IST664 Lab: Extracting Semantics

Edited by Jeff Stanton and Preeti Jagadev

If the end goal of a project is natural language understanding, at some point we must figure out how to associate words, phrases, sentences, and larger structures with the meaning that they convey. Figuring out the meaning of utterances is the goal of semantics. In fact the leading definition of the word semantics is, "the study of the meaning of words." In this lab, we are going to conisder three different approaches to extracting semantics from text.

One of the earliest and most comprehensive efforts to explore semantics on a large scale arose from the work of George Miller at Princeton in the mid-1980s. The database arising from Miller's work, known as WordNet, was an award-winning effort to create a network of interconnected meanings of words. The WordNet project is alive and well in the present day, in fact there is an international organization known as the Global WordNet Association that continues research and development of WordNet. Check it out here:

http://globalwordnet.org

GWA has an annual conference and offers some databases and documentation to the world community for free. These databases, now covering more than 200 languages, represent a massive amount of collective human effort, which is both amazing and illustrative of the core challenge with such resources: The maintenance of manually developed language resources requires lots of manual labor.

Possibly, some of the value of what WordNet provides has been or will eventually be superceded by approaches based on deep learning. We see inklings of this with GloVe word embedding and more sophisticated embedding approaches such as BERT that are initially trained (in an unsupervised mode) on masses of unlabeled natural language text. Even so, WordNet works (and works fast!) without needing to provide any training data, so there are many applications where it is still an appropriate choice. In this first part of the lab, we explore some of the WordNet capabilities afforded by NLTK.

```python
import nltk
nltk.download('wordnet') # Colab does not have it installed by default
nltk.download('omw-1.4') # Colab does not have it installed by default
from nltk.corpus import wordnet as wn

type(wn.synsets) # A key function call (method) that we will use
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
method
```

```
# Let's start by getting data on the word cat. A "synset" is a very basic
# data structure supported by NLTK that can be used to look up synonyms
# and related information for any word that the WordNet folks have included
# in the giant database.
syn = wn.synsets('cat')
type(syn), len(syn)
```

```
(list, 10)
```

```
# The output above shows that the return data structure has 10 elements in a
# list. What are these different list elements?
[type(s) for s in syn]
```

```
[nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset,
 nltk.corpus.reader.wordnet.Synset]
```

```
# Each element in the list is a synset object. We have more than one whenever
# there is more than one sense of the word.

cat0 = syn[0] # Let's look at some of the details for the first synset

print ("Synset name :  ", cat0.name())

# Defining the word
print ("\nSynset meaning : ", cat0.definition())

# list of phrases that use the word in context; not all words have these
print ("\nSynset example : ", cat0.examples())
```

```
Synset name :   cat.n.01

Synset meaning :  feline mammal usually having thick soft fur and no ability to roar: dc

Synset example :  []
```

Note that the synset name has interesting information in it: Of course the word itself comes first, but then the letter after the dot indicates the part of speech. The number after the second dot reveals the variant. So cat.n.01 would be read as "the first noun sense of cat." The fact that cat.n.01 appears as the first synset in the list indicates that linguists believed it to be the most commonly used sense of the word.

WordNet is organized as a tree structure, where we can find more specific and more general terms related to a particular word by tracing up or down the branches and twigs of the tree. A "hypernym" - which you can think of as "higher level name" - is a more general term that encompasses the word we are focusing on. In the other direction, a "hyponym" is an example of a word that is more specific than the word we are focusing on. As a mnemonic, remember that "hyper" means "excess" or "above" as in "hyperactive." On the other hand, "hypo" means below, as in "hypothermia."

```
print ("Synset name :  ", cat0.name()) # Let's show the name again

# Here is the "root" word - the highest level hypernym
print ("\nSynset root hypernym:  ", cat0.root_hypernyms())

# These are the more general terms
print ("\nSynset hypernyms:  ", cat0.hypernyms())

# These are the more specific terms
print ("\nSynset hyponyms:  ", cat0.hyponyms())


```

```
Synset name :   cat.n.01

Synset root hypernym:   [Synset('entity.n.01')]

Synset hypernyms:   [Synset('feline.n.01')]

Synset hyponyms:   [Synset('wildcat.n.03'), Synset('domestic_cat.n.01')]
```

The second and subsequent elements in the synset list (if any) are alternative word senses. If you're a music fan, you might be able to think of another use of the word "cat." In the first line of code below, we extract the second element of the synset list. Use it to show the name, definition, example, root hypernym, hypernyms, and hyponyms for this first synonym of cat.

```
# Exercises: Explore the second synset for "cat."
# Create a new block of code for each of the following exercises.

cat1 = syn[1] # Let's look at some of the details for the second element

# 4.1: Print the name of cat1: What part of speech is it?
print("Synset name :  ", cat1.name())
print("Part of Speech:", cat1.pos())
#Part of Speech N

# 4.2: Print the definition of cat1
print("Definition : ", cat1.definition())

# 4.3: Print the examples of use of cat1 in context
```

```
print("Examples : ", cat1.examples())

# 4.4: Print the root hypernym of cat1
print("Root Hypernym : ", cat1.root_hypernyms())

# 4.5: Print a list of hypernyms of cat1
print("Hypernyms : ", cat1.hypernyms())

# 4.6: Print a list of hyponyms of cat1
print("Hyponyms : ", cat1.hyponyms())
```

```
Synset name :   guy.n.01
Part of Speech: n
Definition :  an informal term for a youth or man
Examples :  ['a nice guy', "the guy's only doing it for some doll"]
Root Hypernym :  [Synset('entity.n.01')]
Hypernyms :  [Synset('man.n.01')]
Hyponyms :  [Synset('sod.n.04')]
```

```
# Given what you saw above, does it make sense now why the root hypernym
# of cat is "entity" rather than something more specific like "animal?"

# Cat is such a common word in English that it has been reused to refer
# to many different kinds of things. Let's go back to the complete list
# to show all of the definitions:

[s.definition() for s in syn]
```

```
['feline mammal usually having thick soft fur and no ability to roar: domestic cats; wildcats',
 'an informal term for a youth or man',
 'a spiteful woman gossip',
 'the leaves of the shrub Catha edulis which are chewed like tobacco or used to make tea; has the effect of a euphoric stimulant',
 'a whip with nine knotted cords',
 'a large tracked vehicle that is propelled by two endless metal belts; frequently used for moving earth in construction and farm work',
 'any of several large cats typically able to roar and living in the wild',
 'a method of examining body organs by scanning them with X rays and using a computer to construct a series of cross-sectional scans along a single axis',
 "beat with a cat-o'-nine-tails",
 'eject the contents of the stomach through the mouth']
```

```
# That's an amazing variety. Let's also glue the corresponding synset name
# to the definition so that we can see the parts of speech and numbering.
[ (s.name(), s.definition())  for s in syn]
```

```
[('cat.n.01',
  'feline mammal usually having thick soft fur and no ability to roar: domestic cats; wildcats'),
 ('guy.n.01', 'an informal term for a youth or man'),
 ('cat.n.03', 'a spiteful woman gossip'),
 ('kat.n.01',
  'the leaves of the shrub Catha edulis which are chewed like tobacco or used to make
tea; has the effect of a euphoric stimulant'),
 ("cat-o'-nine-tails.n.01", 'a whip with nine knotted cords'),
 ('caterpillar.n.02',
  'a large tracked vehicle that is propelled by two endless metal belts; frequently
used for moving earth in construction and farm work'),
 ('big_cat.n.01',
  'any of several large cats typically able to roar and living in the wild'),
 ('computerized_tomography.n.01',
  'a method of examining body organs by scanning them with X rays and using a computer
to construct a series of cross-sectional scans along a single axis'),
 ('cat.v.01', "beat with a cat-o'-nine-tails"),
 ('vomit.v.01', 'eject the contents of the stomach through the mouth')]
```

```
# That last one is British slang, probably arising from the propensity of
# domestic cats to retch hairballs. Anyway. . . We can also get lemmas for
# each synonym entry in our list of 10:
[ (s.name(), s.lemma_names())  for s in syn]
```

```
[('cat.n.01', ['cat', 'true_cat']),
 ('guy.n.01', ['guy', 'cat', 'hombre', 'bozo']),
 ('cat.n.03', ['cat']),
 ('kat.n.01',
  ['kat', 'khat', 'qat', 'quat', 'cat', 'Arabian_tea', 'African_tea']),
 ("cat-o'-nine-tails.n.01", ["cat-o'-nine-tails", 'cat']),
 ('caterpillar.n.02', ['Caterpillar', 'cat']),
 ('big_cat.n.01', ['big_cat', 'cat']),
 ('computerized_tomography.n.01',
  ['computerized_tomography',
   'computed_tomography',
   'CT',
   'computerized_axial_tomography',
   'computed_axial_tomography',
   'CAT']),
 ('cat.v.01', ['cat']),
 ('vomit.v.01',
  ['vomit',
   'vomit_up',
   'purge',
   'cast',
   'sick',
   'cat',
   'be_sick',
   'disgorge',
   'regorge',
   'retch',
   'puke',
   'barf',
   'spew',
   'spue',
   'chuck',
   'upchuck',
   'honk',
   'regurgitate',
   'throw_up'])]
```

