Modern NLP in Python

Please Follow me on LinkedIn for More:

Steve Nouri
https://www.linkedin.com/in/stevenouri/

- Or -

What you can learn about food by analyzing a million Yelp reviews

Before we get started...

whois?

- Patrick Harrison
- Lead Data Scientist @ S&P Global Market Intelligence - ***we are hiring***
- University of Virginia — Systems Engineering
- patrick@skipgram.io / @skipgram

Join Charlottesville Data Science!

- On Meetup.com ... http://www.meetup.com/CharlottesvilleDataScience (http://www.meetup.com/CharlottesvilleDataScience)
- On Slack ... **https://cville.typeform.com/to/UEzMVh (https://cville.typeform.com/to/UEzMVh)**
  - *link invites you to join the Cville team on Slack. Join Cville, then join channel **#datascience**.*

Note: I presented this notebook as a tutorial during the PyData DC 2016 conference (http://pydata.org/dc2016/schedule/presentation/11/). To view the video of the presentation on YouTube, see here (https://www.youtube.com/watch?v=6zm9NC9uRkk).

Our Trail Map

This tutorial features an end-to-end data science & natural language processing pipeline, starting with **raw data** and running through **preparing**, **modeling**, **visualizing**, and **analyzing** the data. We'll touch on the following points:

1. A tour of the dataset
2. Introduction to text processing with spaCy
3. Automatic phrase modeling
4. Topic modeling with LDA
5. Visualizing topic models with pyLDAvis
6. Word vector models with word2vec
7. Visualizing word2vec with t-SNE

...and we might even learn a thing or two about Python along the way.

Let's get started!

The Yelp Dataset

The Yelp Dataset (https://www.yelp.com/dataset_challenge/) is a dataset published by the business review service Yelp (http://yelp.com) for academic research and educational purposes. I really like the Yelp dataset as a subject for machine learning and natural language processing demos, because it's big (but not so big that you need your own data center to process it), well-connected, and anyone can relate to it — it's largely about food, after all!

Note: If you'd like to execute this notebook interactively on your local machine, you'll need to download your own copy of the Yelp dataset. If you're reviewing a static copy of the notebook online, you can skip this step. Here's how to get the dataset:

1. Please visit the Yelp dataset webpage here (https://www.yelp.com/dataset_challenge/)
2. Click "Get the Data"
3. Please review, agree to, and respect Yelp's terms of use!
4. The dataset downloads as a compressed .tgz file; uncompress it
5. Place the uncompressed dataset files (*yelp_academic_dataset_business.json*, etc.) in a directory named *yelp_dataset_challenge_academic_dataset*
6. Place the *yelp_dataset_challenge_academic_dataset* within the *data* directory in the *Modern NLP in Python* project folder

That's it! You're ready to go.

The current iteration of the Yelp dataset (as of this demo) consists of the following data:

- **552K** users
- **77K** businesses
- **2.2M** user reviews

When focusing on restaurants alone, there are approximately **22K** restaurants with approximately **1M** user reviews written about them.

The data is provided in a handful of files in *.json* format. We'll be using the following files for our demo:

- **yelp_academic_dataset_business.json** — *the records for individual businesses*
- **yelp_academic_dataset_review.json** — *the records for reviews users wrote about businesses*

The files are text files (UTF-8) with one *json object* per line, each one corresponding to an individual data record. Let's take a look at a few examples.

In [1]:

```
import os
import codecs

data_directory = os.path.join('..', 'data',
                              'yelp_dataset_challenge_academic_dataset')

businesses_filepath = os.path.join(data_directory,
                                   'yelp_academic_dataset_business.json')

with codecs.open(businesses_filepath, encoding='utf_8') as f:
    first_business_record = f.readline()

print first_business_record
```

```
{"business_id": "vcNAWiLM4dR7D2nwwJ7nCA", "full_address": "4840 E Indian School R
d\nSte 101\nPhoenix, AZ 85018", "hours": {"Tuesday": {"close": "17:00", "open":
"08:00"}, "Friday": {"close": "17:00", "open": "08:00"}, "Monday": {"close": "17:
00", "open": "08:00"}, "Wednesday": {"close": "17:00", "open": "08:00"}, "Thursda
y": {"close": "17:00", "open": "08:00"}}, "open": true, "categories": ["Doctors",
"Health & Medical"], "city": "Phoenix", "review_count": 9, "name": "Eric Goldber
g, MD", "neighborhoods": [], "longitude": -111.98375799999999, "state": "AZ", "st
ars": 3.5, "latitude": 33.499313000000001, "attributes": {"By Appointment Only":
true}, "type": "business"}
```

The business records consist of *key, value* pairs containing information about the particular business. A few attributes we'll be interested in for this demo include:

- **business_id** — *unique identifier for businesses*
- **categories** — *an array containing relevant category values of businesses*

The *categories* attribute is of special interest. This demo will focus on restaurants, which are indicated by the presence of the *Restaurant* tag in the *categories* array. In addition, the *categories* array may contain more detailed information about restaurants, such as the type of food they serve.

The review records are stored in a similar manner — *key, value* pairs containing information about the reviews.

In [2]:
```
review_json_filepath = os.path.join(data_directory,
                                    'yelp_academic_dataset_review.json')

with codecs.open(review_json_filepath, encoding='utf_8') as f:
    first_review_record = f.readline()

print first_review_record
```

```
{"votes": {"funny": 0, "useful": 2, "cool": 1}, "user_id": "Xqd0DzHaiyRqVH3WRG7hz
g", "review_id": "15SdjuK7DmYqUAj6rjGowg", "stars": 5, "date": "2007-05-17", "tex
t": "dr. goldberg offers everything i look for in a general practitioner.  he's n
ice and easy to talk to without being patronizing; he's always on time in seeing
his patients; he's affiliated with a top-notch hospital (nyu) which my parents ha
ve explained to me is very important in case something happens and you need surge
ry; and you can get referrals to see specialists without having to see him first.
really, what more do you need?  i'm sitting here trying to think of any complaint
s i have about him, but i'm really drawing a blank.", "type": "review", "business
_id": "vcNAWiLM4dR7D2nwwJ7nCA"}
```

A few attributes of note on the review records:

- **business_id** — *indicates which business the review is about*
- **text** — *the natural language text the user wrote*

The *text* attribute will be our focus today!

json is a handy file format for data interchange, but it's typically not the most usable for any sort of modeling work. Let's do a bit more data preparation to get our data in a more usable format. Our next code block will do the following:

1. Read in each business record and convert it to a Python `dict`
2. Filter out business records that aren't about restaurants (i.e., not in the "Restaurant" category)
3. Create a `frozenset` of the business IDs for restaurants, which we'll use in the next step

In [3]:
```
import json

restaurant_ids = set()

# open the businesses file
with codecs.open(businesses_filepath, encoding='utf_8') as f:

    # iterate through each line (json record) in the file
    for business_json in f:
```

```
    for business_json in f:

        # convert the json record to a Python dict
        business = json.loads(business_json)

        # if this business is not a restaurant, skip to the next one
        if u'Restaurants' not in business[u'categories']:
            continue

        # add the restaurant business id to our restaurant_ids set
        restaurant_ids.add(business[u'business_id'])

# turn restaurant_ids into a frozenset, as we don't need to change it anymore
restaurant_ids = frozenset(restaurant_ids)

# print the number of unique restaurant ids in the dataset
print '{:,}'.format(len(restaurant_ids)), u'restaurants in the dataset.'
```

```
21,892 restaurants in the dataset.
```

Next, we will create a new file that contains only the text from reviews about restaurants, with one review per line in the file.

In [4]:

```
intermediate_directory = os.path.join('..', 'intermediate')

review_txt_filepath = os.path.join(intermediate_directory,
                                   'review_text_all.txt')
```

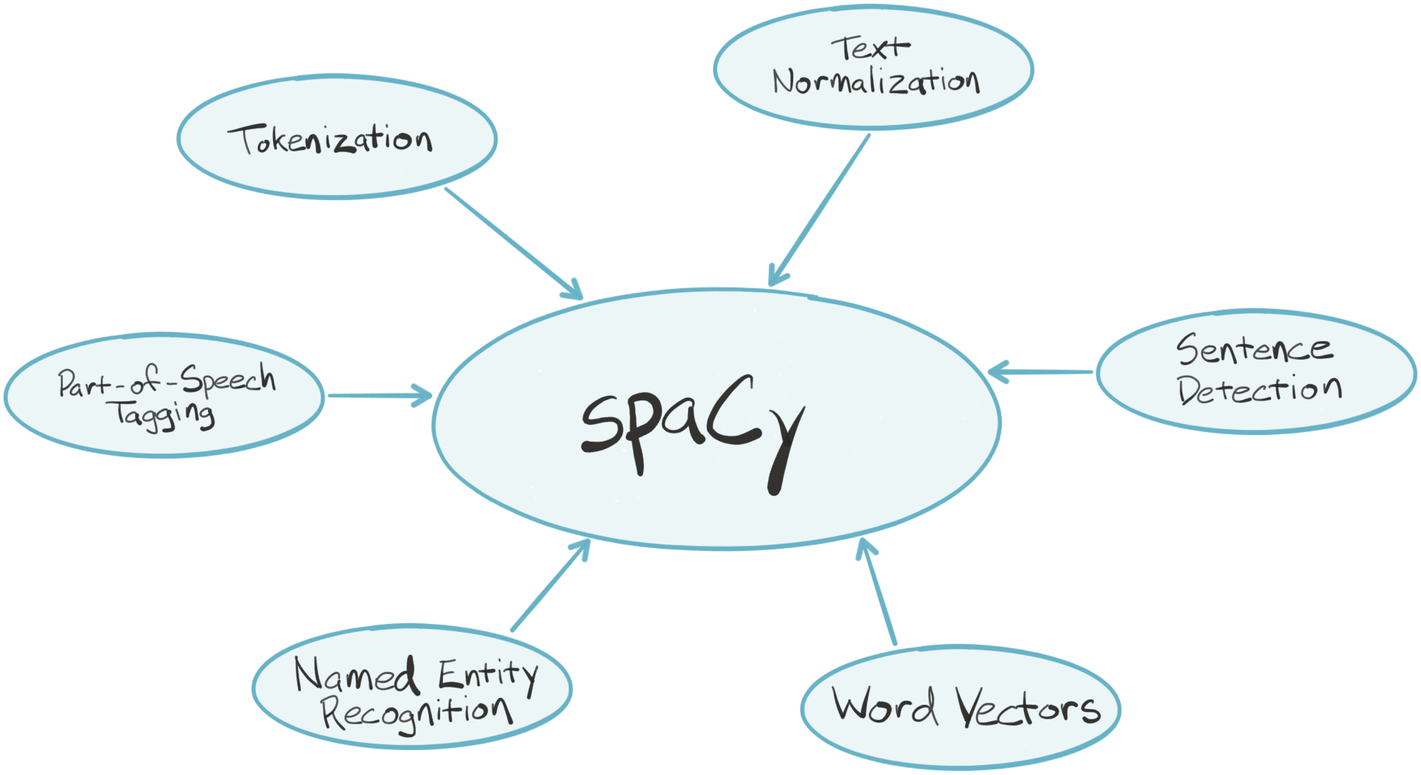
In [5]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute data prep yourself.
if 0 == 1:

    review_count = 0

    # create & open a new file in write mode
    with codecs.open(review_txt_filepath, 'w', encoding='utf_8') as review_txt_f
ile:

        # open the existing review json file
        with codecs.open(review_json_filepath, encoding='utf_8') as review_json_
file:

            # loop through all reviews in the existing file and convert to dict
            for review_json in review_json_file:
                review = json.loads(review_json)

                # if this review is not about a restaurant, skip to the next one
                if review[u'business_id'] not in restaurant_ids:
                    continue

