Bradley Voytek, Ph.D.

UC San Diego

Cognitive and Neural Dynamics Laboratory

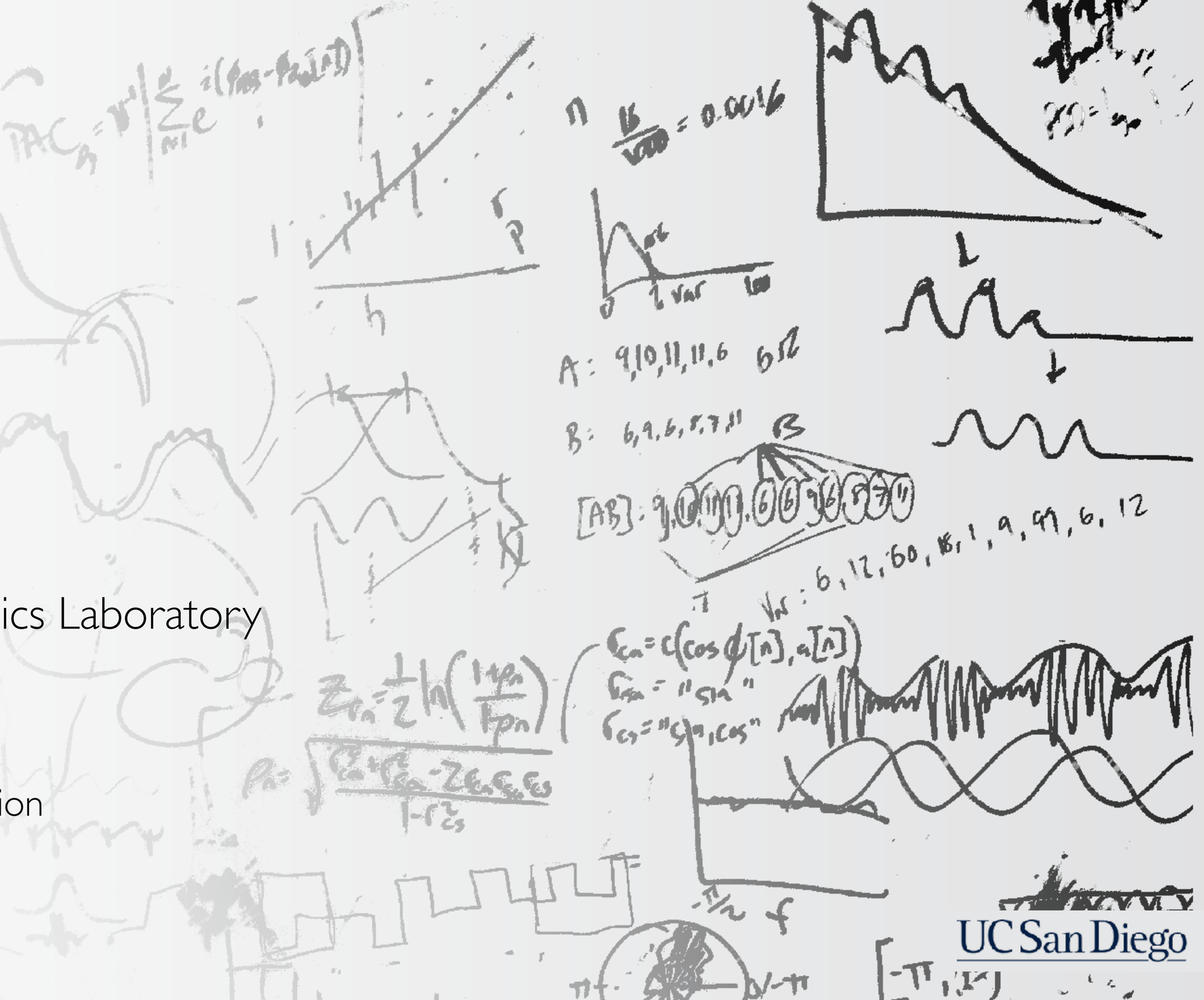
Department of Cognitive Science

Neurosciences Graduate Program

The Institute for Neural Computation

bvoytek@ucsd.edu

@bradleyvoytek



Administrative stuff

- **FINAL REQUIRED LECTURE**
  - Josh Wills: Director of Data Engineering, Slack
  - 2018 February 27 (Tuesday)

Administrative stuff

- **Still anyone without a group?!**

Administrative stuff

- **PYTHON HELP!**
  - http://codecademy.com/learn/learn-python

Administrative stuff

- **GIT HELP!**
  - http://codecademy.com/learn/learn-git

Administrative stuff

- **DATA SCIENCE HELP!**
  - https://software-carpentry.org/

Administrative stuff

- **We will release A5 on Wednesday (instead of Monday), and have it due end of W9.**

- **A4 is now due *Monday* night (giving you an extra day). This is because we released slightly later than originally planned, and Monday sections missed a section last week due to the holiday.**

UC San Diego Halicioglu Data Science Institute Launch Event

The *special objectives* for the *optional* UC San Diego Halicioglu Data Science Institute launch event are to:

- Communicate your results effectively to both experts and laypersons.
- Use data scientific approaches to address questions *specifically concerning civic utility and social good*.

A panel of local Data Science experts from the university, government, and industry will evaluate 4-8 projects, selected by Prof. Voytek for their potential for addressing critical questions of civic utility and/or social good.

These Projects *need not be the complete and final project you will submit for grading*, however they do need to be relatively thorough and complete to be considered for presentation on the afternoon of the launch event.

Deadline: To be considered eligible for presenting at this event, you will need to submit your Project Notebook by Sunday, Feb 25 at 23:59.

This *optional* submission, to be considered for the event, should follow the same outline and rubric as above for the final project notebook. You must have preliminary results, but it can be be a work-in-progress (for example, discussion section and conclusions need not necessarily be fleshed out).

One member from your team must submit this notebook on TritonED, with filename format (filled in with your group number):

'Pr_0XX_HDSIevent.ipynb'

COGS 108
Data Science in Practice

Distributions

UC San Diego

Central limit theorem

For a random sample from a distribution (any distribution!) with (finite) mean and (finite) variance, if n is sufficiently large then the sample mean follows an approximate normal distribution

```
% reset

import random
import numpy as np
from scipy import stats

% config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
from matplotlib import rcParams
% matplotlib inline
rcParams['figure.figsize'] = 8, 6
rcParams['font.family'] = 'sans-serif'
rcParams['font.sans-serif'] = ['Tahoma']
```

```
%%time
# magic that tells you how long the cell takes to run

num_rolls = 100000 # how many simulated die rolls?

# define the function to simulate die rolls
def rollDie(number):
    roll = []
    for i in range(number):
        # note this appends the result at the end of the list every iteration
        roll = np.append(roll, random.randint(1,6))
    return roll

simulated_rolls = rollDie(num_rolls)
```

```
CPU times: user 2.02 s, sys: 102 ms, total: 2.12 s
Wall time: 2.13 s
```

>2.0 seconds to simulate 100,000 die rolls is a long time!

```
%%time

# note this version creates a numpy vector of zeros first, and rather than appending
# a number at the end of a list such as was done above, it rewrites the zeros with
# the result as it moves through the loop.

# this is called initializing your data, and can speed things up
# the reason this is faster is because when you append, as we did above, numpy has to
# grow the array each loop

num_rolls = 100000

def rollDie(number):
    roll = np.zeros((number, 1)) # initialize your vector
    for i in range(number):
        roll[i] = random.randint(1,6)
    return roll

simulated_rolls = rollDie(num_rolls)
```

```
CPU times: user 149 ms, sys: 3.49 ms, total: 152 ms
Wall time: 153 ms
```

<0.2 seconds to simulate 100,000 die rolls; much better!

```
%%time

# now we're going to do it all in native numpy, which uses a (much faster) C basis
# look, no loops!

num_rolls = 100000 # how many simulated die rolls?

simulated_rolls = np.random.randint(low=1, high=7, size=num_rolls)
```

```
CPU times: user 1.96 ms, sys: 1.27 ms, total: 3.22 ms
Wall time: 1.86 ms
```

About 0.002 seconds this time! Orders of magnitude faster than the above, which was orders of magnitude faster than the first method.

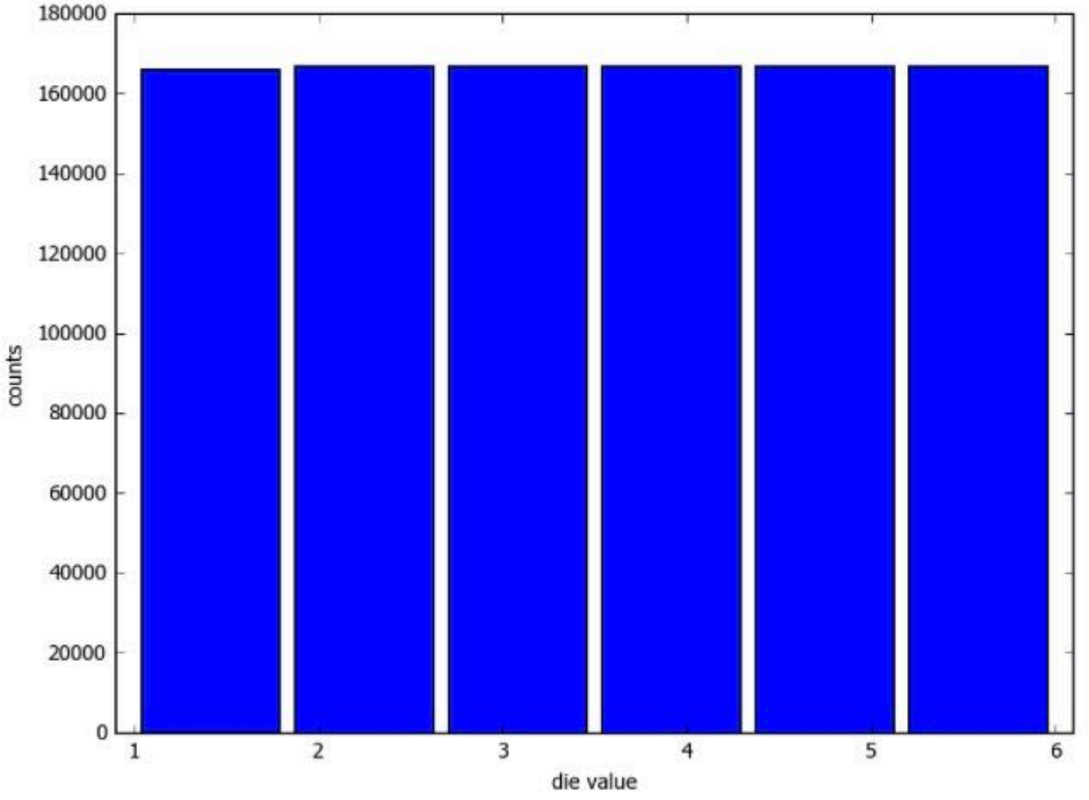
The moral is to always profile your code!

