STATS 701
Data Analysis using Python

Lecture 27: APIs and Graph Processing

Some slides adapted from C. Budak

Previously: Scraping Data from the Web

We used BeautifulSoup to process HTML that we read directly

- We had to figure out where to find the data in the HTML
- This was okay for simple things like Wikipedia…
- ...but what about large, complicated data sets?
  - E.g., Climate data from NOAA; Twitter/reddit/etc.; Google maps

Many websites support APIs, which make these tasks simpler

Instead of scraping for what we want, just ask!

Example: ask Google Maps for a computer repair shop near a given address

Three common API approaches

Via a Python package

Service (e.g., Google maps, ESRI*) provides library for querying DB

Example: `from arcgis.gis import GIS`

Via a command-line tool

Example: `twurl` https://developer.twitter.com/

Via HTTP requests

We submit an HTTP request to a server

Supply additional parameters in URL to specify our query

Example: https://www.yelp.com/developers/documentation/v3/business_search

Ultimately, all three of these approaches end up submitting an HTTP request to a server, which returns information in the form of a JSON or XML file, typically.

* ESRI is a GIS service, to which the university has a subscription: https://developers.arcgis.com/python/

Web service APIs

Step 1: Create URL with query parameters

Example (non-working): www.example.com/search?key1=val1&key2=val2

Step 2: Make an HTTP request

Communicates to the server what kind of action we wish to perform

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

Step 3: Server returns a response to your request

May be as simple as a code (e.g., 404 error)...

...but typically a JSON or XML file (e.g., in response to a DB query)

HTTP Requests

Allows a client to ask a server to perform an action on a resource

E.g., perform a search, modify a file, submit a form

Two main parts of an HTTP request:

URI: specifies a resource on the server

Method: specifies the action to be performed on the resource

HTTP request also includes (optional) additional information

E.g., specifying message encoding, length and language

More information:
https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods
RFC specifying HTTP requests: https://tools.ietf.org/html/rfc7231#section-4

HTTP Request Methods

GET: retrieves information from the server

POST: sends information to the serve (e.g., a file for upload)

PUT: replace the URI with a client-supplied file

DELETE: delete the file indicated by the URI

CONNECT: establishes a tunnel (i.e., connection) with the server

More: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

See also **Representational State Transfer**:
https://en.wikipedia.org/wiki/Representational_state_transfer

Refresher: JSON

JavaScript Object Notation

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

Commonly used by website APIs

Basic building blocks:

- attribute–value pairs
- array data

Example (right) from wikipedia:

- Possible JSON representation of a person

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    },
    {
      "type": "mobile",
      "number": "123 456-7890"
    }
  ],
  "children": [],
  "spouse": null
}
```

Python `json` module

```
1 import json
2 json_string = '{"first_name": "Claude", "last_name":"Shannon",\
3                 "alma_mater":"University of Michigan"}'
4 parsed_json = json.loads(json_string)
5 parsed_json
```

```
{u'alma_mater': u'University of Michigan',
 u'first_name': u'Claude',
 u'last_name': u'Shannon'}
```

```
1 json.dumps(parsed_json)
```

```
'{"alma_mater": "University of Michigan", "first_name": "Claude", "last_name": "Shannon"}'
```

JSON string encoding information about information theorist Claude Shannon

`json.loads` parses a string and returns a JSON object.

`json.dumps` turns a JSON object back into a string.

Python `json` module

```
1 parsed_json
```

```
{u'alma_mater': u'University of Michigan',
 u'first_name': u'Claude',
 u'last_name': u'Shannon'}
```

```
1 parsed_json['alma_mater']
```

```
u'University of Michigan'
```

```
1 parsed_json['first_name']
```

```
u'Claude'
```

```
1 parsed_json['middle_name']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-440-a100eb80552a> in <module>()
----> 1 parsed_json['middle_name']

KeyError: 'middle_name'
```

JSON object returned by `json.loads` acts just like a Python dictionary.

Example: Querying Yelp's Business Search Service

I am sitting at my desk, woefully undercaffeinated

I could open a new tab and search for coffee nearby…

...but why leave the comfort of my Jupyter notebook?

Yelp provides several services under their "Fusion API"

https://www.yelp.com/developers/documentation/v3/get_started

We'll use the business search endpoint

Supports queries that return businesses reviewed on Yelp

https://www.yelp.com/developers/documentation/v3/business_search

Example: Querying Yelp's Business Search Service

```
1  import requests
2  url = 'https://api.yelp.com/v3/businesses/search'
3  headers = {'Authorization': 'Bearer %s' % api_key}
4  url_params = {'term': 'coffee', # Search for coffee...
5                'radius': 1000, # ...within 1000 meters...
6                'location': '1085 S. University'} # ...near my office.
7  r = requests.get(url, headers=headers, params=url_params)
8  r.json()
```

URL to which to direct our request, specified in Yelp's documentation.