                # write the restaurant review as a line in the new file
                # escape newline characters in the original review text
                review_txt_file.write(review[u'text'].replace('\n', '\\n') + '\n
')
                review_count += 1

    print u'''Text from {:,} restaurant reviews
              written to the new txt file.'''.format(review_count)
```

```
else:
    
    with codecs.open(review_txt_filepath, encoding='utf_8') as review_txt_file:
        for review_count, line in enumerate(review_txt_file):
            pass
        
    print u'Text from {:,} restaurant reviews in the txt file.'.format(review_co
unt + 1)
```

```
Text from 991,714 restaurant reviews in the txt file.
CPU times: user 26.7 s, sys: 1.21 s, total: 27.9 s
Wall time: 28.1 s
```

spaCy — Industrial-Strength NLP in Python



spaCy (https://spacy.io) is an industrial-strength natural language processing (*NLP*) library for Python. spaCy's goal is to take recent advancements in natural language processing out of research papers and put them in the hands of users to build production software.

spaCy handles many tasks commonly associated with building an end-to-end natural language processing pipeline:

- Tokenization
- Text normalization, such as lowercasing, stemming/lemmatization
- Part-of-speech tagging
- Syntactic dependency parsing
- Sentence boundary detection
- Named entity recognition and annotation

In the "batteries included" Python tradition, spaCy contains built-in data and models which you can use out-of-the-box for processing general-purpose English language text:

- Large English vocabulary, including stopword lists

- Token "probabilities"
- Word vectors

spaCy is written in optimized Cython, which means it's *fast*. According to a few independent sources, it's the fastest syntactic parser available in any language. Key pieces of the spaCy parsing pipeline are written in pure C, enabling efficient multithreading (i.e., spaCy can release the *GIL*).

In [6]:

```
import spacy
import pandas as pd
import itertools as it

nlp = spacy.load('en')
```

Let's grab a sample review to play with.

In [7]:

```
with codecs.open(review_txt_filepath, encoding='utf_8') as f:
    sample_review = list(it.islice(f, 8, 9))[0]
    sample_review = sample_review.replace('\\n', '\n')

print sample_review
```

```
After a morning of Thrift Store hunting, a friend and I were thinking of lunch, a
nd he suggested Emil's after he'd seen Chris Sebak do a bit on it and had tried i
t a time or two before, and I had not. He said they had a decent Reuben, but to b
e prepared to step back in time.

Well, seeing as how I'm kind of addicted to late 40's and early 50's, and the who
le Rat Pack scene, stepping back in time is a welcomed change in da burgh...as lo
ng as it doesn't involve 1979, which I can see all around me every day.

And yet another shot at finding a decent Reuben in da burgh...well, that's like h
unting the Holy Grail. So looking under one more bush certainly wouldn't hurt.

So off we go right at lunchtime in the middle of...where exactly were we? At firs
t I thought we were lost, driving around a handful of very rather dismal looking 
blocks in what looked like a neighborhood that had been blighted by the building 
of a highway. And then...AHA! Here it is! And yep, there it was. This little unas
suming building with an add-on entrance with what looked like a very old hand pai
nted sign stating quite simply 'Emil's.

We walked in the front door, and entered another world. Another time, and another
place. Oh, and any Big Burrito/Sousa foodies might as well stop reading now. I wo
uldn't want to see you walk in, roll your eyes and say 'Reaaaaaallly?'

This is about as old world bar/lounge/restaurant as it gets. Plain, with a dark w
ood bar on one side, plain white walls with no yinzer pics, good sturdy chairs an
d actual white linens on the tables. This is the kind of neighborhood dive that I
could see Frank and Dino pulling a few tables together for some poker, a fish sam
mich, and some cheap scotch. And THAT is exactly what I love.

Oh...but good food counts too.

We each had a Reuben, and my friend had a side of fries. The Reubens were decent,
but not NY awesome. A little too thick on the bread, but overall, tasty and defin
itely filling. Not too skimpy on the meat. I seriously CRAVE a true, good NY Reub
en, but since I can't afford to travel right now, what I find in da burgh will ha
ve to do. But as we sat and ate, burgers came out to an adjoining table. Those we
re some big thick burgers. A steak went past for the table behind us. That was HU
GE! And when we asked about it, the waitress said 'Yeah, it's huge and really goo
d, and he only charges $12.99 for it, ain't that nuts?' Another table of five cam
```

```
e in, and wham. Fish sandwiches PILED with breaded fish that looked amazing. Yea
h, I want that, that, that and THAT!

My friend also mentioned that they have a Chicken Parm special one day of the wee
k that is only served UNTIL 4 pm, and that it is fantastic. If only I could GET t
here on that week day before 4...

The waitress did a good job, especially since there was quite a growing crowd at
lunchtime on a Saturday, and only one of her. She kept up and was very friendly.

They only have Pepsi products, so I had a brewed iced tea, which was very fresh,
and she did pop by to ask about refills as often as she could. As the lunch hour
went on, they were getting busy.

Emil's is no frills, good portions, very reasonable prices, VERY comfortable neig
hborhood hole in the wall...kind of like Cheers, but in a blue collar neighborhoo
d in the 1950's. Fan-freakin-tastic! I could feel at home here.

You definitely want to hit Mapquest or plug in your GPS though. I am not sure tha
t I could find it again on my own...it really is a hidden gem. I will be making m
y friend take me back until I can memorize where the heck it is.

Addendum: 2nd visit for the fish sandwich. Excellent. Truly. A pound of fish on a
fish-shaped bun (as opposed to da burgh's seemingly popular hamburger bun). The f
ish was flavorful, the batter excellent, and for just $8. This may have been the
best fish sandwich I've yet to have in da burgh.
```

Hand the review text to spaCy, and be prepared to wait...

```
In [8]: %%time
        parsed_review = nlp(sample_review)
```

```
CPU times: user 222 ms, sys: 11.6 ms, total: 234 ms
Wall time: 251 ms
```

...1/20th of a second or so. Let's take a look at what we got during that time...

```
In [9]: print parsed_review
```

```
After a morning of Thrift Store hunting, a friend and I were thinking of lunch, a
nd he suggested Emil's after he'd seen Chris Sebak do a bit on it and had tried i
t a time or two before, and I had not. He said they had a decent Reuben, but to b
e prepared to step back in time.

Well, seeing as how I'm kind of addicted to late 40's and early 50's, and the who
le Rat Pack scene, stepping back in time is a welcomed change in da burgh...as lo
ng as it doesn't involve 1979, which I can see all around me every day.

And yet another shot at finding a decent Reuben in da burgh...well, that's like h
unting the Holy Grail. So looking under one more bush certainly wouldn't hurt.

So off we go right at lunchtime in the middle of...where exactly were we? At firs
t I thought we were lost, driving around a handful of very rather dismal looking
blocks in what looked like a neighborhood that had been blighted by the building
of a highway. And then...AHA! Here it is! And yep, there it was. This little unas
suming building with an add-on entrance with what looked like a very old hand pai
nted sign stating quite simply 'Emil's.

We walked in the front door, and entered another world. Another time, and another
```

```
place. Oh, and any Big Burrito/Sousa foodies might as well stop reading now. I wo
uldn't want to see you walk in, roll your eyes and say 'Reaaaaaallllly?'

This is about as old world bar/lounge/restaurant as it gets. Plain, with a dark w
ood bar on one side, plain white walls with no yinzer pics, good sturdy chairs an
d actual white linens on the tables. This is the kind of neighborhood dive that I
could see Frank and Dino pulling a few tables together for some poker, a fish sam
mich, and some cheap scotch. And THAT is exactly what I love.

Oh...but good food counts too.

We each had a Reuben, and my friend had a side of fries. The Reubens were decent,
but not NY awesome. A little too thick on the bread, but overall, tasty and defin
itely filling. Not too skimpy on the meat. I seriously CRAVE a true, good NY Reub
en, but since I can't afford to travel right now, what I find in da burgh will ha
ve to do. But as we sat and ate, burgers came out to an adjoining table. Those we
re some big thick burgers. A steak went past for the table behind us. That was HU
GE! And when we asked about it, the waitress said 'Yeah, it's huge and really goo
d, and he only charges $12.99 for it, ain't that nuts?' Another table of five cam
e in, and wham. Fish sandwiches PILED with breaded fish that looked amazing. Yea
h, I want that, that, that and THAT!

My friend also mentioned that they have a Chicken Parm special one day of the wee
k that is only served UNTIL 4 pm, and that it is fantastic. If only I could GET t
here on that week day before 4...

The waitress did a good job, especially since there was quite a growing crowd at
lunchtime on a Saturday, and only one of her. She kept up and was very friendly.

They only have Pepsi products, so I had a brewed iced tea, which was very fresh,
and she did pop by to ask about refills as often as she could. As the lunch hour
went on, they were getting busy.

Emil's is no frills, good portions, very reasonable prices, VERY comfortable neig
hborhood hole in the wall...kind of like Cheers, but in a blue collar neighborhoo
d in the 1950's. Fan-freakin-tastic! I could feel at home here.

You definitely want to hit Mapquest or plug in your GPS though. I am not sure tha
t I could find it again on my own...it really is a hidden gem. I will be making m
y friend take me back until I can memorize where the heck it is.

Addendum: 2nd visit for the fish sandwich. Excellent. Truly. A pound of fish on a
fish-shaped bun (as opposed to da burgh's seemingly popular hamburger bun). The f
ish was flavorful, the batter excellent, and for just $8. This may have been the
best fish sandwich I've yet to have in da burgh.
```

Looks the same! What happened under the hood?

What about sentence detection and segmentation?

In [10]:

```python
for num, sentence in enumerate(parsed_review.sents):
    print 'Sentence {}:'.format(num + 1)
    print sentence
    print ''
```

```
Sentence 1:
After a morning of Thrift Store hunting, a friend and I were thinking of lunch, a
nd he suggested Emil's after he'd seen Chris Sebak do a bit on it and had tried i
t a time or two before, and I had not.
```

Sentence 2:
He said they had a decent Reuben, but to be prepared to step back in time.

Sentence 3:
Well, seeing as how I'm kind of addicted to late 40's and early 50's, and the whole Rat Pack scene, stepping back in time is a welcomed change in da burgh...as long as it doesn't involve 1979, which I can see all around me every day.

Sentence 4:
And yet another shot at finding a decent Reuben in da burgh...

Sentence 5:
well, that's like hunting the Holy Grail.

Sentence 6:
So looking under one more bush certainly wouldn't hurt.

Sentence 7:
So off we go right at lunchtime in the middle of...where exactly were we?

Sentence 8:
At first I thought we were lost, driving around a handful of very rather dismal looking blocks in what looked like a neighborhood that had been blighted by the building of a highway.

Sentence 9:
And then...AHA!

Sentence 10:
Here it is!

Sentence 11:
And yep, there it was.

Sentence 12:
This little unassuming building with an add-on entrance with what looked like a very old hand painted sign stating quite simply 'Emil's.

Sentence 13:
We walked in the front door, and entered another world.

Sentence 14:
Another time, and another place.

Sentence 15:
Oh, and any Big Burrito/Sousa foodies might as well stop reading now.

Sentence 16:
I wouldn't want to see you walk in, roll your eyes and say 'Reaaaaaallly?'

Sentence 17:
This is about as old world bar/lounge/restaurant as it gets.

Sentence 18:
Plain, with a dark wood bar on one side, plain white walls with no yinzer pics, good sturdy chairs and actual white linens on the tables.

Sentence 19:
This is the kind of neighborhood dive that I could see Frank and Dino pulling a few tables together for some poker, a fish sammich, and some cheap scotch.

Sentence 20:
And THAT is exactly what I love.

Sentence 21:
Oh...but good food counts too.

Sentence 22:
We each had a Reuben, and my friend had a side of fries.

Sentence 23:
The Reubens were decent, but not NY awesome.

Sentence 24:
A little too thick on the bread, but overall, tasty and definitely filling.

Sentence 25:
Not too skimpy on the meat.

Sentence 26:
I seriously CRAVE a true, good NY Reuben, but since I can't afford to travel right now, what I find in da burgh will have to do.

Sentence 27:
But as we sat and ate, burgers came out to an adjoining table.

Sentence 28:
Those were some big thick burgers.

Sentence 29:
A steak went past for the table behind us.

Sentence 30:
That was HUGE!

Sentence 31:
And when we asked about it, the waitress said 'Yeah, it's huge and really good, and he only charges $12.99 for it, ain't that nuts?'

Sentence 32:
Another table of five came in, and wham.

Sentence 33:
Fish sandwiches PILED with breaded fish that looked amazing.

Sentence 34:
Yeah, I want that, that, that and THAT!