Aside: Code profiling

In software engineering, profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

Source: https://en.wikipedia.org/wiki/Profiling_(computer_programming)

```python
num_rolls = 1000000

simulated_rolls = np.random.randint(low=1, high=7, size=num_rolls)

# np.unique finds the unique values are in our data (1, 2, 3, 4, 5, 6)
# and len asks how many there are (6)
# we then use that to define the number of bins we want in our histogram
die_vals = np.unique(simulated_rolls)
number_of_bins = len(die_vals)
spacing = 0.1

plt.hist(simulated_rolls, bins=number_of_bins, rwidth=1-spacing)
plt.xlim(np.min(die_vals)-spacing, np.max(die_vals)+spacing)
plt.xlabel('die value')
plt.ylabel('counts')
plt.show()
```



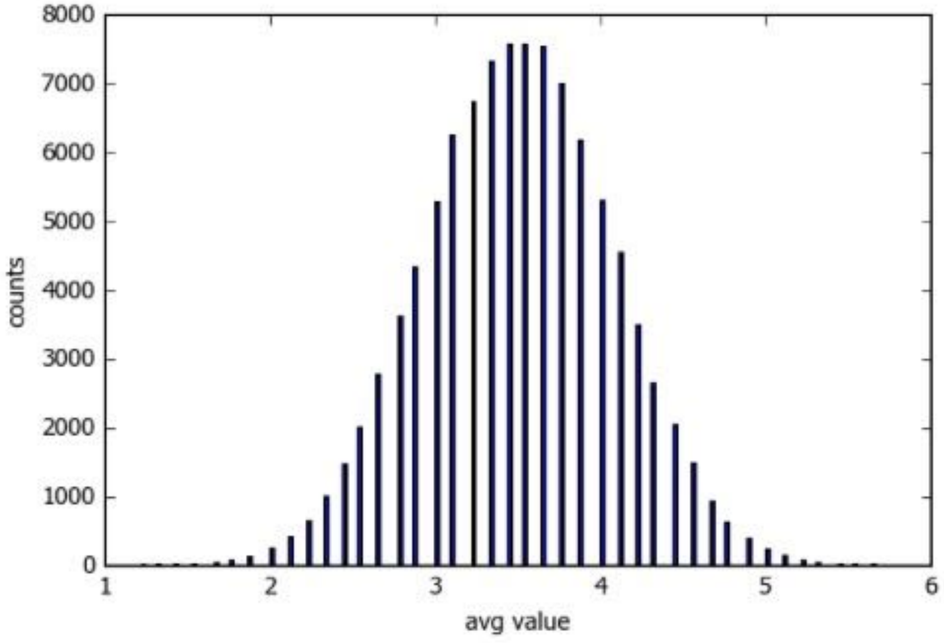
Looks decently uniform!

So let's look at the magic that is the Central Limit Theorem.

First, we break our data vector of die roll results, simulated_rolls, into chunks of 10 rolls each.

Then we average each of those 10 rolls and store the result.

```python
chunk_size = 10 # each chunk has this many rolls
samples = num_rolls//chunk_size # instead of <num_rolls> elements, we now have num_rolls/10
                                # so instead of 1,000,000 we have 100,000
roll_avg = np.zeros((samples, 1)) # initialize your vector

for i in range(samples):
    # for each loop we need to average from 0:9, 10:19, 20:29, and so on
    # this gives us a low range and a high range to average across
    # the low range (0,10,20,etc) is just i*10
    # the high range is that, plus 9
    low_range = i*10
    high_range = low_range+9
    roll_avg[i] = np.mean(simulated_rolls[low_range:high_range]) # average the chunks

# plot the histogram!
plt.hist(roll_avg, 200)
plt.xlabel('avg value')
plt.ylabel('counts')
plt.show()

print(np.mean((1,2,3,4,5,6)), np.mean(roll_avg))
```




```
3.5 3.49934888889
```

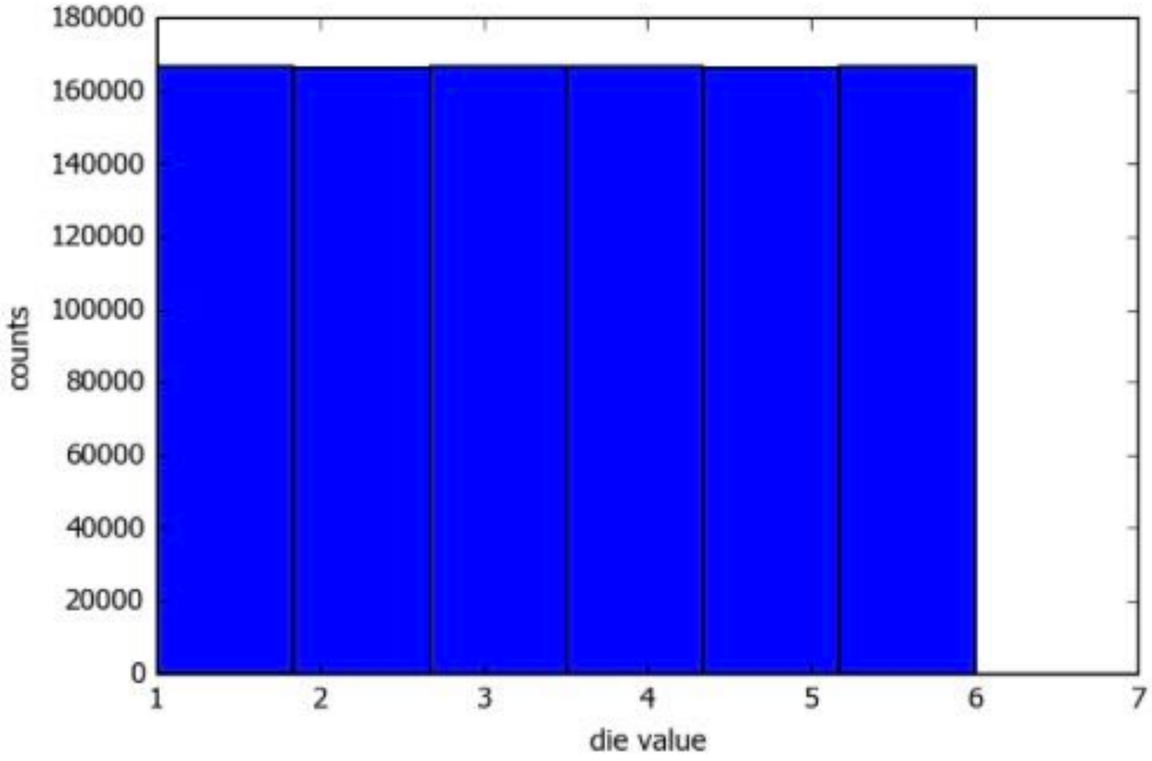
Great! Just like the CLT says, averages of samples will result in normally distributed averages.

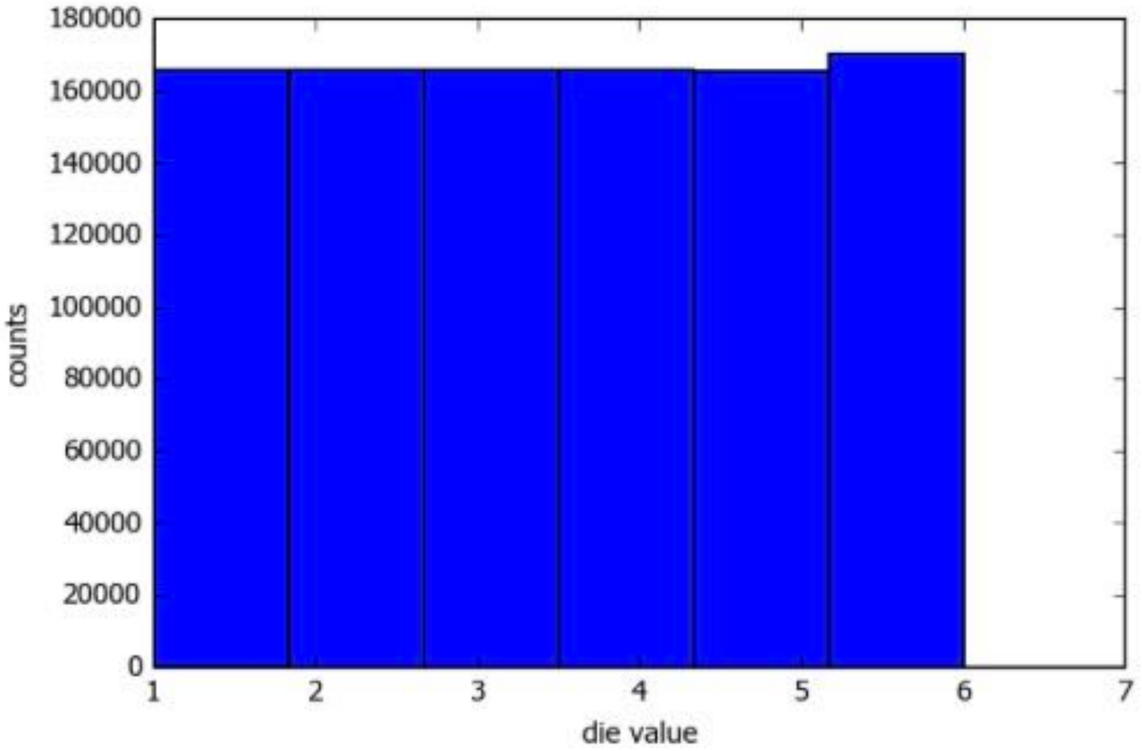
Unsurprisingly, the mean appears to be ~3.5, which is the mean of (1,2,3,4,5,6)

We're going to show how we can leverage the CLT to run statistical analyses that assume normality on data that are otherwise non-normal (such as die roll probabilities).

Note that although we're doing die rolls here, these could just as easily be star-ratings for Yelp, Uber, etc. or thumbs up/down for songs on Pandora or Netflix shows.

```python
# now, let's create a second set of fake die rolls
simulated_biased_rolls = np.random.randint(low=1, high=7, size=num_rolls)

# we're going to replace approximately 0.5% of each non-six result with a 6 to
# simulate a biased die
percent_bias = 0.005
replace_size = np.around((num_rolls/number_of_bins)*percent_bias)
replace_size = replace_size.astype(int)

# loop from 1 to 5
for i in range((number_of_bins-1)):
    # find each instance of i+1 as the die result
    idx = np.where(simulated_biased_rolls==(i+1))
    replace_idx = np.random.choice(idx[0], size=replace_size, replace=True)
    simulated_biased_rolls[replace_idx] = 6

plt.hist(simulated_rolls, 6)
plt.xlabel('die value')
plt.ylabel('counts')
plt.show()

plt.hist(simulated_biased_rolls, 6)
plt.xlabel('die value')
plt.ylabel('counts')
plt.show()
```

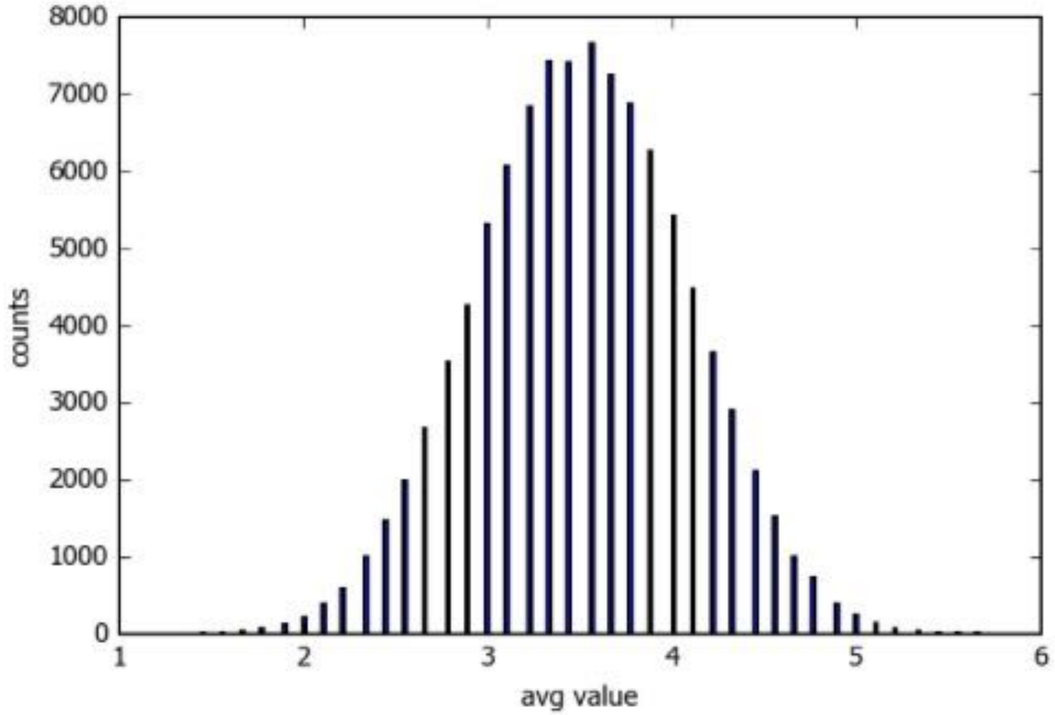




Okay it looks like there's a *little* bump at 6, but it's not hugely obvious.