If you examined the output above carefully, you will see a lemma_name for guy.n.01, which is "bozo." In the late 1920s, this was a slang word which had the connotation of a big, muscular person of sub-normal intelligence. Gradually over the next century, this evolved into meaning a goofy or incompetent person. The use of this sense of the word peaked in 2012 and has been declining since. This anecdote illustrates the difficulty of maintaining an information resource like wordnet: Language usage changes over time, sometimes quite rapidly, and keeping up with those changes is time-consuming, expensive work.

```
# The elements of each of list shown above (as the second part of the tuple)
# are plain words - the lemma names - representing the synonym set.
# This could come in handy later, so let's make sure we know how to
# extract each synonym.

[s.lemma_names()[0] for s in syn]
```

```
['cat',
 'guy',
 'cat',
 'kat',
 "cat-o'-nine-tails",
 'Caterpillar',
 'big_cat',
 'computerized_tomography',
 'cat',
 'vomit']
```

```
from enum import unique
# Now repeat the process by finding the synset for an adjectival word, like
# good, bad, great, horrid. etc. Show the list of lemma_names for that word.
# As a related task, reduce that list of lemma names to its unique set
# in order to eliminate duplicates. Can you figure out
# how to do all that with just one line of code?

# 4.7: Generate a unique set of lemmas for an adjective of your choice.

unique_lemmas = set(lemma for syn in wn.synsets("horrid", pos=wn.ADJ) for lemma in syn.l
print(unique_lemmas)

unique

#used Source https://www.phind.com/

```

```
{'outrageous', 'horrid', 'hideous', 'horrific'}
```

```
enum.unique
def unique(enumeration)

Class decorator for enumerations ensuring unique member values.
```

```
# There are a couple more useful things we can do with a synset. First, we can
# ask WordNet for the part of speech for each entry:
from tabulate import tabulate # To make a neat table

takesyn = wn.synsets('take') # The word "take" has many senses - noun and verb

poslist = [(s.lemma_names()[0], s.pos(), s.definition()) for s in takesyn]

print(tabulate(poslist,  headers=["Word", "POS", "Definition"]))
```

```
Word         POS    Definition
-----------  -----  ------------------------------------------------------------------
return       n      the income or profit arising from such transactions as the sale of l
take         n      the act of photographing a scene or part of a scene without interrup
take         v      carry out
take         v      require (time or space)
lead         v      take somebody somewhere
take         v      get into one's hands, take physically
assume       v      take on a certain form, attribute, or aspect
take         v      interpret something in a certain way; convey a particular meaning or
bring        v      take something or somebody with oneself somewhere
take         v      take into one's possession
take         v      travel or go by means of a certain kind of transportation, or a cert
choose       v      pick out, select, or choose from a number of alternatives
accept       v      receive willingly something given or offered
fill         v      assume, as of positions or roles
consider     v      take into consideration for exemplifying purposes
necessitate  v      require as useful, just, or proper
take         v      experience or feel or submit to
film         v      make a film or photograph of something
remove       v      remove something concrete, as by lifting, pushing, or taking off, or
consume      v      serve oneself to, or consume regularly
take         v      accept or undergo, often unwillingly
take         v      make use of or accept for some purpose
take         v      take by force
assume       v      occupy or take on
accept       v      admit into a group or community
take         v      ascertain or determine by measuring, computing or take a reading fro
learn        v      be a student of a certain subject
claim        v      take as an undesirable consequence of some event or state of affairs
take         v      head into a specified direction
aim          v      point or cause to go (blows, weapons, or objects such as photographi
take         v      be seized or affected in a specified way
carry        v      have with oneself; have on one's person
lease        v      engage for service under a term of contract
subscribe    v      receive or obtain regularly
```

```
take         v      buy, select
take         v      to get into a position of having, e.g., safety, comfort
take         v      have sex with; archaic use
claim        v      lay claim to; as of an idea
accept       v      be designed to hold or take
contain      v      be capable of holding or containing
take         v      develop a habit
drive        v      proceed along in a vehicle
take         v      obtain by winning
contract     v      be stricken by an illness, fall victim to an illness
```

Having all of the most common words in a language organized based on their hypernyms and hyponyms supports some interesting operations. For example, the noun senses of "dog" and "cat" that refer to pets both have mammal as a "container" word. So we can traverse our way upward from "cat" to find the common ancestor word and then traverse back down to "dog." If we started with "cat" and we wanted to get to "doctor" it would probably take a lot more steps, because the common ancestor word would be much more general.

This leads to an interesting possibility: We can calculate the similarity between any pair of words by measuring the length of the "path" along the twigs and branches that connects two words. Here's an example to illustrate:

```python
# Pay close attention: the "synset" method looks up ONE synset if it
# exists. We have to specify exactly which synset we are talking about,
# so that's why we use something like bird.n.01 to refer to the first
# noun sense of bird. Earlier in this lab we used the "synsets" method
# which will look up all of the available synsets for a word. So "synset"
# and "synsets" do slightly different jobs.
birdsyn = wn.synset('bird.n.01')
goatsyn = wn.synset('goat.n.01')
sheepsyn = wn.synset('sheep.n.01')

birdsyn.path_similarity(goatsyn) # Bird to goat
# These distances are normalized to be on a scale of 0 to 1 where 0
# is least similar and 1 is most similar.

```

```
0.1111111111111111
```

```python
# Does this value make sense?
birdsyn.path_similarity(sheepsyn) # Bird to sheep
# Not Similar at all
```

```
0.1111111111111111
```

```
# How about goat to sheep?
goatsyn.path_similarity(sheepsyn)
#Shows Some Correlation
```

```
0.3333333333333333
```

Discuss With Your Partner

Just above, you have three measures of word similarity. The scores have been normalized to a scale of 0 to 1 and they represent the "path length" between the two words in the synset hierarchy. Discuss these scores with your partner and make sure you agree on why two of them are identical and the other one is higher. The idea of "path length" means that there must be a traversal path between every pair of words in the database, yes? What would that look like for a noun and a verb?

```
# As with many things related to language, there is often an alternative way
# to do something. Leacock-Chodorow similarity also uses the path lengths,
# but also considers how deep the least common ancestor is in the hierarchy.
# Resnik similarity also considers the relative frequency of a word in a
# corpus you provide. We show the earlier value of path similarity here just
# for the sake of comparison.
nltk.download('wordnet_ic')
from nltk.corpus import wordnet_ic
brown_ic = wordnet_ic.ic('ic-brown.dat')

birdsyn.path_similarity(goatsyn), birdsyn.lch_similarity(goatsyn), birdsyn.res_similarit
```

```
[nltk_data] Downloading package wordnet_ic to /root/nltk_data...
[nltk_data]   Unzipping corpora/wordnet_ic.zip.
(0.1111111111111111, 1.4403615823901665, 5.2175784741185165)
```

```
# Obviously, these other similarity measures are calibrated on different
# scales from path similarity. Add code to produce the Leacock-Chodorow
# and the Resnick similarity for sheepsyn to goatsyn

# 4.8: Compute L-C and Res similarity for sheepsyn to goatsyn

brown_ic = wordnet_ic.ic('ic-brown.dat')

sheepsyn = wn.synset('sheep.n.01')
goatsyn = wn.synset('goat.n.01')

path_similarity = sheepsyn.path_similarity(goatsyn)
lch_similarity = sheepsyn.lch_similarity(goatsyn)
res_similarity = sheepsyn.res_similarity(goatsyn, brown_ic)

print("Path Similarity:", path_similarity)
print("L-C Similarity:", lch_similarity)
print("Resnick Similarity:", res_similarity)

#used source https://www.phind.com/