```
Sentence 35:
My friend also mentioned that they have a Chicken Parm special one day of the wee
k that is only served UNTIL 4 pm, and that it is fantastic.

Sentence 36:
If only I could GET there on that week day before 4...



Sentence 37:
The waitress did a good job, especially since there was quite a growing crowd at
lunchtime on a Saturday, and only one of her.

Sentence 38:
She kept up and was very friendly.



Sentence 39:
They only have Pepsi products, so I had a brewed iced tea, which was very fresh,
and she did pop by to ask about refills as often as she could.

Sentence 40:
As the lunch hour went on, they were getting busy.



Sentence 41:
Emil's is no frills, good portions, very reasonable prices, VERY comfortable neig
hborhood hole in the wall...

Sentence 42:
kind of like Cheers, but in a blue collar neighborhood in the 1950's.

Sentence 43:
Fan-freakin-tastic!

Sentence 44:
I could feel at home here.



Sentence 45:
You definitely want to hit Mapquest or plug in your GPS though.

Sentence 46:
I am not sure that I could find it again on my own...it really is a hidden gem.

Sentence 47:
I will be making my friend take me back until I can memorize where the heck it i
s.



Sentence 48:
Addendum: 2nd visit for the fish sandwich.

Sentence 49:
Excellent.

Sentence 50:
```

```
Truly.

Sentence 51:
A pound of fish on a fish-shaped bun (as opposed to da burgh's seemingly popular
hamburger bun).

Sentence 52:
The fish was flavorful, the batter excellent, and for just $8.

Sentence 53:
This may have been the best fish sandwich I've yet to have in da burgh.
```

What about named entity detection?

In [11]:

```
for num, entity in enumerate(parsed_review.ents):
    print 'Entity {}:'.format(num + 1), entity, '-', entity.label_
    print ''
```

```
Entity 1: Thrift Store - ORG

Entity 2: Emil - PERSON

Entity 3: Chris Sebak - PERSON

Entity 4: two - CARDINAL

Entity 5: Reuben - PERSON

Entity 6: Rat Pack - ORG

Entity 7: 1979 - DATE

Entity 8: every day - DATE

Entity 9: Reuben - PERSON

Entity 10: one - CARDINAL

Entity 11: Emil - PERSON

Entity 12: Frank - PERSON

Entity 13: Dino - PERSON

Entity 14: Reuben - PERSON

Entity 15: Reubens - PERSON

Entity 16: Reuben - PERSON

Entity 17: HUGE - ORG

Entity 18: 12.99 - MONEY

Entity 19: five - CARDINAL

Entity 20: one day - DATE

Entity 21: UNTIL - ORG
Entity 22: 4 pm - TIME

Entity 23: that week day - DATE

Entity 24: Saturday - DATE

Entity 25: only one - CARDINAL

Entity 26: Pepsi - ORG

Entity 27: the lunch hour - TIME

Entity 28: Emil - PERSON

Entity 29: 1950 - DATE

Entity 30: Mapquest - LOC

Entity 31: 2nd - CARDINAL

Entity 32: Truly - PERSON

Entity 33: 8 - MONEY
```

What about part of speech tagging?

In [12]:

```python
token_text = [token.orth_ for token in parsed_review]
token_pos = [token.pos_ for token in parsed_review]

pd.DataFrame(zip(token_text, token_pos),
             columns=['token_text', 'part_of_speech'])
```

Out[12]:

	token_text	part_of_speech
0	After	ADP
1	a	DET
2	morning	NOUN
3	of	ADP
4	Thrift	PROPN
5	Store	PROPN
6	hunting	NOUN
7	,	PUNCT
8	a	DET
9	friend	NOUN
10	and	CONJ
11	I	PRON
12	were	VERB
13	thinking	VERB
14	of	ADP

15	lunch	NOUN
16	,	PUNCT
17	and	CONJ
18	he	PRON
19	suggested	VERB
20	Emil	PROPN
21	's	PART
22	after	ADP
23	he	PRON
24	'd	VERB
25	seen	VERB
26	Chris	PROPN
27	Sebak	PROPN
28	do	VERB
29	a	DET
...	...	...
855	flavorful	ADJ
856	,	PUNCT
857	the	DET
858	batter	NOUN
859	excellent	ADJ
860	,	PUNCT
861	and	CONJ
862	for	ADP
863	just	ADV
864	$	SYM
865	8	NUM
866	.	PUNCT
867	This	DET
868	may	VERB
869	have	VERB
870	been	VERB
871	the	DET
872	best	ADJ
873	fish	NOUN
874	sandwich	NOUN

875	I	PRON
876	've	VERB
877	yet	ADV
878	to	PART
879	have	VERB
880	in	ADP
881	da	PROPN
882	burgh	NOUN
883	.	PUNCT
884	\n	SPACE

885 rows × 2 columns

What about text normalization, like stemming/lemmatization and shape analysis?

In [13]:

```
token_lemma = [token.lemma_ for token in parsed_review]
token_shape = [token.shape_ for token in parsed_review]

pd.DataFrame(zip(token_text, token_lemma, token_shape),
             columns=['token_text', 'token_lemma', 'token_shape'])
```

Out[13]:

	token_text	token_lemma	token_shape
0	After	after	Xxxxx
1	a	a	x
2	morning	morning	xxxx
3	of	of	xx
4	Thrift	thrift	Xxxxx
5	Store	store	Xxxxx
6	hunting	hunting	xxxx
7	,	,	,
8	a	a	x
9	friend	friend	xxxx
10	and	and	xxx
11	I	i	X
12	were	be	xxxx
13	thinking	think	xxxx
14	of	of	xx
15	lunch	lunch	xxxx
16	,	,	,
17	and	and	xxx

18	he	he	xx
19	suggested	suggest	xxxx
20	Emil	emil	Xxxx
21	's	's	'x
22	after	after	xxxx
23	he	he	xx
24	'd	would	'x
25	seen	see	xxxx
26	Chris	chris	Xxxxx
27	Sebak	sebak	Xxxxx
28	do	do	xx
29	a	a	x
...	...	...	...
855	flavorful	flavorful	xxxx
856	,	,	,
857	the	the	xxx
858	batter	batter	xxxx
859	excellent	excellent	xxxx
860	,	,	,
861	and	and	xxx
862	for	for	xxx
863	just	just	xxxx
864	$	< / td><td>	
865	8	8	d
866	.	.	.
867	This	this	Xxxx
868	may	may	xxx
869	have	have	xxxx
870	been	be	xxxx
871	the	the	xxx
872	best	best	xxxx
873	fish	fish	xxxx
874	sandwich	sandwich	xxxx
875	I	i	X
876	've	have	'xx
877	yet	yet	xxx

878	to	to	xx
879	have	have	xxxx
880	in	in	xx
881	da	da	xx
882	burgh	burgh	xxxx
883	.	.	.
884	\n	\n	\n

885 rows × 3 columns

What about token-level entity analysis?

In [14]:

```
token_entity_type = [token.ent_type_ for token in parsed_review]
token_entity_iob = [token.ent_iob_ for token in parsed_review]

pd.DataFrame(zip(token_text, token_entity_type, token_entity_iob),
             columns=['token_text', 'entity_type', 'inside_outside_begin'])
```

Out[14]:

	token_text	entity_type	inside_outside_begin
0	After		O
1	a		O
2	morning		O
3	of		O
4	Thrift	ORG	B
5	Store	ORG	I
6	hunting		O
7	,		O
8	a		O
9	friend		O
10	and		O
11	I		O
12	were		O
13	thinking		O
14	of		O
15	lunch		O
16	,		O
17	and		O
18	he		O
19	suggested		O
20	Emil	PERSON	B

20	Emil	PERSON	B
21	's		O
22	after		O
23	he		O
24	'd		O
25	seen		O
26	Chris	PERSON	B
27	Sebak	PERSON	I
28	do		O
29	a		O
...	...	...	...
855	flavorful		O
856	,		O
857	the		O
858	batter		O
859	excellent		O
860	,		O
861	and		O
862	for		O
863	just		O
864	$		O
865	8	MONEY	B
866	.		O
867	This		O
868	may		O
869	have		O
870	been		O
871	the		O
872	best		O
873	fish		O
874	sandwich		O
875	I		O
876	've		O
877	yet		O
878	to		O
879	have		O
880	in		O

881	da		O
882	burgh		O
883	.		O
884	\n		O

885 rows × 3 columns

What about a variety of other token-level attributes, such as the relative frequency of tokens, and whether or not a token matches any of these categories?

- stopword
- punctuation
- whitespace
- represents a number
- whether or not the token is included in spaCy's default vocabulary?

In [15]:

```
token_attributes = [(token.orth_,
                     token.prob,
                     token.is_stop,
                     token.is_punct,
                     token.is_space,
                     token.like_num,
                     token.is_oov)
                    for token in parsed_review]

df = pd.DataFrame(token_attributes,
                  columns=['text',
                           'log_probability',
                           'stop?',
                           'punctuation?',
                           'whitespace?',
                           'number?',
                           'out of vocab.?'])

df.loc[:, 'stop?':'out of vocab.?'] = (df.loc[:, 'stop?':'out of vocab.?']
                                       .applymap(lambda x: u'Yes' if x else u''
))

