Lower Case
`[A-F]`	Upper Case (but only A – F)
`[0-9]`	`\d`
`[a-zA-Z0-9_]`	`\w`

Invalid:

`[a-Z]`

`[F-A]`

`[9-0]`

Negation of ranges

`[^0-9]`	Anything BUT digits
`[^a]`	Anything BUT a lower case a
`[^A-Z]`	Anything BUT upper case letters
`[^,]`	Anything BUT ,

Greedy matching: The Default

Greedy: matching as much as possible in the given string.

Example:

```
s = 'the cat in the hat’
match = re.search(r'^(.*)(at)(.*)$’, s)
```

Note: () is used for matching multiple groups!

Now, what do we have in three groups:

```
match.group(1),     match.group(2),        match.group(3)?
'the cat in the h’  'at’                   ''
```

Non-greedy matching by adding an extra ? to the end of wildcard

Non-greedy versions try to match as minimally as possible.

`??`, `*?`, `+?`, and `{}?`

Example:

```
s = 'the cat in the hat’
match = re.search(r'^(.*?)(at)(.*)$’, s)
```

Now, what do we have in three groups:

```
match.group(1),       match.group(2),         match.group(3)?
'the cat in the h'   'at'                    ''
'the c'               'at'                    'in the hat'
```

Defining alternatives using pipe |

- Use alternatives in regular expressions by putting them in brackets:
  `(success|failure code: [0-9]+|maybe[!?]*)`

- Pipe is never greedy. As the target string is scanned:
  - patterns separated by `'|'` are tried from left to right.
  - When one pattern completely matches, that branch is accepted.
  - Once A matches, B will not be tested further
  - Even if it would produce a longer overall match.

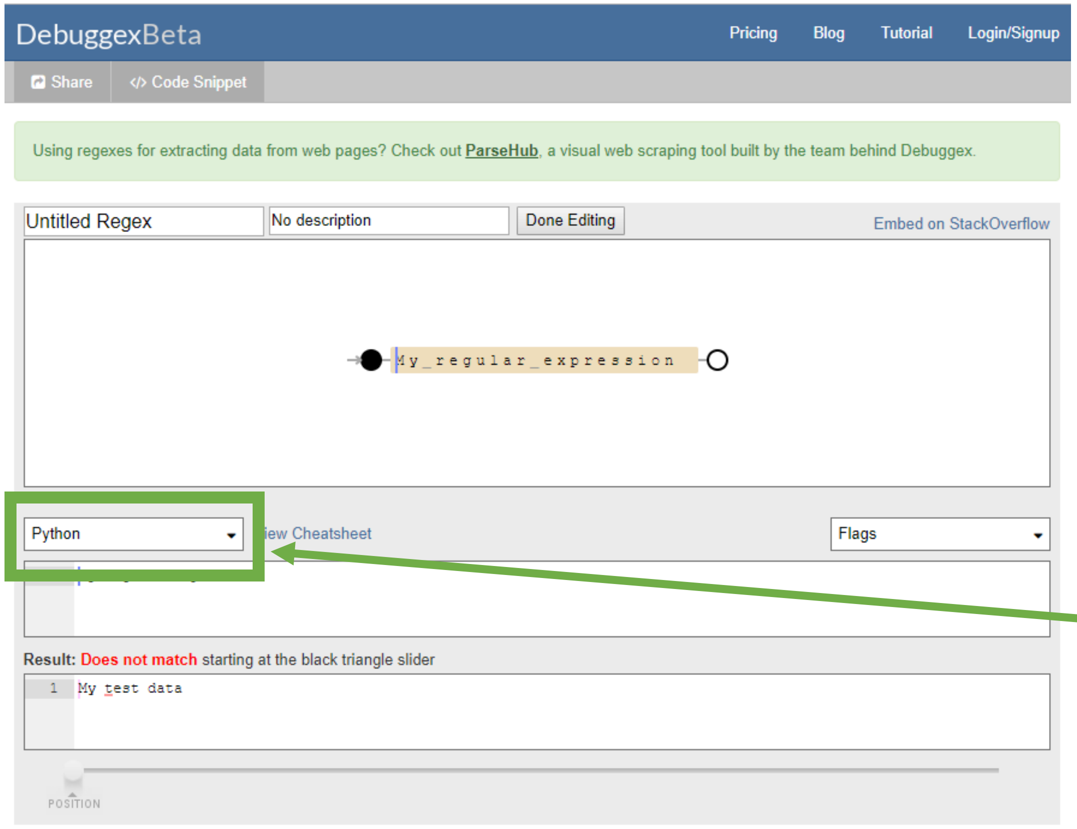
- What does this match? `^(T|t)oday`

Optional parameters

- The option flag can be added as an extra argument.
  - e.g., `re.search(pat, str, `**`re.IGNORECASE`**`)`
- `re.IGNORECASE:` Ignore upper/lowercase differences.
- `re.DOTALL:` Make the '.' special character match any character at all, including a newline (by default, the '.' matches anything *except* a newline).
- `re.MULTILINE:` Within a multi-line string, let ^ and $ match the start and end of each line. Normally ^/$ match the start and end of the whole string.

Useful online debugging tools

Visually test if code works in real-time: www.debuggex.com



Using regexes for extracting data from web pages? Check out ParseHub, a visual web scraping tool built by the team behind Debuggex.

Untitled Regex

No description

My_regular_expression

Python

Result: Does not match starting at the black triangle slider

```
My test data
```

Make sure to set this!

What you should know

- Read/write regular expressions for matching patterns
- How to use the Python `re` library functions to **search** for matches, **extract** them, **substitute** text for them
- How to specify and **extract groups** in a match
- **Greedy** vs. **non-greedy** matching

Case study: identify quotes from famous people in news articles

- Extract text in quotation marks that follow some patterns.
- Use a regular expression to capture phrases that include these quote-signaling words (18 out of top 50 verbs):
  - *"describe", "explain", "say", "tell", "note", "add", "acknowledge", "offer", "point", "caution", "advise", "emphasize", "see", "suggest", "comment", "continue", "confirm", "accord".*
- If any of the 18 verbs (or their verb tenses) appears within five tokens before or after a person's full or last name.
  - E.g., "We are getting close to the truth." according to the chief executive John Lee.
  - John Lee, the chief executive of HK says: "CityU is awesome."