Let's look at the histogram of the samples from the biased rolls and see if we see anything obvious.

```
chunk_size = 10
samples = num_rolls//chunk_size
biased_roll_avg = np.zeros((samples, 1))

for i in range(samples):
    low_range = i*10
    high_range = low_range+9
    # average the chunks
    biased_roll_avg[i] = np.mean(simulated_biased_rolls[low_range:high_range])

# plot the histogram!
plt.hist(biased_roll_avg, 200)
plt.xlabel('avg value')
plt.ylabel('counts')
plt.show()
```



Well, okay. Looks about the same as above...

What's the mean of this one?

```
np.mean(biased_roll_avg)
```

```
3.5129466666666662
```

Just *ever so slightly* greater than the 3.5 we'd expect.

Is this significant though? How can we check?

This is where the "art" of data science starts to come into play!

How can we determine (statistically) if our die is loaded?

We can start with the assumption that any two distributions of random samples of the means of fair die will be centered around 3.5.

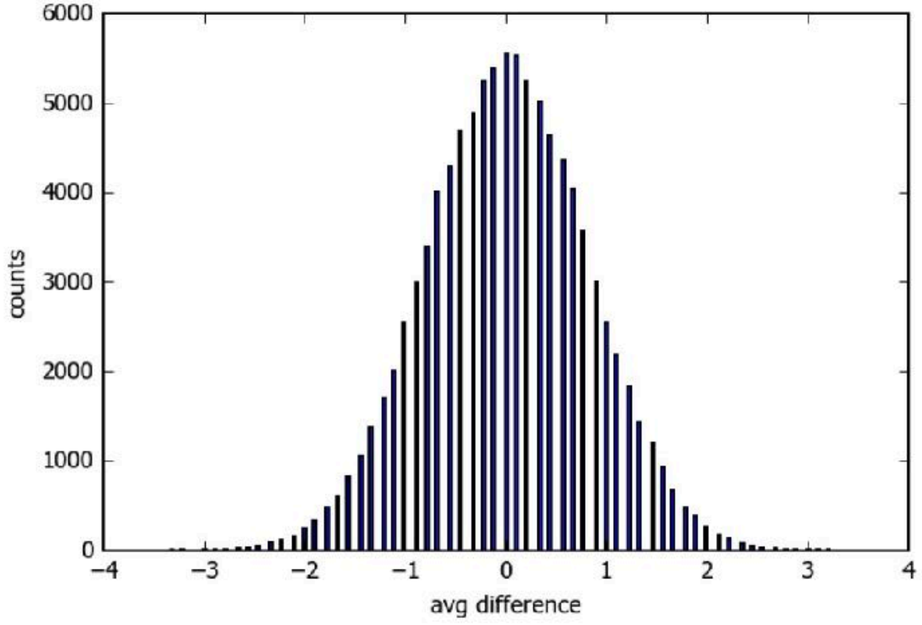
Additionally we can say that, given this, if we take a random sample of the mean of 10 die rolls from die_1, and a random sample of the mean of 10 die rolls from die_2, the difference of those means should itself be normally distributed (thanks CLT!) around 0.

That is, any differeces should cancel out, given enough data.

If, however, one of the die is biased--such as toward 6, in our case--that bias should force the mean of the samples to be slightly *greater* than the expected mean of 3.5, and should force the mean of the *differences* between the die to be slightly greater than 0.

Armed with this, we can now perform a very simple independent samples t-ttest of the distribution of the differences against the assumption that the mean of the differences should be 0.

```python
## get the distribution of the differences of means
chunk_size = 10
samples = num_rolls//chunk_size
roll_diff = np.zeros((samples, 1))

for i in range(samples):
    low_range = i*10
    high_range = low_range+9
    # get the mean of the differences between the die
    clean_mean = np.mean(simulated_rolls[low_range:high_range])
    biased_mean = np.mean(simulated_biased_rolls[low_range:high_range])
    roll_diff[i] = biased_mean-clean_mean

# plot the histogram!
plt.hist(roll_diff, 199)
plt.xlabel('avg difference')
plt.ylabel('counts')
plt.show()
```




```
np.mean(roll_diff)
```

```
0.013315555555555556
```

The distribution of differences between means looks normal, too!

Also note that the mean of these differences is, indeed, slightly greater than 0.

Now we can use a standard t-test to look at how big this difference is, and whether it's "significant".

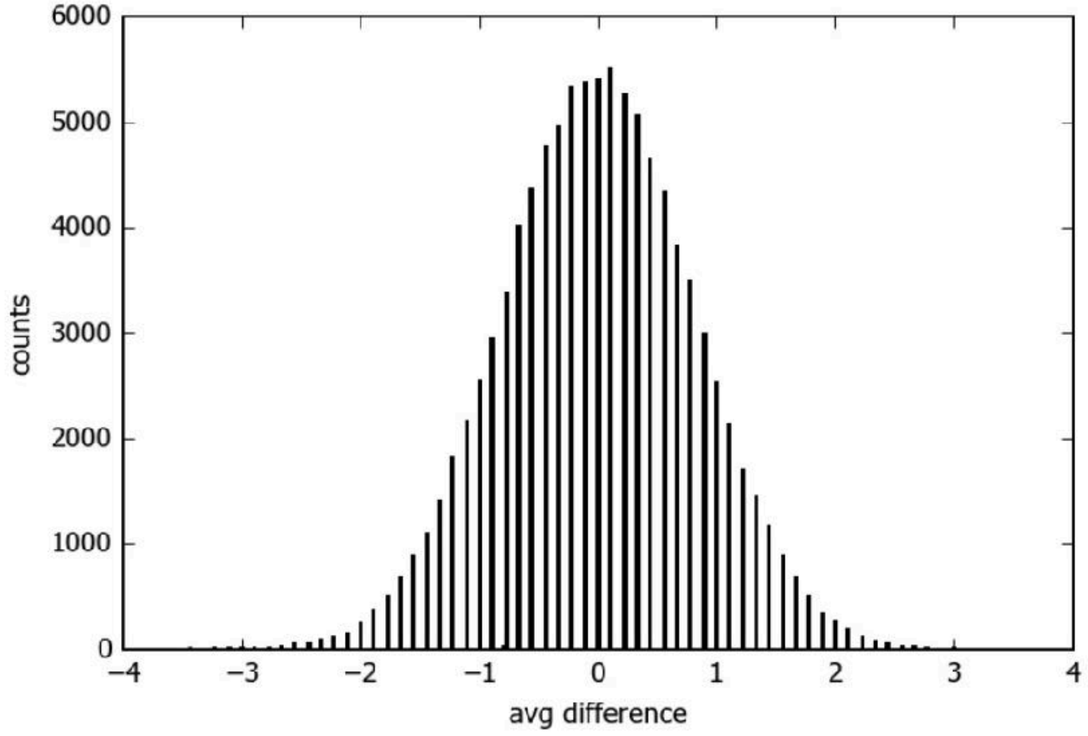
```
# compare our differences against the null, which is a mean of 0
t,p = stats.ttest_1samp(roll_diff, popmean=0)
print(['t-value: ', t[0], '; p-value: ', p[0]])
```

```
['t-value: ', 5.2327376260479124, '; p-value: ', 1.6735361051218107e-07]
```

Yes! It is significant.

And just to sanity check, let's do the same thing, but this time compare our un-biased die against a second simulation of an un-biased die.

```python
# simulate a second, unbiased set of rolls
simulated_rolls_taketwo = np.random.randint(low=1, high=7, size=num_rolls)

chunk_size = 10
samples = num_rolls//chunk_size
unbiased_roll_diff = np.zeros((samples, 1))

for i in range(samples):
    low_range = i*10
    high_range = low_range+9
    # get the mean of the differences between the die
    clean_mean = np.mean(simulated_rolls[low_range:high_range])
    clean_mean_taketwo = np.mean(simulated_rolls_taketwo[low_range:high_range])
    unbiased_roll_diff[i] = clean_mean_taketwo-clean_mean

# plot the histogram!
plt.hist(unbiased_roll_diff, 399)
plt.xlabel('avg difference')
plt.ylabel('counts')
plt.show()

# compare our differences against the null, which is a mean of 0
t,p = stats.ttest_1samp(unbiased_roll_diff, popmean=0)
print(['t-value: ', t[0], '; p-value: ', p[0]])
```




```
['t-value: ', -0.68255648596119545, '; p-value: ', 0.49488871930334943]
```

No difference between them, whatsoever.

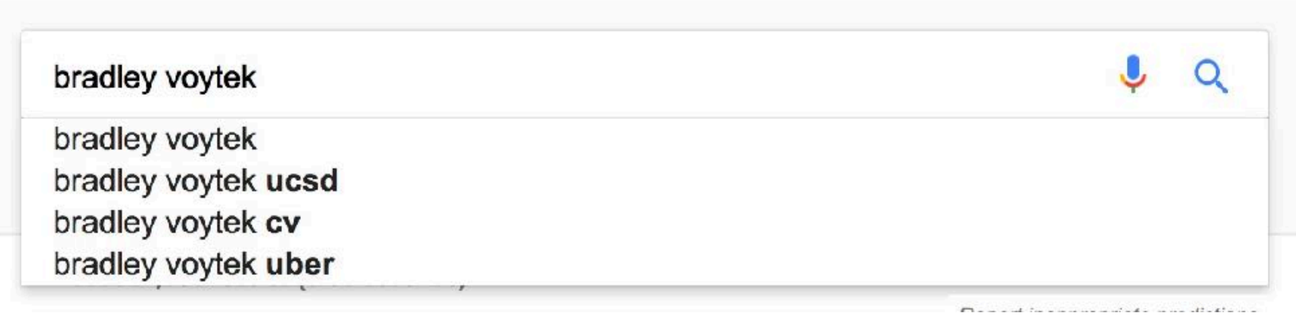
Good job, statistics!