df
```

Out[15]:

	text	log_probability	stop?	punctuation?	whitespace?	number?	out of vocab.?
0	After	-9.091193	Yes				
1	a	-3.929788	Yes				
2	morning	-9.529314					
3	of	-4.275874	Yes				
4	Thrift	-14.550483					
5	Store	-11.719210					
6	hunting	-10.961483					
7		-3.454960		Yes			

7	,	-3.454960		Yes			
8	a	-3.929788	Yes				
9	friend	-8.210516					
10	and	-4.113108	Yes				
11	I	-3.791565	Yes				
12	were	-6.673175	Yes				
13	thinking	-8.442947					
14	of	-4.275874	Yes				
15	lunch	-10.572958					
16	,	-3.454960		Yes			
17	and	-4.113108	Yes				
18	he	-5.931905	Yes				
19	suggested	-10.656719					
20	Emil	-15.862375					
21	's	-4.830559					
22	after	-7.265652	Yes				
23	he	-5.931905	Yes				
24	'd	-7.075287					
25	seen	-7.973224					
26	Chris	-10.966099					
27	Sebak	-19.502029					Yes
28	do	-5.246997	Yes				
29	a	-3.929788	Yes				
...	...	...	...	...	...	...	...
855	flavorful	-14.094742					
856	,	-3.454960		Yes			
857	the	-3.528767	Yes				
858	batter	-12.895466					
859	excellent	-10.147964					
860	,	-3.454960		Yes			
861	and	-4.113108	Yes				
862	for	-4.880109	Yes				
863	just	-5.630868	Yes				
864	$	-7.450107					
865	8	-8.940966				Yes	
866	.	-3.067898		Yes			
867	This	-6.783917	Yes				

868	may	-7.678495	Yes				
869	have	-5.156485	Yes				
870	been	-6.670917	Yes				
871	the	-3.528767	Yes				
872	best	-7.492557					
873	fish	-10.166230					
874	sandwich	-11.186007					
875	I	-3.791565	Yes				
876	've	-6.593011					
877	yet	-8.229137	Yes				
878	to	-3.856022	Yes				
879	have	-5.156485	Yes				
880	in	-4.619072	Yes				
881	da	-10.829142					
882	burgh	-16.942732					
883	.	-3.067898		Yes			
884	\n	-6.050651			Yes		

885 rows × 7 columns

If the text you'd like to process is general-purpose English language text (i.e., not domain-specific, like medical literature), spaCy is ready to use out-of-the-box.

I think it will eventually become a core part of the Python data science ecosystem — it will do for natural language computing what other great libraries have done for numerical computing.

Phrase Modeling

Phrase modeling is another approach to learning combinations of tokens that together represent meaningful multi-word concepts. We can develop phrase models by looping over the the words in our reviews and looking for words that *co-occur* (i.e., appear one after another) together much more frequently than you would expect them to by random chance. The formula our phrase models will use to determine whether two tokens A and B constitute a phrase is:

$$\frac{count(A\,B) - count_{min}}{count(A) * count(B)} * N > threshold$$

...where:

- $count(A)$ is the number of times token A appears in the corpus
- $count(B)$ is the number of times token B appears in the corpus
- $count(A\,B)$ is the number of times the tokens $A\,B$ appear in the corpus *in order*
- N is the total size of the corpus vocabulary
- $count_{min}$ is a user-defined parameter to ensure that accepted phrases occur a minimum number of times

- $count_{min}$ is a user-defined parameter to ensure that accepted phrases occur a minimum number of times
- $threshold$ is a user-defined parameter to control how strong of a relationship between two tokens the model requires before accepting them as a phrase

Once our phrase model has been trained on our corpus, we can apply it to new text. When our model encounters two tokens in new text that identifies as a phrase, it will merge the two into a single new token.

Phrase modeling is superficially similar to named entity detection in that you would expect named entities to become phrases in the model (so *new york* would become *new_york*). But you would also expect multi-word expressions that represent common concepts, but aren't specifically named entities (such as *happy hour*) to also become phrases in the model.

We turn to the indispensible **gensim** (https://radimrehurek.com/gensim/index.html) library to help us with phrase modeling — the **Phrases** (https://radimrehurek.com/gensim/models/phrases.html) class in particular.

In [16]:

```
from gensim.models import Phrases
from gensim.models.word2vec import LineSentence
```

As we're performing phrase modeling, we'll be doing some iterative data transformation at the same time. Our roadmap for data preparation includes:

1. Segment text of complete reviews into sentences & normalize text
2. First-order phrase modeling → *apply first-order phrase model to transform sentences*
3. Second-order phrase modeling → *apply second-order phrase model to transform sentences*
4. Apply text normalization and second-order phrase model to text of complete reviews

We'll use this transformed data as the input for some higher-level modeling approaches in the following sections.

First, let's define a few helper functions that we'll use for text normalization. In particular, the `lemmatized_sentence_corpus` generator function will use spaCy to:

- Iterate over the 1M reviews in the `review_txt_all.txt` we created before
- Segment the reviews into individual sentences
- Remove punctuation and excess whitespace
- Lemmatize the text

... and do so efficiently in parallel, thanks to spaCy's `nlp.pipe()` function.

In [17]:

```
def punct_space(token):
    """
    helper function to eliminate tokens
    that are pure punctuation or whitespace
    """
    
    return token.is_punct or token.is_space

def line_review(filename):
    """
    generator function to read in reviews from the file
    and un-escape the original line breaks in the text
    """
    
    with codecs.open(filename, encoding='utf_8') as f:
        for review in f:
            yield review.replace('\\n', '\n')

def lemmatized_sentence_corpus(filename):
    """
    generator function to use spaCy to parse reviews,
    lemmatize the text, and yield sentences
    """

    for parsed_review in nlp.pipe(line_review(filename),
                                  batch_size=10000, n_threads=4):

        for sent in parsed_review.sents:
            yield u' '.join([token.lemma_ for token in sent
                             if not punct_space(token)])
```

In [18]:

```
unigram_sentences_filepath = os.path.join(intermediate_directory,
                                          'unigram_sentences_all.txt')
```

Let's use the `lemmatized_sentence_corpus` generator to loop over the original review text, segmenting the reviews into individual sentences and normalizing the text. We'll write this data back out to a new file (`unigram_sentences_all`), with one normalized sentence per line. We'll use this data for learning our phrase models.

In [19]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute data prep yourself.
if 0 == 1:

    with codecs.open(unigram_sentences_filepath, 'w', encoding='utf_8') as f:
        for sentence in lemmatized_sentence_corpus(review_txt_filepath):
            f.write(sentence + '\n')
```

```
CPU times: user 11 µs, sys: 12 µs, total: 23 µs
Wall time: 36 µs
```

If your data is organized like our `unigram_sentences_all` file now is — a large text file with one document/sentence per line — gensim's **LineSentence** (https://radimrehurek.com/gensim/models/word2vec.html#gensim.models.word2vec.LineSentence) class provides a convenient iterator for working with other gensim components. It *streams* the documents/sentences from disk, so that you never have to hold the entire corpus in RAM at once. This allows you to scale your modeling pipeline up to potentially very large corpora.

In [20]:

```
unigram_sentences = LineSentence(unigram_sentences_filepath)
```

Let's take a look at a few sample sentences in our new, transformed file.

In [21]:

```
for unigram_sentence in it.islice(unigram_sentences, 230, 240):
    print u' '.join(unigram_sentence)
    print u''
```

```
no it be not the best food in the world but the service greatly help the percepti
on and it do not taste bad

so back in the late 90 there use to be this super kick as cinnamon ice cream like
an apple pie ice cream without the apple or the pie crust

so delicious

however now there be some shit tastic replacement that taste like vanilla ice cre
however now there be some shit tastic replacement that taste like vanilla ice cre
am with last year 's red hot in the middle totally gross

fortunately our server be nice enough to warn me about the change and bring me a
sample so i only have to suffer the death of a childhood memory rather than also
have to pay for it

the portion be big and fill just do not come for the ice cream

i have pretty much be eat at various king pretty regularly since i be a child whe
n my parent would take my sister and i into the fox chapel location often

lately me and my girl have be visit the heidelburg location

i love the food it really taste homemade much like something a grandmother would
make complete with gob of butter and side dish

price be low selection be great but do not expect fine dining by any mean
```

Next, we'll learn a phrase model that will link individual words into two-word phrases. We'd expect words that together represent a specific concept, like "`ice cream`", to be linked together to form a new, single token: "`ice_cream`".

In [22]:
```
bigram_model_filepath = os.path.join(intermediate_directory, 'bigram_model_all')
```

In [23]:
```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute modeling yourself.
if 0 == 1:

    bigram_model = Phrases(unigram_sentences)

    bigram_model.save(bigram_model_filepath)

# load the finished model from disk
bigram_model = Phrases.load(bigram_model_filepath)
```

```
CPU times: user 5.91 s, sys: 3.14 s, total: 9.05 s
Wall time: 11 s
```

Now that we have a trained phrase model for word pairs, let's apply it to the review sentences data and explore the results.

In [24]:
```
bigram_sentences_filepath = os.path.join(intermediate_directory,
                                         'bigram_sentences_all.txt')
```

In [25]:
```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute data prep yourself.
if 0 == 1:

    with codecs.open(bigram_sentences_filepath, 'w', encoding='utf_8') as f:

        for unigram_sentence in unigram_sentences:

            bigram_sentence = u' '.join(bigram_model[unigram_sentence])
            f.write(bigram_sentence + '\n')
```

```
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 8.11 µs
```

In [26]:

```
bigram_sentences = LineSentence(bigram_sentences_filepath)
```

In [27]:

```
for bigram_sentence in it.islice(bigram_sentences, 230, 240):
    print u' '.join(bigram_sentence)
    print u''
```

```
no it be not the best food in the world but the service greatly help the percepti
on and it do not taste bad

so back in the late 90 there use to be this super kick as cinnamon ice_cream like
an apple_pie ice_cream without the apple or the pie crust

so delicious

however now there be some shit tastic replacement that taste like vanilla_ice cre
am with last year 's red hot in the middle totally gross

fortunately our server be nice enough to warn me about the change and bring me a
sample so i only have to suffer the death of a childhood_memory rather_than also
have to pay for it

the portion be big and fill just do not come for the ice_cream

i have pretty much be eat at various king pretty regularly since i be a child whe
n my parent would take my sister and i into the fox_chapel location often

lately me and my girl have be visit the heidelburg location

i love the food it really taste homemade much like something a grandmother would
make complete with gob of butter and side dish

price be low selection be great but do not expect fine_dining by any mean
```

Looks like the phrase modeling worked! We now see two-word phrases, such as "ice_cream" and "apple_pie", linked together in the text as a single token. Next, we'll train a *second-order* phrase model. We'll apply the second-order phrase model on top of the already-transformed data, so that incomplete word combinations like "vanilla_ice cream" will become fully joined to "vanilla_ice_cream". No disrespect intended to Vanilla Ice (https://www.youtube.com/watch?v=rog8ou-ZepE), of course.

In [28]:

```
trigram_model_filepath = os.path.join(intermediate_directory,
                                      'trigram_model_all')
```

In [29]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute modeling yourself.
if 0 == 1:

    trigram_model = Phrases(bigram_sentences)

    trigram_model.save(trigram_model_filepath)

# load the finished model from disk
# load the finished model from disk
trigram_model = Phrases.load(trigram_model_filepath)
```

```
CPU times: user 4.85 s, sys: 3.17 s, total: 8.02 s
Wall time: 9.58 s
```

We'll apply our trained second-order phrase model to our first-order transformed sentences, write the results out to a new file, and explore a few of the second-order transformed sentences.

In [30]:

```
trigram_sentences_filepath = os.path.join(intermediate_directory,
                                          'trigram_sentences_all.txt')
```

In [31]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute data prep yourself.
if 0 == 1:

    with codecs.open(trigram_sentences_filepath, 'w', encoding='utf_8') as f:
        
        for bigram_sentence in bigram_sentences:
            
            trigram_sentence = u' '.join(trigram_model[bigram_sentence])
            
            f.write(trigram_sentence + '\n')
```

```
CPU times: user 8 µs, sys: 4 µs, total: 12 µs
Wall time: 21.9 µs
```

In [32]:

```
trigram_sentences = LineSentence(trigram_sentences_filepath)
```

In [33]:

```
for trigram_sentence in it.islice(trigram_sentences, 230, 240):
    print u' '.join(trigram_sentence)
    print u''
```

```
no it be not the best food in the world but the service greatly help the percepti
on and it do not taste bad

so back in the late 90 there use to be this super kick as cinnamon_ice_cream like
an apple_pie ice_cream without the apple or the pie crust

so delicious

however now there be some shit tastic replacement that taste like vanilla_ice_cre
am with last year 's red hot in the middle totally gross

fortunately our server be nice enough to warn me about the change and bring me a
sample so i only have to suffer the death of a childhood_memory rather_than also
have to pay for it

the portion be big and fill just do not come for the ice_cream

i have pretty much be eat at various king pretty regularly since i be a child whe
n my parent would take my sister and i into the fox_chapel location often

lately me and my girl have be visit the heidelburg location

i love the food it really taste homemade much like something a grandmother would
make complete with gob of butter and side dish

price be low selection be great but do not expect fine dining by any mean
```

Looks like the second-order phrase model was successful. We're now seeing three-word phrases, such as "`vanilla_ice_cream`" and "`cinnamon_ice_cream`".

The final step of our text preparation process circles back to the complete text of the reviews. We're going to run the complete text of the reviews through a pipeline that applies our text normalization and phrase models.

In addition, we'll remove stopwords at this point. *Stopwords* are very common words, like *a*, *the*, *and*, and so on, that serve functional roles in natural language, but typically don't contribute to the overall meaning of text. Filtering stopwords is a common procedure that allows higher-level NLP modeling techniques to focus on the words that carry more semantic weight.

Finally, we'll write the transformed text out to a new file, with one review per line.