Content moderation on Reddit

- Platforms employ regex for real-time content moderation.
    - Reddit's use of patterns like (asshole|slur|abuse) to flag potentially harmful language in comments or posts.
- This helps in filtering negative content by triggering reviews.
- Such tools can help to maintain community standards through toxicity analysis in high-volume online social environments.

Cross-platform user identification

- Use regex to identify links to other social media accounts.
  - Extract Instagram handles in tweets: instagram\.com/[a-zA-Z0-9_]+
  - Revealing user interests and user migration.
- Helpful for tracking user behaviors across platforms.
  - Taobao product recommendation based on post viewed on RedNote.
- This simple technique is also valuable for analyzing how users share content across sites, which is good for viral marketing.

Text classification

Goal: Assign 'objects' from a universe to two or more *classes* or *categories*

Examples:

Tasks	Object	Categories
Tagging	Word	Part of Speech
Sense disambiguation	Word	The word's senses
Information retrieval	Document	Relevant/not relevant
Named entity recognition	Document	People/location/…
Sentiment classification	Document	Positive/negative/…
News classification	Document	Sport/science/finance…
……		

Sentiment Analysis

- Attempt to identify affective state of text with NLP methods.
- How do we measure sentiment (score or category)?
- At what level? Word or sentence?

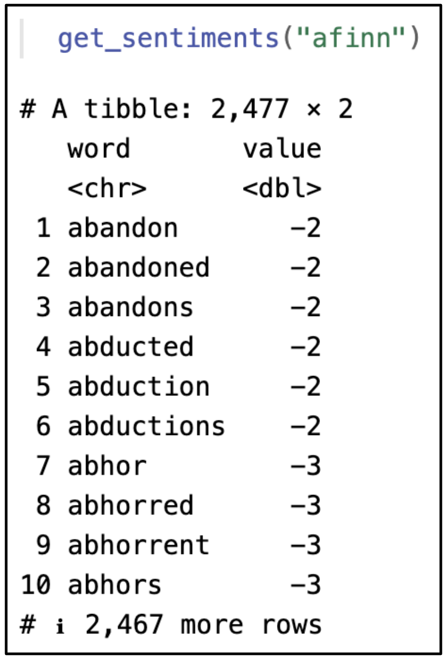
Sentiment Analysis

Dictionary-based approach

- Lexicon: each word is assigned a sentiment score.
- Sum sentiment score of all words in the given text.
- **Challenge**: which words are associated with positive/negative sentiment?

Weaknesses:

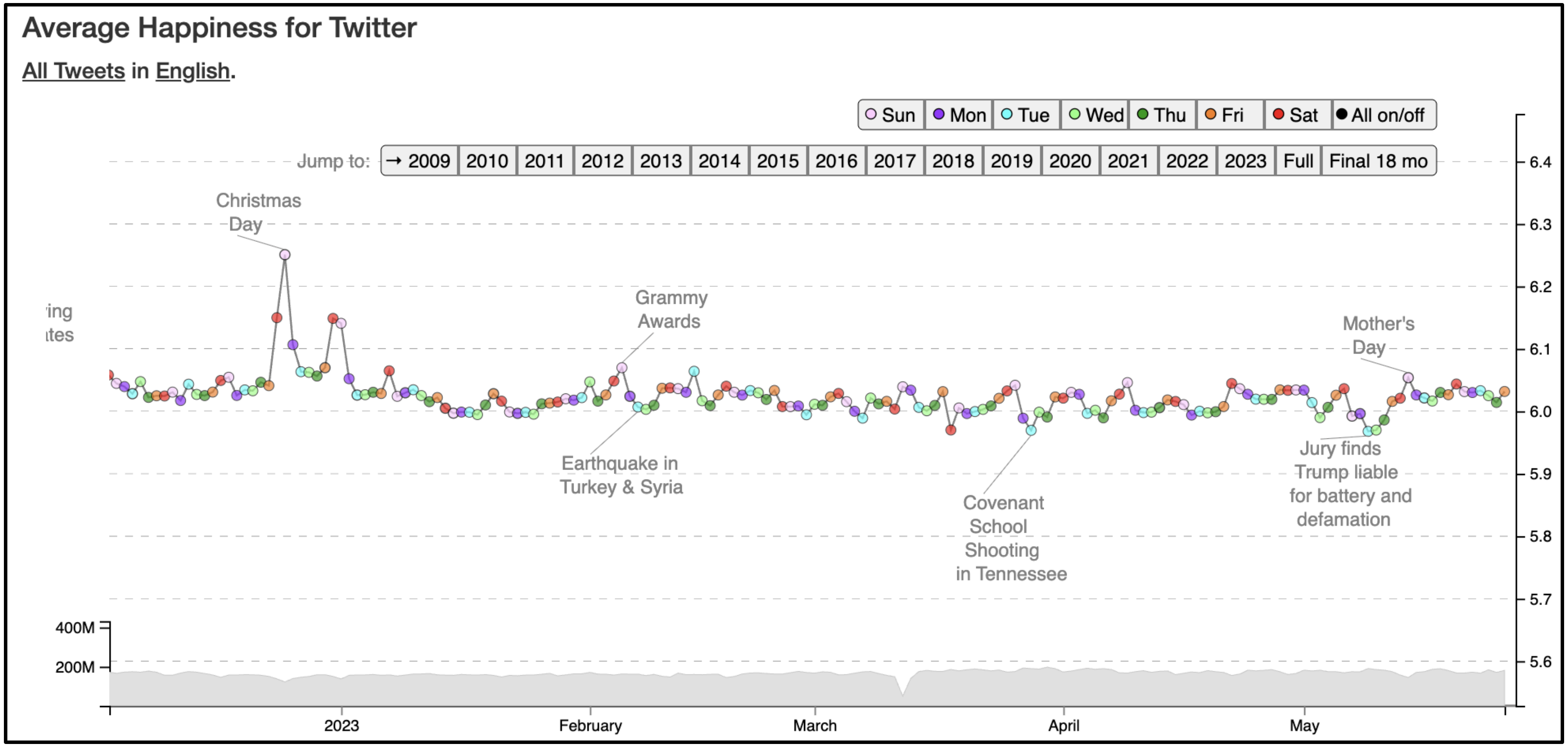
- Manual annotation
- Can be very subjective / imprecise
- Does not consider the word's context
- Ignores word phrases



```
get_sentiments("afinn")
```

