# Preprocessing, vectorizing, and comparing text

CS 759/859 Natural Language Processing Lecture 3

Samuel Carton, University of New Hampshire

# Representing text numerically

Before we try to do anything computational with text, we need to create a representation our computer can actually work with

We often want this to include preprocessing and normalization that makes it easier to treat similar texts similarly

E.g.

- Case
- Stemming
- Tokenization
- Synonymy

# Case study: text similarity

Very frequent basic NLP task: how similar are these two texts?

Especially in comparison to these other two texts?

Why might we want to do this?

- Web search
- Classification based on similar labeled examples
- Plagiarism detection
- Etc.

I will use text similarity as a motivating task for preprocessing and vectorizing text.

# Our corpus

Four short movie reviews:

**Review 0: "The film was a delight--I was riveted."**
Review 1: "It's the most delightful and riveting movie."
Review 2: "It was a terrible flick, the worst I have ever seen."
Review 3: "I have a feeling the film was recut poorly.“

**Question**: Which review is most similar to review 0?

# Our corpus

Four short movie reviews:

Review 0: "The film was a delight--I was riveted."
**Review 1: "It's the most delightful and riveting movie."**
Review 2: "It was a terrible flick, the worst I have ever seen."
Review 3: "I have a feeling the film was recut poorly.“

**Answer**: review 1.

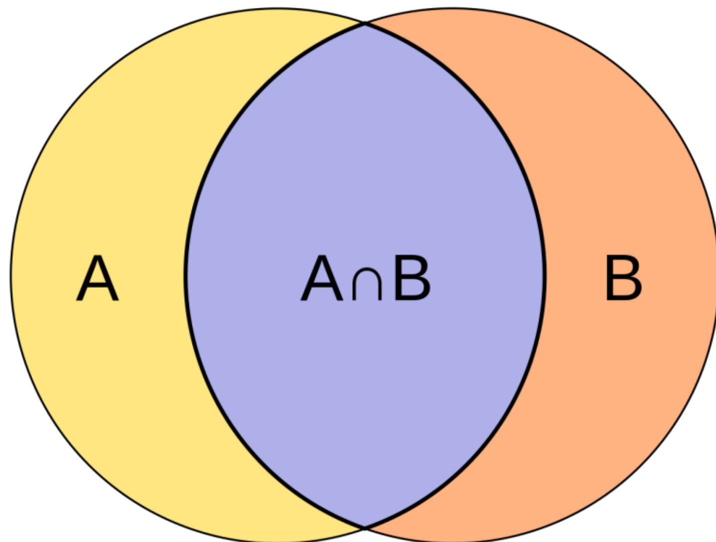
- But can we come up with a similarity metric that reflects that fact?

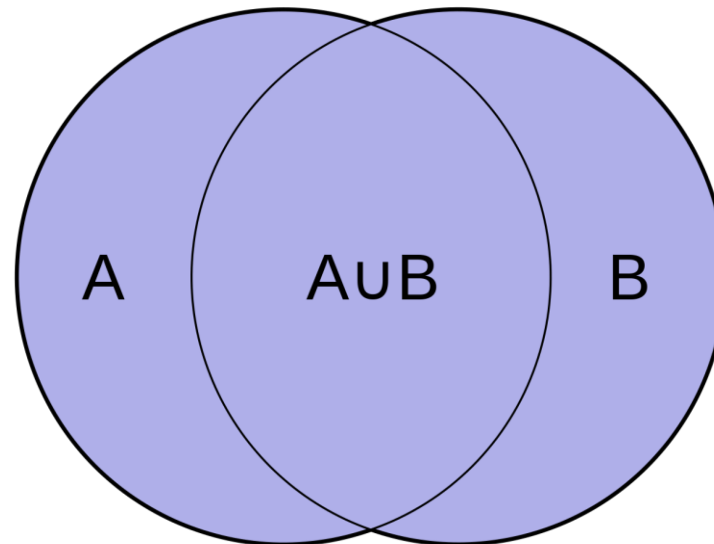
# Jaccard similarity

Very basic discrete similarity metric.

Given two sets, divide size of intersection by size of union

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$




https://en.wikipedia.org/wiki/Jaccard_index

# Bag-of-words representations

Simplest representation of text is as a **“bag-of-words”** without respect to order

This is also called a **unigram** or **1-gram** representation.

But how to identify distinct unigrams in text?

**Naïve solution:** split on whitespaces

- We’ll improve on this in a bit

**Example**:

"The film was a delight--I was riveted.“ →

```
['The', 'film', 'was', 'a', 'delight--I', 'was', 'riveted.']
```

# Basics of Colab, tokenization, & Jaccard similarity

**Headings:**

**Basics of Google Colab**

Our "dataset"

Jaccard similarity

Bag-of-words representation

Whitespace tokenization

Jaccard similarity on whitespace tokens

Text preprocessing with NLTK

# Jaccard similarity for reviews 0 and 2

**Review 0:** "The film was a delight--I was riveted.“ →

['The', 'film', 'was', 'a', 'delight--I', 'was', 'riveted.’]