In [34]:

```
trigram_reviews_filepath = os.path.join(intermediate_directory,
                                        'trigram_transformed_reviews_all.txt')
```

In [35]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute data prep yourself.
if 0 == 1:

    with codecs.open(trigram_reviews_filepath, 'w', encoding='utf_8') as f:

        for parsed_review in nlp.pipe(line_review(review_txt_filepath),
                                      batch_size=10000, n_threads=4):

            # lemmatize the text, removing punctuation and whitespace
            unigram_review = [token.lemma_ for token in parsed_review
                              if not punct_space(token)]

            # apply the first-order and second-order phrase models
            bigram_review = bigram_model[unigram_review]
            trigram_review = trigram_model[bigram_review]

            # remove any remaining stopwords
            trigram_review = [term for term in trigram_review
                              if term not in spacy.en.STOPWORDS]

            # write the transformed review as a line in the new file
            trigram_review = u' '.join(trigram_review)
            f.write(trigram_review + '\n')
```

```
CPU times: user 5 µs, sys: 1e+03 ns, total: 6 µs
Wall time: 11.9 µs
```

Let's preview the results. We'll grab one review from the file with the original, untransformed text, grab the same review from the file with the normalized and transformed text, and compare the two.

In [36]:

```
print u'Original:' + u'\n'

for review in it.islice(line_review(review_txt_filepath), 11, 12):
    print review

print u'----' + u'\n'
print u'Transformed:' + u'\n'
print u'Transformed:' + u'\n'

with codecs.open(trigram_reviews_filepath, encoding='utf_8') as f:
    for review in it.islice(f, 11, 12):
        print review
```

```
Original:

A great townie bar with tasty food and an interesting clientele. I went to check
this place out on the way home from the airport one Friday night and it didn't di
sappoint. It is refreshing to walk into a townie bar and not feel like the music
stops and everyone in the place is staring at you - I'm guessing the mixed crowd
of older hockey fans, young men in collared shirts, and thirtysomethings have pro
bably seen it all during their time at this place.

The staff was top notch - the orders were somewhat overwhelming as they appeared
short-staffed for the night, but my waitress tried to keep a positive attitude fo
r my entire visit. The other waiter was wearing a hooded cardigan, and I wanted t
o steal it from him due to my difficulty in finding such a quality article of clo
thing.

We ordered a white pizza - large in size, engulfed in cheese, full of garlic flav
or, flavorful hot sausage. An overall delicious pizza, aside from 2 things: 1, wa
y too much grease (I know this comes with the territory, but still, it is sometim
es unbearable); 2, CANNED MUSHROOMS - the worst thing to come out of a can. Ever.
I would rather eat canned Alpo than canned mushrooms. And if the mushrooms were
n't canned, they were just the worst mushrooms I've ever consumed. The mushroom d
ebacle is enough to lower the review by an entire star - disgusting!

My advice for the place is keep everything awesome - random music from the jukebo
x, tasty food, great prices, good crowd and staff - and get some decent mushroom
s; why they spoil an otherwise above average pie with such inferior crap, I'll ne
ver know.

----

Transformed:

great townie bar tasty food interesting clientele check place way home airport fr
iday_night disappoint refresh walk townie bar feel like music stop place star gue
ss mixed crowd old hockey_fan young_man collared_shirt thirtysomethings probably
time place staff top_notch order somewhat overwhelming appear short staff night w
aitress try positive_attitude entire visit waiter wear hooded cardigan want steal
difficulty quality article clothing order white pizza large size engulf cheese ga
rlic flavor flavorful hot sausage overall delicious pizza aside_from 2 thing 1 wa
y grease know come territory unbearable 2 canned mushrooms bad thing come eat alp
o canned_mushroom mushroom bad mushroom consume mushroom debacle lower review ent
ire star disgusting advice place awesome random music jukebox tasty food great pr
ice good crowd staff decent mushroom spoil above_average pie inferior crap know
```

You can see that most of the grammatical structure has been scrubbed from the text — capitalization, articles/conjunctions, punctuation, spacing, etc. However, much of the general semantic *meaning* is still present. Also, multi-word concepts such as "`friday_night`" and "`above_average`" have been joined into single tokens, as expected. The review text is now ready for higher-level modeling.

Topic Modeling with Latent Dirichlet Allocation (*LDA*)

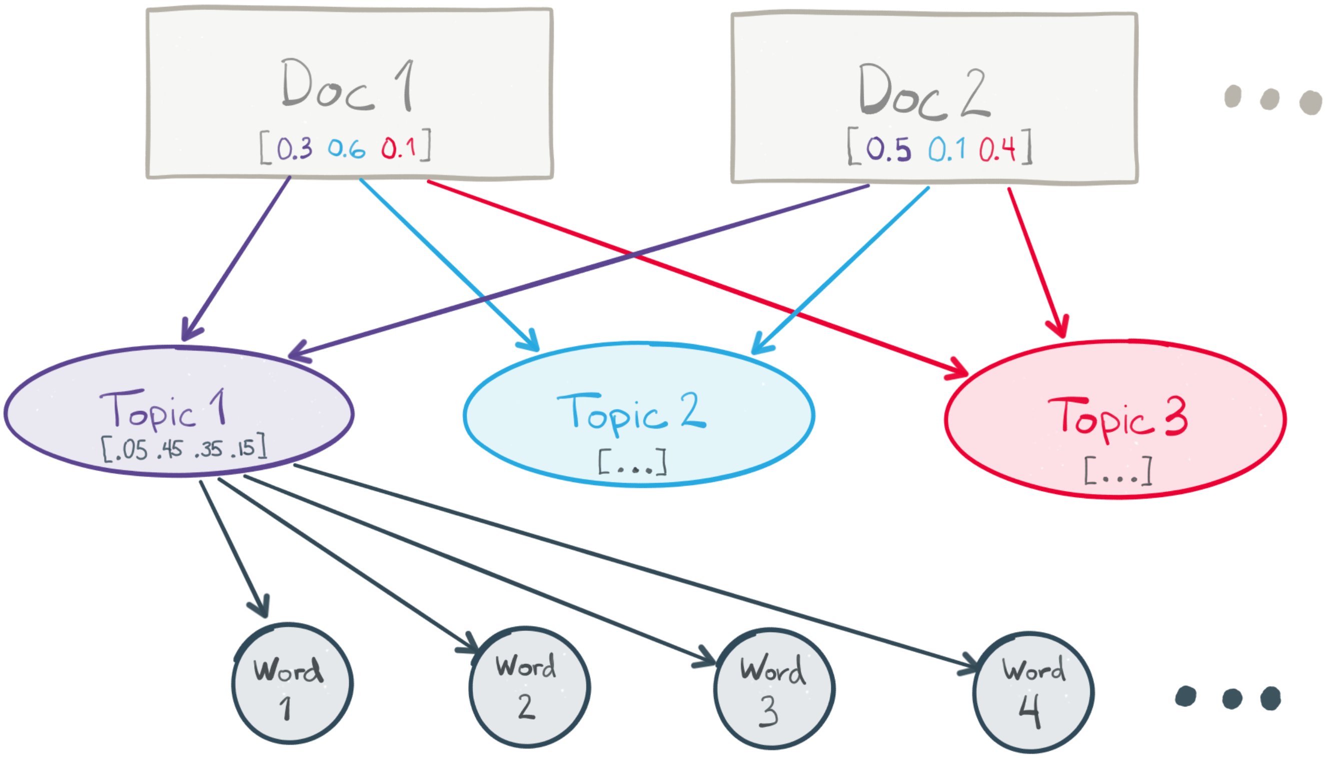
Topic modeling is family of techniques that can be used to describe and summarize the documents in a corpus

Topic modeling is family of techniques that can be used to describe and summarize the documents in a corpus according to a set of latent "topics". For this demo, we'll be using *Latent Dirichlet Allocation* (http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf) or LDA, a popular approach to topic modeling.

In many conventional NLP applications, documents are represented a mixture of the individual tokens (words and phrases) they contain. In other words, a document is represented as a *vector* of token counts. There are two layers in this model — documents and tokens — and the size or dimensionality of the document vectors is the number of tokens in the corpus vocabulary. This approach has a number of disadvantages:

- Document vectors tend to be large (one dimension for each token ⇒ lots of dimensions)
- They also tend to be very sparse. Any given document only contains a small fraction of all tokens in the vocabulary, so most values in the document's token vector are 0.
- The dimensions are fully indepedent from each other — there's no sense of connection between related tokens, such as *knife* and *fork*.

LDA injects a third layer into this conceptual model. Documents are represented as a mixture of a pre-defined number of *topics*, and the *topics* are represented as a mixture of the individual tokens in the vocabulary. The number of topics is a model hyperparameter selected by the practitioner. LDA makes a prior assumption that the (document, topic) and (topic, token) mixtures follow *Dirichlet* (https://en.wikipedia.org/wiki/Dirichlet_distribution) probability distributions. This assumption encourages documents to consist mostly of a handful of topics, and topics to consist mostly of a modest set of the tokens.



LDA is fully unsupervised. The topics are "discovered" automatically from the data by trying to maximize the likelihood of observing the documents in your corpus, given the modeling assumptions. They are expected to capture some latent structure and organization within the documents, and often have a meaningful human interpretation for people familiar with the subject material.

We'll again turn to gensim to assist with data preparation and modeling. In particular, gensim offers a high-performance parallelized implementation of LDA with its **LdaMulticore** (https://radimrehurek.com/gensim/models/ldamulticore.html) class.

```
In [37]: from gensim.corpora import Dictionary, MmC
```

In [37]:

```
from gensim.corpora import Dictionary, MmCorpus
from gensim.models.ldamulticore import LdaMulticore

import pyLDAvis
import pyLDAvis.gensim
import warnings
import cPickle as pickle
```

The first step to creating an LDA model is to learn the full vocabulary of the corpus to be modeled. We'll use gensim's **Dictionary** (https://radimrehurek.com/gensim/corpora/dictionary.html) class for this.

In [38]:

```
trigram_dictionary_filepath = os.path.join(intermediate_directory,
                                           'trigram_dict_all.dict')
```

In [39]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to learn the dictionary yourself.
if 0 == 1:

    trigram_reviews = LineSentence(trigram_reviews_filepath)

    # learn the dictionary by iterating over all of the reviews
    trigram_dictionary = Dictionary(trigram_reviews)

    # filter tokens that are very rare or too common from
    # the dictionary (filter_extremes) and reassign integer ids (compactify)
    trigram_dictionary.filter_extremes(no_below=10, no_above=0.4)
    trigram_dictionary.compactify()

    trigram_dictionary.save(trigram_dictionary_filepath)

# load the finished dictionary from disk
trigram_dictionary = Dictionary.load(trigram_dictionary_filepath)
```

```
CPU times: user 50.8 ms, sys: 10.7 ms, total: 61.5 ms
Wall time: 65.5 ms
```

Like many NLP techniques, LDA uses a simplifying assumption known as the *bag-of-words* model (https://en.wikipedia.org/wiki/Bag-of-words_model). In the bag-of-words model, a document is represented by the counts of distinct terms that occur within it. Additional information, such as word order, is discarded.

Using the gensim Dictionary we learned to generate a bag-of-words representation for each review. The `trigram_bow_generator` function implements this. We'll save the resulting bag-of-words reviews as a matrix.

In the following code, "bag-of-words" is abbreviated as `bow`.