```
{'businesses': [{'alias': 'comet-coffee-ann-arbor',
   'categories': [{'alias': 'coffee', 'title': 'Coffee & Tea'}],
   'coordinates': {'latitude': 42.2784368818938,
    'longitude': -83.7414034863831},
   'display_phone': '(734) 222-0579',
   'distance': 508.08706397561326,
```

Documentation: https://www.yelp.com/developers/documentation/v3/business_search

Example: Querying Yelp's Business Search Service

```
1  import requests
2  url = 'https://api.yelp.com/v3/businesses/search'
3  headers = {'Authorization': 'Bearer %s' % api_key}
4  url_params = {'term': 'coffee', # Search for coffee...
5                'radius': 1000, # ...within 1000 meters...
6                'location': '1085 S. University'} # ...near my office.
7  r = requests.get(url, headers=headers, params=url_params)
8  r.json()
```

```
{'businesses': [{'alias': 'comet-coffee-ann-arbor',
   'categories': [{'alias': 'coffee', 'title': 'Coffee
   'coordinates': {'latitude': 42.2784368818938,
    'longitude': -83.7414034863831},
   'display_phone': '(734) 222-0579',
   'distance': 508.08706397561326,
```

Yelp requires that we obtain an API key to use for authentication. You must register with Yelp to obtain such a key.

Documentation: https://www.yelp.com/developers/documentation/v3/business_search

Example: Querying Yelp's Business Search Service

```
1  import requests
2  url = 'https://api.yelp.com/v3/businesses/search'
3  headers = {'Authorization': 'Bearer %s' % api_key}
4  url_params = {'term': 'coffee', # Search for coffee
5                'radius': 1000, # ...within 1000 mete
6                'location': '1085 S. University'} # ...near my office.
7  r = requests.get(url, headers=headers, params=url_params)
8  r.json()
```

We are going to pass a dictionary of parameter values for `requests` to use in constructing a GET request for us.

The resulting URL looks like this (can be access with `r.url`):
https://api.yelp.com/v3/businesses/search?term=coffee&radius=1000&location=1085+S.+University
Notice that if you try to follow that link, you'll get an error asking for an authentication token.

```
'longitude': -83.741034863831},
'display_phone': '(734) 222-0579',
'distance': 508.08706397561326,
```

Documentation: https://www.yelp.com/developers/documentation/v3/business_search

Example: Querying Yelp's Business Search Service

```
1  import requests
2  url = 'https://api.yelp.com/v3/businesses/search'
3  headers = {'Authorization': 'Bearer %s' % api_key}
4  url_params = {'term': 'coffee', # Search for coffee...
5                'radius': 1000, # ...within 1000 meters...
6                'location': '1085 S. University'} # ...near my office.
7  r = requests.get(url, headers=headers, params=url_params)
8  r.json()
```

```
{'businesses': [{'alias': 'comet-coffee-a
   'categories': [{'alias': 'coffee', 'ti
   'coordinates': {'latitude': 42.2784368
    'longitude': -83.7414034863831},
   'display_phone': '(734) 222-0579',
   'distance': 508.08706397561326,
```

This line actually submits the GET request to the URL, and includes the authorization header and our search parameters. `requests` handles all the annoying formatting and construction of the HTTP request for us.

Documentation: https://www.yelp.com/developers/documentation/v3/business_search

Example: Querying Yelp's Business Search Service

```
1  import requests
2  url = 'https://api.yelp.com/v3/businesses/search'
3  headers = {'Authorization': 'Bearer %s' % api_key}
4  url_params = {'term': 'coffee', # Search for coffee...
5                'radius': 1000, # ...within 1000 meters...
6                'location': '1085 S. University'} # ...near my office.
7  r = requests.get(url, headers=headers, params=url_params)
8  r.json()
```