COGS 108
Data Science in Practice

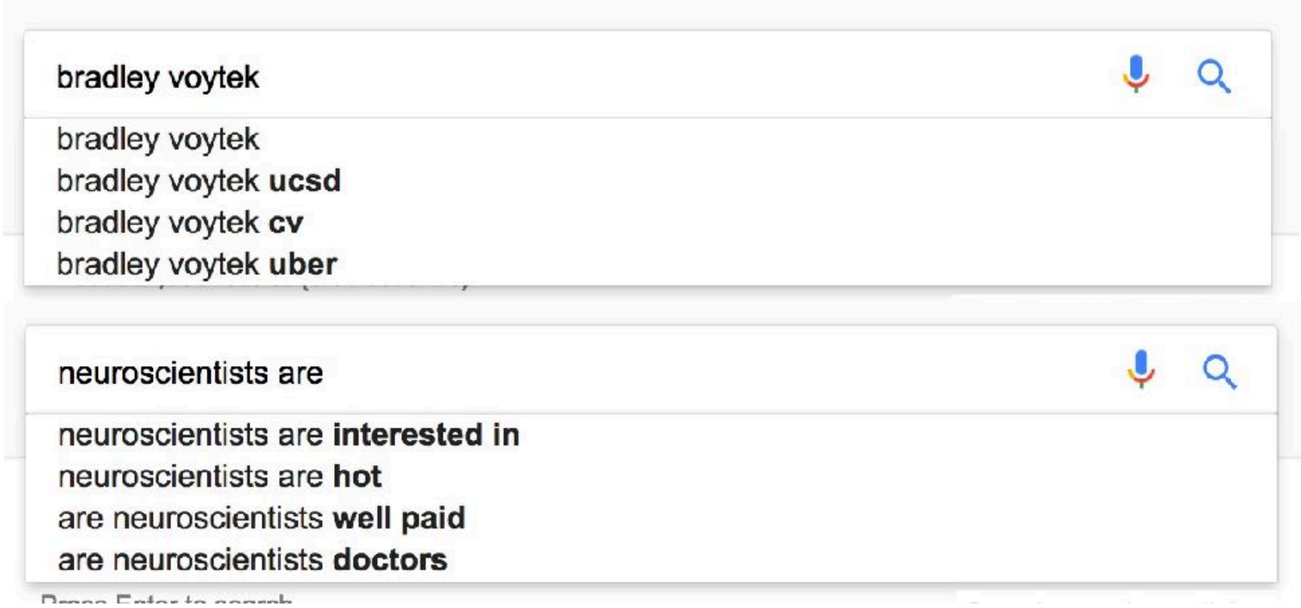
Text mining and NLP

UC San Diego

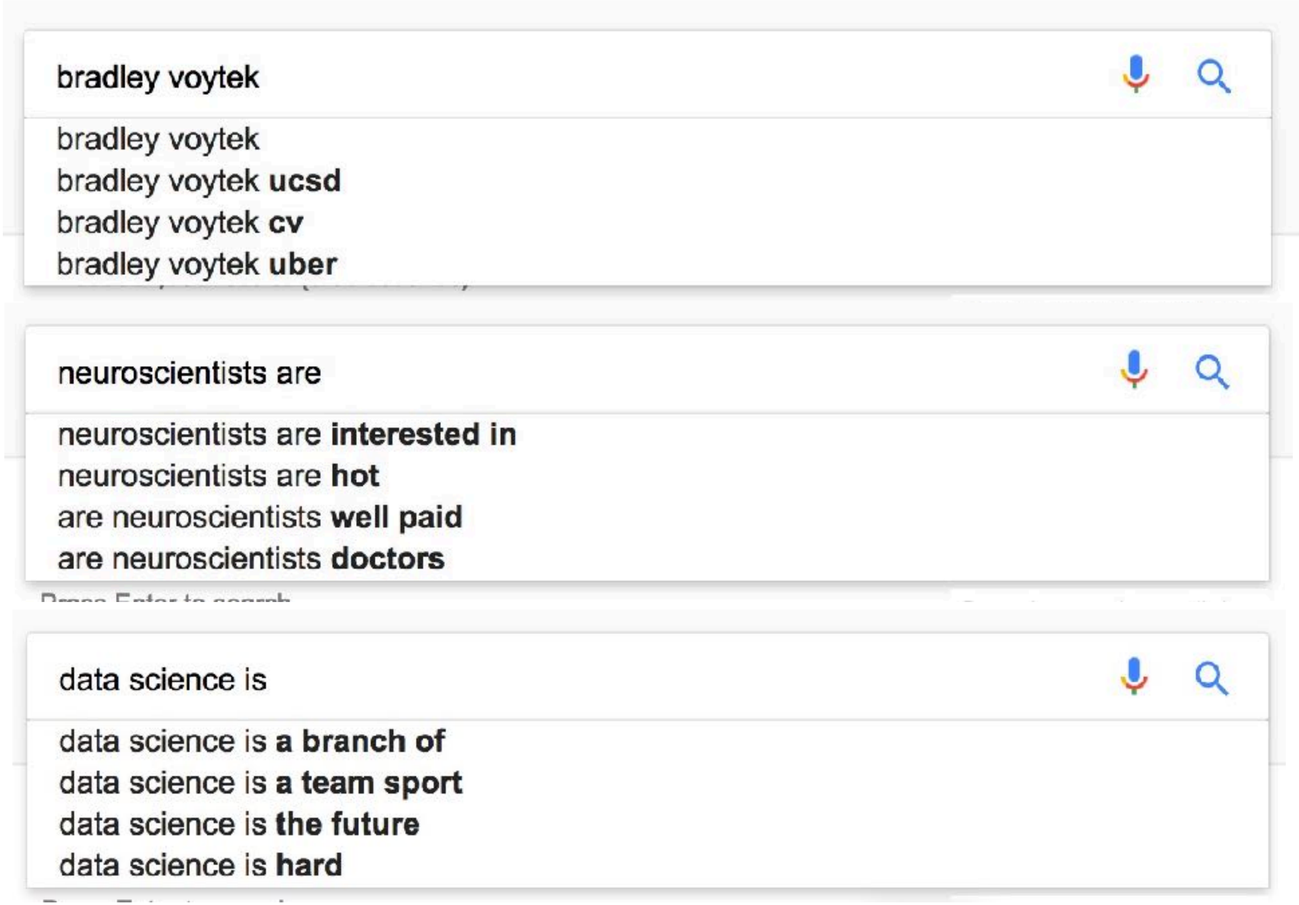
What does text mining get us?



What does text mining get us?



What does text mining get us?



Simple word visualization

Simple word visualization

UC San Diego

Text basics

- cognitive science is the study of cognition
- data science is the use of data to uncover knowledge, including the cognitive

Text basics

- cognitive science is the study of cognition
- data science is the use of data to uncover knowledge, including the cognitive

[“cognitive”, “science”, ”is”, “the”, “study”, “of”, “cognition”, “data”, “use”, “to”, “uncover”, “knowledge”, “including”]

Vector space model

- cognitive science is the study of cognition
- data science is the use of data uncover knowledge, including the cognitive

[“cognitive”, “science”, ”is”, “the”, “study”, “of”, “cognition”, “data”, “use”, “uncover”, “knowledge”, “including”]

[1 1 1 1 1 1 1 0 0 0 0 0]

[1 1 1 2 0 1 0 2 1 1 1 1]

TF-IDF

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF

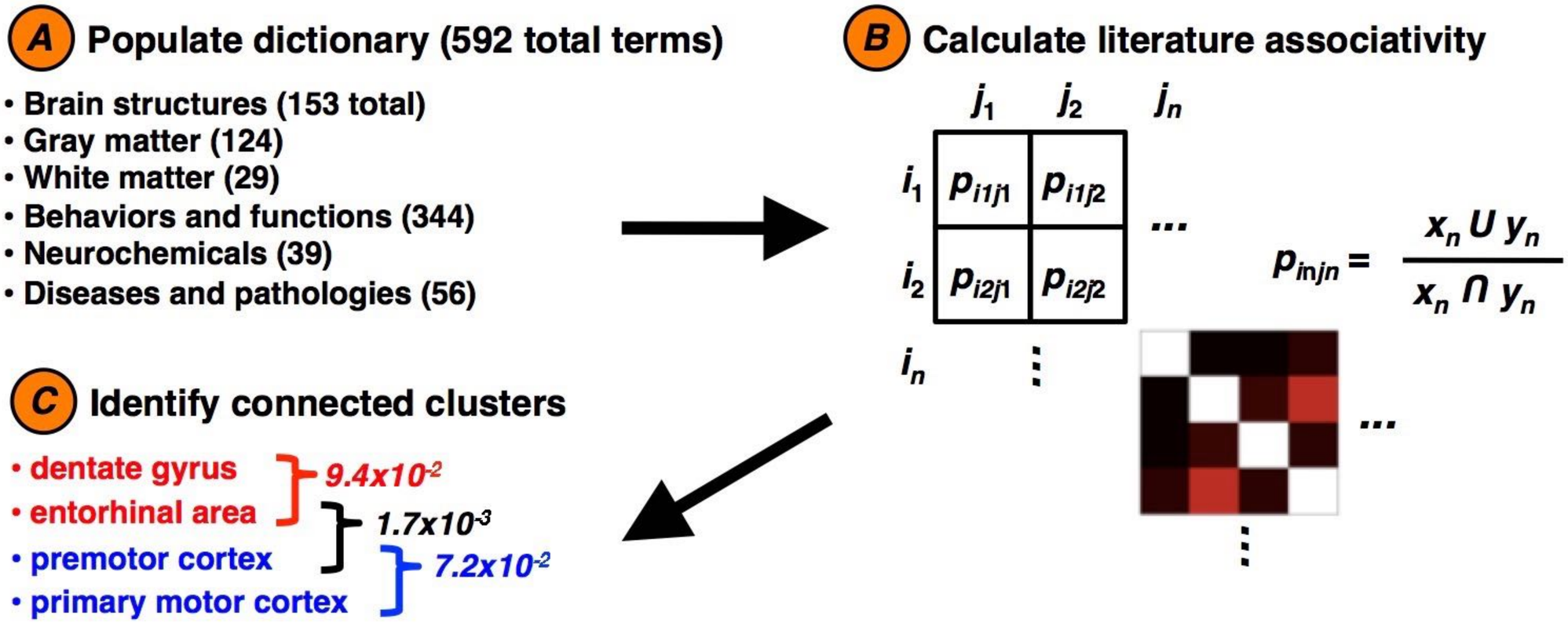
Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