**Review 2:** "It was a terrible flick, the worst I have ever seen." →

['It', 'was', 'a', 'terrible', 'flick,', 'the', 'worst', 'I', 'have', 'ever', 'seen.’]

**Intersection:** {'was', 'a’}

**Union:**

{'terrible', 'flick,', 'seen.', 'riveted.', 'worst', 'I', 'The', 'a', 'delight--I', 'the', 'It', 'have', 'was', 'film', 'ever’}

**Jaccard similarity:** 0.133

# Jaccard similarity for reviews 0 and 1

**Review 0:** "The film was a delight--I was riveted.“ →

```
['The', 'film', 'was', 'a', 'delight--I', 'was', 'riveted.’]
```

**Review 1:** "It's the most delightful and riveting movie.“ →

```
["It's", 'the', 'most', 'delightful', 'and', 'riveting', 'movie.’]
```

**Intersection:** {}

**Union:**

```
{"It's", 'delightful', 'and', 'the', 'most', 'was', 'film',
'riveted.', 'movie.', 'The', 'riveting', 'a', 'delight--I’}
```

**Jaccard similarity:** 0

**What went wrong here?**

# Preprocessing text

Various transformations we can perform on text in order to iron out superficial differences and home in on the types of similarity we are interested in

What preprocessing you do depends on your application

Basics include:

- **Lower-casing**
- Removing punctuation
- Removing common words, aka “stopwords”
- Removing unicode characters
    - Often needed for web text
- And a bunch of other stuff. Often some trial-and-error here

# Lower-casing

Very simple and obvious thing to do, but smooths out some differences

**Review 0:** "The film was a delight--I was riveted.“ →

`[‘the', 'film', 'was', 'a', 'delight--i', 'was', 'riveted.’]`

**Review 1:** "It's the most delightful and riveting movie.“ →

`[“it's", 'the', 'most', 'delightful', 'and', 'riveting', 'movie.’]`

Intersection: `{'the’}`

**Union:**

`{“it's", 'delightful', 'and', 'the', 'most', 'was', 'film', 'riveted.', 'movie.', 'riveting', 'a', 'delight--i’}`

**Jaccard similarity:** 0.083

**Better, but we’re still not beating .133**

# Lower-casing

Simplest way to do it in Python is just to use `str.lower()`

Not always appropriate!

- **Semantic differences**: "I told Frank to close up the shop" vs. "I have to be frank with you."
- **Syntactic differences:** "The …." vs. "the …"

**Rule of thumb:**

- Until we get to Transformer-based models, lower-casing should be a standard part of your preprocessing pipeline.

# Tokenization

**Tokenization**: splitting up a string sequence like into its component tokens

- **Token**: whatever pieces we are dividing text into
  - Often equates to individual words and pieces of punctuation
  - But transformer-based models use pieces of words
- Not as trivial as just splitting on whitespace

**Naïve example:**

**Review 0:** "The film was a delight--I was riveted.“ →

```
[‘the', 'film', 'was', 'a', 'delight--i', 'was', 'riveted.’]
```

**More sophisticated example:**

**Review 0:** “the film was a delight--I was riveted.“ → `['the', 'film', 'was', 'a', 'delight', '--', 'i', 'was', 'riveted', '.’]`

# NLTK

Natural Language ToolKit (NLTK): https://www.nltk.org/index.html

Most popular Python text processing package

Competes with SpacY, which is also good

Has lots of basic NLP functionality: tokenization, stemming, parsing, etc.

- We'll only be doing tokenization and stemming

# Tokenization

**Review 0:** "The film was a delight--I was riveted.“ →

['the', 'film', 'was', 'a', 'delight', '--', 'i', 'was', 'riveted', '.’]

**Review 1:** "It's the most delightful and riveting movie.“ →

[“it's", 'the', 'most', 'delightful', 'and', 'riveting', 'movie.’]