```
{'businesses': [{'alias': 'comet-cof
   'categories': [{'alias': 'coffee'
   'coordinates': {'latitude': 42.27
    'longitude': -83.7414034863831},
   'display_phone': '(734) 222-0579'
   'distance': 508.08706397561326,
```

`requests` packages up the JSON object returned by Yelp, if we ask for it. Recall that we can naturally go back and forth between JSON formatted files and dictionaries, so it makes sense that `r.json()` is a dictionary.

Documentation: https://www.yelp.com/developers/documentation/v3/business_search

```
1  r = requests.get(url, headers=headers, params=url_params)
2  [res['alias'] for res in r.json()['businesses']]
```

```
['comet-coffee-ann-arbor',
 'lab-ann-arbor',
 'mighty-good-coffee-ann-arbor',
 'roasting-plant-coffee-ann-arbor',
 'the-common-cup-ann-arbor',
 'pastry-peddler-bakery-and-cafe-ann-arbor',
 'freds-ann-arbor',
 'espresso-royale-ann-arbor-6',
 'starbucks-ann-arbor-3',
 'sweetwaters-coffee-and-tea-ann-arbor-4',
 'biggby-coffee-ann-arbor-6',
 'berts-cafe-ann-arbor',
 'espresso-royale-ann-arbor-5',
 'elixir-vitae-coffee-and-tea-ann-arbor',
 'starbucks-ann-arbor-14',
 'zingermans-next-door-ann-arbor',
 'kirkland-and-ellis-café-ann-arbor',
 'insomnia-cookies-ann-arbor',
 'sweeting-ann-arbor',
 'starbucks-ann-arbor-19']
```

The `businesses` attribute of the JSON object returned by Yelp is a list of dictionaries, one dictionary per result. The name of each business is stored in its `alias` key.

See Yelp's documentation for more information on the structure of the returned JSON object. https://www.yelp.com/developers/documentation/v3/business_search

More interesting API services

National Oceanic and Atmospheric Administration (NOAA)
https://www.ncdc.noaa.gov/cdo-web/webservices/v2

ESRI ArcGIS
https://developers.arcgis.com/python/

MediaWiki (includes API for accessing Wikipedia pages)
https://www.mediawiki.org/wiki/API:Main_page

Open Movie Database (OMDb)
https://omdbapi.com/

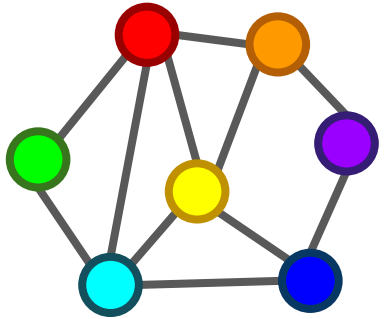
Major League Baseball
http://statsapi.mlb.com/docs

Of course, these are just examples. Just about every large tech company provides an API, as do most groups/agencies that collect data.

Graphs: a Brief Introduction

A **graph** (also called a **network**) is:

- a set of **nodes** (also called **vertices**) V
- And a set of **edges** (also called **links**) E
  - each edge is a pair of nodes from V
- Edges may be **directed** or **undirected**



An undirected graph on seven vertices.

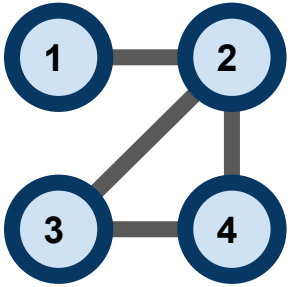
Very useful for representing phenomena in the real world

- Food webs, neurons, gene regulation, social networks, politics, WWW…
- Any situation with actors/objects interacting with one another

Graphs are fundamentally different from relational data

- vertex and edge structure makes it hard to represent graph as a “table”

Representing Graphs



The three structures below all encode this undirected graph.

Adjacency matrix

- Each vertex gets a row in a matrix A
- A_{ij} is 1 if edge from node i to node j, 0 otherwise
- A is symmetric when graph is undirected

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Edge list

- Graph represented by a list of its edges
- (i,j) appears iff there is an edge from node i to node j

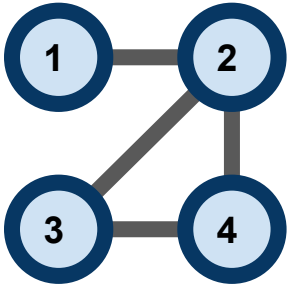
```
[(1,2),(2,3),
    (2,4),(3,4)]
```

Edge “dictionary”

- Graph is stored as a dictionary-lke structure
- Each vertex is a key, value is a list of its neighbors