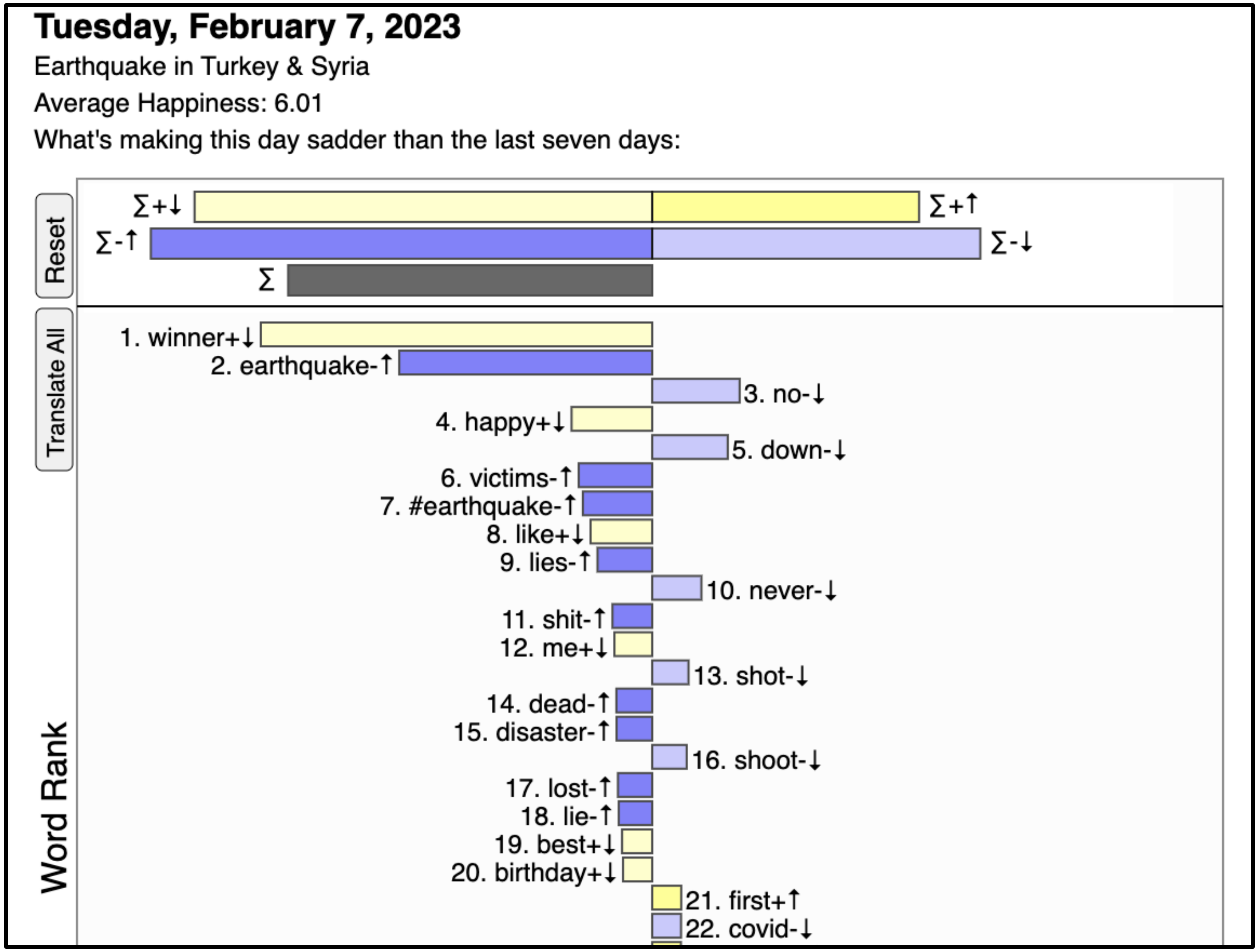
```
# A tibble: 2,477 × 2
   word        value
   <chr>       <dbl>
 1 abandon        -2
 2 abandoned      -2
 3 abandons       -2
 4 abducted       -2
 5 abduction      -2
 6 abductions     -2
 7 abhor          -3
 8 abhorred       -3
 9 abhorrent      -3
10 abhors         -3
# ℹ 2,467 more rows
```

Demo: Hedonometer of happiness



https://hedonometer.org/timeseries/en_all/?from=2022-12-03&to=2023-05-26&viz=wordshift

Measure happiness via word change


Tuesday, February 7, 2023
Earthquake in Turkey & Syria
Average Happiness: 6.01
What's making this day sadder than the last seven days:

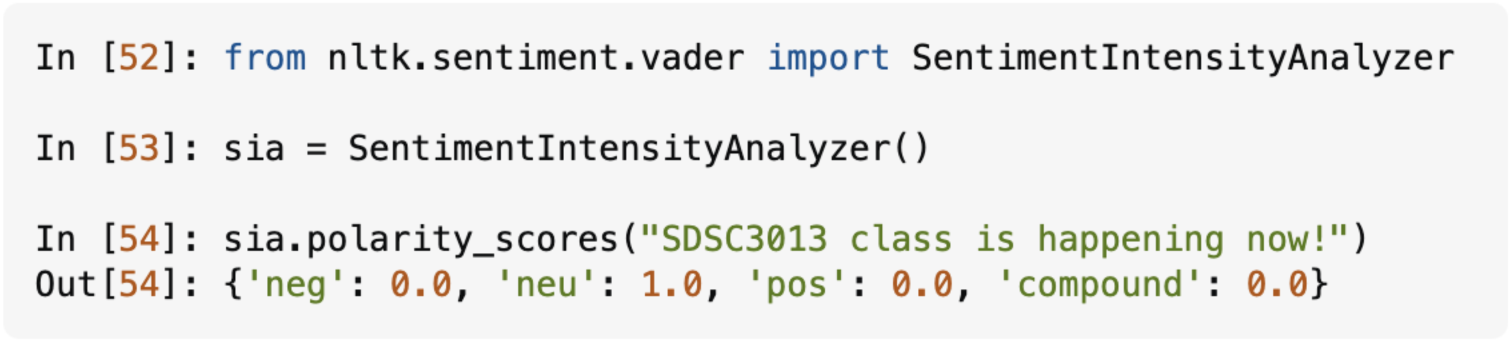
Sentiment Analysis

Dictionary-based approach with pre-defined rules

- E.g., VADER (Valence Aware Dictionary and sEntiment Reasoner)
  - https://github.com/cjhutto/vaderSentiment (Incorporated in NLTK!)
- Utilizes a pre-defined lexicon of words, each with a sentiment score (valence)
- Incorporates a set of rules to account for linguistic nuances:
  - **Punctuation**: Exclamation points can amplify sentiment.
  - **Capitalization**: All-caps can indicate increased intensity.
  - **Degree Modifiers**: Words like "very" or "slightly" modify sentiment intensity.
  - **Negation**: Words like "not" or "never" reverse words’ sentiment.
  - **Conjunctions**: "But" can shift the focus and sentiment within a sentence.
- Other tools:
  - sentimentr: https://github.com/trinker/sentiment

Sentiment Analysis

- Strengths of VADER:
  - Optimized for social media language (e.g., informal, emojis, slang)
  - No need of training data (labeled text is expensive/rare)
  - Provides a more nuanced analysis than simple word counting

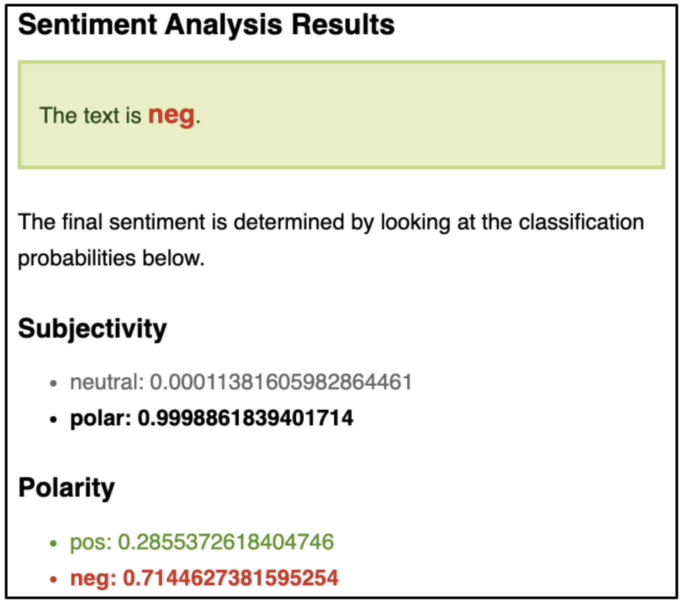


```
In [52]: from nltk.sentiment.vader import SentimentIntensityAnalyzer

In [53]: sia = SentimentIntensityAnalyzer()

In [54]: sia.polarity_scores("SDSC3013 class is happening now!")
Out[54]: {'neg': 0.0, 'neu': 1.0, 'pos': 0.0, 'compound': 0.0}
```

Limitations of lexicon-based models

- Difficult to evaluate mixed-sentiment sentences.
  - e.g. Your essay's introduction is good, but the conclusions are weak.
- Hard to deal with words with multiple meanings.

I am not going to miss you.



Sentiment Analysis Results

The text is **neg**.

The final sentiment is determined by looking at the classification probabilities below.

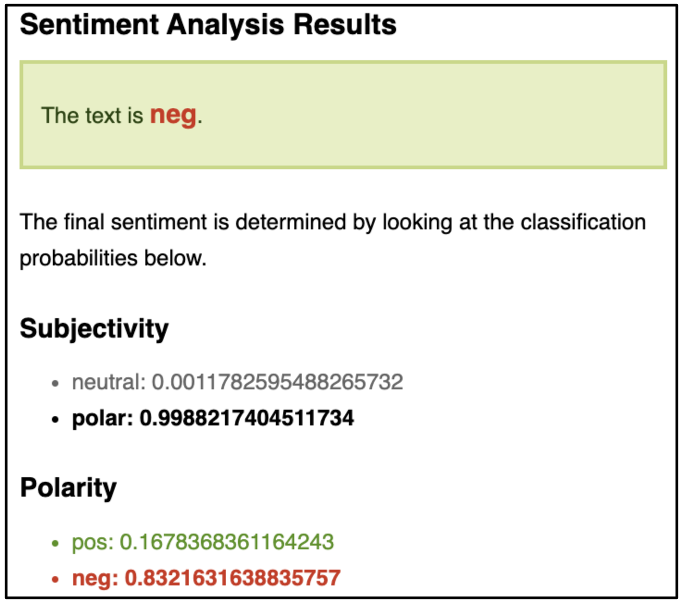
Subjectivity

- neutral: 0.00011381605982864461
- **polar: 0.9998861839401714**

Polarity

- pos: 0.2855372618404746
- **neg: 0.7144627381595254**

I am not going to miss using this product.



Sentiment Analysis Results

The text is **neg**.

The final sentiment is determined by looking at the classification probabilities below.