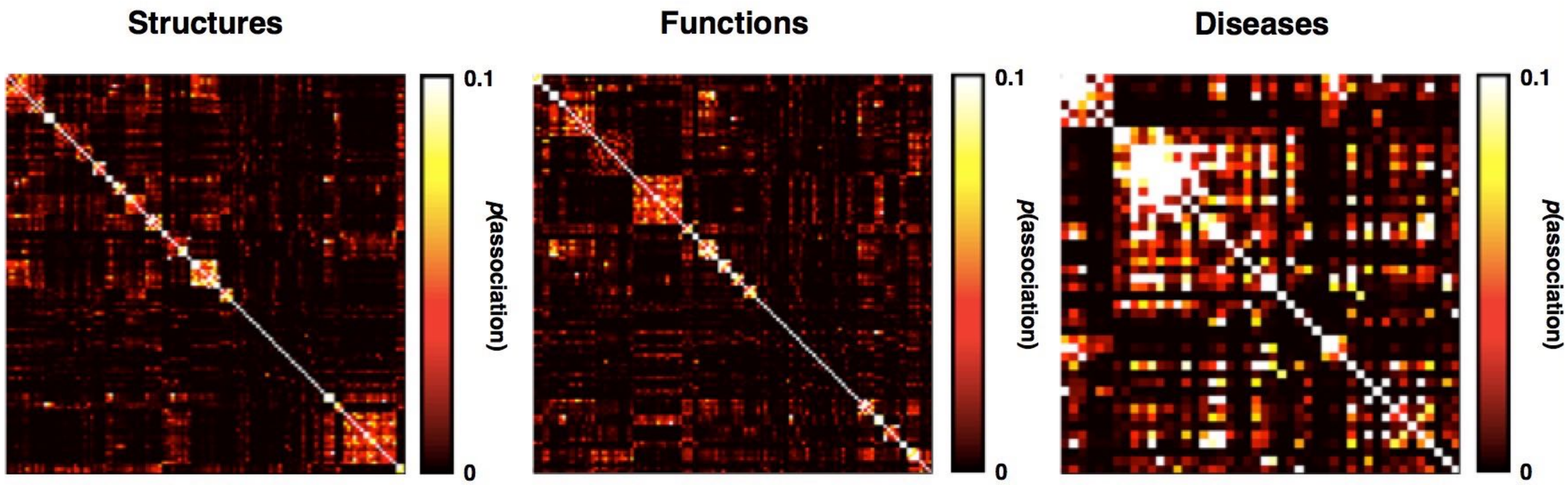
brainSCANr



Source: Voytek & Voytek, *J Neurosci Methods* 2012

UC San Diego

brainSCANr



Source: Voytek & Voytek, *J Neurosci Methods* 2012

brainSCANr

Language comprehension
Language processing
Language production
Lexical processing
Lexical retrieval
Phonological encoding
Picture naming
Semantic processing
Sentence comprehension
Sentence production
Syntactic processing
Word comprehension
Word production

Anxiety
Bipolar disorder
Depression
Obsessive compulsive disorder
Panic disorder
Schizophrenia
Social phobia

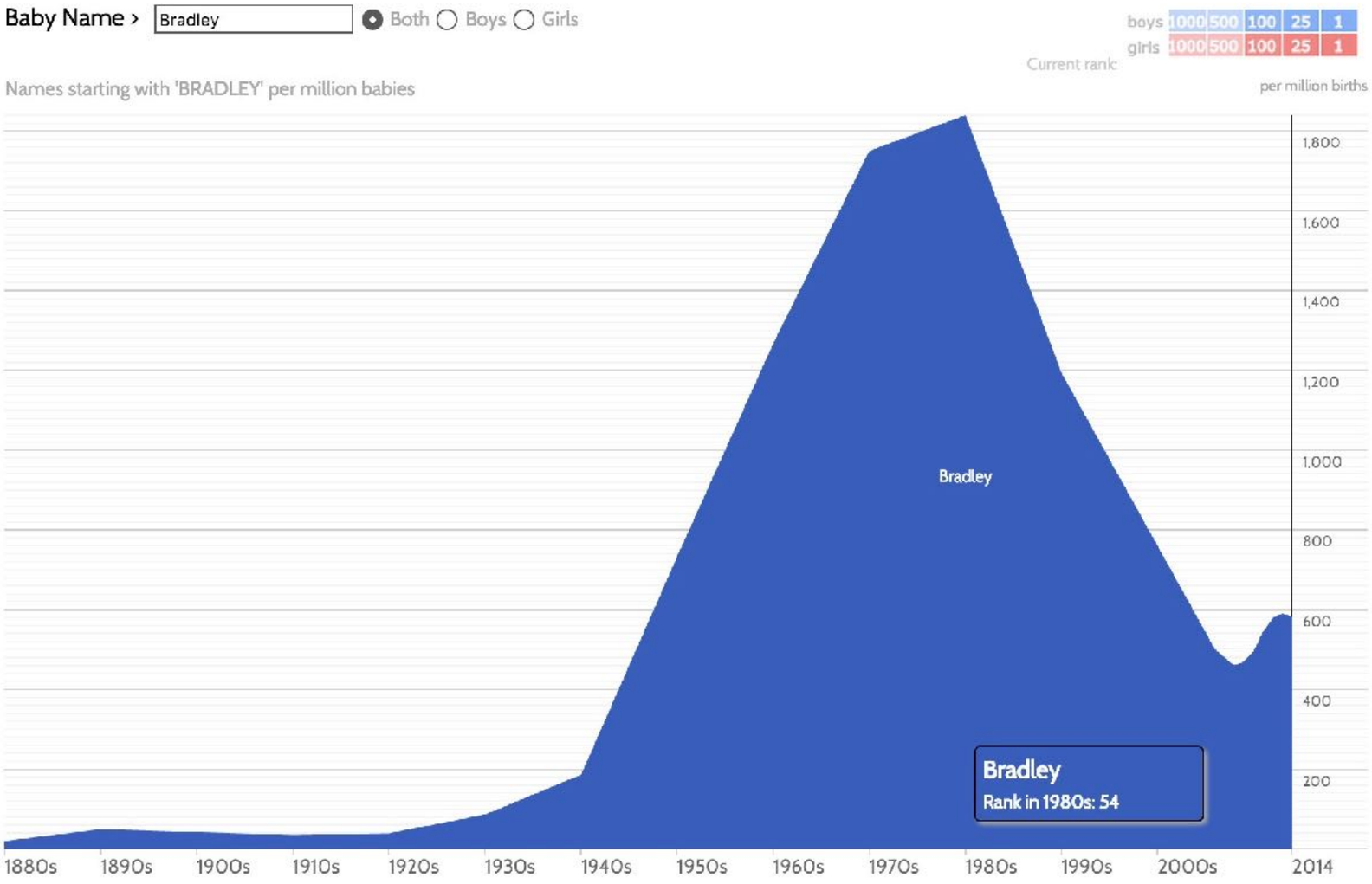
Language comprehension
Sentence comprehension
Syntactic processing
Broca's aphasia
Wernicke's aphasia
Broca's area
Wernicke's area

Parkinson's
Parkinson's disease
Caudate nucleus
Globus pallidus
Putamen
Substantia nigra

