Representing text numerically

CS 780/880 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.

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.“

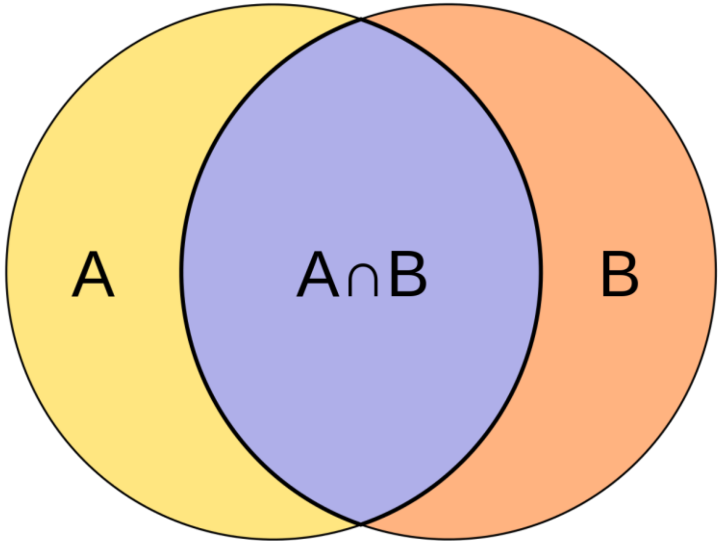
Which review is most similar to review 0?

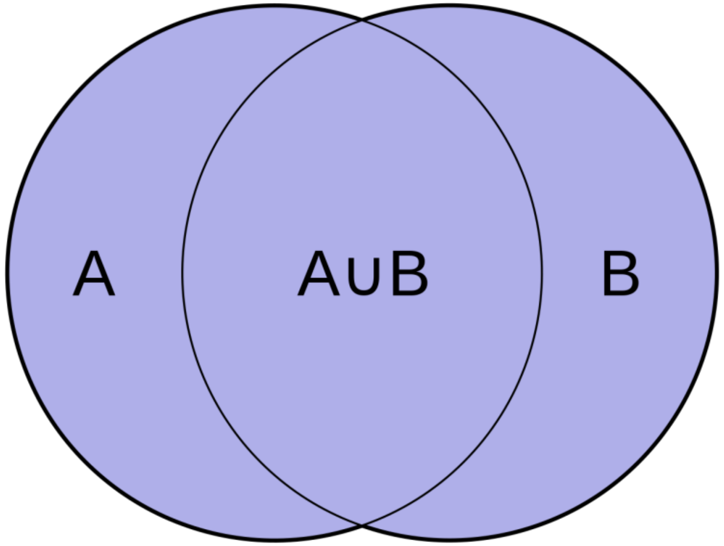
Jaccard similarity

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

Example:

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

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

Jaccard similarity for text

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

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

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

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

Intersection: [none]

Union:

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

Jaccard similarity: 0

Jaccard similarity for text

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

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

"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

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

Tokenization

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

- Slightly different from words

Example:

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

To do this: NLTK

NLTK

Natural Language ToolKit: 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

NLTK tokenization

```python
from nltk import WordPunctTokenizer

tokenizer = WordPunctTokenizer()
processed_tokens = [tokenizer.tokenize(review) for review in lowercased_reviews]
print('\n'.join([str(ts) for ts in processed_tokens]))

```

```
['the', 'film', 'was', 'a', 'delight', '--', 'i', 'was', 'riveted', '.']
['it', "'", 's', 'the', 'most', 'delightful', 'and', 'riveting', 'movie', '.']
['it', 'was', 'a', 'terrible', 'flick', ',', 'the', 'worst', 'i', 'have', 'ever', 'seen', '.']
['i', 'have', 'a', 'feeling', 'the', 'film', 'was', 'recut', 'poorly', '.']
```

https://www.nltk.org/api/nltk.tokenize.html

Stemming

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

So we can match e.g. ‘delight’ with ‘delightful’

Example:

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

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

→ `['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?

NLTK stemming

Very simple

```
[11]
from nltk import PorterStemmer

stemmer = PorterStemmer()
stemmed_tokens = [[stemmer.stem(t) for t in ts] for ts in processed_tokens ]

print('\n'.join([str(ss) for ss in stemmed_tokens]))
```

```
['the', 'film', 'wa', 'a', 'delight', '--', 'i', 'wa', 'rivet', '.']
['it', "'", 's', 'the', 'most', 'delight', 'and', 'rivet', 'movi', '.']
['it', 'wa', 'a', 'terribl', 'flick', ',', 'the', 'worst', 'i', 'have', 'ever', 'seen', '.']
['i', 'have', 'a', 'feel', 'the', 'film', 'wa', 'recut', 'poorli', '.']
```

Jaccard similarity (after preprocessing)

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

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

"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

Jaccard similarity (after preprocessing)

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

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

"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

Problem

"The film was a delight--I was riveted." vs. "It's the most delightful and riveting movie."
→Jaccard similarity .267
`{'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 in Python

Not implemented in standard Python

Implemented in wildly popular and ubiquitous numpy library

```
[15] import numpy as np
```

Basic operations

```
[16] v0 = np.array([1,2,3])
     v1 = np.array([4,5,6])
```

```
[17] v0 + v1

array([5, 7, 9])
```

```
[18] v0 - v1

array([-3, -3, -3])
```