Subjectivity

- neutral: 0.0011782595488265732
- **polar: 0.9988217404511734**

Polarity

- pos: 0.1678368361164243
- **neg: 0.8321631638835757**

Sentiment prediction is domain-specific and context-dependent!

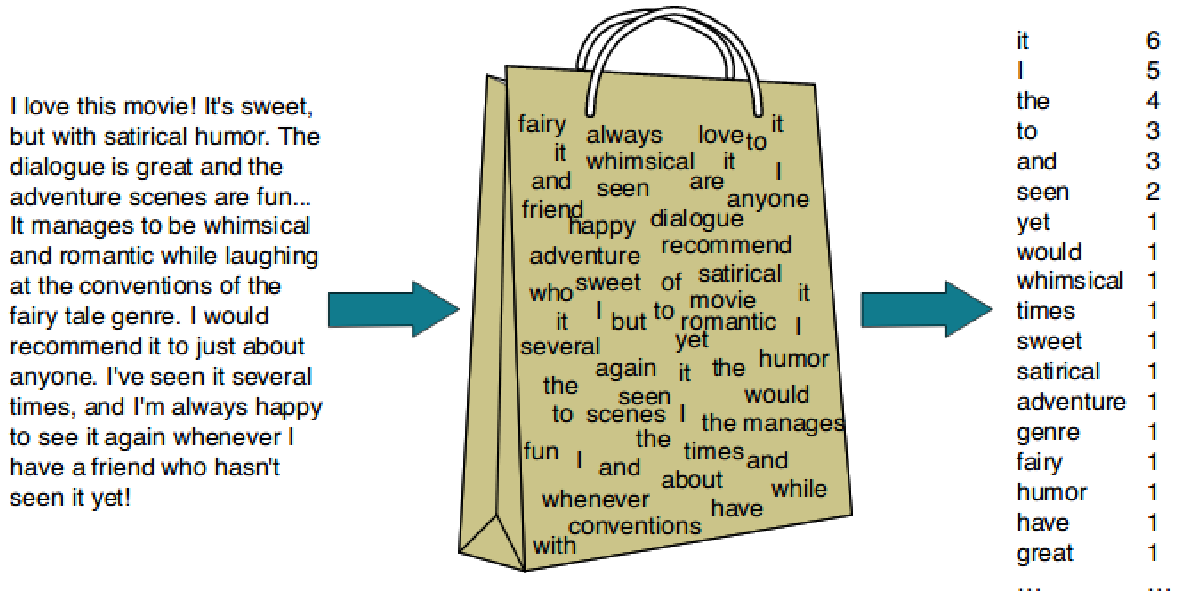
Sentiment Analysis

Machine learning based approach

- Training data is needed
  - texts (X) with human-annotated sentiment labels (y)
- Feature engineering
  - Bag of words, n-grams
  - Theme / Topic of words
  - Word embeddings
- Model evaluation
  - AUC-ROC
  - F1-score

Bag of words

- **Do text pre-processing first**
- Count word frequencies for each document
- Represent each document as a numerical vector

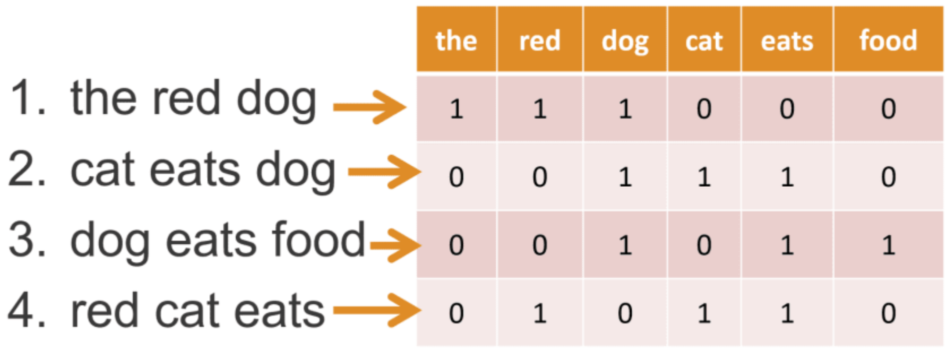


I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!

it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
...	...

https://koushik1102.medium.com/nlp-bag-of-words-and-tf-idf-explained-fd1f49dce7c4

Bag of words



	the	red	dog	cat	eats	food
1. the red dog	1	1	1	0	0	0
2. cat eats dog	0	0	1	1	1	0
3. dog eats food	0	0	1	0	1	1
4. red cat eats	0	1	0	1	1	0

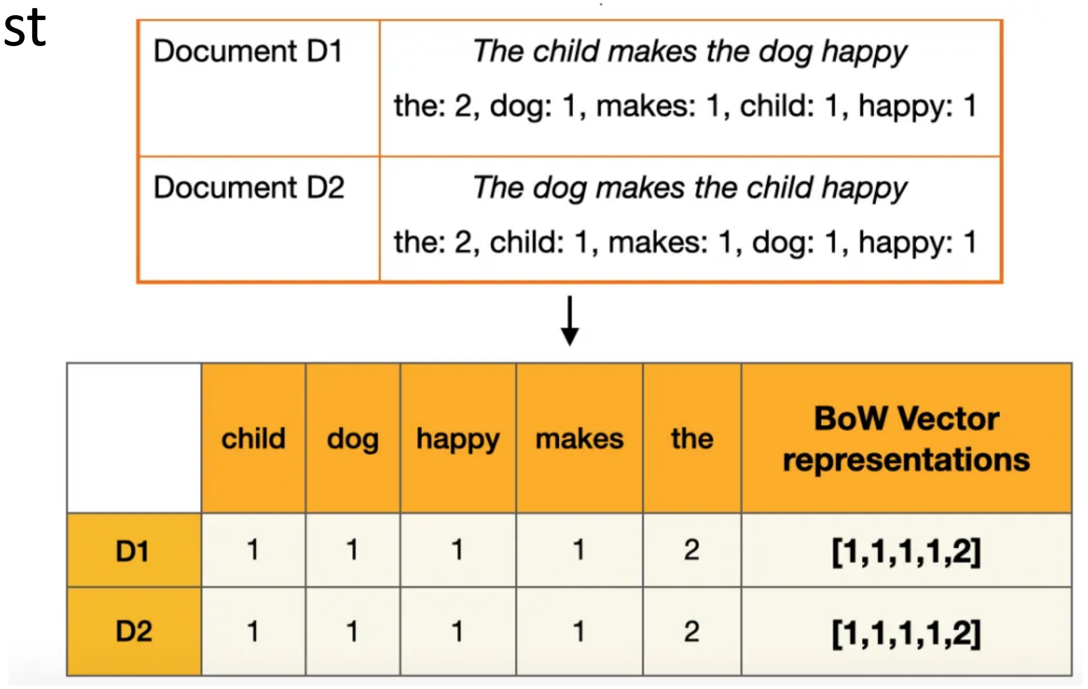
- Easy to understand, explain, and implement
- Scalable to a large collection of documents
- Generalize to many NLP tasks
  - Document similarity
  - Text classification
  - Text clustering

