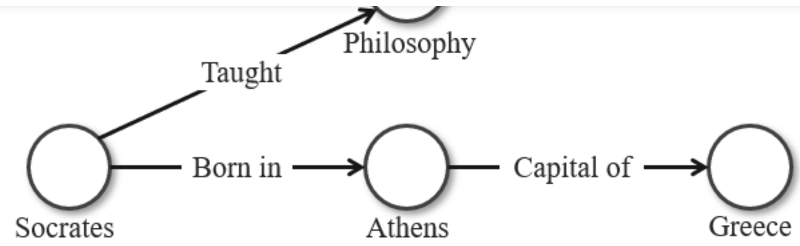
Chris Thornton Follow
Feb 9, 2020 · 4 min read ★ · Listen

Auto-Generated Knowledge Graphs

Utilize an ensemble of web scraping bots, computational linguistics, natural language processing algorithms and graph theory.

Knowledge graphs are a tool of data science that deal with interconnected **entities** (people, organizations, places, events, etc.). Entities are the **nodes** which are connected via **edges**. Knowledge graphs consist of these **entity pairs** that can be traversed to uncover meaningful connections in unstructured data.



There are issues inherent with graph databases, one being the manual effort required to construct them. In this article I will discuss my research and implementations of automatic generation using web scraping bots, computational linguistics, natural language processing (NLP) algorithms and graph theory (with python code provided).

Web Scraping

The first step in constructing a knowledge graph is to gather your sources. One document may be enough for some purposes, but if you want to go deeper and crawl the web for more information there are multiple ways to achieve this using web scraping. Wikipedia is a decent starting point, as the site functions as a user-generated content database with citations to mostly reliable secondary sources, which vet data from primary sources.

Side Note*: Always check your sources. Believe it or not, not all information on the internet is true! For a heuristic based solution, cross-reference other sites or opt for SEO metrics as a proxy for trust-signals.*

I will avoid screen-scraping wherever possible by using a direct python wrapper for the Wikipedia API.

The following function searches Wikipedia for a given topic and extracts information from the target page and its internal links.

```
import wikipediaapi  # pip install wikipedia-api
import pandas as pd
import concurrent.futures
from tqdm import tqdm

def wiki_scrape(topic_name, verbose=True):
    def wiki_link(link):
        try:
            page = wiki_api.page(link)
            if page.exists():
                return {'page': link, 'text': page.text, 'link': page.fullurl,
                        'categories': list(page.categories.keys())}
        except:
            return None

    wiki_api = wikipediaapi.Wikipedia(language='en',
        extract_format=wikipediaapi.ExtractFormat.WIKI)
    page_name = wiki_api.page(topic_name)
    if not page_name.exists():
        print('Page {} does not exist.'.format(topic_name))
        return

    page_links = list(page_name.links.keys())
    progress = tqdm(desc='Links Scraped', unit='', total=len(page_links)) if verbose else None
    sources = [{'page': topic_name, 'text': page_name.text, 'link': page_name.fullurl,
                'categories': list(page_name.categories.keys())}]

    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        future_link = {executor.submit(wiki_link, link): link for link in page_links}
        for future in concurrent.futures.as_completed(future_link):
            data = future.result()
            sources.append(data) if data else None
            progress.update(1) if verbose else None
    progress.close() if verbose else None

    namespaces = ('Wikipedia', 'Special', 'Talk', 'LyricWiki', 'File', 'MediaWiki',
```

```
        sources['topic'] = topic_name
        print('Wikipedia pages scraped:', len(sources))

        return sources
```

wikipedia_scrape.py hosted with ❤ by GitHub view raw

Let's test this function on the topic: "Financial crisis of 2007–08"

```
wiki_data = wiki_scrape('Financial crisis of 2007-08')
```

Output:

Wikipedia pages scraped: 798

topic	page	text	link	categories
Financial crisis of 2007-08	Financial crisis of 2007-08	The financial crisis of 2007-08, also known as the global financial crisis a…	https://en.wikipedia.org/wiki/Financial_crisis_of_2007%E2%80%9308	['2000s economic history', '2007 in economics', '2008 in economics', 'All Wi…
Financial crisis of 2007-08	10-Q	Form 10-Q, (also known as a 10-Q or 10Q) is a quarterly report mandated by…	https://en.wikipedia.org/wiki/Form_10-Q	['Articles with short description', 'SEC filings']
Financial crisis of 2007-08	1970s energy crisis	The 1970s energy crisis occurred when the Western world, particularly the Un…	https://en.wikipedia.org/wiki/1970s_energy_crisis	['1970s economic history', '1979 in economics', 'All NPOV disputes', 'All ar…
Financial crisis of 2007-08	1973 oil crisis	The 1973 oil crisis began in October 1973 when the members of the Organizat…	https://en.wikipedia.org/wiki/1973_oil_crisis	['1973 in economics', '1973 in international relations', 'All articles …
Financial crisis of 2007-08	1973-1975 recession	The 1973-1975 recession or 1970s recession was a period of economic sta…	https://en.wikipedia.org/wiki/1973%E2%80%931975_recession	['1970s economic history', '1973 in economics', '1974 in economics', '1975 i…
Financial crisis of 2007-08	1973-74 stock market crash	The 1973-74 stock market crash caused a bear market between January 1973 and D…	https://en.wikipedia.org/wiki/1973%E2%80%9374_stock_market_crash	['1973 in economics', '1974 in economics', 'Stock market crashes', 'Use…
Financial crisis of 2007-08	1973-75 recession	The 1973-1975 recession or 1970s recession was a period of economic sta…	https://en.wikipedia.org/wiki/1973%E2%80%931975_recession	['1970s economic history', '1973 in economics', '1974 in economics', '1975 i…
Financial crisis of 2007-08	1976 IMF crisis	The 1976 IMF Crisis was a financial crisis in the United Kingdom in 1976 w…	https://en.wikipedia.org/wiki/1976_IMF_crisis	['1976 in the United Kingdom', 'Economic history of the United Kingdom', 'Financi…
Financial crisis of 2007-08	1979 oil crisis	The 1979 (or second) oil crisis or oil shock occurred in the world due to dec…	https://en.wikipedia.org/wiki/1979_oil_crisis	['1979 in economics', '1979 in international relations', 'All articles …

If you want to extract a single page use the below function:

```
def wiki_page(page_name):
    wiki_api = wikipediaapi.Wikipedia(language='en',
            extract_format=wikipediaapi.ExtractFormat.WIKI)
    page_name = wiki_api.page(page_name)
    if not page_name.exists():
        print('Page {} does not exist.'.format(page_name))
        return

    page_data = pd.DataFrame({
        'page': page_name,
```

```
    return page_data
```

wikipedia_scrape_page.py hosted with ❤ by GitHub view raw

Computational Linguistics & NLP Algorithms

Knowledge graphs can be constructed automatically from text using **part-of-speech** and **dependency parsing**. The extraction of entity pairs from grammatical patterns is fast and scalable to large amounts of text using NLP library **SpaCy**.

The following function defines entity pairs as **entities/noun chunks with subject — object dependencies connected by a root verb**. Other rules-of-thumb can be used to produce different types of connections. This kind of connection can be referred to as a **subject-predicate-object triple**.

```
import pandas as pd
import re
import spacy
import neuralcoref

nlp = spacy.load('en_core_web_lg')
neuralcoref.add_to_pipe(nlp)


def get_entity_pairs(text, coref=True):
    # preprocess text
    text = re.sub(r'\n+', '.', text)  # replace multiple newlines with period
    text = re.sub(r'\[\d+\]', ' ', text)  # remove reference numbers
    text = nlp(text)
    if coref:
        text = nlp(text._.coref_resolved)  # resolve coreference clusters

    def refine_ent(ent, sent):
        unwanted_tokens = (
            'PRON',  # pronouns
            'PART',  # particle
            'DET',  # determiner
            'SCONJ',  # subordinating conjunction
            'PUNCT',  # punctuation
            'SYM',  # symbol
            'X',  # other
        )
        ent_type = ent.ent_type_  # get entity type
        if ent_type == '':
            ent_type = 'NOUN_CHUNK'
            ent = ' '.join(str(t.text) for t in
                           nlp(str(ent)) if t.pos_
                           not in unwanted_tokens and t.is_stop == False)
        elif ent_type in ('NOMINAL', 'CARDINAL', 'ORDINAL') and str(ent).find(' ') == -1:
            refined = ''
            for i in range(len(sent) - ent.i):
                if ent.nbor(i).pos_ not in ('VERB', 'PUNCT'):
                    refined += ' ' + str(ent.nbor(i))
                else:
```

```python
    sentences = [sent.string.strip() for sent in text.sents]  # split text into sentences
    ent_pairs = []
    for sent in sentences:
        sent = nlp(sent)
        spans = list(sent.ents) + list(sent.noun_chunks)  # collect nodes
        spans = spacy.util.filter_spans(spans)
        with sent.retokenize() as retokenizer:
            [retokenizer.merge(span, attrs={'tag': span.root.tag,
                                            'dep': span.root.dep}) for span in spans]
        deps = [token.dep_ for token in sent]

        # limit our example to simple sentences with one subject and object
        if (deps.count('obj') + deps.count('dobj')) != 1\
                or (deps.count('subj') + deps.count('nsubj')) != 1:
            continue

        for token in sent:
            if token.dep_ not in ('obj', 'dobj'):  # identify object nodes
                continue
            subject = [w for w in token.head.lefts if w.dep_
                       in ('subj', 'nsubj')]  # identify subject nodes
            if subject:
                subject = subject[0]
                # identify relationship by root dependency
                relation = [w for w in token.ancestors if w.dep_ == 'ROOT']
                if relation:
                    relation = relation[0]
                    # add adposition or particle to relationship
                    if relation.nbor(1).pos_ in ('ADP', 'PART'):
                        relation = ' '.join((str(relation), str(relation.nbor(1))))
                else:
                    relation = 'unknown'

                subject, subject_type = refine_ent(subject, sent)
                token, object_type = refine_ent(token, sent)

                ent_pairs.append([str(subject), str(relation), str(token),
                                  str(subject_type), str(object_type)])

    ent_pairs = [sublist for sublist in ent_pairs
                          if not any(str(ent) == '' for ent in sublist)]
    pairs = pd.DataFrame(ent_pairs, columns=['subject', 'relation', 'object',
                                             'subject_type', 'object_type'])
    print('Entity pairs extracted:', str(len(ent_pairs)))

    return pairs
```