```

```
Path Similarity: 0.3333333333333333
L-C Similarity: 2.538973871058276
Resnick Similarity: 8.005695458684853
```

```
# OK, one last WordNet operation: Antonyms. If we want to find a word with
# the opposite meaning, WordNet can provide us with choices:
syn = wn.synsets('good') # Grab all of the synonyms for good
[(s.name(), s.definition()) for s in syn] # Display them
```

```
[('good.n.01', 'benefit'),
 ('good.n.02', 'moral excellence or admirableness'),
 ('good.n.03', 'that which is pleasing or valuable or useful'),
 ('commodity.n.01', 'articles of commerce'),
 ('good.a.01',
  'having desirable or positive qualities especially those suitable for a thing
specified'),
 ('full.s.06', 'having the normally expected amount'),
 ('good.a.03', 'morally admirable'),
 ('estimable.s.02', 'deserving of esteem and respect'),
 ('beneficial.s.01', 'promoting or enhancing well-being'),
 ('good.s.06', 'agreeable or pleasing'),
 ('good.s.07', 'of moral excellence'),
 ('adept.s.01', 'having or showing knowledge and skill and aptitude'),
 ('good.s.09', 'thorough'),
 ('dear.s.02', 'with or in a close or intimate relationship'),
 ('dependable.s.04', 'financially sound'),
 ('good.s.12', 'most suitable or right for a particular purpose'),
 ('good.s.13', 'resulting favorably'),
 ('effective.s.04', 'exerting force or influence'),
 ('good.s.15', 'capable of pleasing'),
 ('good.s.16', 'appealing to the mind'),
 ('good.s.17', 'in excellent physical condition'),
 ('good.s.18', 'tending to promote physical well-being; beneficial to health'),
 ('good.s.19', 'not forged'),
 ('good.s.20', 'not left to spoil'),
 ('good.s.21', 'generally admired'),
 ('well.r.01',
  "(often used as a combining form) in a good or proper or satisfactory manner or to a
high standard (`good' is a nonstandard dialectal variant for `well')"),
 ('thoroughly.r.02',
  "completely and absolutely (`good' is sometimes used informally for `thoroughly')")]
```

```
# We'll choose the first adjectival form for finding antonyms:
goodsyn = wn.synset('good.a.01')

# Now get the antonym from the lemma
[l.antonyms() for l in goodsyn.lemmas()]
```

```
[[Lemma('bad.a.01.bad')]]
```

```
# Now look up the antonym(s) for the adjectival sense of bad.

# 4.9: Look up the antonym for bad

badsyn = wn.synset('bad.a.01')
[l.antonyms() for l in badsyn.lemmas()]
```

```
[[Lemma('good.a.01.good')]]
```

Part 2 - Word Sentiment

Using WordNet, a truly massive effort to build a tree of semantics for words in various languages, we've looked up synonyms, antonyms, definitions, and examples for a variety of words. We've also gotten our first indication of quantitative relations between words, i.e., the similarity measures between pairs of words. The tree structure of WordNet was intentionally designed so that similar words were close to each other and semantically different words further away.

But what can we do with this kind of information? One of the first practical analytical methods for using semantics arises from efforts to measure sentiment in fragments of natural language - basically whether a phrase, sentence, paragraph or other structure is trying to say something positive or negative. Positive versus negative is perhaps the simplest form of sentiment, but naturally there are other forms as well: good vs. bad, happy vs. sad, useful vs. useless, etc. Maybe there is some way that we could use the structure of WordNet to get at these forms of sentiment. Let's start simple and then work our way to some more complex options.

```
# First, we need a dataset to work on. Anything with some comments and
# a sentiment indicator will do. This URL refers to a project on Github
# that Dennis Pan posted. If the URL is unavailable, find another CSV file
# with text comments and a sentiment score.
import pandas as pd
data = pd.read_csv("https://raw.githubusercontent.com/dennisypan/Quick-N-Dirty-Sentiment
print(list(data)) # Show column names
```

```
['Unnamed: 0', 'Business', 'Comments', 'Rating', 'Sentiment Label', 'Sentiment Score', '
```

```
# So we will be examining the Comments for hints about sentiment along with
# Sentiment Label and Sentiment Score, which were set up by the creator of the
# dataset to help with machine learning tasks.

# Show the number of rows and columns in this data set
data.shape
```

```
(70, 7)
```

```
# A very small dataset, but fine for what we are doing here.

# View a few comments. Note that the name of this data field ('Comments') is
# peculiar to the Dennis Pan Yelp/Sushi data. If you change the dataset, you may
# need to change the name of the field where the comments are found.
data['Comments'][0:5]
```

	Comments
0	Boyfriend and I ordered togo sushi and picked ...
1	I am pleased to recommend this restaurant in S...
2	great food been going to japan town since i wa...
3	Small sushi boat restaurant located in SJ Japa...
4	Sushi Maru is one of the many gems of SJ's Jap...

dtype: object

```
#
# For the super bonus challenge at the end, you might need to use a regular
# expression to remove punctuation and digits from the text. You could put
# that code here. This regex matches anything that is not an alpha character
# or whitespace: [^a-zA-Z\s]
# Don't worry about this for now, though. We want to first see the results
# with punctuation and other stuff left in the text.
#
```

```
# We know that the word "good" has many senses, but they are generally
# all positive as we saw in the WordNet work above. What if we just find
# all the comments where a good-related word appears in the text?
# Would that tell us anything about sentiment?

# First, let's see if we can just match one word
for g in data['Comments']:
  if 'good' in g.lower():
    print(g)
```

```
I am pleased to recommend this restaurant in San Jose's "Japantown"right next door to Ur
Sushi Maru is one of the many gems of SJ's Japan town. They have a rotating sushi belt,
Sushi Maru is a conveyor belt sushi restaurant and a popular lunch spot during the week.
Food: Sushi, bento, udon, donburi.....all consistent and good flavors. If you are seatec
```

```
I struggle with this joint.  To me it's a "tweener" meaning it's sometimes "just-ok" whi
Delicious food and great service, but a little expensive...  I'm a big fan of their pump
I've come here once before and the first time was good. This time... no. The food on the
This place is amazing! The staff was so friendly and we were seated quickly (it was also
This is my favorite sushi place in San Jose! Super good, super high quality too. Great p
Stopped by for a quick lunch on the way back from Santa Cruz. Although it was memorials
The plusses: Food is very good, service is friendly, reasonable price, and we didn't hav
You need to chose the guy in the bar to make Nanook but not the women. She made it in he
So yummy here, we where here on Christmas Eve. And we wanted sushi, thank good it was op
This is my favorite sushi place. The staff and chefs are friendly and always working har
Best bento place ever! We eat here like every month. We get the Chicken Teriyaki and Kat
I came here for lunch today with my friend and his Mom, and was surprised when I saw tha
I had the Chicken teriyaki and Salmon along with the Eel and Octopus Sushi and it was ve
Our families favorite sushi spot here in Japantown. Love the options and the chefs reall
The menu is slightly limited, not many rolls (no special rolls, rolls come in 2-6 pieces
Food was pretty good with decent price. The place has both sushi belt and tables so you
It was my first time coming here for lunch with my dad! We came around 1pm and decided t
Sat at the bar and the service was terrible.  Fish was presliced and the portion of fish
First time here and the people were very nice and the food was good. Enjoyed the beef ta
My go to sushi bar in San Jose.  I have come here for many years and when I read yelp re
My sister in law and brother in law took me here for the first tine.  I must say, I LOVE
Not much of a wait during 12:30 lunch time. It is somewhat busy. There's a sushi boat co
Showed up and there was no table my kid could get a high chair for without obstructing t
Walking around sj japantown in the carmeet and me and my sibs decide to eat some Japanes
Sushi Maru in Jtown is top notch value. I would say arguably the best value sushi in San
```

```python
# Before moving on, make sure to take a careful look at the list of comments
# above. Each should contain the word "good." What do you observe?
# I see words like Great and Pleased
# Now look for good AND all of its synonyms. Notice the use of set().
lemmas = set([s.lemma_names()[0] for s in wn.synsets('good')])

# This is going to be a longer list, so let's just count the matches for now
# rather than printing them all.
matches = 0

for g in data['Comments']:
  for l in lemmas:
    if l in g.lower():
      matches += 1


print(matches)
```

```
44
```