**Intersection:** {‘the’, ‘.'}

**Union:**

{‘it', "'s", 'delightful', 'and', 'most', 'was', 'film', 'riveted', ‘.', 'movie', 'riveting', 'a', 'delight’, ‘--', ‘i’}

**Jaccard similarity:** 0.133

**So now we’ve matched .133… are we done?**

# Stemming

**Stemming**: chop off affixes that distinguish plural versus singular and different tenses of words

- So we can match e.g. ‘delight’ with ‘delightful’, ‘riveting’ with ‘riveted’

**Example:**

“the film was a delight--I was riveted.“

tokenization → `['the', 'film', 'was', 'a', 'delight', '--', 'i', 'was', 'riveted', '.']`

stemming → `['the', 'film', 'wa', 'a', 'delight', '--', 'i', 'wa', 'rivet', '.']`

Contrast with **lemmatization**, which would recover the dictionary versions of the words

- But if all we’re doing is comparing, why would we care?
- So in practice, lemmatization is hardly ever done

# How do NLTK tokenization & stemming work

Default options for NLTK are Punkt tokenizer and Porter stemmer

- But there other options

**Punkt tokenizer**

- Uses a trained ML model to recognize where to split up sentences
- Trained on a big corpus of English-language text
- https://www.nltk.org/api/nltk.tokenize.punkt.html

**Porter stemmer**

- Uses a bunch of hand-written rules that are specific to the English language
- Old algorithm (1979)
- https://tartarus.org/martin/PorterStemmer/

You can treat these as black boxes for now, though we'll learn more about modeling as we move forward.

# Preprocessing with NLTK

**Headings:**

- Text preprocessing with NLTK
  - Non-NLTK basics
    - Lower-casing
    - Removing punctuation
  - Tokenization
  - Stemming
- Jaccard similarity with better tokenization

# Jaccard similarity (after preprocessing)

Review 0: "The film was a delight--I was riveted.“ →

['the', 'film', 'wa', 'a', 'delight', '--', 'i', 'wa', 'rivet', '.’]

Review 1: "It's the most delightful and riveting movie.“ →

['it', "'", 's', 'the', 'most', 'delight', 'and', 'rivet', 'movi', '.’]

Intersection: {'delight', 'rivet', '.', 'the'}

Union:

{"'", 'movi', 'most', 'delight', '--', '.', 'the', 'a', 'it', 'and', 'rivet', 'film', 'i', 's', 'wa’}

Jaccard similarity: .267 **Yay!**

# Jaccard similarity (after preprocessing)

Review 0: "The film was a delight--I was riveted.“ →

`['the', 'film', 'wa', 'a', 'delight', '--', 'i', 'wa', 'rivet', '.’]`

Review 2: "It was a terrible flick, the worst I have ever seen." →

`['it', 'wa', 'a', 'terribl', 'flick', ',', 'the', 'worst', 'i', 'have', 'ever', 'seen', '.’]`

Intersection: `{'the', 'wa', 'i', '.', 'a’}`

Union:

`{'flick', 'delight', 'seen', 'worst', '--', '.', 'the', 'a', 'it', 'rivet', 'have', ',', 'film', 'terribl', 'i', 'ever', 'wa’}`

Jaccard similarity: .294 **…dang.**

# Problem

"The film was a delight--I was riveted."

vs.

"It's the most delightful and riveting movie."

→Jaccard similarity .267

Intersection: `{'delight', 'rivet', '.', 'the'}`

"The film was a delight--I was riveted."

vs.

"It was a terrible flick, the worst I have ever seen."

→ Jaccard similarity .294

Intersection: `{'the', 'wa', 'i', '.', 'a'}`

What's the problem?

# Vectors

To go beyond very simple preprocessing, you really need to **vectorize** your text.

A **vector** is a 1-dimensional set of values, usually numeric.

Examples:

[0.1 8.2 11.7 0.5]

[1 2 3 4 5]

[True False True True False]

[1 0 0 1 1 0]

Different from a list because you are generally operating on the whole vector at once rather than iterating through it.

# Vector operations

In many ways can be treated a single number

**Addition**: [1 2 3] + [4 5 6] = [5 7 9]
**Subtraction**: [1 2 3] - [4 5 6] = [-3 -3 -3]
**Division**: [1 2 3] / [4 5 6] = [0.25 0.4 0.5]
**Multiplication**: [1 2 3] * [4 5 6] = [4 10 18]
**Power**:[1 2 3] ^ 2 = [1 4 9]

But there are certain operations that are only defined for vectors:
**Dot product**: [1 2 3] · [4 5 6] = sum([1 2 3] * [4 5 6]) = 32

There is a **lot** of stuff that can be done with vectors (see: all of linear algebra)
We will focus on just what we need to know to do the things we want to do