In [40]:

```
trigram_bow_filepath = os.path.join(intermediate_directory,
                                    'trigram_bow_corpus_all.mm')
```

In [41]:

```
def trigram_bow_generator(filepath):
    """
    generator function to read reviews from a file
    and yield a bag-of-words representation
    """

    for review in LineSentence(filepath):
        yield trigram_dictionary.doc2bow(review)
```

In [42]:
```
%%time

# this is a bit time consuming - make the if statement True
# if you want to build the bag-of-words corpus yourself.
if 0 == 1:

    # generate bag-of-words representations for
    # all reviews and save them as a matrix
    MmCorpus.serialize(trigram_bow_filepath,
                       trigram_bow_generator(trigram_reviews_filepath))

# load the finished bag-of-words corpus from disk
trigram_bow_corpus = MmCorpus(trigram_bow_filepath)
```

```
CPU times: user 143 ms, sys: 25.7 ms, total: 169 ms
Wall time: 172 ms
```

With the bag-of-words corpus, we're finally ready to learn our topic model from the reviews. We simply need to pass the bag-of-words matrix and Dictionary from our previous steps to `LdaMulticore` as inputs, along with the number of topics the model should learn. For this demo, we're asking for 50 topics.

In [43]:
```
lda_model_filepath = os.path.join(intermediate_directory, 'lda_model_all')
```

In [44]:
```
%%time

# this is a bit time consuming - make the if statement True
# if you want to train the LDA model yourself.
if 0 == 1:

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')

        # workers => sets the parallelism, and should be
        # set to your number of physical cores minus one
        lda = LdaMulticore(trigram_bow_corpus,
                           num_topics=50,
                           id2word=trigram_dictionary,
                           workers=3)

    lda.save(lda_model_filepath)

# load the finished LDA model from disk
lda = LdaMulticore.load(lda_model_filepath)
```

```
CPU times: user 130 ms, sys: 182 ms, total: 312 ms
Wall time: 337 ms
```

Our topic model is now trained and ready to use! Since each topic is represented as a mixture of tokens, you can manually inspect which tokens have been grouped together into which topics to try to understand the patterns the model has discovered in the data.

In [45]:
```
def explore_topic(topic_number, topn=25):
    """
    accept a user-supplied topic number and
    print out a formatted list of the top terms
    """

    print u'{:20} {}'.format(u'term', u'frequency') + u'\n'
    for term, frequency in lda.show_topic(topic_number, topn=25):
        print u'{:20} {:.3f}'.format(term, round(frequency, 3))
```

In [46]:

```
explore_topic(topic_number=0)
```

```
term                 frequency

taco                 0.053
salsa                0.029
chip                 0.027
mexican              0.027
burrito              0.020
order                0.016
like                 0.013
try                  0.012
margarita            0.011
guacamole            0.010
come                 0.009
fresh                0.009
bean                 0.009
cheese               0.008
rice                 0.008
chicken              0.008
meat                 0.008
tortilla             0.007
flavor               0.007
nacho                0.007
'                    0.007
fish_taco            0.007
chipotle             0.006
little               0.006
sauce                0.006
```

The first topic has strong associations with words like *taco*, *salsa*, *chip*, *burrito*, and *margarita*, as well as a handful of more general words. You might call this the **Mexican food** topic!

It's possible to go through and inspect each topic in the same way, and try to assign a human-interpretable label that captures the essence of each one. I've given it a shot for all 50 topics below.

In [47]:

```
topic_names = {0: u'mexican',
               1: u'menu',
               2: u'thai',
               3: u'steak',
               4: u'donuts & appetizers',
               5: u'specials',
               6: u'soup',
               7: u'wings, sports bar',
               8: u'foreign language',
               9: u'las vegas',
               10: u'chicken',
               11: u'aria buffet',
               12: u'noodles',
               13: u'ambience & seating',
               14: u'sushi',
               15: u'arizona',
               16: u'family',
               17: u'price',
               18: u'sweet',
               19: u'waiting',
               20: u'general',
               21: u'tapas',
               22: u'dirty',
               23: u'customer service',
               24: u'restrooms',
               25: u'chinese',
               26: u'gluten free',
               27: u'pizza',
               28: u'seafood',
               29: u'amazing',
               30: u'eat, like, know, want',
               31: u'bars',
               32: u'breakfast',
               33: u'location & time',
               34: u'italian',
               35: u'barbecue',
               36: u'arizona',
               37: u'indian',
               38: u'latin & cajun',
               39: u'burger & fries',
               40: u'vegetarian',
               41: u'lunch buffet',
               42: u'customer service',
               43: u'taco, ice cream',
               44: u'high cuisine',
               45: u'healthy',
               46: u'salad & sandwich',
               47: u'greek',
               48: u'poor experience',
               49: u'wine & dine'}
```

In [48]:

```
topic_names_filepath = os.path.join(intermediate_directory, 'topic_names.pkl')

with open(topic_names_filepath, 'w') as f:
    pickle.dump(topic_names, f)
```

You can see that, along with **mexican**, there are a variety of topics related to different styles of food, such as **thai**, **steak**, **sushi**, **pizza**, and so on. In addition, there are topics that are more related to the overall restaurant *experience*, like **ambience & seating**, **good service**, **waiting**, and **price**.

Beyond these two categories, there are still some topics that are difficult to apply a meaningful human interpretation to, such as topic 30 and 43.

Manually reviewing the top terms for each topic is a helpful exercise, but to get a deeper understanding of the topics and how they relate to each other, we need to visualize the data — preferably in an interactive format. Fortunately, we have the fantastic **pyLDAvis** (https://pyldavis.readthedocs.io/en/latest/readme.html) library to help with that!

pyLDAvis includes a one-line function to take topic models created with gensim and prepare their data for visualization.

In [49]:

```
LDAvis_data_filepath = os.path.join(intermediate_directory, 'ldavis_prepared')
```

In [50]:

```
%%time

# this is a bit time consuming - make the if statement True
# if you want to execute data prep yourself.
if 0 == 1:
    LDAvis_prepared = pyLDAvis.gensim.prepare(lda, trigram_bow_corpus,
                                               trigram_dictionary)

    with open(LDAvis_data_filepath, 'w') as f:
        pickle.dump(LDAvis_prepared, f)

# load the pre-prepared pyLDAvis data from disk
with open(LDAvis_data_filepath) as f:
    LDAvis_prepared = pickle.load(f)
```

```
CPU times: user 442 ms, sys: 28.4 ms, total: 471 ms
Wall time: 526 ms
```

`pyLDAvis.display(...)` displays the topic model visualization in-line in the notebook.

In [51]:
```
pyLDAvis.display(LDAvis_prepared)
```

Out[51]:

Wait, what am I looking at again?

There are a lot of moving parts in the visualization. Here's a brief summary:

- On the left, there is a plot of the "distance" between all of the topics (labeled as the *Intertopic Distance Map*)
    - The plot is rendered in two dimensions according a *multidimensional scaling (MDS)* (https://en.wikipedia.org/wiki/Multidimensional_scaling) algorithm. Topics that are generally similar should be appear close together on the plot, while *dis*similar topics should appear far apart.
    - The relative size of a topic's circle in the plot corresponds to the relative frequency of the topic in the corpus.
    - An individual topic may be selected for closer scrutiny by clicking on its circle, or entering its number in the "selected topic" box in the upper-left.
- On the right, there is a bar chart showing top terms.
    - When no topic is selected in the plot on the left, the bar chart shows the top-30 most "salient" terms in the corpus. A term's *saliency* is a measure of both how frequent the term is in the corpus and how "distinctive" it is in distinguishing between different topics.
    - When a particular topic is selected, the bar chart changes to show the top-30 most "relevant" terms for the selected topic. The relevance metric is controlled by the parameter λ, which can be adjusted with a slider above the bar chart.
        - Setting the λ parameter close to 1.0 (the default) will rank the terms solely according to their probability within the topic.
        - Setting λ close to 0.0 will rank the terms solely according to their "distinctiveness" or "exclusivity" within the topic — i.e., terms that occur *only* in this topic, and do not occur in other topics.
        - Setting λ to values between 0.0 and 1.0 will result in an intermediate ranking, weighting term probability and exclusivity accordingly.
- Rolling the mouse over a term in the bar chart on the right will cause the topic circles to resize in the plot on the left, to show the strength of the relationship between the topics and the selected term.

A more detailed explanation of the pyLDAvis visualization can be found here (https://cran.r-project.org/web/packages/LDAvis/vignettes/details.pdf). Unfortunately, though the data used by gensim and pyLDAvis are the same, they don't use the same ID numbers for topics. If you need to match up topics in gensim's `LdaMulticore` object and pyLDAvis' visualization, you have to dig through the terms manually.

Analyzing our LDA model

The interactive visualization pyLDAvis produces is helpful for both:

1. Better understanding and interpreting individual topics, and
2. Better understanding the relationships between the topics.

For (1), you can manually select each topic to view its top most freqeuent and/or "relevant" terms, using different values of the λ parameter. This can help when you're trying to assign a human interpretable name or "meaning" to each topic.

For (2), exploring the *Intertopic Distance Plot* can help you learn about how topics relate to each other, including potential higher-level structure between groups of topics.

In our plot, there is a stark divide along the x-axis, with two topics far to the left and most of the remaining 48 far to the right. Inspecting the two outlier topics provides a plausible explanation: both topics contain many non-English words, while most of the rest of the topics are in English. So, one of the main attributes that distinguish the reviews in the dataset from one another is their language.

This finding isn't entirely a surprise. In addition to English-speaking cities, the Yelp dataset includes reviews of businesses in Montreal and Karlsruhe, Germany, often written in French and German, respectively. Multiple languages isn't a problem for our demo, but for a real NLP application, you might need to ensure that the text you're processing is written in English (or is at least tagged for language) before passing it along to some downstream processing. If that were the case, the divide along the x-axis in the topic plot would immediately alert you to a potential data quality issue.

The y-axis separates two large groups of topics — let's call them "super-topics" — one in the upper-right quadrant and the other in the lower-right quadrant. These super-topics correlate reasonably well with the pattern we'd noticed while naming the topics:

- The super-topic in the *lower*-right tends to be about *food*. It groups together the **burger & fries**, **breakfast**, **sushi**, **barbecue**, and **greek** topics, among others.
- The super-topic in the *upper*-right tends to be about other elements of the *restaurant experience*. It groups together the **ambience & seating**, **location & time**, **family**, and **customer service** topics, among others.

So, in addition to the 50 direct topics the model has learned, our analysis suggests a higher-level pattern in the data. Restaurant reviewers in the Yelp dataset talk about two main things in their reviews, in general: (1) the food, and (2) their overall restaurant experience. For this dataset, this is a very intuitive result, and we probably didn't need a sophisticated modeling technique to tell it to us. When working with datasets from other domains, though, such high-level patterns may be much less obvious from the outset — and that's where topic modeling can help.

Describing text with LDA

Beyond data exploration, one of the key uses for an LDA model is providing a compact, quantitative description of natural language text. Once an LDA model has been trained, it can be used to represent free text as a mixture of the topics the model learned from the original corpus. This mixture can be interpreted as a probability distribution across the topics, so the LDA representation of a paragraph of text might look like 50% *Topic A*, 20% *Topic B*, 20% *Topic C*, and 10% *Topic D*.

To use an LDA model to generate a vector representation of new text, you'll need to apply any text preprocessing steps you used on the model's training corpus to the new text, too. For our model, the preprocessing steps we used include:

1. Using spaCy to remove punctuation and lemmatize the text
2. Applying our first-order phrase model to join word pairs
3. Applying our second-order phrase model to join longer phrases
4. Removing stopwords
5. Creating a bag-of-words representation

Once you've applied these preprocessing steps to the new text, it's ready to pass directly to the model to create an LDA representation. The `lda_description(...)` function will perform all these steps for us, including printing the resulting topical description of the input text.