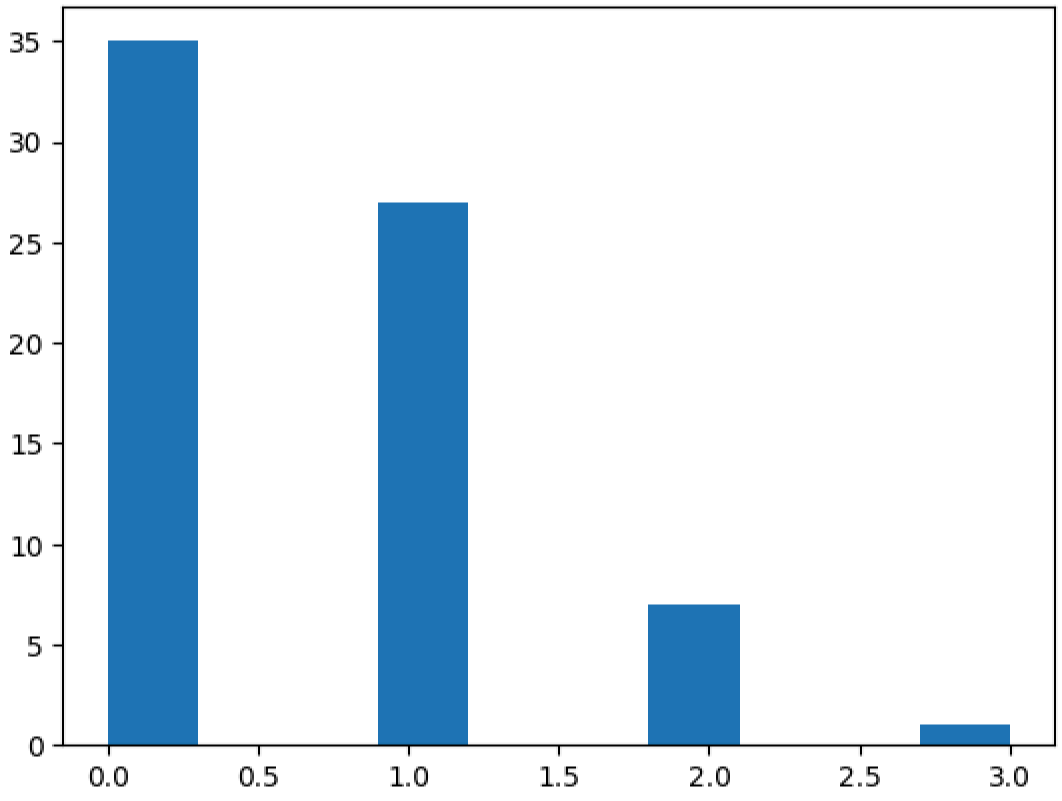
```python
# That's useful: We got a lot of mentions of all good-related words
# in the 70 comments stored in this dataset. This time we will count
# how many matches we get per comment.

lemmas = set([s.lemma_names()[0] for s in wn.synsets('good')])
```

```

# Here we will assign matches into a list containing 70 elements
matches = [0] * data.shape[0] # Fill a list with 70 zeroes

# This loop uses an enumerator, which is a cool Python solution to a
# common programming problem. The "i" that the enumerator produces is the
# index of the corresponding g. In this case the "g"s are the comments
# extracted from each successive row.
for i, g in enumerate(data['Comments']): # Loop over all comments
  for l in lemmas:  # Loop over all lemmas of good
    if l in g.lower(): # If we get a match, increment the count
      matches[i] += 1

len(matches) # We should have 70 entries
```

```
70
```

```
# What's the distribution of counts
import matplotlib.pyplot as plt
plt.hist(matches)
```

```
(array([35.,  0.,  0., 27.,  0.,  0.,  7.,  0.,  0.,  1.]),
 array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8, 2.1, 2.4, 2.7, 3. ]),
 <BarContainer object of 10 artists>)
```



Discuss With Your Partner

Interpret the histogram above. What does the leftmost bar signify? It looks like there are 35 of something? What?

Also, what's the maximum number of matches found in a comment?

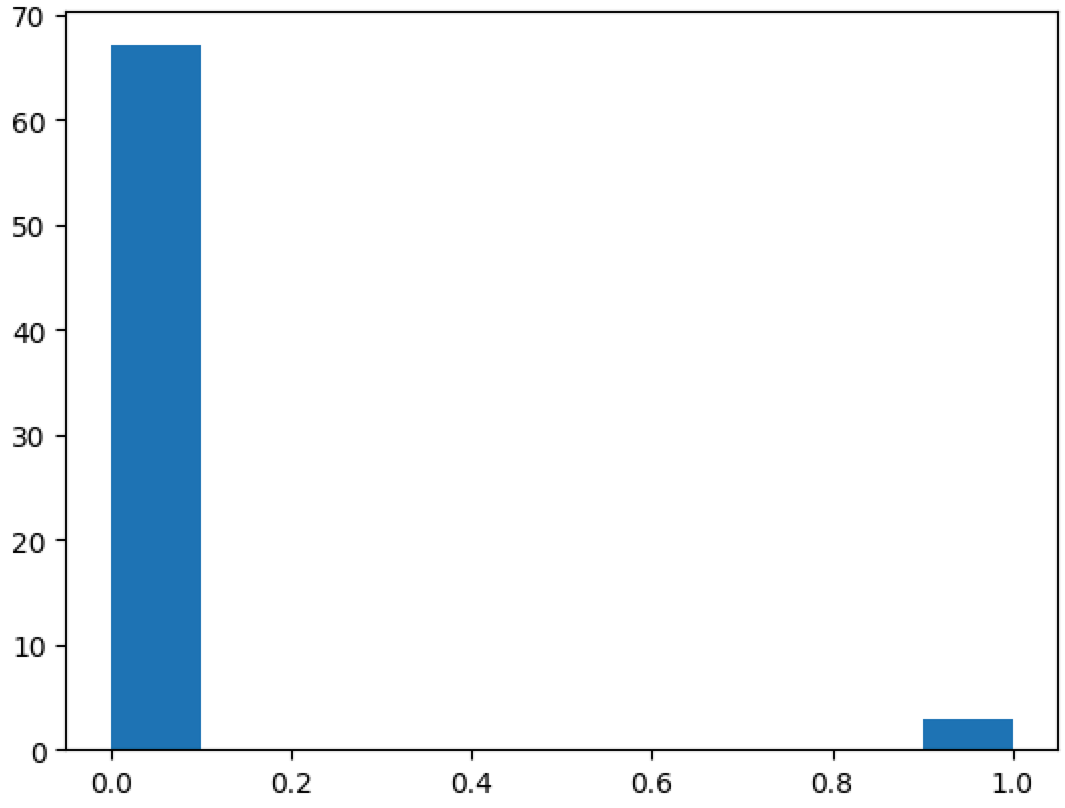
```
#35 Potential Matches
```

```
# Let's use pandas to calculate the simple Pearson's correlation between
# the sentiment score in the data set and the count of good-related words.
x = pd.Series(data['Sentiment Score'])
y = pd.Series(matches)
x.corr(y)
```

```
np.float64(0.12068240259212179)
```

```
# Well, that's not a very strong result, but what if we used the word
# bad instead of the word good. Would that make any difference?
# (Use a separate code block for each task.)

# 4.11: Copy the loop from four blocks above and change from good to bad

lemmas = set([s.lemma_names()[0] for s in wn.synsets('bad')])

matches = [0] * data.shape[0] # Fill a list with 70 zeroes


for i, g in enumerate(data['Comments']): # Loop over all comments
  for l in lemmas:  # Loop over all lemmas of good
    if l in g.lower(): # If we get a match, increment the count
      matches[i] += 1

# 4.12: Create a histogram of matches
import matplotlib.pyplot as plt
plt.hist(matches)

# 4.13: Write a comment interpreting the histogram
#35 Bad or not correlated matches and 1 potential decent match

# 4.14: Calculate the correlation between Sentiment Score and matches
x = pd.Series(data['Sentiment Score'])
y = pd.Series(matches)
x.corr(y)
```

```
np.float64(-0.1877341356444923)
```



The approach we took above depends on the a priori assumption that the positive and negative meanings of words as determined by WordNet are the most appropriate way to score these comments. But what if we turned the puzzle around and let the comments "tell" us which words are the best? We can use a simple geometric technique called Linear Discriminant Analysis to accomplish this for us. Don't worry about the math for this: Just keep in mind that we are going to divide the dataset into two groups - a positive group and a negative group - and then find out to what degree each word functions differently in those two groups.