```
{1:[2],2:[1,3,4],
  3:[3,4], 4:[2,3]}
```

Representing Graphs



The three structures below all encode this undirected graph.

Adjacency matrix

- Each vertex gets a row in a matrix A
- A_{ij} is 1 if edge from node i to node j, 0 otherwise
- A is symmetric when graph is undirected

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Which of these (or other) representations to use will depend on the task. Each representation has its own advantages and disadvantages with respect to memory usage and speed of access.

Edge list

- Graph represented by a list of its edges
- (i,j) appears iff there is an edge from node i to node j

Edge “dictionary”

- Graph is stored as a dictionary-lke structure
- Each vertex is a key, value is a list of its neighbors

```
{1:[2],2:[1,3,4],
 3:[3,4], 4:[2,3]}
```

Graphs in Python: `igraph`

`igraph` is a set of tools for performing graph processing and analysis

- Implementations in Python, R and C

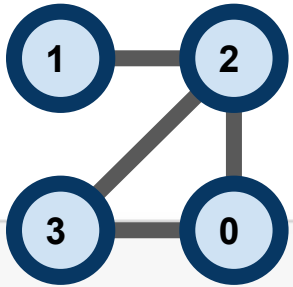
Includes functions for generating graphs from several common models

- Including the stochastic block model and preferential attachment model
- https://en.wikipedia.org/wiki/Stochastic_block_model
- https://en.wikipedia.org/wiki/Preferential_attachment

Includes graph visualization tools

- see also the `graphviz` package, https://pypi.python.org/pypi/graphviz

Using `igraph`



I've changed vertex 4 to vertex 0 because `igraph` numbers the vertices from 0 by default.

```
1  import igraph as ig
2  g = ig.Graph()
3  g
```

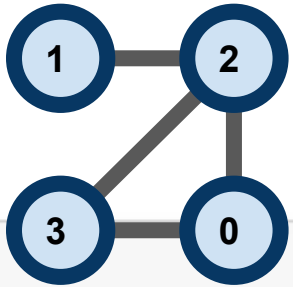
```
<igraph.Graph at 0x10757ed68>
```

Create an `igraph` object representing a graph. Initially, it is an empty graph (no nodes, no edges).

```
1  g.add_vertices(4)
2  g.add_edges([(1,2),(2,3),(2,0),(3,0)])
3  print(g)
```

```
IGRAPH U--- 24 20 --
+ edges:
1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2
0--3 1--2 2--3 0--2 0--3
```

Using `igraph`



I've changed vertex 4 to vertex 0 because `igraph` numbers the vertices from 0 by default.

```
1  import igraph as ig
2  g = ig.Graph()
3  g
```

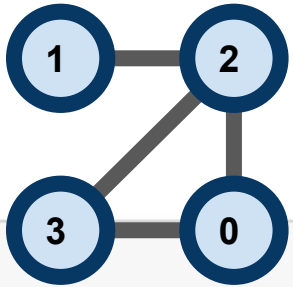
```
<igraph.Graph at 0x10757ed68>
```

`add_vertices` method adds vertices to the graph. Argument specifies number of vertices to add. By default, vertices are numbered sequentially in the order that they are created.

```
1  g.add_vertices(4)
2  g.add_edges([(1,2),(2,3),(2,0),(3,0)])
3  print(g)
```

```
IGRAPH U--- 24 20 --
+ edges:
1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2
0--3 1--2 2--3 0--2 0--3
```

Using `igraph`



I've changed vertex 4 to vertex 0 because `igraph` numbers the vertices from 0 by default.

```
1  import igraph as ig
2  g = ig.Graph()
3  g
```

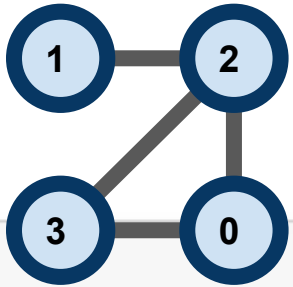
```
<igraph.Graph at 0x10757ed68>
```

```
1  g.add_vertices(4)
2  g.add_edges([(1,2),(2,3),(2,0),(3,0)])
3  print(g)
```

Add edges to the graph. Edges are specified by pairs, with no attention to order. Trying to add an edge to a vertex that doesn't exist in the graph is an error.

```
IGRAPH U--- 24 20 --
+ edges:
1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2
0--3 1--2 2--3 0--2 0--3
```

Using `igraph`



I've changed vertex 4 to vertex 0 because `igraph` numbers the vertices from 0 by default.