UC San Diego

Source: Voytek & Voytek, *J Neurosci Methods* 2012

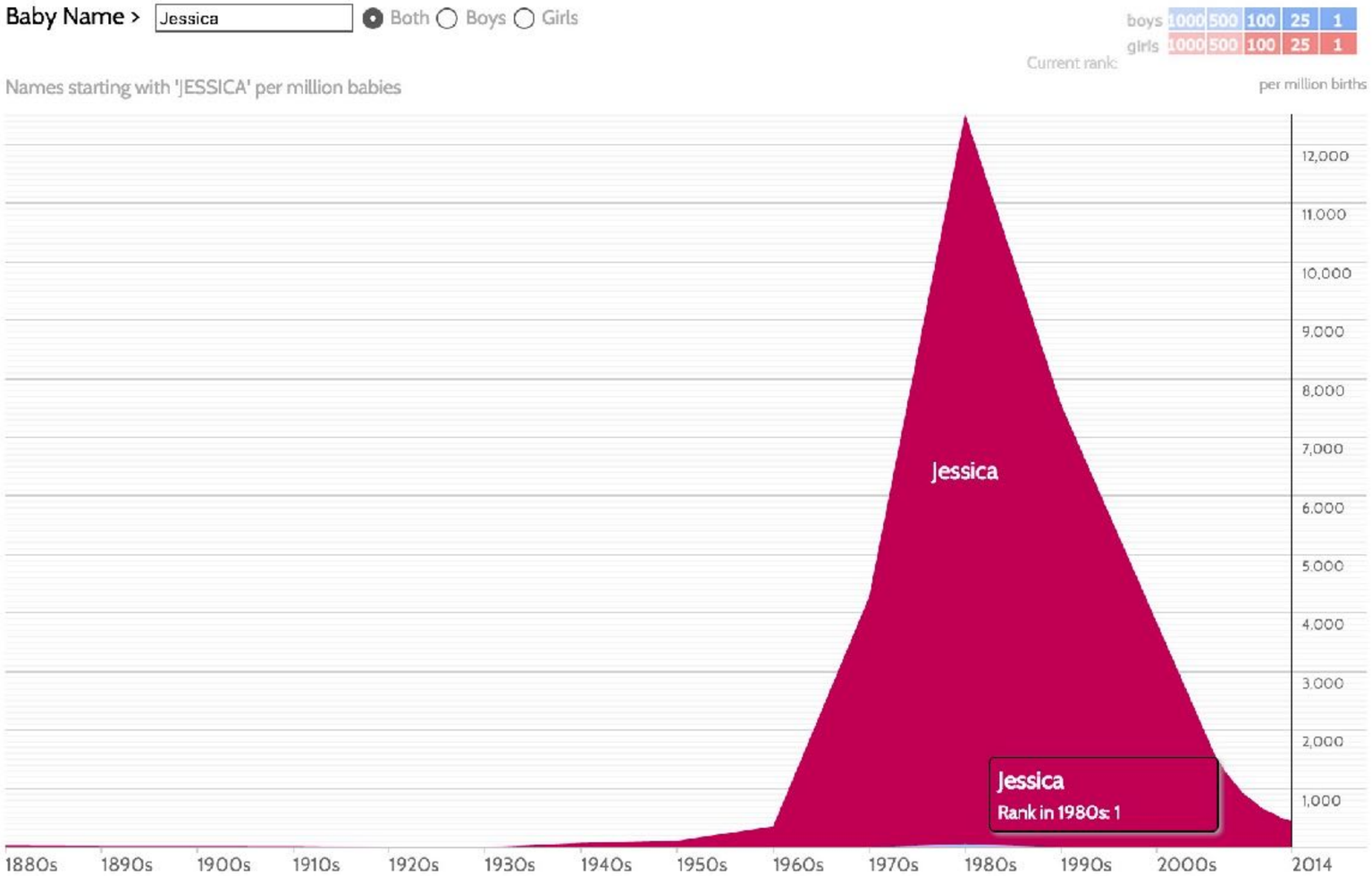
Names over time



Source: babynamewizard.com

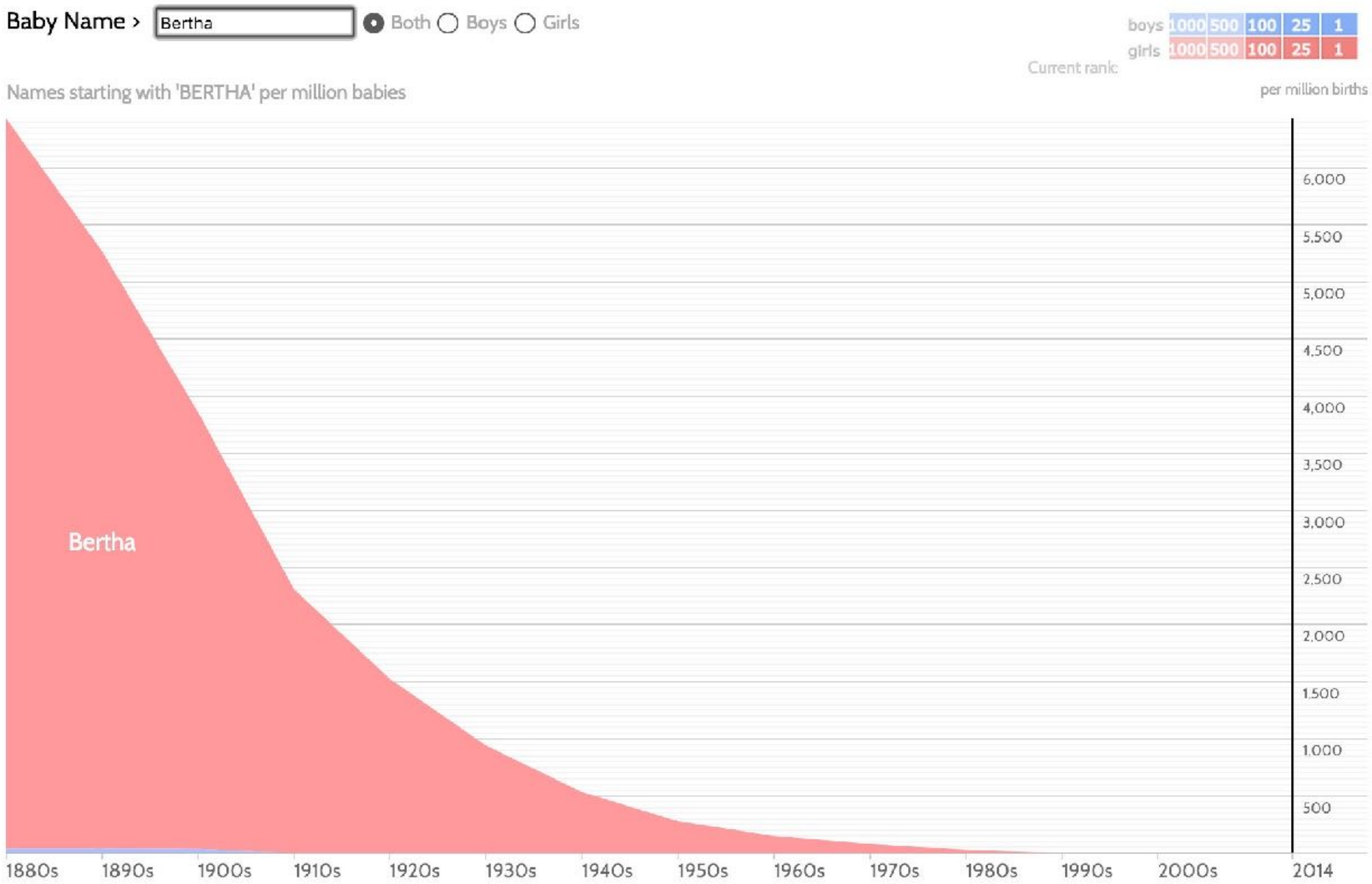
UC San Diego

Names over time



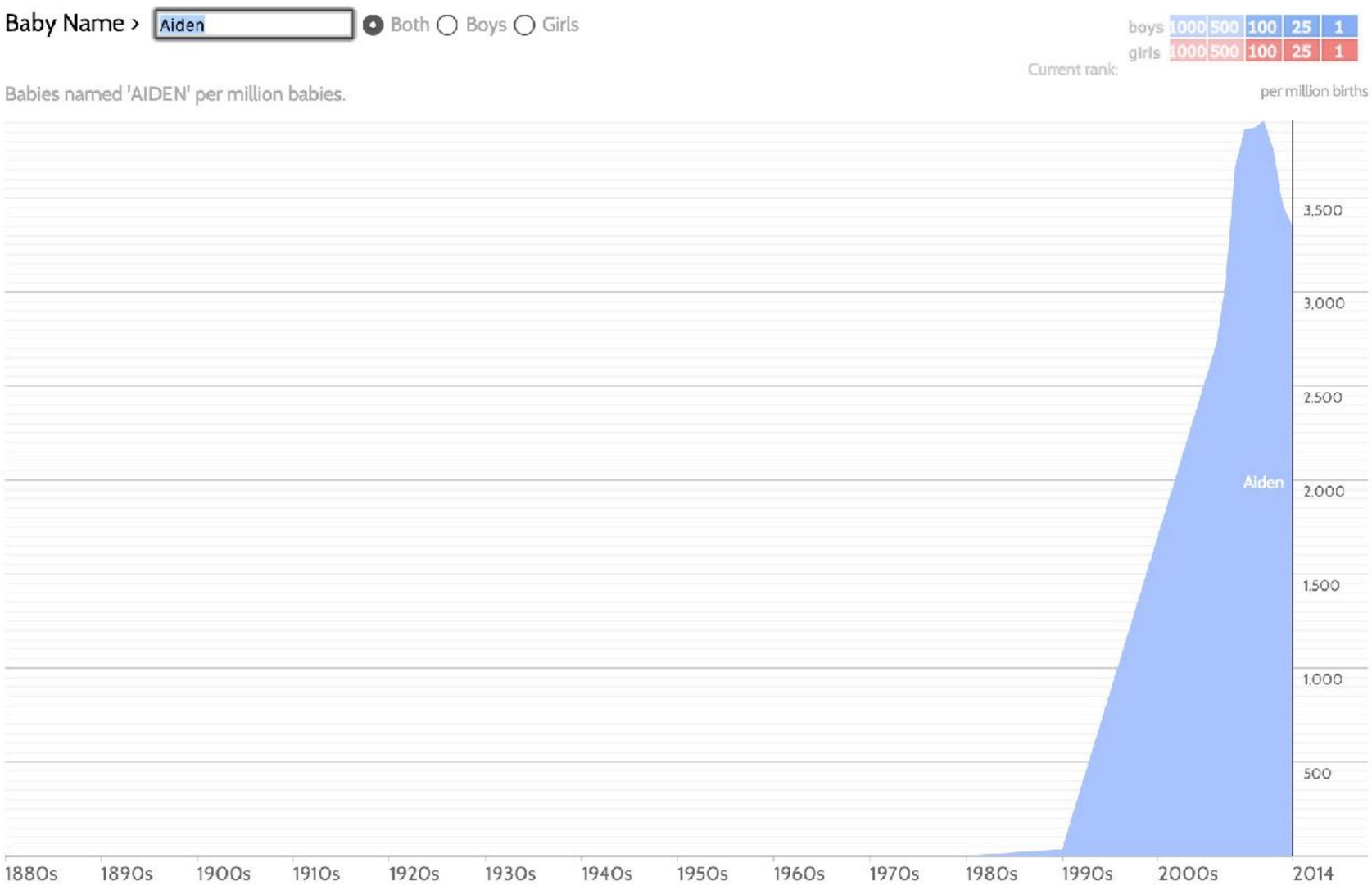
Source: babynamewizard.com

Names over time



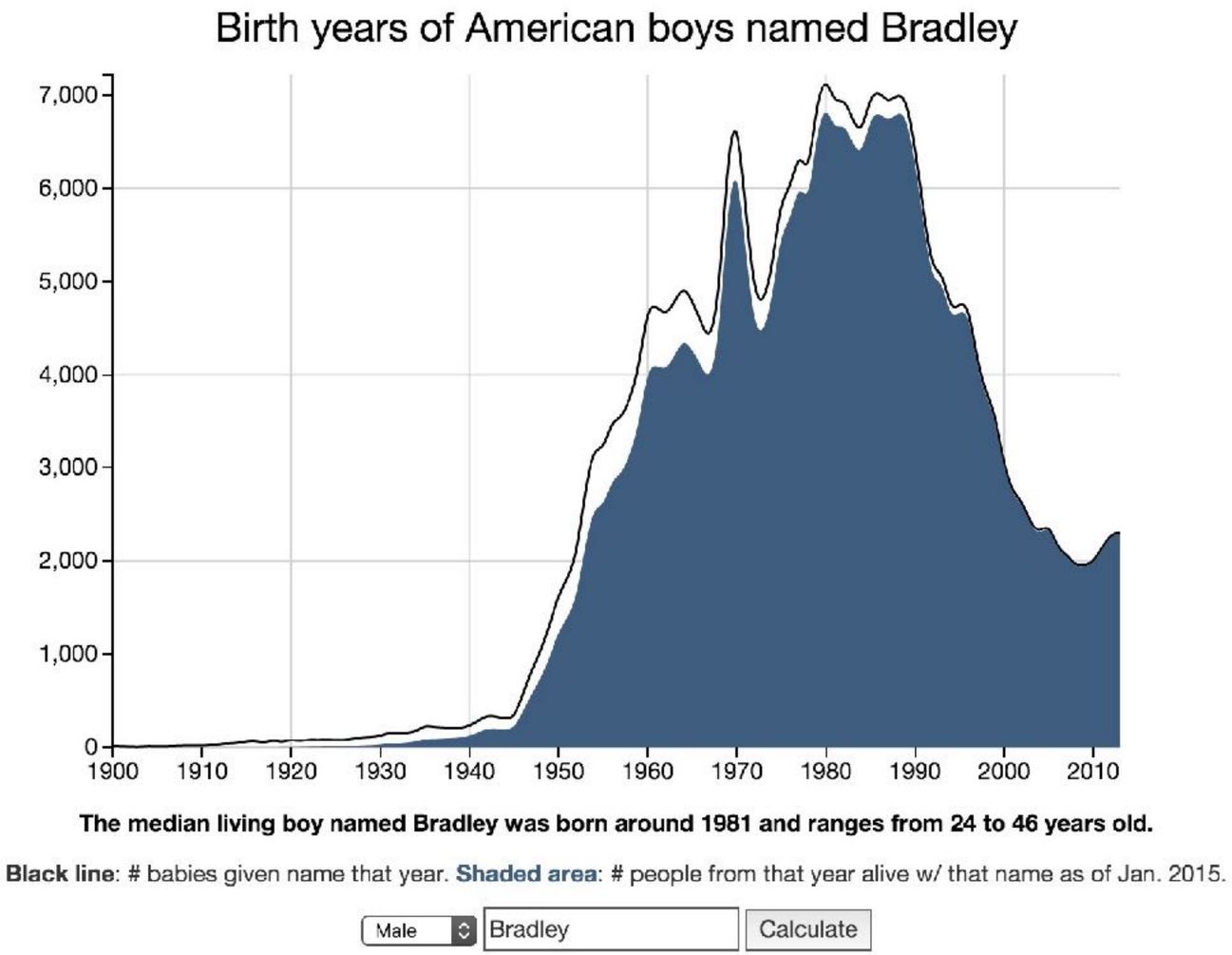
Source: babynamewizard.com

Names over time



Source: babynamewizard.com

Names over time

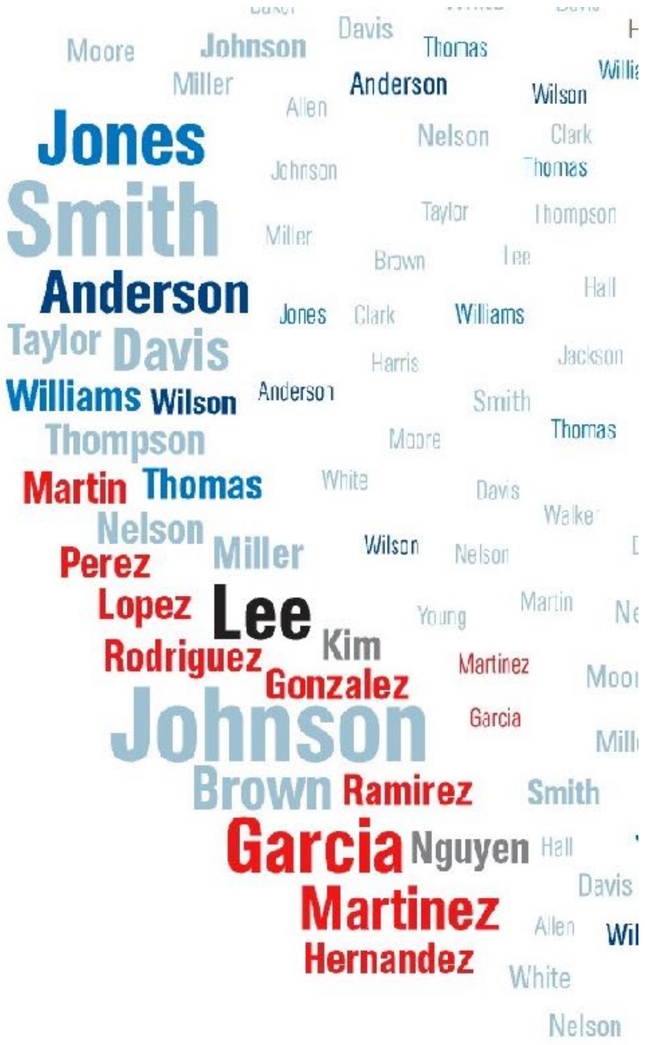


Source: http://rhiever.github.io/

Names over space

Source: *National Geographic*

Names over space



Source: *National Geographic*

Visualizing narrative

Visualization of Narrative Structure

Can books be summarized through their emotional trajectory and character relationships? Can a graphic representation of a book provide an at-a-glance impression and an invitation to explore the details?