We are going to use a vectorizer to represent what each word does in each document. As a reminder, TF-IDF stands for "term frequency, inverse document frequency" which is an alternative to the raw count of each word in a document that gives more weight to rare words.

```python
# To perform the LDA calculation, we need a TF-IDF matrix.
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.tokenize.casual import casual_tokenize

# We're keeping it simple for now, no special preprocessing. For example,
# we're leaving in stop words.
tfidfingest = TfidfVectorizer(tokenizer=casual_tokenize)
type(tfidfingest) # We've created an instance of our vectorizer
```

```
sklearn.feature_extraction.text.TfidfVectorizer
def __init__(*, input='content', encoding='utf-8', decode_error='strict',
strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None,
analyzer='word', stop_words=None, token_pattern='(?u)\\b\\w\\w+\\b', ngram_range=(1,
1), max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False,
dtype=np.float64, norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)
:class:`~sklearn.feature_extraction.text.HashingVectorizer`, see
:ref:`sphx_glr_auto_examples_text_plot_document_clustering.py`.

Read more in the :ref:`User Guide <text_feature_extraction>`.

Parameters
----------
```

```
# Now we have the tool we need to create the TF-IDF matrix: tfidfingest can
# be used to "eat up" the text data from our dataset.
tfidfmatrix = tfidfingest.fit_transform(raw_documents=data['Comments']).toarray()

tfidfmatrix.shape, type(tfidfmatrix)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/feature_extraction/text.py:517: UserWarr
  warnings.warn(
((70, 1254), numpy.ndarray)
```

```
# The 1254 columns is saying that we have 1254 unique terms in our vectorized
# dataset. Usually we would want to work a lot harder to get this number down.

# Remember that an array in numpy is a matrix-like object containing
# values all of the same type.

# What's in our array?
tfidfmatrix[:10,:5] # Just the first ten rows and five columns
```

```
array([[0.11255423, 0.        , 0.053985  , 0.        , 0.        ],
       [0.        , 0.15665963, 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.22535615, 0.        , 0.0540444 , 0.        , 0.07541902],
       [0.0478254 , 0.        , 0.20644883, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.15721287, 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
```

```
# Typical sparse array: Mostly zeroes. The non-zero values are fractional,
# showing that this is TF-IDF and not an array of simple counts.

# Now let's subdivide our original dataset based on whether each row was classed
# as a positive or negative comment.

# Here we are creating what Python programmers sometimes call a "mask."
# The first mask will let us select the positive cases. The second mask will
# let us select the negative cases.

classlist = [(c=="Positive" or c=="Neutral") for c in data['Sentiment Label']] # Positiv
notclasslist = [not c for c in classlist] # Negative mask

print(classlist[:8]) # Compare the first 8 results from the two masks
print(notclasslist[:8]) # Entries in these two lists should be boolean inverses
```

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

```
# We can use the masks to select the cases we want and the mean method
# to summarize our tfidf for each subset of the data

# Centroid means "a point in space that is in the middle of a bunch of other points"
poscentroid = tfidfmatrix[classlist].mean(axis=0)
negcentroid = tfidfmatrix[notclasslist].mean(axis=0)

separator = poscentroid - negcentroid # The difference in means for each word

separator
```

```
array([ 0.05894595, -0.11399296,  0.01746025, ...,  0.00252529,
        0.00154828,  0.01642816])
```

We just did quite a bit of math in a couple easy steps. When we computed each of the two centroids by taking the mean down the column, we summarized the "activity" of each word across all of the comments in each of the two subsets of the data. So for example, if "good" is important in positive reviews, the mean of its TF-IDF values across all the positive reviews should be high.

The final step in the previous code block is really interesting. By subtracting the two centroids, we get a representation of whether a word does something *different* in the two datasets. We call it "separator" because it represents a kind of dividing structure between the positive and negative data subsets.

```
# Now let's take our separator and put it together in a little
# dataset with the list of tokens, so we can see what's going on.
wordsent = pd.DataFrame({
    "Token":list(tfidfingest.get_feature_names_out()),
    "Sentiment":list(separator)
})

# Here we sort the cases from lowest to highest, based on the
# value that we took from separator.
wordsent.sort_values(by=['Sentiment'], inplace=True)

wordsent.shape, list(wordsent) # Summarize our data frame
```

```
((1254, 2), ['Token', 'Sentiment'])
```

```
# After sorting, the "most negative" words are at the beginning of
# the data frame and the "most positive" words are at the end. Let's
# Take a look:

wordsent.head(15) # Give us 15 words that are most negative
```

	Token	Sentiment
1	"	-0.113993
1076	the	-0.050822
1214	which	-0.048907
200	but	-0.047988
948	she	-0.044887
436	fish	-0.044311
87	alright	-0.043009
78	agedashi	-0.043009
1104	tofu	-0.043009
425	few	-0.042864
762	ordered	-0.042253
1168	visit	-0.038870
684	most	-0.038832
1206	were	-0.036540
1219	will	-0.036013

```
wordsent.tail(15) # Give us 15 words that are most positive
```

	Token	Sentiment
161	best	0.021562
1074	that	0.021747
312	delicious	0.021767
806	place	0.022676
252	come	0.022697
209	can	0.024593
1101	to	0.024680
761	order	0.025505
1033	sushi	0.027182
491	great	0.027710
455	fresh	0.028815
510	have	0.030804
111	are	0.036204
100	and	0.039881
0	!	0.058946

Well, not everything in those lists makes sense, but remember that this is a really small dataset and we did not make the effort to filter out stop words or punctuation. Still, there's some interesting stuff in there, and we've shown how with a few simple steps, we can create a kind of "empirical" sentiment value for each word.

The next step also uses a cool matrix math trick. By creating *dot products* of the separator vector with the TF-IDF vector for each of our 70 comments, we can summarize the similarity/distance of a given comment from the dividing structure. Maybe the resulting document scores will be useful for something.

```
docusent = tfidfmatrix.dot(separator)
docusent
```

```
array([ 0.03190957,  0.00489625,  0.02053524, -0.1131945 ,  0.0068361 ,
        0.03714363,  0.05276984, -0.1372704 ,  0.01527414,  0.04051255,
       -0.13587406,  0.03995574,  0.00276345, -0.00473419,  0.03760517,
        0.02651981,  0.06201844,  0.00034177, -0.01263898,  0.03052205,
        0.02293302,  0.0197362 , -0.13294862,  0.02436511,  0.03068126,
        0.01019862,  0.00424415,  0.01116213,  0.04171756,  0.03642798,
        0.03830759,  0.0108454 ,  0.04254235,  0.00893868,  0.03582491,
        0.03351333,  0.03417034,  0.03088434, -0.00089552,  0.04566911,
        0.04567206,  0.05179615,  0.03851501,  0.04892801,  0.03642214,
        0.02451584,  0.03657144,  0.03245421, -0.12116866,  0.01874153,
        0.02458296, -0.00526663,  0.04728941,  0.0562507 ,  0.03587462,
        0.04009349, -0.12562397,  0.0534958 ,  0.02060276,  0.02848552,
        0.03458464,  0.04533432,  0.04616221,  0.01689287,  0.02849751,
        0.0065034 ,  0.02493422, -0.01567791,  0.05086638,  0.02925724])
```

```
x = pd.Series(data['Sentiment Score']) # Here's the sentiment score from the data
y = pd.Series(docusent) # Here's our calculated separation of each document from the div
x.corr(y).round(3)
```

```
np.float64(0.618)
```

That's a way better result than what we got by counting WordNet synonyms. We shouldn't get too excited though, because we did not hold out a test set to verify our results, so it is likely that these results are peculiar to this sample of texts.

The important thing to note is this: If we have a training criterion (in this case we used Sentiment Label), we can easily process our text in a way that shows us which words are associated with positive sentiment and which words are associated with negative sentiment. Of course sentiment is only one small aspect of semantics as a whole: A word like "fresh" can be strongly positive while still containing many other shades of meaning.

We used a simple mathematical method to tease out this thin slice of semantics in a somewhat primitive way. Because of the rapid evolution of deep learning techniques and their proven success in so many NLP tasks, many researchers believe that the age of maintaining large, manually-created dictionaries is over.