Representing bag-of-words as a vector

Basic idea: each text is a vector the size of the vocabulary, with the number in each slot representing the count of that word in that text

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

the	1.0
film	1.0
wa	2.0
a	1.0
delight	1.0
--	1.0
i	1.0
rivet	1.0
.	1.0
it	0.0
'	0.0
s	0.0
most	0.0
and	0.0
movi	0.0
terribl	0.0
flick	0.0
,	0.0
worst	0.0
have	0.0
ever	0.0
seen	0.0
feel	0.0
recut	0.0
poorli	0.0

Jaccard similarity for (binary) vectors

We can (hackily) still do Jaccard similarity if we binarize our vectors to be only 0 and 1

```python
def binary_vector_jaccard(v0, v1, verbose=True):

    intersection_vector = v0 * v1
    union_vector = 1-((1-v0) * (1-v1))
    similarity = intersection_vector.sum() / max(union_vector.sum(), 0.1) #so that we never divide by zero
    if verbose:
        print(f'\nVector-based Jaccard similarity:')
        print(f'Vector 1:     {v0}')
        print(f'Vector 2:     {v1}')
        print(f'\nIntersection: {intersection_vector}')
        print(f'Union:        {union_vector}')
        print(f'\nJaccard similarity: {similarity:.3f}')

```

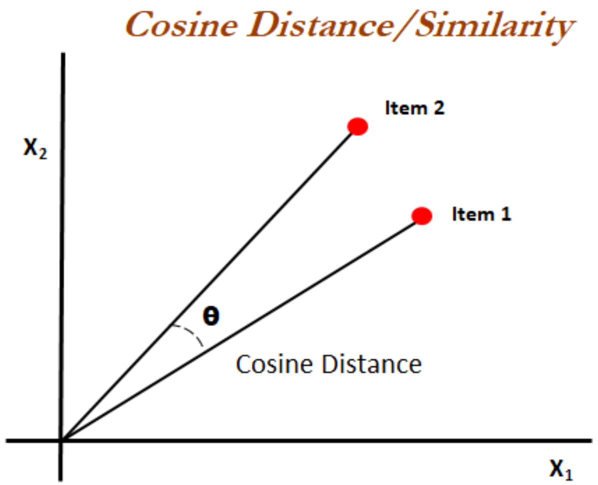
But it’s probably good to learn a similarity metric that can handle continuous values

Cosine similarity

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

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

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)

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

Manual TF-IDF

Lots of counting. See associated Colab Notebook

Easy TF-IDF: Scikit-Learn

Repeat all the preprocessing

```
# Just a reminder of what our corpus looks like
reviews
```

```
['The film was a delight--I was riveted.',
 "It's the most delightful and riveting movie.",
 'It was a terrible flick, the worst I have ever seen.',
 'I have a feeling the film was recut poorly.']
```

```
# It's actually a little awkward to integrate NLTK text preprocessing into scikit-learn vectorization,
# so perhaps the simplest thing to do is preprocess the text and stitch it back together with spaces
# before passing it to scikit-learn
#We'll use the tokenizer and stemmer we defined above to redo this

lowercased_reviews = [review.lower() for review in reviews]
tokenized_reviews = [tokenizer.tokenize(review) for review in lowercased_reviews]
stemmed_tokens = [[stemmer.stem(token) for token in review_tokens] for review_tokens in tokenized_reviews]
preprocessed_reviews = [' '.join(review_tokens) for review_tokens in stemmed_tokens]

preprocessed_reviews

```

Easy TF-IDF: Scikit-Learn

Then create and use a TfidfVectorizor (or a CountVectorizer if you just want counts)

```
# https://scikit-learn.org/

from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity as sklearn_cosine_similarity

```

```
[69]
# But the TF_IDF vectorizer works the exact same way, only it also does all that IDF stuff
tf_idf_vectorizer = TfidfVectorizer()

tf_idf_vectorized_reviews = tf_idf_vectorizer.fit_transform(preprocessed_reviews)

```

```
[72]
# And these IDFs will be reflected in the vectorized corpus array
print(tf_idf_vectorized_reviews.toarray())
```

```
[[0.    0.406 0.    0.    0.406 0.    0.    0.    0.    0.    0.    0.    0.406 0.    0.    0.269 0.658 0.   ]
 [0.441 0.348 0.    0.    0.    0.    0.    0.348 0.441 0.441 0.    0.    0.348 0.    0.    0.23  0.    0.   ]
 [0.    0.    0.38  0.    0.    0.38  0.3   0.3   0.    0.    0.    0.    0.    0.38  0.38  0.198 0.243 0.38 ]
 [0.    0.    0.    0.451 0.355 0.    0.355 0.    0.    0.    0.451 0.451 0.    0.    0.    0.235 0.288 0.   ]]
```

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. List-based Jaccard similarity:

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
```

Other similarity/distance metrics

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

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

Concluding thoughts

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.“

With preprocessing and frequency normalization, we conquered several of the problems we identified at the beginning.

But what about synonymy (e.g. “film” versus “flick”)?

And what about word order?

And what if we’re more interested in sentence structure than lexical similarity?