```
1  import igraph as ig
2  g = ig.Graph()
3  g
```

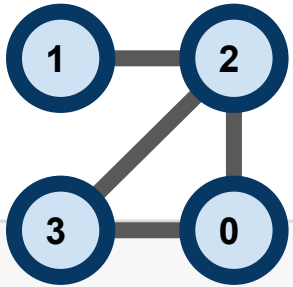
```
<igraph.Graph at 0x10757ed68>
```

```
1  g.add_vertices(4)
2  g.add_edges([(1,2),(2,3),(2,0),(3,0)])
3  print(g)
```

By default, we create an undirected graph. Directed graphs are also available by specifying `directed=True` in `ig.Graph()` initialization.

```
IGRAPH U--- 24 20 --
+ edges:
1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2
0--3 1--2 2--3 0--2 0--3
```

Using `igraph`



I've changed vertex 4 to vertex 0 because `igraph` numbers the vertices from 0 by default.

```
1  import igraph as ig
2  g = ig.Graph()
3  g
```

```
<igraph.Graph at 0x10757ed68>
```

```
1  g.add_vertices(4)
2  g.add_edges([(1,2),(2,3),(2,0),(3,0)])
3  print(g)
```

```
IGRAPH U--- 24 20 --
+ edges:
1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2 0--3 1--2 2--3 0--2
0--3 1--2 2--3 0--2 0--3
```

You can check that this encodes the graph above.

Generating graphs

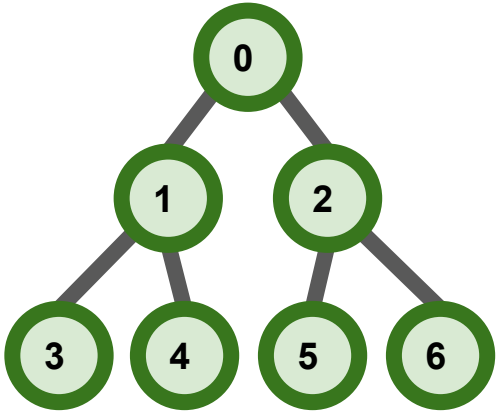
`igraph` includes functions for generating deterministic graphs, such as k-regular trees (i.e., a tree in which every non-leaf has the same number of children).

```
1  g = ig.Graph.Tree(7,2)
2  print(g)
```

```
IGRAPH U--- 7 6 --
+ edges:
0--1 0--2 1--3 1--4 2--5 2--6
```

```
1  geom = ig.Graph.GRG(100,0.2)
2  geom.summary()
```

```
'IGRAPH U--- 100 484 -- \n+ attr: x (v), y (v)'
```



Generating graphs

```
1  g = ig.Graph.Tree(7,2)
2  print(g)
```

```
IGRAPH U--- 7 6 --
+ edges:
0--1 0--2 1--3 1--4 2--5 2--6
```

`igraph` also includes functions for generating random graphs, such as this geometric random graph, in which each node has a corresponding (random) point in the unit square, and nodes form edges if and only if their positions are within some radius of one another.

```
1  geom = ig.Graph.GRG(100,0.2)
2  geom.summary()
```

```
'IGRAPH U--- 100 484 -- \n+ attr: x (v), y (v)'
```

This generates a geometric random graph on n=100 nodes with radius 0.2.

Generating graphs

`igraph` also includes functions for generating random graphs, such as this geometric random graph, in which each node has a corresponding (random) point in the unit square, and nodes form edges if and only if their positions are within some radius of one another.

```
1  g = ig.Graph.Tree(7,2)
2  print(g)
```

```
IGRAPH U--- 7 6 --
+ edges:
0--1 0--2 1--3 1--4 2--5 2--6
```

```
1  geom = ig.Graph.GRG(100,0.2)
2  geom.summary()
```

```
'IGRAPH U--- 100 484 -- \n+ attr: x (v), y (v)'
```

Summary tells us that the graph has 100 nodes and 484 edges. The nodes have attributes, namely their (x,y) coordinates in the unit square.

Graph properties

```
1  geom = ig.Graph.GRG(10,0.4)
2  geom.degree()
```

```
[2, 3, 1, 6, 5, 2, 4, 4, 4, 1]
```

Compute the degree sequence of the graph. The degree of a vertex is the number of edges incident upon that vertex.