In [52]:

```
def get_sample_review(review_number):
    """
    retrieve a particular review index
    from the reviews file and return it
    """

    return list(it.islice(line_review(review_txt_filepath),
                          review_number, review_number+1))[0]
```

In [53]:

```
def lda_description(review_text, min_topic_freq=0.05):
    """
    accept the original text of a review and (1) parse it with spaCy,
    (2) apply text pre-proccessing steps, (3) create a bag-of-words
    representation, (4) create an LDA representation, and
    (5) print a sorted list of the top topics in the LDA representation
    """

    # parse the review text with spaCy
    parsed_review = nlp(review_text)

    # lemmatize the text and remove punctuation and whitespace
    unigram_review = [token.lemma_ for token in parsed_review
                      if not punct_space(token)]

    # apply the first-order and secord-order phrase models
    bigram_review = bigram_model[unigram_review]
    trigram_review = trigram_model[bigram_review]

    # remove any remaining stopwords
    trigram_review = [term for term in trigram_review
                      if not term in spacy.en.STOPWORDS]

    # create a bag-of-words representation
    review_bow = trigram_dictionary.doc2bow(trigram_review)

    # create an LDA representation
    review_lda = lda[review_bow]

    # sort with the most highly related topics first
    review_lda = sorted(review_lda, key=lambda (topic_number, freq): -freq)

    for topic_number, freq in review_lda:
        if freq < min_topic_freq:
            break

        # print the most highly related topic names and frequencies
        print '{:25} {}'.format(topic_names[topic_number],
                                round(freq, 3))
```

```
In [54]: sample_review = get_sample_review(50)
         print sample_review
```

```
Best French toast ever!! Love the friendly atmosphere, and especially the breakfa
st. Never been disappointed. You have to try French toast with raisin bread to
o... yummy!
```

```
In [55]: lda_description(sample_review)
```

```
breakfast                 0.4
amazing                   0.341
customer service          0.192
```

```
In [56]: sample_review = get_sample_review(100)
         print sample_review
```

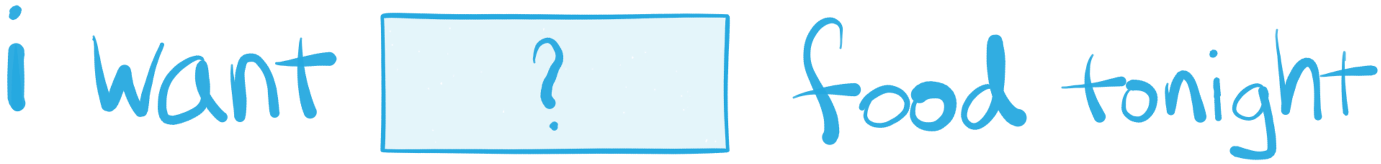
```
I went there for dinner last night with a client.  This is second time I visited.
I had a scotch and he had a Guiness.  The (-1) is for drink selection. Just stock
some better beers and higher end scotch and you're five stars.

We started with the meatballs covered with Provolone and other blessed goodness.
These did not disappoint.  I had a four cheese pizza with sweet sausage and garli
c.  It was fantastic.  They have so many good dishes, but I wanted the pizza last
night.  I couldn't finish the pizza - way to go big medium pizza.

I finished up with a coffee.  The parking can be a bit of a challenge on the stre
et, but it's a small town atmosphere in Carnegie, PA.  I love the downtown there.
```

```
In [57]: lda_description(sample_review)
```

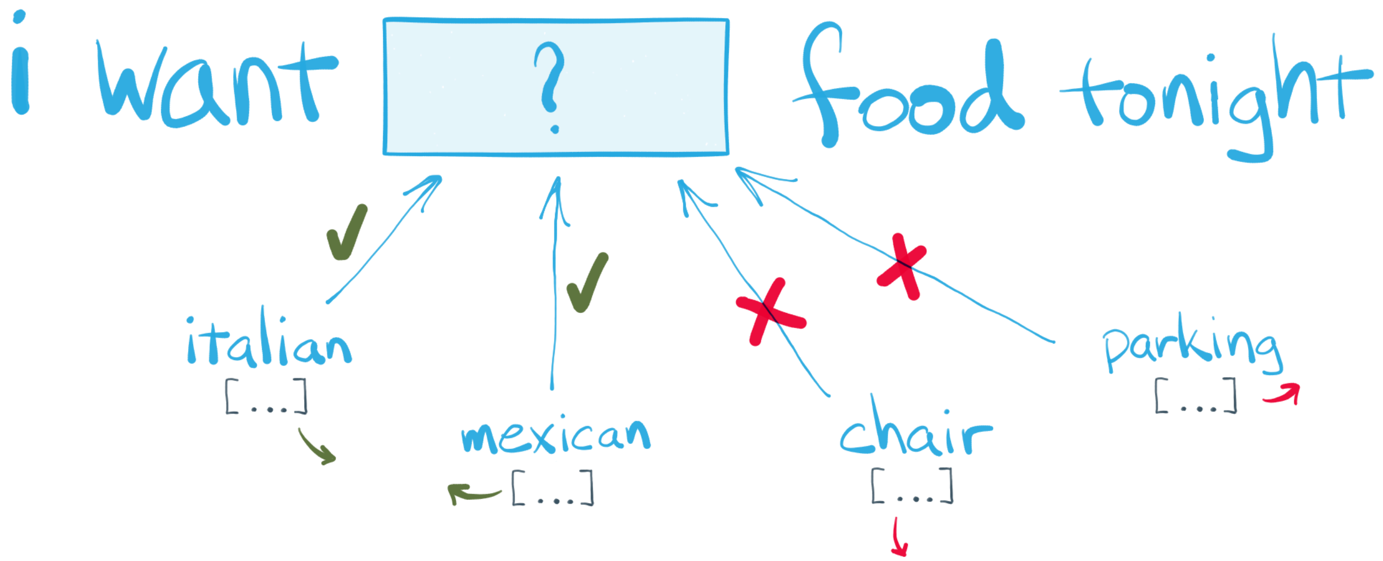
```
pizza                     0.28
bars                      0.193
amazing                   0.172
wine & dine               0.149
```

Word Vector Embedding with Word2Vec

Pop quiz! Can you complete this text snippet?



You just demonstrated the core machine learning concept behind word vector embedding models!



The goal of *word vector embedding models*, or *word vector models* for short, is to learn dense, numerical vector representations for each term in a corpus vocabulary. If the model is successful, the vectors it learns about each term should encode some information about the *meaning* or *concept* the term represents, and the relationship between it and other terms in the vocabulary. Word vector models are also fully unsupervised — they learn all of these meanings and relationships solely by analyzing the text of the corpus, without any advance knowledge provided.

Perhaps the best-known word vector model is word2vec (https://arxiv.org/pdf/1301.3781v3.pdf), originally proposed in 2013. The general idea of word2vec is, for a given *focus word*, to use the *context* of the word — i.e., the other words immediately before and after it — to provide hints about what the focus word might mean. To do this, word2vec uses a *sliding window* technique, where it considers snippets of text only a few tokens long at a time.

At the start of the learning process, the model initializes random vectors for all terms in the corpus vocabulary. The model then slides the window across every snippet of text in the corpus, with each word taking turns as the focus word. Each time the model considers a new snippet, it tries to learn some information about the focus word based on the surrouding context, and it "nudges" the words' vector representations accordingly. One complete pass sliding the window across all of the corpus text is known as a training *epoch*. It's common to train a word2vec model for multiple passes/epochs over the corpus. Over time, the model rearranges the terms' vector representations such that terms that frequently appear in similar contexts have vector representations that are *close* to each other in vector space.

For a deeper dive into word2vec's machine learning process, see here (https://arxiv.org/pdf/1411.2738v4.pdf).

Word2vec has a number of user-defined hyperparameters, including:

- The dimensionality of the vectors. Typical choices include a few dozen to several hundred.
- The width of the sliding window, in tokens. Five is a common default choice, but narrower and wider windows are possible.
- The number of training epochs.

For using word2vec in Python, gensim (https://rare-technologies.com/deep-learning-with-word2vec-and-gensim/) comes to the rescue again! It offers a highly-optimized (https://rare-technologies.com/word2vec-in-python-part-two-optimizing/), parallelized (https://rare-technologies.com/parallelizing-word2vec-in-python/) implementation of the word2vec algorithm with its Word2Vec (https://radimrehurek.com/gensim/models/word2vec.html) class.

```
In [58]: from gensim.models import Word2Vec
In [58]: from gensim.models import Word2Vec

         trigram_sentences = LineSentence(trigram_sentences_filepath)
         word2vec_filepath = os.path.join(intermediate_directory, 'word2vec_model_all')
```

We'll train our word2vec model using the normalized sentences with our phrase models applied. We'll use 100-dimensional vectors, and set up our training process to run for twelve epochs.

```
In [59]: %%time

         # this is a bit time consuming - make the if statement True
         # if you want to train the word2vec model yourself.
         if 0 == 1:

             # initiate the model and perform the first epoch of training
             food2vec = Word2Vec(trigram_sentences, size=100, window=5,
                                 min_count=20, sg=1, workers=4)

             food2vec.save(word2vec_filepath)

             # perform another 11 epochs of training
             for i in range(1,12):

                 food2vec.train(trigram_sentences)
                 food2vec.save(word2vec_filepath)

         # load the finished model from disk
         food2vec = Word2Vec.load(word2vec_filepath)
         food2vec.init_sims()

         print u'{} training epochs so far.'.format(food2vec.train_count)

         12 training epochs so far.
         CPU times: user 5.43 s, sys: 891 ms, total: 6.32 s
         Wall time: 7.12 s
```

On my four-core machine, each epoch over all the text in the ~1 million Yelp reviews takes about 5-10 minutes.

```
In [60]: print u'{:,} terms in the food2vec vocabulary.'.format(len(food2vec.vocab))

         50,835 terms in the food2vec vocabulary.
```

Let's take a peek at the word vectors our model has learned. We'll create a pandas DataFrame with the terms as the row labels, and the 100 dimensions of the word vector model as the columns.

```
In [90]: # build a list of the terms, integer indices,
         # and term counts from the food2vec model vocabulary
         ordered_vocab = [(term, voc.index, voc.count)
                          for term, voc in food2vec.vocab.iteritems()]

         # sort by the term counts, so the most common terms appear first
         ordered_vocab = sorted(ordered_vocab, key=lambda (term, index, count): -count)

         # unzip the terms, integer indices, and counts into separate lists
         ordered_terms, term_indices, term_counts = zip(*ordered_vocab)

         # create a DataFrame with the food2vec vectors as data,
         # and the terms as row labels
         word_vectors = pd.DataFrame(food2vec.syn0norm[term_indices, :],
                                     index=ordered_terms)
```

```
                                index=ordered_terms)

word_vectors
```

Out[90]:

	0	1	2	3	4	5	6
the	-0.035762	-0.173890	-0.035782	-0.007144	0.032371	-0.065272	-
be	-0.074780	-0.049524	0.085974	-0.098892	0.141556	0.024878	-
and	-0.070505	-0.026918	0.028344	-0.099909	0.127974	-0.058155	-
i	-0.161238	0.050831	-0.081706	-0.084479	0.053073	-0.102327	-
a	-0.083491	-0.033712	-0.124125	-0.110776	-0.033046	-0.089950	0
to	-0.012082	0.033135	-0.063183	-0.057252	-0.018721	-0.017931	-
it	0.025022	0.081581	0.127987	-0.188015	0.041450	-0.126222	0
have	-0.140812	-0.070552	0.022102	0.001077	0.109890	-0.061365	0
of	-0.036341	-0.054903	0.000644	-0.010602	0.168195	-0.058505	-
not	-0.075276	0.109047	0.055135	0.052251	0.209437	0.084334	-
for	-0.102976	0.001369	-0.069402	-0.122936	0.028278	-0.074256	-
in	-0.053390	-0.175599	-0.091688	-0.153791	0.003205	0.013146	-
we	-0.015929	-0.019187	-0.186680	-0.240963	0.077926	-0.122313	-
that	-0.026609	0.085940	0.118164	0.011576	0.156952	-0.061402	-
but	-0.063436	0.089140	-0.057425	-0.093110	0.066531	-0.079715	-
with	0.042318	-0.186670	-0.230563	0.076302	0.216593	-0.056183	0
my	-0.002067	-0.023159	0.035879	0.036316	-0.110738	-0.033034	-
this	0.139739	0.184600	0.137359	-0.109916	0.021484	-0.018423	-
you	-0.171262	-0.119866	0.063801	-0.087287	-0.061923	0.023105	-
on	0.057481	0.044937	-0.063766	-0.007839	0.161119	-0.047322	-
they	-0.235321	-0.026314	0.143165	-0.170460	0.042189	-0.019444	-
food	-0.164133	0.007745	0.058311	-0.169839	-0.042278	0.004095	0
do	0.091428	-0.132115	0.105080	0.135949	0.038100	0.066993	-
good	-0.239592	-0.232940	-0.005036	-0.028226	0.149816	-0.133312	-
place	0.025479	0.130311	0.119834	-0.096365	0.013793	0.074431	-
so	0.021455	0.079794	0.192058	-0.093809	-0.094279	-0.147522	-
get	0.009313	-0.101684	-0.163864	-0.159002	0.018936	-0.056202	-
go	0.031094	-0.126839	-0.054429	-0.221885	-0.063464	0.024554	0
at	0.102501	-0.095756	-0.216304	-0.107230	-0.112544	-0.036979	-
as	0.141091	0.073669	0.109637	-0.112564	-0.167600	-0.059139	-
...	...	...	...	...	...	...	.
hard_boiled_egg	0.025154	0.060949	-0.064816	0.071975	0.087870	-0.034552	0
poached_quail_egg	0.060616	0.001519	-0.052908	0.013590	0.031732	-0.020164	0

egg_foo	0.022816	0.077288	-0.122898	-0.022765	-0.137979	-0.131516
sherri	-0.089536	-0.030370	0.011311	-0.094766	-0.121970	0.143681
hindrance	-0.033325	0.011520	0.027370	0.240223	-0.004098	-0.047842
eggdrop_soup	-0.028369	0.040039	-0.005378	-0.088736	0.015203	-0.150734
arbitrarily	0.203714	-0.047405	-0.045261	0.000302	-0.074105	-0.011420
faisant	0.046053	0.083873	0.057943	0.174203	-0.121259	-0.043806
marian	-0.035330	0.146843	-0.173594	-0.010971	-0.150150	0.082224
9:30p	0.119010	0.002159	-0.000148	-0.102635	-0.016918	-0.075822
extra_chasu	-0.225889	-0.131615	0.046431	0.017999	0.119188	-0.075226
lavosh_wrap	0.134420	-0.032055	0.012240	0.024420	0.031334	-0.002414
dum_dum	-0.002741	-0.137371	0.030704	-0.030365	-0.134645	-0.036521
triplet	-0.010495	0.057432	-0.019535	-0.044881	0.042409	-0.094355
nantucket	-0.124356	0.141918	-0.038579	0.035650	-0.157662	0.048110
gurl	-0.119794	0.025898	-0.070130	-0.027929	0.113244	0.076868
nordstroms	-0.030410	-0.026861	-0.016836	0.097363	-0.098189	0.080675
eau_de	-0.146450	0.000788	-0.094391	0.146833	-0.051392	0.026519
extra_.5	-0.124972	-0.032596	-0.017800	0.106415	-0.086728	-0.039636
hazelnut_crunch	0.055723	-0.098708	0.013225	0.098075	-0.021967	-0.046137
bella_notte	0.209296	-0.102068	-0.059274	-0.061223	0.032078	0.042433
homebrew	0.054390	-0.050279	-0.181006	0.001028	-0.064048	0.029383
conveyor_belt_oven	-0.099600	-0.127618	0.070998	-0.040632	-0.022066	-0.021832
meekly	-0.087812	0.020389	0.040041	-0.046436	0.084847	0.022525
foccaccia	0.118130	0.028629	-0.063642	0.006247	0.072471	-0.086754
clyde	0.025336	-0.044486	-0.081030	-0.049451	-0.215602	-0.004157
original_g_spicy	0.058115	-0.036521	-0.119183	-0.040159	0.163193	0.043903
potaotes	0.009098	0.057424	-0.156997	-0.057388	0.030169	-0.095243
desert_botanical_gardens	0.073653	0.200249	-0.088580	-0.032873	-0.161853	0.066677
mi_match	0.025482	0.122178	0.062693	0.150734	-0.028056	0.091268

50835 rows × 100 columns

Holy wall of numbers! This DataFrame has 50,835 rows — one for each term in the vocabulary — and 100 colums. Our model has learned a quantitative vector representation for each term, as expected.

Put another way, our model has "embedded" the terms into a 100-dimensional vector space.

So... what can we do with all these numbers?

The first thing we can use them for is to simply look up related words and phrases for a given term of interest.

The first thing we can use them for is to simply look up related words and phrases for a given term of interest.

In [63]:

```
def get_related_terms(token, topn=10):
    """
    look up the topn most similar terms to token
    and print them as a formatted list
    """

    for word, similarity in food2vec.most_similar(positive=[token], topn=topn):

        print u'{:20} {}'.format(word, round(similarity, 3))
```

What things are like Burger King?

In [64]:

```
get_related_terms(u'burger_king')
```

```
mcdonalds            0.895
wendy_'s             0.855
mcd_'s               0.853
mcdonald_'s          0.852
denny_'s             0.816
bk                   0.808
carl_'s_jr.          0.8
red_robin            0.792
mickey_d_'s          0.771
sonic                0.765
```

The model has learned that fast food restaurants are similar to each other! In particular, *mcdonalds* and *wendy's* are the most similar to Burger King, according to this dataset. In addition, the model has found that alternate spellings for the same entities are probably related, such as *mcdonalds*, *mcdonald's* and *mcd's*.

When is happy hour?

In [65]:

```
get_related_terms(u'happy_hour')
```

```
hh                   0.874
reverse_happy_hour   0.801
happy_hr             0.771
during_happy_hour    0.672
mon_fri              0.634
3_6pm                0.632
hh.                  0.631
4_7pm                0.625
special              0.621
happy_hour_3_6pm     0.618
```

The model has noticed several alternate spellings for happy hour, such as *hh* and *happy hr*, and assesses them as highly related. If you were looking for reviews about happy hour, such alternate spellings would be very helpful to know.

Taking a deeper look — the model has turned up phrases like *3-6pm*, *4-7pm*, and *mon-fri*, too. This is especially interesting, because the model has no advance knowledge at all about what happy hour is, and what time of day it should be. But simply by scanning through restaurant reviews, the model has discovered that the concept of happy hour has something very important to do with that block of time around 3-7pm on weekdays.

Let's make pasta tonight. Which style do you want?

In [66]:

```
get_related_terms(u'pasta', topn=20)
```

```
lasagna                 0.798
spaghetti               0.773
bolognese               0.757
fettuccine              0.748
penne                   0.745
rigatoni                0.743
angel_hair              0.721
linguine                0.716
angel_hair_pasta        0.712
penne_pasta             0.712
carbonara               0.705
fettucini               0.704
alfredo                 0.703
tortellini              0.703
manicotti               0.7
ziti                    0.698
gnocci                  0.694
ravioli                 0.694
linguini                0.693
risotto                 0.691
```

Word algebra!

No self-respecting word2vec demo would be complete without a healthy dose of *word algebra*, also known as *analogy completion*.

The core idea is that once words are represented as numerical vectors, you can do math with them. The mathematical procedure goes like this:

1. Provide a set of words or phrases that you'd like to add or subtract.
2. Look up the vectors that represent those terms in the word vector model.
3. Add and subtract those vectors to produce a new, combined vector.
4. Look up the most similar vector(s) to this new, combined vector via cosine similarity.
5. Return the word(s) associated with the similar vector(s).

But more generally, you can think of the vectors that represent each word as encoding some information about the *meaning* or *concepts* of the word. What happens when you ask the model to combine the meaning and concepts of words in new ways? Let's see.

In [67]:

```
def word_algebra(add=[], subtract=[], topn=1):
    """
    combine the vectors associated with the words provided
    in add= and subtract=, look up the topn most similar
    terms to the combined vector, and print the result(s)
    """
    answers = food2vec.most_similar(positive=add, negative=subtract, topn=topn)
    
    for term, similarity in answers:
        print term
```

breakfast + lunch = ?

Let's start with a softball!

Let's start with a softball.

```
In [68]: word_algebra(add=[u'breakfast', u'lunch'])
```

```
brunch
```

OK, so the model knows that *brunch* is a combination of *breakfast* and *lunch*. What else?

lunch - day + night = ?

```
In [69]: word_algebra(add=[u'lunch', u'night'], subtract=[u'day'])
```

```
dinner
```

Now we're getting a bit more nuanced. The model has discovered that:

- Both *lunch* and *dinner* are meals
- The main difference between them is time of day
- Day and night are times of day
- Lunch is associated with day, and dinner is associated with night

What else?

taco - mexican + chinese = ?

```
In [70]: word_algebra(add=[u'taco', u'chinese'], subtract=[u'mexican'])
```

```
dumpling
```

Here's an entirely new and different type of relationship that the model has learned.

- It knows that tacos are a characteristic example of Mexican food
- It knows that Mexican and Chinese are both styles of food
- If you subtract *Mexican* from *taco*, you're left with something like the concept of a *"characteristic type of food"*, which is represented as a new vector
- If you add that new *"characteristic type of food"* vector to Chinese, you get *dumpling*.

What else?

bun - american + mexican = ?

```
In [71]: word_algebra(add=[u'bun', u'mexican'], subtract=[u'american'])
```

```
tortilla
```

The model knows that both *buns* and *tortillas* are the doughy thing that goes on the outside of your real food, and that the primary difference between them is the style of food they're associated with.

What else?

filet mignon - beef + seafood = ?

```
In [72]: word_algebra(add=[u'filet_mignon', u'seafood'], subtract=[u'beef'])
```

```
raw_oyster
```

The model has learned a concept of *delicacy*. If you take filet mignon and subtract beef from it, you're left with a vector that roughly corresponds to delicacy. If you add the delicacy vector to *seafood*, you get *raw oyster*.

What else?

coffee - drink + snack = ?

```
In [73]: word_algebra(add=[u'coffee', u'snack'], subtract=[u'drink'])
```

```
pastry
```

The model knows that if you're on your coffee break, but instead of drinking something, you're eating something... that thing is most likely a pastry.

What else?

Burger King + fine dining = ?

```
In [74]: word_algebra(add=[u'burger_king', u'fine_dining'])
```

```
denny_'s
```

Touché. It makes sense, though. The model has learned that both Burger King and Denny's are large chains, and that both serve fast, casual, American-style food. But Denny's has some elements that are slightly more upscale, such as printed menus and table service. Fine dining, indeed.

What if we keep going?

Denny's + fine dining = ?

```
In [75]: word_algebra(add=[u"denny_'s", u'fine_dining'])
```

```
applebee_'s
```

This seems like a good place to land... what if we explore the vector space around *Applebee's* a bit, in a few different directions? Let's see what we find.

Applebee's + italian = ?

```
In [76]: word_algebra(add=[u"applebee_'s", u'italian'])
```

```
olive_garden
```

Applebee's + pancakes = ?

```
In [77]: word_algebra(add=[u"applebee_'s", u'pancakes'])
```

```
ihop
```

Applebee's + pizza = ?

```
In [78]: word_algebra(add=[u"applebee_'s", u'pizza'])
```

```
pizza_hut
```

You could do this all day. One last analogy before we move on...

wine - grapes + barley = ?

```
In [79]: word_algebra(add=[u'wine', u'barley'], subtract=[u'grapes'])
```

```
beer
```

Word Vector Visualization with t-SNE

t-Distributed Stochastic Neighbor Embedding (https://lvdmaaten.github.io/publications/papers/JMLR_2008.pdf), or *t-SNE* for short, is a dimensionality reduction technique to assist with visualizing high-dimensional datasets. It attempts to map high-dimensional data onto a low two- or three-dimensional representation such that the relative distances between points are preserved as closely as possible in both high-dimensional and low-dimensional space.