To close this section and get a little practice, let's go back to the vectorization process, take out stop words, and then reproduce the analysis demonstrated above:

```
#
# 4.15: Return to the call to TfidfVectorizer above, and add this argument:
#
# stop_words="english"
#
# Then rerun all the code up to here. Add a comment saying whether the
# correlation value from the last step above changed. If the correlation
# got larger, add a comment saying why you think that happened.
#

tfidfingest = TfidfVectorizer(tokenizer=casual_tokenize,stop_words="english")
type(tfidfingest)
```

```
sklearn.feature_extraction.text.TfidfVectorizer
def __init__(*, input='content', encoding='utf-8', decode_error='strict',
strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None,
analyzer='word', stop_words=None, token_pattern='(?u)\\b\\w\\w+\\b', ngram_range=(1,
1), max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False,
dtype=np.float64, norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)
```

```
Convert a collection of raw documents to a matrix of TF-IDF features.

Equivalent to :class:`CountVectorizer` followed by
:class:`TfidfTransformer`.

For an example of usage, see
```

```
tfidfmatrix = tfidfingest.fit_transform(raw_documents=data['Comments']).toarray()

tfidfmatrix.shape, type(tfidfmatrix)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/feature_extraction/text.py:517: UserWarr
  warnings.warn(
((70, 1067), numpy.ndarray)
```

```
tfidfmatrix[:10,:5]
```

```
array([[0.13499958, 0.        , 0.06475059, 0.        , 0.        ],
       [0.        , 0.1939126 , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.28272417, 0.        , 0.06780227, 0.        , 0.09461814],
       [0.06031177, 0.        , 0.26034898, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.18882644, 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ]])
```

```
classlist = [(c=="Positive" or c=="Neutral") for c in data['Sentiment Label']]
notclasslist = [not c for c in classlist]

print(classlist[:8])
print(notclasslist[:8])
```

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

```
poscentroid = tfidfmatrix[classlist].mean(axis=0)
negcentroid = tfidfmatrix[notclasslist].mean(axis=0)

separator = poscentroid - negcentroid
```

```python

separator
```

```
array([ 0.07135942, -0.11761953,  0.02174546, ...,  0.00275552,
        0.00198711,  0.01787102])
```

```python
wordsent = pd.DataFrame({
    "Token":list(tfidfingest.get_feature_names_out()),
    "Sentiment":list(separator)
})

# Here we sort the cases from lowest to highest, based on the
# value that we took from separator.
wordsent.sort_values(by=['Sentiment'], inplace=True)

wordsent.shape, list(wordsent) # Summarize our data frame
```

```
((1067, 2), ['Token', 'Sentiment'])
```

```python
wordsent.head(15)
```

	Token	Sentiment
1	"	-0.117620
382	fish	-0.058388
652	ordered	-0.049955
79	alright	-0.049836
74	agedashi	-0.049836
950	tofu	-0.049836
564	mackeral	-0.049507
990	unremarkable	-0.049507
1003	visit	-0.048828
5	'	-0.045314
916	taste	-0.044826
710	presliced	-0.044514
307	dont	-0.044514
481	included	-0.044514
1061	yesterday	-0.043310

Next steps: Generate code with wordsent View recommended plots New interactive sheet

```
wordsent.tail(15)
```

	Token	Sentiment
129	belt	0.023250
324	eat	0.023367
232	conveyor	0.023456
403	friendly	0.023814
495	japanese	0.025454
131	best	0.026060
273	delicious	0.026678
944	time	0.026699
687	place	0.027536
214	come	0.028791
651	order	0.032082
427	great	0.032324
397	fresh	0.034100
896	sushi	0.034815
0	!	0.071359

```
docusent = tfidfmatrix.dot(separator)
docusent
```

```
array([ 0.03965399, -0.01647032,  0.03401018, -0.11942917,  0.0167424 ,
        0.03566194,  0.03876762, -0.12955748,  0.02005237,  0.03460744,
       -0.13908266,  0.03676213,  0.02382777,  0.00851573,  0.04307965,
        0.03997313,  0.04848422,  0.01773535,  0.00732781,  0.02793696,
        0.03630905,  0.00333785, -0.14868937,  0.02184578,  0.00531369,
        0.00873881,  0.03682533,  0.00445387,  0.04677694,  0.04654283,
        0.03278815,  0.02138001,  0.04392893,  0.01494492,  0.03830548,
        0.02465944,  0.03305323,  0.05364975,  0.00179854,  0.04755566,
        0.03654433,  0.04369266,  0.0227806 ,  0.05607645,  0.04500973,
        0.01645125,  0.03249932,  0.03870749, -0.12453174,  0.02264018,
        0.02467409,  0.02842332,  0.0476348 ,  0.04559384,  0.03061465,
        0.04451069, -0.14738249,  0.04355967,  0.03178673,  0.02514258,
        0.03025625,  0.0377525 ,  0.05321739,  0.00991499,  0.03378639,
        0.01514721,  0.02618785, -0.0021797 ,  0.03722392,  0.00455033])
```

```
x = pd.Series(data['Sentiment Score']) # Here's the sentiment score from the data
y = pd.Series(docusent) # Here's our calculated separation of each document from the div
```

```
x.corr(y).round(3)
```

```
np.float64(0.663)
```

```
#0.618 to 0.663 - Slightly better - StopWords removed some of the Good/Bad Matches
```

Part 3: Latent Dirichlet Allocation

What have we done so far? We used WordNet and found that large, expert teams of researchers can create a tree structure for the meanings of words. We can find word similarity, synonyms, antonyms, hypernyms, and hyponyms from this structure. We also found that given a training criterion and a bunch of labeled data, we can tease out some aspects of the meanings of words using data analysis rather than a manual dictionary-building process. Now let's synthesize some of that thinking with a new technique that groups words together and uses those groupings to create compact representations of documents.

This new technique is known as Latent Dirichlet Allocation, and its application to NLP was pioneered by David Blei and Andrew Ng. The name sounds complicated, but Latent is the idea that there are unobservable "topics" that can be used to represent documents, Dirichlet refers to a family of statistical distributions that are good for representing probabilities, and Allocation just means that we are going to use the statistical distributions to allocate words to topics and topics to documents.

We'll start again with a vectorizer, this time just a simple count of words:

```
# Begin by creating a new vectorizer instance
from sklearn.feature_extraction.text import CountVectorizer

# Create an instance of the count vectorizer. Do some filtering:
# A word has to appear in at least two documents
# A word can't appear in more than 90% of the documents
# Take out stop words
counter = CountVectorizer(min_df=2, max_df=0.9,
                          tokenizer=casual_tokenize, stop_words='english')

# Use the comments from the dataset to generate a sparse matrix of word counts
countmatrix = counter.fit_transform(raw_documents=data['Comments']).toarray()

countmatrix.shape, type(countmatrix) # Rows? Columns? Type?
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/feature_extraction/text.py:517: UserWarr
  warnings.warn(
((70, 408), numpy.ndarray)
```

```python
# When we want to do calculations or transformations on a sparse matrix, we
# usually need to convert it to a data frame first.
countdf = pd.DataFrame(countmatrix, columns=counter.get_feature_names_out())

countdf.shape, type(countdf) # Rows? Columns? Type?
```

```
((70, 408), pandas.core.frame.DataFrame)
```

```python
# Take a look at the first few rows of the data frame using the head() method.
# There should be no fractional values, showing that it is a count matrix
# rather than tf-idf.

#
# 4.16 - Show the first few rows of the data frame.
#
countdf.head(10)
```

	!	"	$	%	&	'	(	)	,	-	...	white	won't	work	worst	worth	wouldn't	wrong	y
0	3	0	1	0	0	0	4	4	4	4	...	0	0	0	0	0	0	1	
1	0	2	0	0	0	1	1	1	2	0	...	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	
3	0	0	0	0	0	0	0	0	2	0	...	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	4	1	...	0	0	0	0	0	0	0	
5	6	0	1	0	1	1	3	3	3	0	...	0	0	0	0	1	0	0	
6	1	0	3	0	0	0	1	1	3	0	...	0	0	0	0	0	0	0	
7	0	0	0	0	0	0	1	1	3	0	...	0	0	0	0	0	0	0	
8	4	0	0	0	0	0	1	1	9	0	...	1	0	0	0	0	0	0	
9	0	0	0	0	0	0	0	0	7	0	...	0	0	0	0	0	0	0	

10 rows × 408 columns

We've vectorized our documents with word counts, creating a document term matrix. Next, we will use Sci-Kit Learn to fit a Latent Dirichlet Allocation model. This is a stochastic process, because Bayesian estimation methods are used to overcome serious complexities in the math. In order to demonstrate a result that is consistent across runs, we set a random number seed here.

We also must set a specific number of topics that we want to model. There are empirical guides for deciding how many topics are needed to model a particular dataset. For now, we will use a relatively small number and just consider it to be an acceptable guess.