Disadvantages?

https://aiml.com/what-are-the-advantages-and-disadvantages-of-bag-of-words-model/

Bag of words weakness

- Cannot handle out-of-vocabulary words
- Sparse matrix is very inefficient
- All words are equally important
- Word order is lost

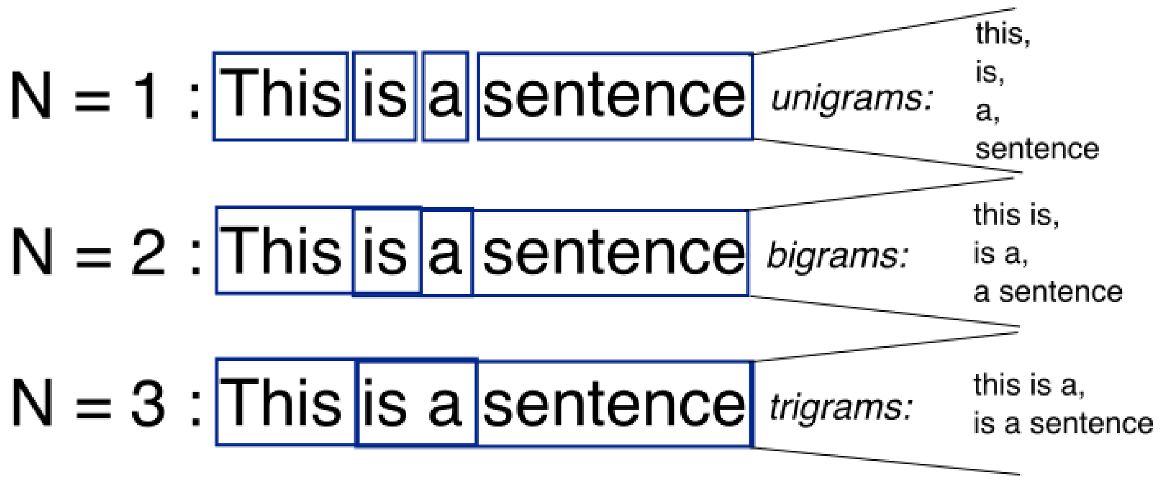


Document D1	*The child makes the dog happy* the: 2, dog: 1, makes: 1, child: 1, happy: 1
Document D2	*The dog makes the child happy* the: 2, child: 1, makes: 1, dog: 1, happy: 1

	child	dog	happy	makes	the	**BoW Vector representations**
D1	1	1	1	1	2	**[1,1,1,1,2]**
D2	1	1	1	1	2	**[1,1,1,1,2]**

N-grams

- Can capture word context (to some degree)
- Significant increase in the model's vocabulary size
- Can use highly frequent phrases to reduce complexity



https://www.kaggle.com/code/samuelcortinhas/nlp3-bag-of-words-and-similarity

Not all words are equal

SOCIAL
MEDIA
COMMUNICATION
NETWORKING
MARKETING
BUSINESS
INFORMATION
COMMERCIAL
WEB
CONNECTION
NETWORK
COMPUTER
TWITTER
APPLICATION
CONTENT
EDITORIAL
ONLINE
MULTIMEDIA
COMMUNITY
BLOGGING
RELATIONSHIP
SUCCESS
SHARE
COLLAGE
INSTAGRAM
BLOG
CONNECT
INTERNET
TECHNOLOGY
ABSTRACT
VIDEO
FACEBOOK
CHAT
ADVERTISING
WEBSITE
SERVICE
BRAND
HAND
FOLLOW
FRIENDS
RECOMMEND
PUBLISHING
POPULAR
WORK
LIKE
FORUM
COMPANY
SEO
APP

Word Cloud: Love-Hate Visual That Still Works (Sometimes)

TF-IDF

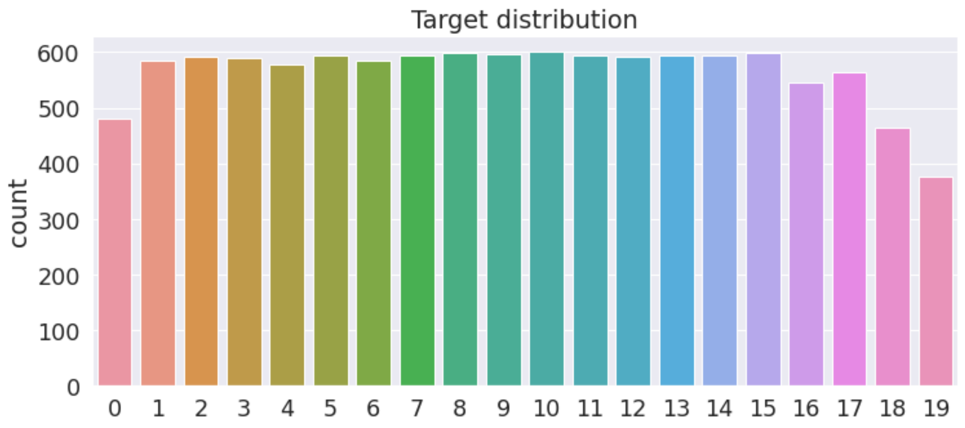
- Term Frequency: how many times a word appears in a text.
- Inverse Document Frequency: how rare a word is across corpus.
- TF-IDF: TF x IDF assigns a weight to each word, which reflects its importance within the text and its uniqueness across the corpus.
- Common steps: text preprocessing; represent texts as vectors; each word is a dimension, with the TF-IDF scores as its values.