# Vectors and vectorizing text

**Headings:**

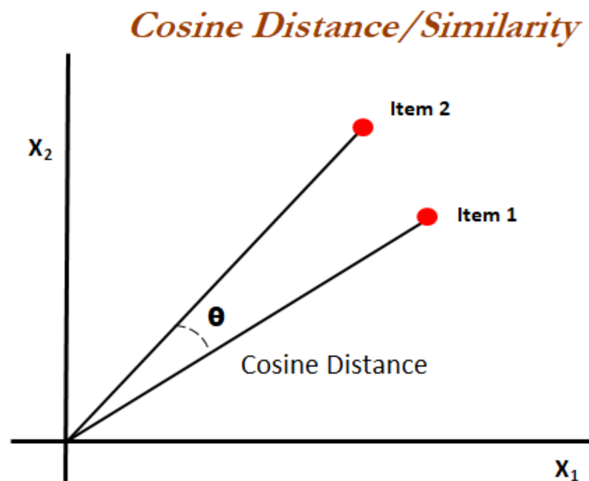
- Vectors with Numpy
  - Basic operations
  - Different dimensionalities
    - Convenient things you can do with numpy tensors
- Vectorizing text the hard way
  - Bag-of-words
    - Preliminary counting
    - From counts to vectors
- Jaccard similarity on vectors

# Cosine similarity

Given two vectors, defined as the **dot product** of the vectors divided by the product of the magnitudes of the two vectors

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2} \sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

https://en.wikipedia.org/wiki/Cosine_similarity




https://www.oreilly.com/library/view/statistics-for-machine/9781788295758/eb9cd609-e44a-40a2-9c3a-f16fc4f5289a.xhtml

# Jaccard vs cosine similarity

```
Text 1: The film was a delight--I was riveted.
Text 2: It's the most delightful and riveting movie.

Count vector 1: [1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Count vector 2: [1. 0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Jaccard similarity: 0.267
Cosine similarity: 0.365
```

# Jaccard vs cosine similarity

```
Text 1: The film was a delight--I was riveted.
Text 2: It was a terrible flick, the worst I have ever seen.

Count vector 1: [1. 1. 2. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Count vector 2: [1. 0. 1. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 0. 0.
1. 1. 1. 1. 1. 1. 1. 0. 0. 0.]

Jaccard similarity: 0.294
Cosine similarity: 0.480
```

Are we done? (still no)

# Cosine similarity

**Headings:**

Cosine similarity

# TF-IDF

**TF-IDF**: Term Frequency – Inverse Document Frequency

**Basic idea:** When we make a vector representation of a bag of words, **upweight** rare words and **downweight** common words

The value at slot *i* for a given sequence *s* should be the **term frequency** of word *i* within *s*, divided by the **document frequency** of word *i* in the corpus as a whole

# TF-IDF

**Headings:**

TF-IDF

Cosine similarity with TF-IDF

# Cosine similarity revisited

```
Text 1: The film was a delight--I was riveted.

Text 2: It's the most delightful and riveting movie.

Vector 1: [0.1 0.2 0.267 0.133 0.2 0.4 0.133 0.2 0.1 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]

Vector 2: [0.1 0. 0. 0. 0.2 0. 0. 0.2 0.1 0.2 0.4 0.4 0.4 0.4 0.4 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]

Cosine similarity: 0.162
```

```
Text 1: The film was a delight--I was riveted.

Text 3: It was a terrible flick, the worst I have ever seen.

Vector 1: [0.1 0.2 0.267 0.133 0.2 0.4 0.133 0.2 0.1 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]

Vector 2: [0.077 0. 0.103 0.103 0. 0. 0.103 0. 0.077 0.154 0. 0. 0. 0. 0. 0.308 0.308 0.308 0.308 0.154
0.308 0.308 0. 0. 0. ]

Cosine similarity: 0.135
```

So now we’ve finally (finally!) come up with a similarity metric that captures our intuitions

# Other similarity/distance metrics

**Euclidean**: Euclidean (l2) distance between the two vectors in vector space

- https://en.wikipedia.org/wiki/Euclidean_distance
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.euclidean.html (scipy implementation)

**Manhattan distance**: L1 distance between the two vectors in vector space

- https://en.wikipedia.org/wiki/Taxicab_geometry
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cityblock.html#scipy.spatial.distance.cityblock

**Others**: https://docs.scipy.org/doc/scipy/reference/spatial.distance.html#module-scipy.spatial.distance

- But really, 95% of people use Jaccard, Euclidean or cosine distance

# Vectorizing & similarity with Scikit-learn

**Headings:**

Vectorizing text with Scikit-Learn

Other similarity metrics

# Concluding thoughts

**Concepts**

- Bag-of-words representations (i.e. unigrams, 1-grams) of text
- Preprocessing text
  - Lower-casing
  - Tokenization
  - Stemming
- Vectorizing text
- Similarity metrics
  - Jaccard similarity
  - Cosine distance
  - Others
- TF-IDF

**Toolkits**

- NLTK
  - For tokenization & stemming
- Numpy
  - For manipulating vectors
- Scikit-Learn
  - For vectorizing lists of texts automatically
  - Also includes implementations of similarity metrics (Jaccard, cosine, etc)