When we fit our model, we will first obtain a word-topic matrix. This shows the strength of connection between each term in our document-term matrix and each of the topics. In the Sci-Kit Learn implementation, this output is formatted as "pseudocounts," i.e., the number of times that the term could be expected to show up for that topic (in light of the model's prediction). You may encounter other software for doing LDiA that shows term-topic probabilities, in which case all values will show as <= 1.0.

```
# Sci-Kit learn has a Latent Dirichlet Allocation fitting algorithm that we
# can use.
from sklearn.decomposition import LatentDirichletAllocation as LDIA
from numpy import random as rnd
rnd.seed(123)

numtopics = 9 # Later on you will change this as an exercise.

# Create an instance of the LDIA analyzer
ldiamodel = LDIA(n_components=numtopics, learning_method='batch')

# Provide the pandas data frame of word counts for it to work on
ldiamodel = ldiamodel.fit(countdf)

ldiamodel.components_.shape # What is the resulting data frame like?
```

```
(9, 408)
```

```
# Each row repesents one topic, and each entry in a row
# pertains to one of the terms in our count matrix. Let's examine
# just the first five entries in each row. Remember that these are
# pseudocounts and not probability values.
ldiamodel.components_[:numtopics,:5]
```

```
array([[ 1.11107766,  4.11114844,  1.11109594,  0.11111111,  0.11111111],
       [ 0.11125124,  0.11111111,  0.97980161,  0.11111111,  1.11112435],
       [ 0.11111899,  2.11111287,  2.11111606,  0.11111111,  0.11111111],
       [ 9.89858297,  0.11111111,  1.60645729,  0.11111111,  0.87929276],
       [ 0.11111112,  0.11111112,  0.11111112,  0.11111112,  0.11111112],
       [ 7.64300236, 20.11106664,  7.44997875,  1.11111118,  0.11111111],
       [42.79141612,  0.11111648,  2.40814966,  1.11111101,  1.34291621],
       [ 3.11128055,  0.11111112,  0.11111112,  0.11111112,  0.11111112],
       [ 2.11115898,  0.11111111,  2.11117846,  0.11111112,  0.11111111]])
```

```
# Here's a function for showing the top ten words for each topic. This was
# adapted from an example in the SciKit Learn documentation.

# The model is the fitted model, the vectorizer is the initalized instance
# of the count vectorizer, and top_words sets how many of the most
# influential words you want to see.
def selected_topics(model, vectorizer, top_words=10):
    for idx, topic in enumerate(model.components_):
        print("Topic %d:" % (idx)) # One of these headers for each topic

        # This uses a list comprehension to iterate over the words
        # in each topic, picking out the highest coefficient values.
        print([(vectorizer.get_feature_names_out()[i], topic[i])
                        for i in topic.argsort()[:-top_words - 1:-1]])
```

```
# Here we call our function
selected_topics(ldiamodel, counter, top_words=10)
```

```
Topic 0:
[(',', np.float64(15.111341076976649)), ('sushi', np.float64(12.111324065910685)), ("it'
Topic 1:
[('sushi', np.float64(7.725661982034617)), ('order', np.float64(6.080713573853641)), (',
Topic 2:
[('sushi', np.float64(9.76940533760243)), (',', np.float64(6.328024454704488)), ('fresh'
Topic 3:
[(',', np.float64(23.011727108471494)), ('sushi', np.float64(14.669325698676126)), ('!',
Topic 4:
[('beef', np.float64(0.11111114754318115)), ('restaurants', np.float64(0.1111111376599 5
Topic 5:
[(',', np.float64(116.21548303789476)), ('sushi', np.float64(39.51141284908605)), ('"',
Topic 6:
[('!', np.float64(42.791416124688794)), (',', np.float64(42.0269537617841)), ('sushi', r
Topic 7:
[('!', np.float64(3.111280548726866)), ('tai', np.float64(1.1112210558920412)), ('love',
Topic 8:
[(',', np.float64(7.111408635367725)), ('rolls', np.float64(3.1112080974970975)), ('sush
```

Take a close look at the output shown above. If two words in a topic both have high values, it means that they co-occur quite frequently in documents where that topic predominates. If you look closely (and ignore the punctuation), you'll see that there are natural semantic connections among words in a topic. For example, even if you did not know that sushi was a kind of food, you can see the sushi-food connection in multiple topics.

If you think about it, our linear discriminant analysis (previous section; using TF-IDF) gave us a very general view of sentiment, simply by comparing the importance of words in two subsets of data. Here we have created a much fuller account of what words occur together in order to build a model that divides words up into semantic groupings. As with other semantic techniques we will encounter, this one relies of the "distributional hypothesis" - the idea that words with connected meanings tend to occur together.

The connections between terms and topics, as shown in the previous code block, it only half the picture, though. We also need the connections between topics and documents. SciKit learn can produce this for us using the transform method applied to our original sparse matrix (which is organized with one document/comment in each row). We should end up with the same number of rows as we originally had comments and the same number of columns as we have topics.

```
# Use transform() to get the topic vectors. We should have one
# vector for each document in our sushi data.
topicvectors = ldiamodel.transform(X=countmatrix)
topicvectors.shape, type(topicvectors)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X
  warnings.warn(
((70, 9), numpy.ndarray)
```

```
# Let's see the coefficients for the first 10 rows in our sushi data
print(topicvectors[0:10,:].round(4))
```

```
[[0.0015 0.8206 0.0015 0.0015 0.0015 0.0015 0.1691 0.0015 0.0015]
 [0.0022 0.0022 0.0022 0.0022 0.0022 0.6663 0.3185 0.0022 0.0022]
 [0.0093 0.0093 0.0093 0.0093 0.0093 0.0093 0.9259 0.0093 0.0093]
 [0.0026 0.0026 0.0026 0.9793 0.0026 0.0026 0.0026 0.0026 0.0026]
 [0.0021 0.0021 0.0021 0.0021 0.0021 0.9832 0.0021 0.0021 0.0021]
 [0.0016 0.0016 0.0016 0.0016 0.0016 0.0016 0.9871 0.0016 0.0016]
 [0.0018 0.0018 0.0018 0.2545 0.0018 0.7327 0.0018 0.0018 0.0018]
 [0.0058 0.0058 0.0059 0.0059 0.0058 0.9532 0.0059 0.0058 0.0059]
 [0.0018 0.0018 0.0018 0.0018 0.0018 0.9859 0.0018 0.0018 0.0018]
 [0.0015 0.0015 0.0015 0.0015 0.0015 0.9878 0.0015 0.0015 0.0015]]
```

Make sure to scan each row in the output above. Generally, each document will load most heavily on just one of the modeled topics. So in each row there should be a lot of very small numbers and one larger number in the range of 0.80 up to 0.99. If you see a row that has two or more "mid-sized" numbers, it means that the document in question could not be easily "assigned" to just one topic.

```
# Each of those rows is a probability vector and therefore each
# row should sum to 1 (within rounding error). As probabilities they
# represent the strength of "attachment" for each of our topics to each
# of our documents. Just as a diagnostic, let's check this assumption.
print( [ round(tv.sum(),1) for tv in topicvectors] )
```

```
[np.float64(1.0), np.float64(1.0), np.float64(1.0), np.float64(1.0), np.float64(1.0), np
```

Some of our topic coefficients are likely to be predictive of sentiment. This would indicate that a particular grouping of words (e.g., good, sushi, friendly, place, decent, service) connects strongly with the semantic idea of positivity or negativity. We will use a simple predictive technique, ordinary least squares regression, to probe whether a particular topic vector might be useful in predicting sentiment.

```python
# We will use OLS regression from sklearn
from sklearn import linear_model # Use regular OLS regression

reg = linear_model.LinearRegression() # Create an instance of the class

# Train the instance: Note that you can only use n-1 of the topic probabilities.
# Because each row sums to one, there is redundant information if you try to
# include all of the topic vectors and you will likely end up with a
# multicollinear model. This slicer, 0:numtopics, cuts off the last topic.
reg.fit(topicvectors[:,0:numtopics], data['Sentiment Score'])

# Calculate predicted values based on this regression model so we can get
# a zero order correlation as we did above.
sentpred = reg.predict(topicvectors[:,0:numtopics])

x = pd.Series(data['Sentiment Score'])
y = pd.Series(sentpred)
x.corr(y)
```

```
np.float64(0.5683062551922705)
```

How does that value, just above, compare with our linear discriminant analysis? Because we have used Latent Dirichlet Allocation as a kind of unsupervised data reduction technique (taking hundreds of columns from our original document term matrix and squashing down to just a few topic vectors), we are much less likely to be overfitting. So if our overall correlation bewteen actual and predicted values is a bit lower than what we obtained from the discriminant analysis, we probably achieved a boost in generalizability insofar as this topic model could be used effectively on novel documents.