get_entity_pairs.py hosted with ❤ by GitHub   view raw

Call the function on the main topic page:

```
pairs = get_entity_pairs(wiki_data.loc[0,'text'])
```

Output:

Entity pairs extracted: 71

subject	relation	object	subject_type	object_type
Dow Jones Industrial Average	hit	Dow Jones Industrial Average peak closing price	NOUN_CHUNK	NOUN_CHUNK
Bank of America	purchased	Merrill Lynch	ORG	ORG
The Federal Reserve	took over	American International Group	ORG	ORG
The Reserve Primary Fund	broke	buck	ORG	NOUN_CHUNK
Congress	passed	the Emergency Economic Stabilization Act	ORG	LAW
Two of the three Big Three automobile manufacturers	received	bailout	CARDINAL	NOUN_CHUNK
Congress	approved	the American Recovery and Reinvestment Act	ORG	ORG
Dow Jones	hit	Dow Jones lowest level	NOUN_CHUNK	NOUN_CHUNK
Low interest rates	encouraged	mortgage lending	NOUN_CHUNK	NOUN_CHUNK
implicit guarantee	created	moral hazard	NOUN_CHUNK	NOUN_CHUNK

Coreference resolution significantly improves entity pair extraction by normalizing the text, removing redundancies, and assigning entities to pronouns (*see my article on coreference resolution below)*.

Coreference Resolution in Python

Integrate Neural Network-Based Coreference Resolution into your NLP Pipeline using NeuralCoref

towardsdatascience.com

It may also be worthwhile to train a custom entity recognizer model if your use-case is domain-specific (healthcare, legal, scientific).