We visualized character interactions and relative emotional content for three very different books: a haunting memory play, a metaphysical mood piece, and a children's fantasy classic. A dynamic graph of character relationships displays the evolution of connections between characters throughout the book. Emotional strength and valence of each sentence are shown in a color-coded sentiment plot. Hovering over the sentence bars reveals the text of the original sentences. The emotional path of each character through the book can be traced by clicking on the character names in the graph. This highlights the corresponding sentences in the sentiment plot where that character appears. Click on the links below to see each visualization.

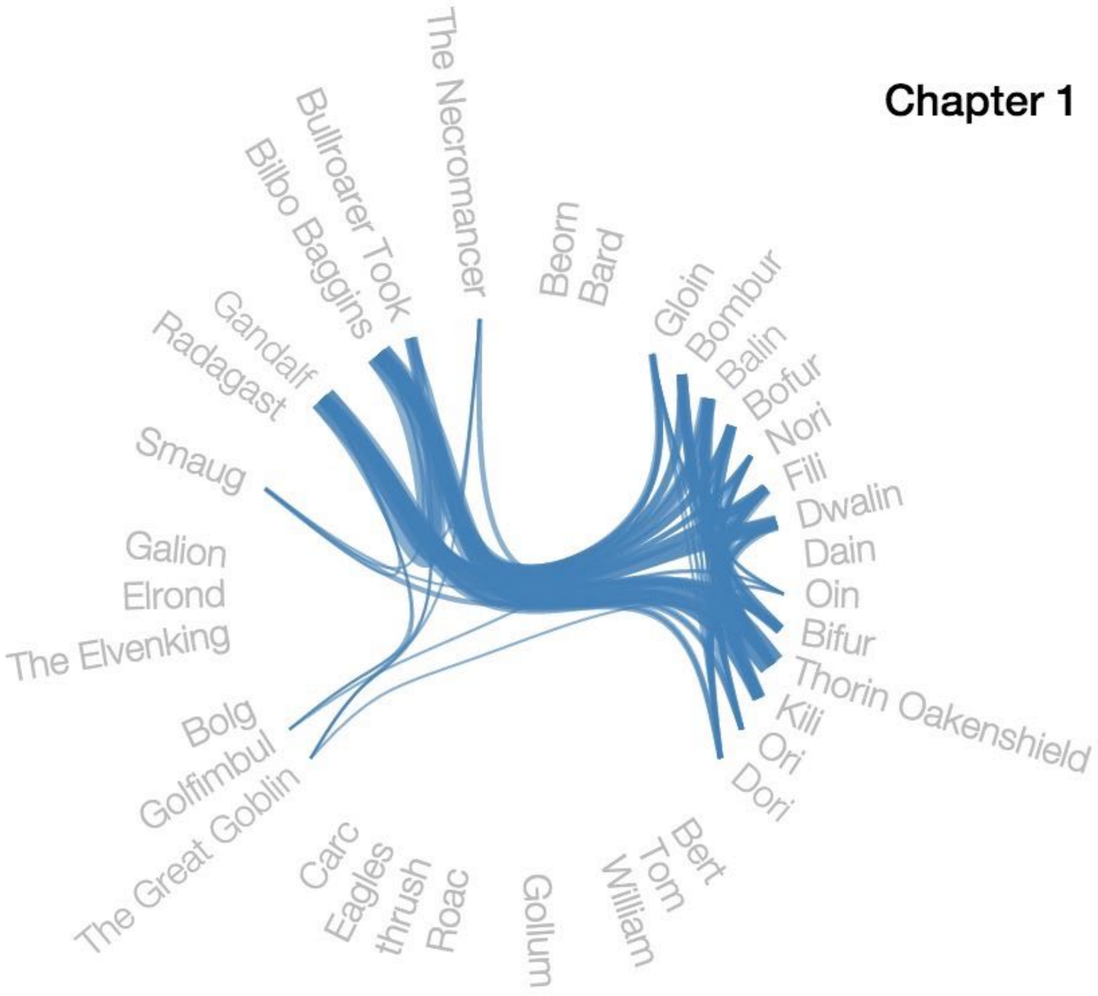
Best viewed in Google Chrome.

Source: Natalia Bilenko & Asako Miyakawa

Visualizing *The Hobbit*

Hover over a character name in a graph.
Yellow links show connections in the selected chapter.
Yellow names show connections in the whole book.

The bars below show emotion intensity for each sentence.
Click on a character in the graph to see where they appear.
Hover over each bar to read the original sentence.



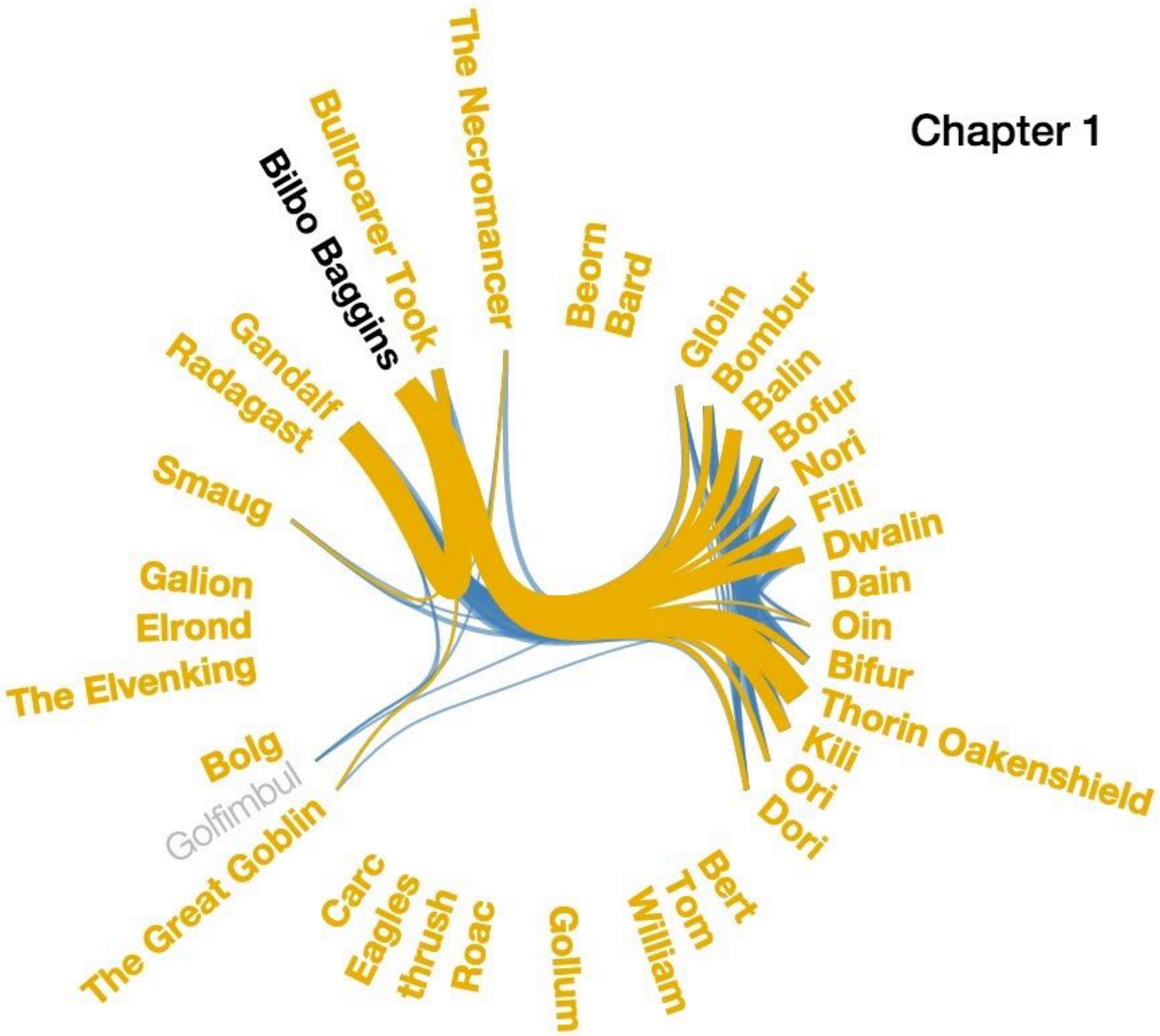
Source: Natalia Bilenko & Asako Miyakawa

UC San Diego

Visualizing *The Hobbit*

Hover over a character name in a graph.
Yellow links show connections in the selected chapter.
Yellow names show connections in the whole book.

The bars below show emotion intensity for each sentence.
Click on a character in the graph to see where they appear.
Hover over each bar to read the original sentence.