```
1  geom.betweenness()
```

```
[0.0, 0.5, 0.0, 5.0, 1.5, 1.0, 0.0, 0.0, 0.0, 0.0]
```

```
1  geom = ig.Graph.GRG(100,0.2)
2  len(geom.cliques(min=3,max=3))
```

```
1189
```

```
1  geom.diameter()
```

```
9
```

Graph properties

```
1  geom = ig.Graph.GRG(10,0.4)
2  geom.degree()
```

```
[2, 3, 1, 6, 5, 2, 4, 4, 4, 1]
```

```
1  geom.betweenness()
```

```
[0.0, 0.5, 0.0, 5.0, 1.5, 1.0, 0.0, 0.0, 0.0, 0.0]
```

Betweenness of a vertex measures how central it is to the graph; essentially, how many paths between other pairs of vertices pass through this vertex?

```
1  geom = ig.Graph.GRG(100,0.2)
2  len(geom.cliques(min=3,max=3))
```

```
1189
```

```
1  geom.diameter()
```

```
9
```

Graph properties

```
1  geom = ig.Graph.GRG(10,0.4)
2  geom.degree()
```

```
[2, 3, 1, 6, 5, 2, 4, 4, 4, 1]
```

```
1  geom.betweenness()
```

```
[0.0, 0.5, 0.0, 5.0, 1.5, 1.0, 0.0, 0.0, 0.0, 0.0]
```

```
1  geom = ig.Graph.GRG(100,0.2)
2  len(geom.cliques(min=3,max=3))
```

```
1189
```

Count the number of triangles in the graph. The number of triangles is a good, albeit simple, measure of the extent to which the graph in question displays the **small world** phenomenon.

```
1  geom.diameter()
```

```
9
```

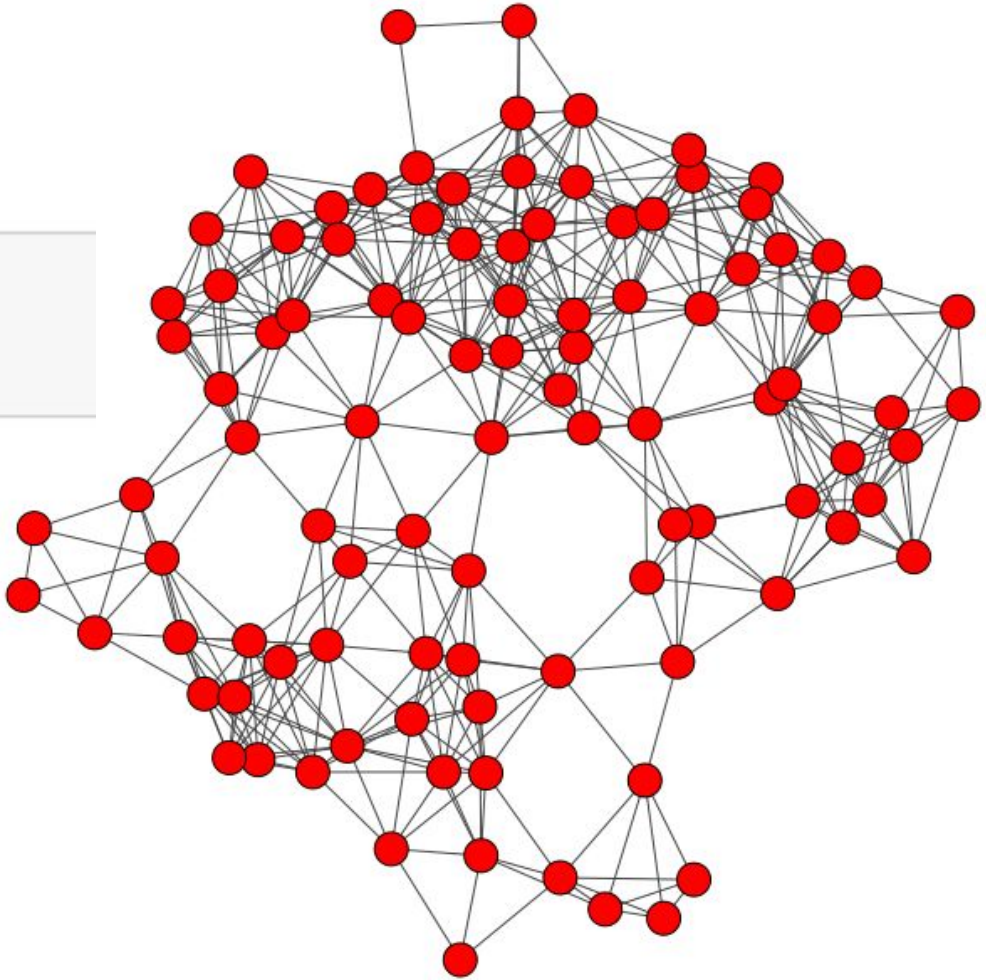
https://en.wikipedia.org/wiki/Small-world_network
https://en.wikipedia.org/wiki/Six_degrees_of_separation

Graph visualization