Let's examine the regression coefficients: Note the largest (absolute value) of these and figure out which topic that coefficient belongs to. For example, counting from 0, if coefficient 4 was equal to 0.437 and that was the largest value, go back a few cells and look at the top words for Topic 4.

```python
reg.coef_
```

```
array([-0.08680103, -0.0502043 , -0.03314594,  0.00835262,  0.43663825,
       -0.22528869,  0.0213544 ,  0.01771591, -0.08862122])
```

```python
# Now add some additional diagnostic output. Use an enumerator to find the
# documents where Topic 4 represented a probability value larger than 0.90.
# Print out the comment corresponding to that document:


for idx, tv in enumerate(topicvectors):
  # 4.16: Eliminate this print statement

  # 4.17: Include an if statement in this for loop to test the probability
  # value for Topic 4 in this particular row
   if tv[4] > 0.90:


  # 4.18: Inside the if statement, use idx as the row slicer to get the
  # 'Comment' field from the original data set. Print that comment.
    print(data['Comments'][idx])
```

```
# Experimentation with the number of topics.

# 4.19: Go back to where numtopics is set and change to the next highest
# integer. Rerun the code from there down to here. Interpret the results.
# Is this new number of topics a better choice? Why or why not?


```

```

from sklearn.decomposition import LatentDirichletAllocation as LDIA
from numpy import random as rnd
rnd.seed(123)

numtopics = 10

ldiamodel = LDIA(n_components=numtopics, learning_method='batch')


ldiamodel = ldiamodel.fit(countdf)

ldiamodel.components_.shape
```

```
(10, 408)
```

```
ldiamodel.components_[:numtopics,:5]
```

```
array([[ 1.09998423,  4.10001713,  1.09999314,  0.1       ,  0.1       ],
       [ 4.10000541,  0.1       ,  1.1000006 ,  0.1       ,  0.1       ],
       [ 2.09993542,  0.1       ,  2.0999981 ,  0.1       ,  0.1       ],
       [10.76924887,  0.1       ,  3.01923988,  0.1       ,  2.10000472],
       [ 0.1       ,  0.1       ,  0.1       ,  0.1       ,  0.1       ],
       [ 9.16876328, 16.09999185,  6.19757346,  1.09999995,  0.10006249],
       [27.36188138,  6.09999101,  2.08316814,  1.10000004,  1.09993278],
       [10.10015341,  0.1       ,  0.1       ,  0.1       ,  0.1       ],
       [ 2.10002799,  0.1       ,  2.10002667,  0.1       ,  0.1       ],
       [ 0.1       ,  0.1       ,  0.1       ,  0.1       ,  0.1       ]])
```

```
def selected_topics(model, vectorizer, top_words=10):
    for idx, topic in enumerate(model.components_):
        print("Topic %d:" % (idx)) # One of these headers for each topic

        # This uses a list comprehension to iterate over the words
        # in each topic, picking out the highest coefficient values.
        print([(vectorizer.get_feature_names_out()[i], topic[i])
                        for i in topic.argsort()[:-top_words - 1:-1]])
```

```
selected_topics(ldiamodel, counter, top_words=10)
```

```
Topic 0:
[(',', np.float64(15.100085428639991)), ('sushi', np.float64(12.100074494139163)), ("it'
Topic 1:
[('sushi', np.float64(12.10705386512421)), (',', np.float64(8.396179578076055)), ('roll'
Topic 2:
[('sushi', np.float64(10.099988171622941)), (',', np.float64(7.099909710918744)), ('lunc
Topic 3:
[(',', np.float64(22.428594433987776)), ('sushi', np.float64(17.663120830281386)), ('foc
Topic 4:
[('yummy', np.float64(0.10000000401296019)), ('restaurants', np.float64(0.10000000330279
Topic 5:
[(',', np.float64(104.42092971640608)), ('sushi', np.float64(36.81713226464581)), (')',
Topic 6:
[(',', np.float64(49.15422044380697)), ('!', np.float64(27.361881382235232)), ('sushi',
Topic 7:
[('!', np.float64(10.100153408152059)), ('took', np.float64(4.100048691800069)), ('busy'
Topic 8:
[(',', np.float64(7.1001354407748565)), ('rolls', np.float64(3.1000537430451103)), ('got
Topic 9:
[('yummy', np.float64(0.10000000401296019)), ('restaurants', np.float64(0.10000000330279
```

```
topicvectors = ldiamodel.transform(X=countmatrix)
topicvectors.shape, type(topicvectors)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X
  warnings.warn(
((70, 10), numpy.ndarray)
```

```
print(topicvectors[0:10,:].round(4))
```

```
[[0.0013 0.9882 0.0013 0.0013 0.0013 0.0013 0.0013 0.0013 0.0013 0.0013]
 [0.002  0.002  0.002  0.002  0.002  0.002  0.9824 0.002  0.002  0.002 ]
 [0.0083 0.0083 0.0083 0.925  0.0083 0.0083 0.0083 0.0083 0.0083 0.0083]
 [0.0023 0.0023 0.0023 0.9791 0.0023 0.0023 0.0023 0.0023 0.0023 0.0023]
 [0.0019 0.182  0.0019 0.0019 0.0019 0.8029 0.0019 0.0019 0.0019 0.0019]
 [0.0014 0.0014 0.0014 0.0014 0.0014 0.6547 0.3337 0.0014 0.0014 0.0014]
 [0.0016 0.0016 0.0016 0.5969 0.0016 0.3899 0.0016 0.0016 0.0016 0.0016]
 [0.0053 0.0053 0.0053 0.0053 0.0053 0.9526 0.0053 0.0053 0.0053 0.0053]
 [0.0016 0.0016 0.0016 0.0016 0.0016 0.9857 0.0016 0.0016 0.0016 0.0016]
 [0.0014 0.0014 0.0014 0.0014 0.0014 0.9877 0.0014 0.0014 0.0014 0.0014]]
```

```
print( [ round(tv.sum(),1) for tv in topicvectors] )
```

```
[np.float64(1.0), np.float64(1.0), np.float64(1.0), np.float64(1.0), np.float64(1.0), np
```

```
from sklearn import linear_model

reg = linear_model.LinearRegression()


reg.fit(topicvectors[:,0:numtopics], data['Sentiment Score'])


sentpred = reg.predict(topicvectors[:,0:numtopics])

x = pd.Series(data['Sentiment Score'])
y = pd.Series(sentpred)
x.corr(y)
```

```
np.float64(0.4434410053645366)
```

```
#0.5683062551922705 to 0.4434410053645366  -- Scores Dropped - The original Method was m
```

```
# Experimentation with the number of topics, take two.

# 4.20: Go back to where numtopics is set and change it to a small value
# such as three or five. Rerun the code from there down to here.
# Interpret the results.
# Is this new number of topics a better choice? Why or why not?
```

```

from sklearn.decomposition import LatentDirichletAllocation as LDIA
from numpy import random as rnd
rnd.seed(123)

numtopics = 2

ldiamodel = LDIA(n_components=numtopics, learning_method='batch')


ldiamodel = ldiamodel.fit(countdf)

ldiamodel.components_.shape
```

```
(2, 408)
```

```
ldiamodel.components_[:numtopics,:5]
```

```
array([[34.65957608, 22.53859278,  4.29849445,  1.50592742,  3.40572782],
       [32.34042392,  4.46140722, 13.70150555,  1.49407258,  0.59427218]])
```

```python
def selected_topics(model, vectorizer, top_words=10):
    for idx, topic in enumerate(model.components_):
        print("Topic %d:" % (idx)) # One of these headers for each topic

        # This uses a list comprehension to iterate over the words
        # in each topic, picking out the highest coefficient values.
        print([(vectorizer.get_feature_names_out()[i], topic[i])
                        for i in topic.argsort()[:-top_words - 1:-1]])
```

```python
selected_topics(ldiamodel, counter, top_words=10)
```

```
Topic 0:
[(',', np.float64(88.66581396041687)), ('sushi', np.float64(46.18444256001666)), ('!', r
Topic 1:
[(',', np.float64(127.33418603958245)), ('sushi', np.float64(66.81555743998267)), ('belt
```

```python
topicvectors = ldiamodel.transform(X=countmatrix)
topicvectors.shape, type(topicvectors)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X
  warnings.warn(
((70, 2), numpy.ndarray)
```

```python
print(topicvectors[0:10,:].round(4))
```

```
[[0.0084 0.9916]
 [0.9813 0.0187]
 [0.9391 0.0609]
 [0.3777 0.6223]
 [0.2932 0.7068]
 [0.457  0.543 ]
 [0.6361 0.3639]
 [0.03   0.97  ]
 [0.0114 0.9886]
 [0.0101 0.9899]]
```