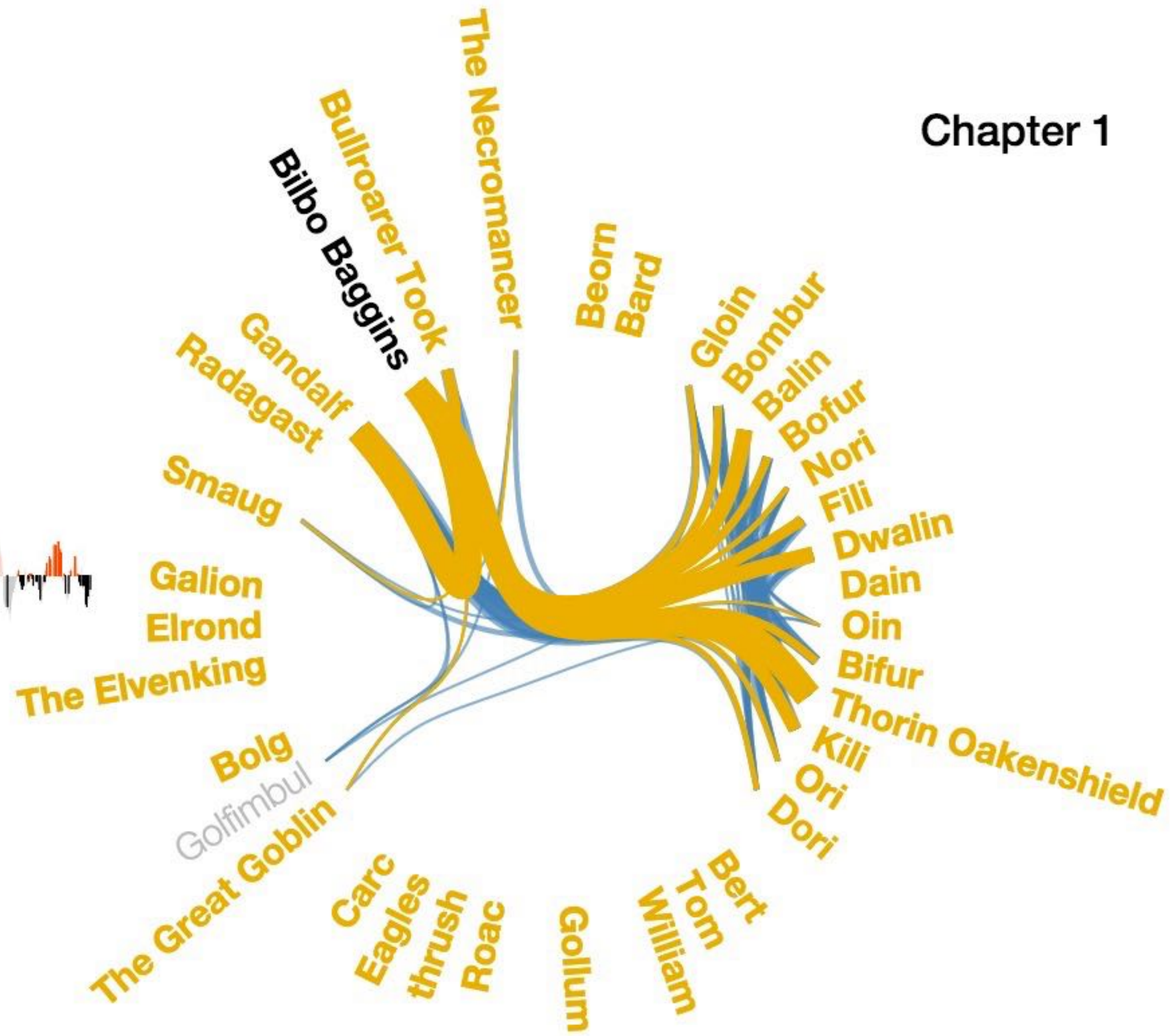
Source: Natalia Bilenko & Asako Miyakawa

UC San Diego

Visualizing *The Hobbit*

Hover over a character name in a graph.
Yellow links show connections in the selected chapter.
Yellow names show connections in the whole book.

The bars below show emotion intensity for each sentence.
Click on a character in the graph to see where they appear.
Hover over each bar to read the original sentence.



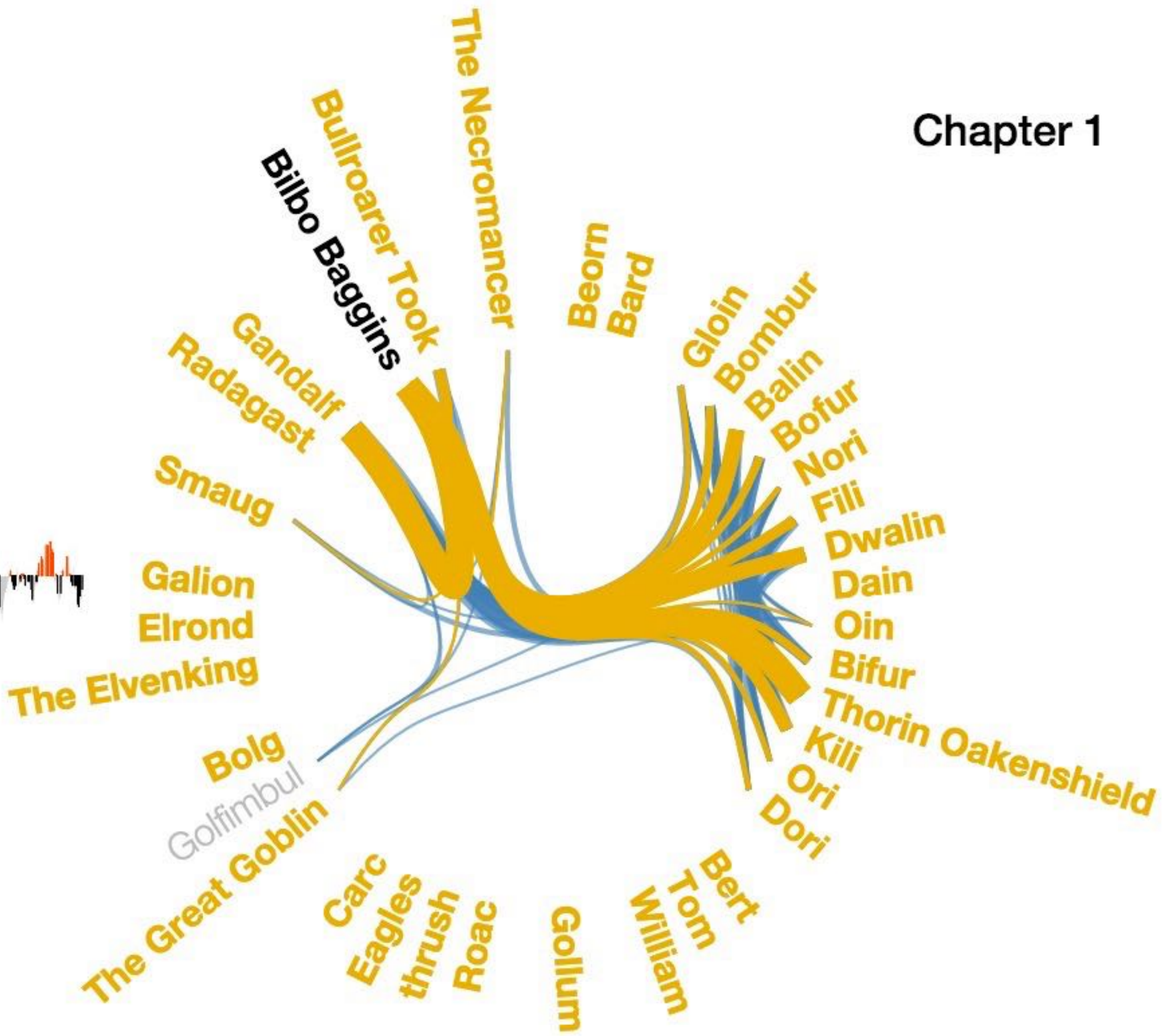
Source: Natalia Bilenko & Asako Miyakawa

UC San Diego

Visualizing *The Hobbit*

Hover over a character name in a graph.
Yellow links show connections in the selected chapter.
Yellow names show connections in the whole book.

The bars below show emotion intensity for each sentence.
Click on a character in the graph to see where they appear.
Hover over each bar to read the original sentence.



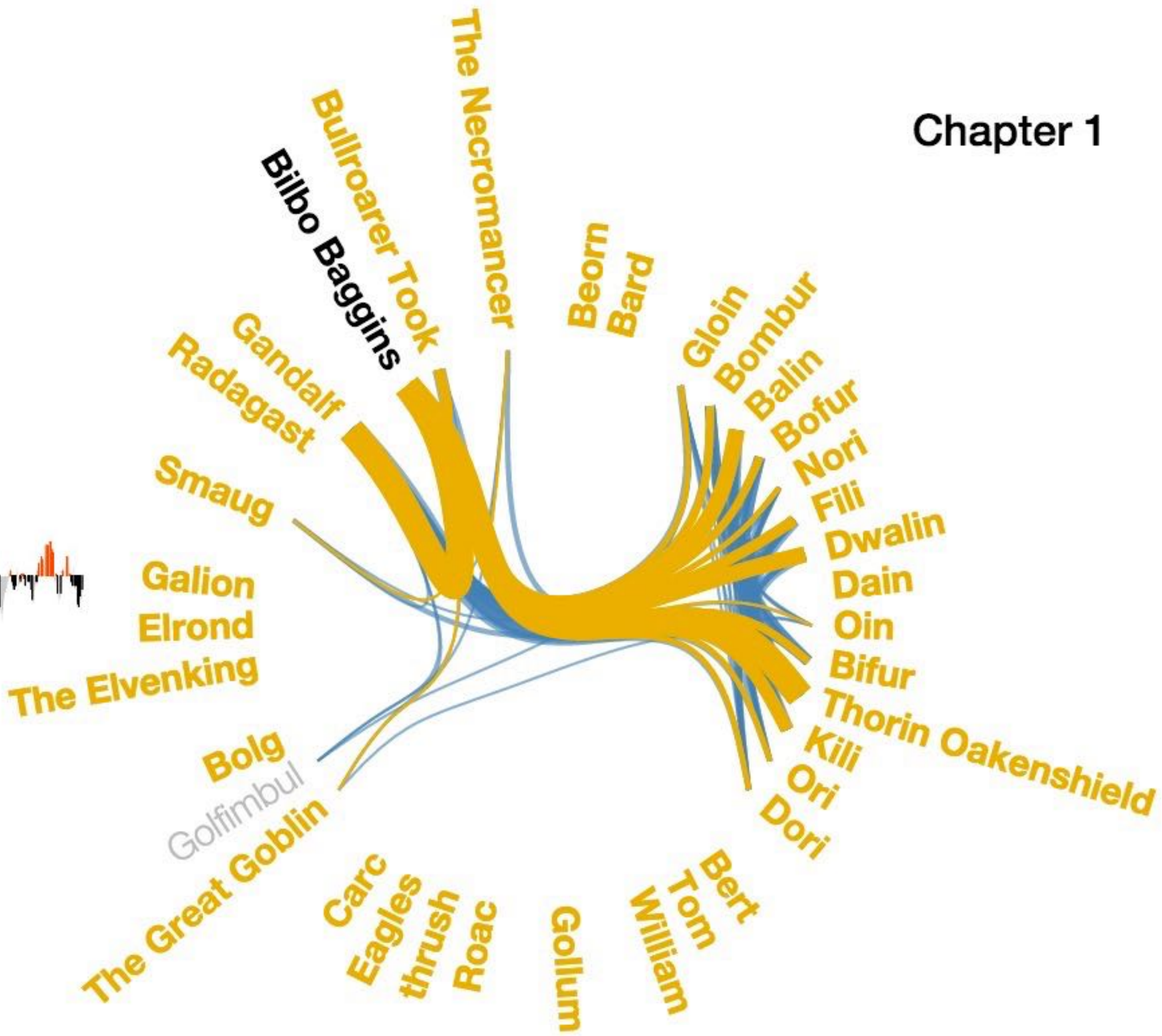
Source: Natalia Bilenko & Asako Miyakawa

UC San Diego

Visualizing *The Hobbit*

Hover over a character name in a graph.
Yellow links show connections in the selected chapter.
Yellow names show connections in the whole book.

The bars below show emotion intensity for each sentence.
Click on a character in the graph to see where they appear.
Hover over each bar to read the original sentence.



Source: Natalia Bilenko & Asako Miyakawa

UC San Diego

Bradley Voytek, Ph.D.

UC San Diego

Cognitive and Neural Dynamics Laboratory

Department of Cognitive Science

Neurosciences Graduate Program

The Institute for Neural Computation

bvoytek@ucsd.edu

@bradleyvoytek