$$\mathrm{tf}(t,d) = \log(1 + f_{t,d}) \qquad \mathrm{idf}(t,D) = \log\left(\frac{N}{n_t}\right)$$

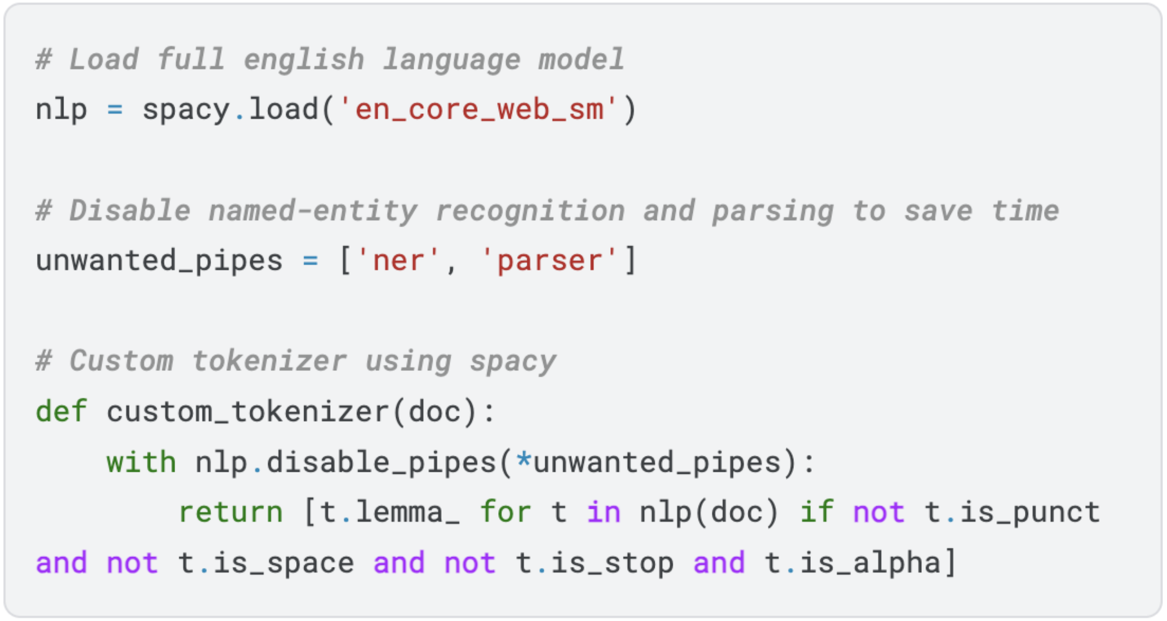
$$w_{t,d} = \mathrm{tf}(t,d) \times \mathrm{idf}(t,D)$$

News classification (TF-IDF)

```python
# nlp core
import spacy

# sklearn
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics import f1_score, classification_report
from sklearn.naive_bayes import MultinomialNB

# Fetch data
train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes’))
test = fetch_20newsgroups(subset='test', remove=('headers', 'footers', 'quotes'))
```



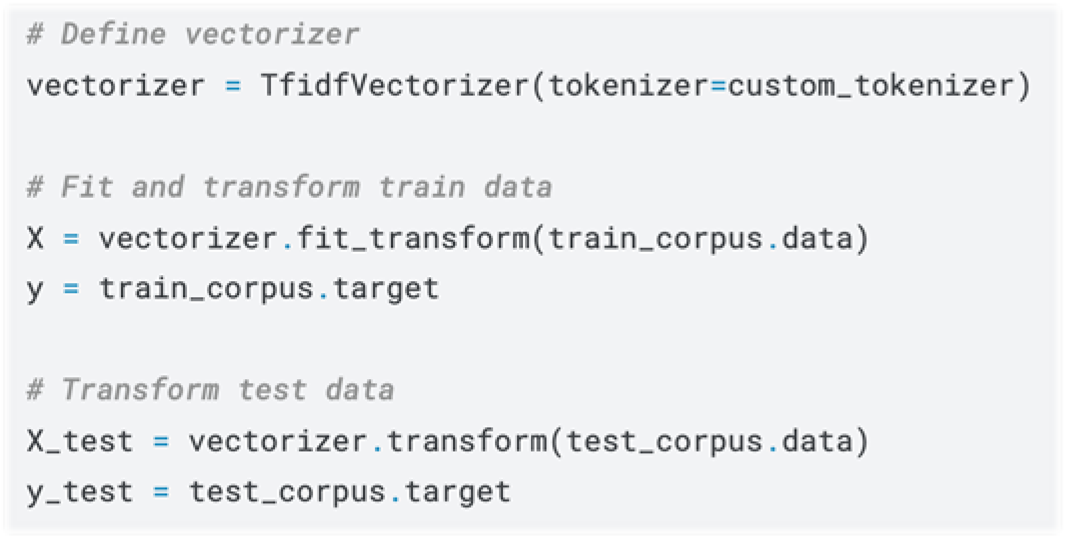
News classification (TF-IDF)



```
# Load full english language model
nlp = spacy.load('en_core_web_sm')

# Disable named-entity recognition and parsing to save time
unwanted_pipes = ['ner', 'parser']

# Custom tokenizer using spacy
def custom_tokenizer(doc):
    with nlp.disable_pipes(*unwanted_pipes):
        return [t.lemma_ for t in nlp(doc) if not t.is_punct
and not t.is_space and not t.is_stop and t.is_alpha]
```

https://www.kaggle.com/code/samuelcortinhas/nlp5-text-classification-with-naive-bayes