```
1  import igraph as ig
2  layout = geom.layout('kk')
3  ig.plot(geom, layout=layout)
```

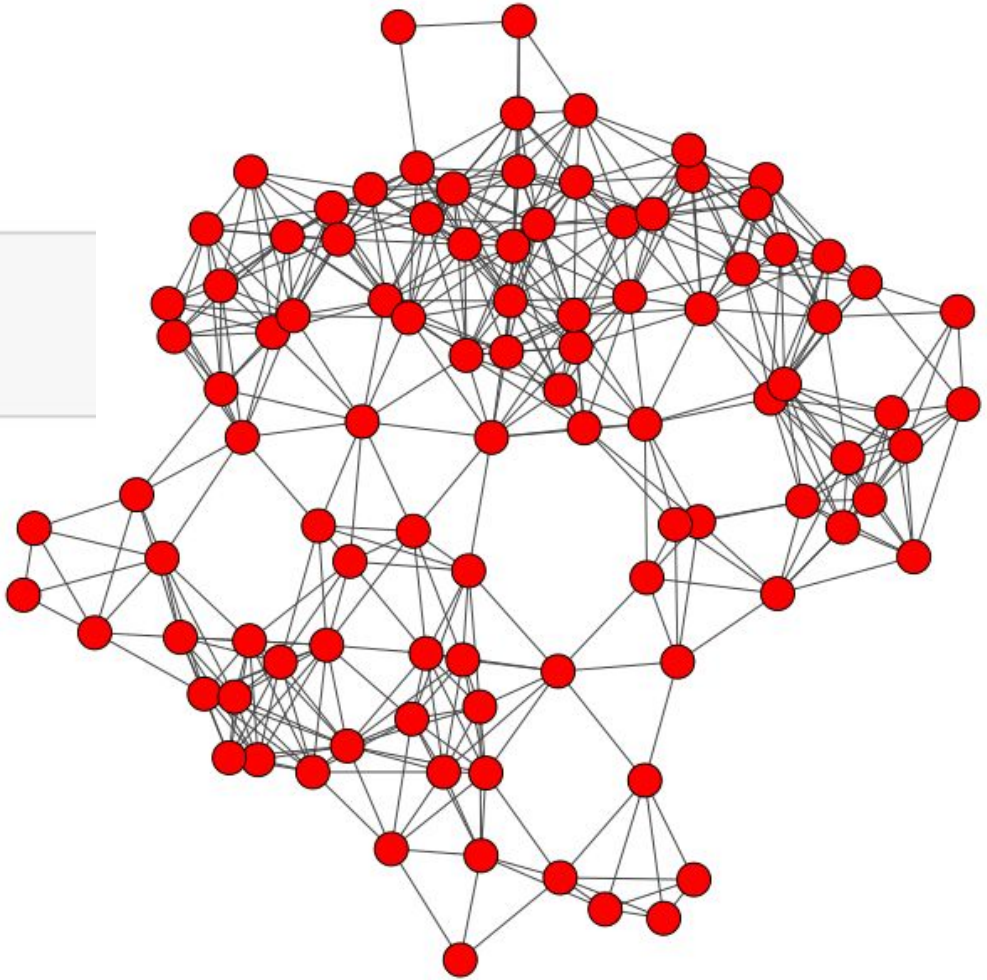
Layout object specifies how to arrange the nodes when visualizing the graph. `kk` is the Kamada-Kawai algorithm, which models the nodes as masses connected by springs. http://igraph.org/python/doc/tutorial/tutorial.html#layout-algorithms

Full disclosure: because `igraph` does not include support for inline image display like `matplotlib` does, I actually ran this code directly in the Python interpreter.



Graph visualization

```
1  import igraph as ig
2  layout = geom.layout('kk')
3  ig.plot(geom, layout=layout)
```



We can add more information to the plot, such as vertex- or edge-level labels, change vertex sizes and colors, change edge weights and colors, etc. More information: http://igraph.org/python/doc/tutorial/tutorial.html#vertex-attributes-controlling-graph-plots

Full disclosure: because `igraph` does not include support for inline image display like `matplotlib` does, I actually ran this code directly in the Python interpreter.

Working with graphs

For most of my own research, I use MATLAB and numpy

- Using adjacency matrix representation, so want fast matrix operations

I find `igraph` primarily useful for visualization

- Less so for running experiments

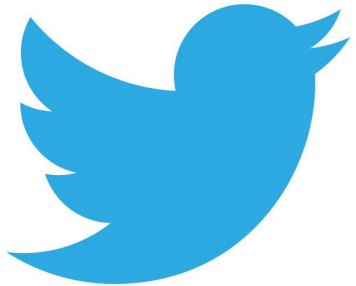
But support for many different generative models is also quite nice

Note: the R igraph package is better supported than the Python one

Example Project: Exploring the Twitter Graph

It is natural to represent Twitter as a directed graph

- Each user is a node
- Edge from user A to user B if A follows B



Idea:

- select a user of twitter
- construct the neighborhood of that user in the twitter graph
- E.g., graph formed by that user, his/her friends, and friends of friends

Example Project: Exploring the Twitter Graph

Project sketch:

- Choose a user, say, Albert-Laszlo Barabasi https://twitter.com/barabasi
  - Create a graph with one node, labeled @barabasi
- Choose a neighborhood size (e.g., 2 for friends and friends of friends)
- Use the twitter API to access followers and followees of Prof. Barabasi
  - https://developer.twitter.com/en/docs/accounts-and-users/follow-search-get-users/overview
  - **Note:** this requires a twitter account and App registration
- For each follower and followee, add a corresponding node and edge
- Recurse, repeating on all neighbors of @barabasi, and so on
  - (until we have explored to the desired neighborhood size)

Readings

Required:

None

Recommended:

Getting started with the Python `requests` package

http://docs.python-requests.org/en/master/user/quickstart/

Mozilla overview of HTTP methods

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

RFC Specifying HTTP methods https://tools.ietf.org/html/rfc7231#section-4

`igraph` tutorial http://igraph.org/python/doc/tutorial/tutorial.html

STATS 701
Data Analysis using Python

Closing Remarks

First, a word of thanks

Peter Knoop
Programmer & Senior Analyst
LSA IT

Seth Meyer
Research Computing Lead
ARC-TS

Without these two gentlemen, the second half of this course would not have been possible. If you see them, please thank them for their help!

Second, more words of thanks

Roger Fan
PhD Student
Department of Statistics

Topics We Surveyed

Regular expressions

Markup languages

Databases

UNIX Command Line

MapReduce

Spark

TensorFlow

APIs

We've only scratched the surface on all of these topics. The best way to learn more is to pick a project and start working on it. For example, pick a simple statistical model and implement it in TensorFlow, then apply that model to data, perhaps scraped from the web somewhere.

Topics We Surveyed

Regular expressions

Markup languages

Databases

UNIX Command Line

MapReduce

Spark

TensorFlow

APIs

We've only scratched the surface on all of these topics. The best way to learn more is to pick a project and start working on it. For example, pick a simple statistical model and implement it in TensorFlow, then apply that model to data, perhaps scraped from the web somewhere.

But these topics are constantly changing
- New software versions
- New tools
- New frameworks

It's a lot of work to keep up!

Keeping up with new tools

Find a few blogs/twitter feeds to follow

Forums: e.g., HackerNews, Reddit

Read papers on the arXiv

Most good papers will describe what framework(s) they used

Keeping up with changes in the software ecosystem is a part of the job, **especially in industry**, and requires time and effort.

Finding Projects

If you are currently doing research:

- At least one thing we discussed this semester should apply to your project!
- Speak to your supervisor about Flux allocation or buying GCP time

If you aren't:

- Find an interesting question, and answer it
- Interesting data set? Visualization? Simulation?
- Consider Amazon AWS or GoogleCloud for compute resources

Finding Projects

If you are currently doing research:

At least one thing we discussed this semester should apply to your project!

Speak to your supervisor about Flux allocation or buying GCP time

If you aren't:

Find an interesting question, and answer it

Interesting data set? Visualization? Simulation?

Consider Amazon AWS or GoogleCloud for compute resources

"I picked this card shuffling problem up off the street.
Find a problem that sparks your interest, and pursue it!"
-Persi Diaconis (paraphrased)

Thanks!