News classification (TF-IDF)

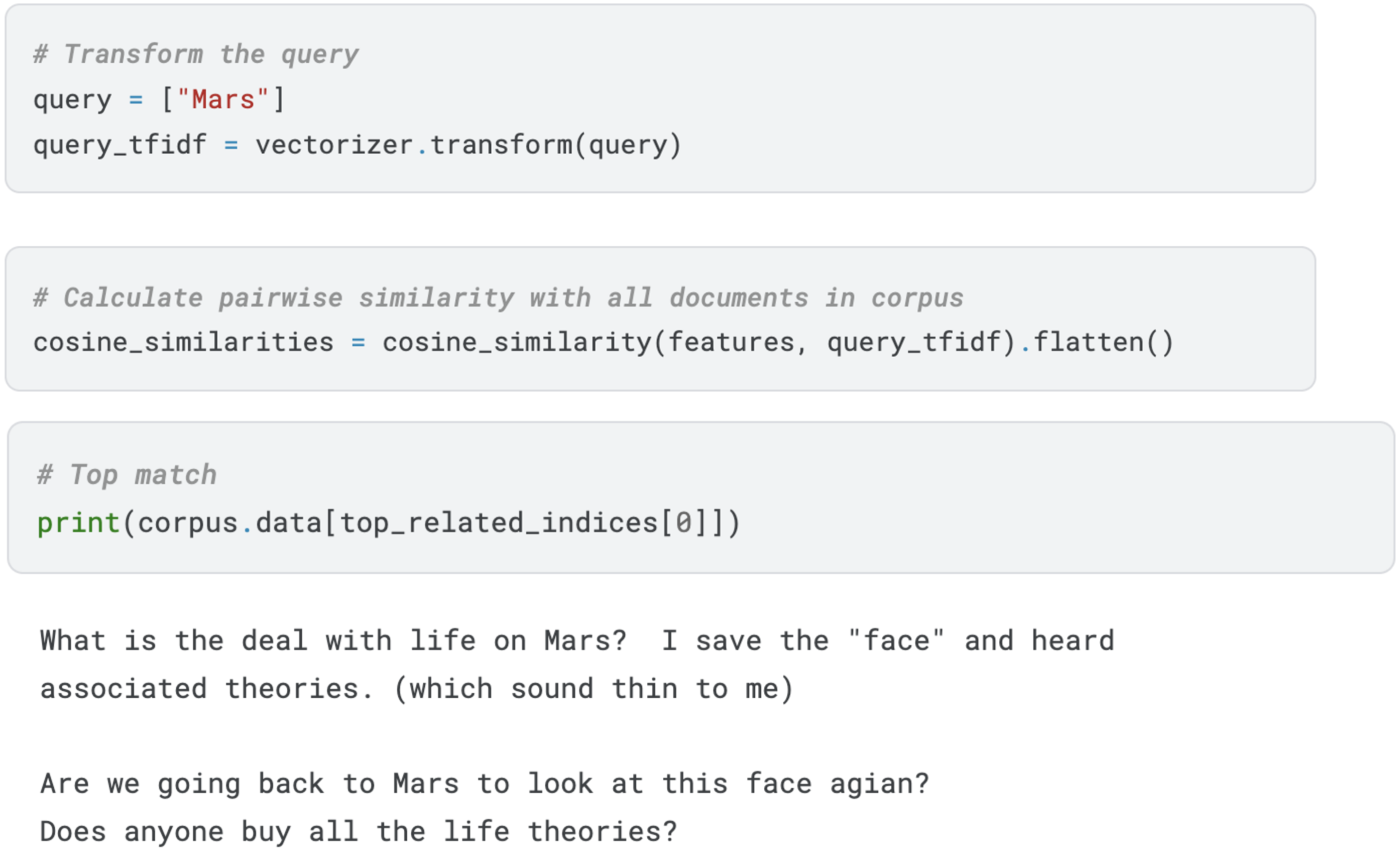


```python
# Define vectorizer
vectorizer = TfidfVectorizer(tokenizer=custom_tokenizer)

# Fit and transform train data
X = vectorizer.fit_transform(train_corpus.data)
y = train_corpus.target

# Transform test data
X_test = vectorizer.transform(test_corpus.data)
y_test = test_corpus.target
```

```python
clf = MultinomialNB() # can change it to logit model
clf.fit(X, y)

test_preds = clf.predict(X_test)

print('Test set F1-score:', f1_score(y_test, test_preds, average='macro'))
```

Document search with TF-IDF

```python
# Transform all textual data into TF-IDF representation
features = vectorizer.transform(doc_corpus)
```



```python
# Transform the query
query = ["Mars"]
query_tfidf = vectorizer.transform(query)
```

```python
# Calculate pairwise similarity with all documents in corpus
cosine_similarities = cosine_similarity(features, query_tfidf).flatten()
```

```python
# Top match
print(corpus.data[top_related_indices[0]])
```

```
What is the deal with life on Mars?  I save the "face" and heard
associated theories. (which sound thin to me)

Are we going back to Mars to look at this face agian?
Does anyone buy all the life theories?
```

https://www.kaggle.com/code/samuelcortinhas/nlp4-tf-idf-and-document-search

Sentiment analysis with TF-IDF

Build the same machine learning pipeline:

- Obtain labeled data (supervised learning)
- Feature engineering / transformation
  - Text preprocessing
  - TF-IDF vs. BoW & N-grams
- Model comparison / evaluation
  - Logit vs. Knn vs. Naive Bayes
  - AUC-ROC
  - F1-score

Course Notes

- Unsupervised learning next week
  - Cross-validation
  - PCA, Clustering
- Text analysis (V2) the week after
  - LIWC, LDA, Word2vec
  - Higher level traits
- Homework 1 released
  - Due on Oct 17!