Graph Theory

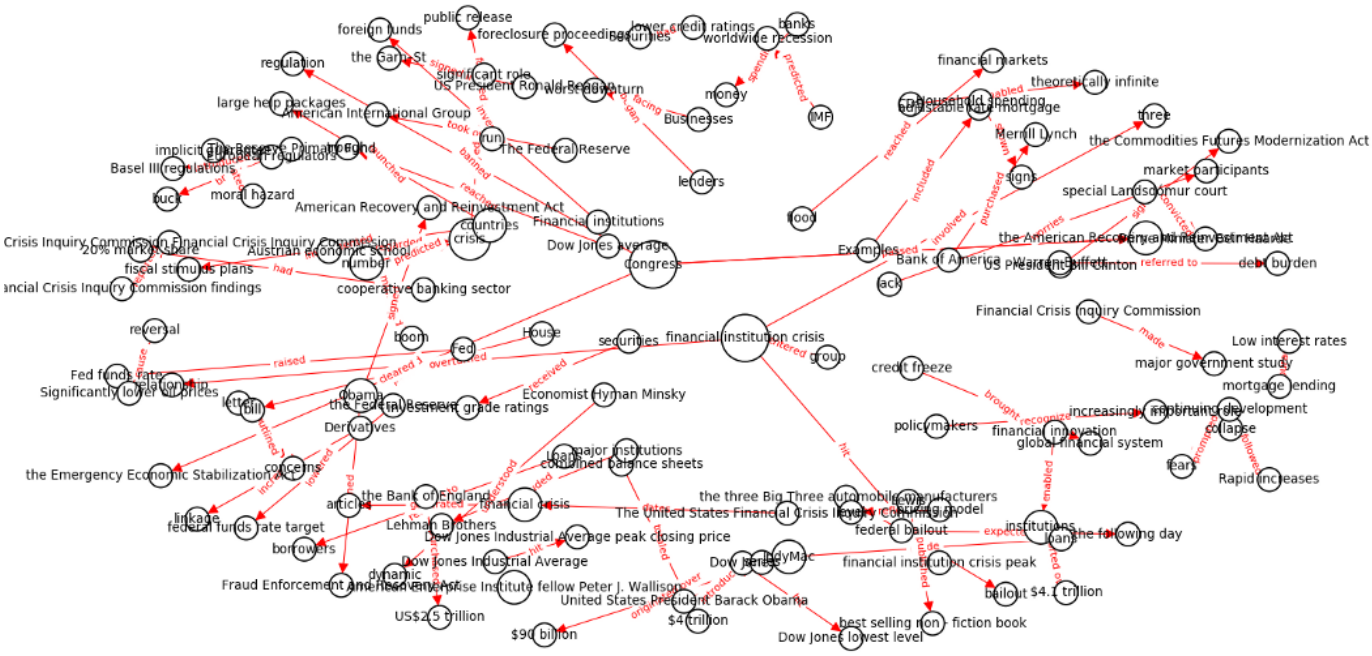
Next, lets draw the network using the **NetworkX** library. I will create a **directed multigraph** network with nodes sized in proportion to **degree centrality**.

```
import networkx as nx
```

```
        create_using=nx.MultiDiGraph())
    node_deg = nx.degree(k_graph)
    layout = nx.spring_layout(k_graph, k=0.15, iterations=20)
    plt.figure(num=None, figsize=(120, 90), dpi=80)
    nx.draw_networkx(
        k_graph,
        node_size=[int(deg[1]) * 500 for deg in node_deg],
        arrowsize=20,
        linewidths=1.5,
        pos=layout,
        edge_color='red',
        edgecolors='black',
        node_color='white',
        )
    labels = dict(zip(list(zip(pairs.subject, pairs.object)),
                pairs['relation'].tolist()))
    nx.draw_networkx_edge_labels(k_graph, pos=layout, edge_labels=labels,
                                font_color='red')
    plt.axis('off')
    plt.show()
```

draw_kg.py hosted with ❤ by GitHub view raw

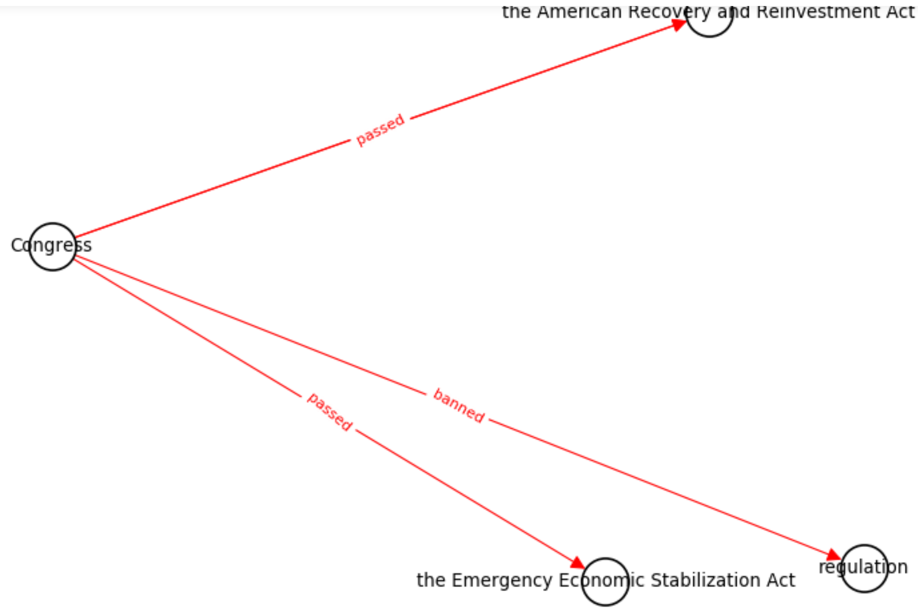
```
draw_kg(pairs)
```



```
            create_using=nx.MultiDiGraph())
    edges = nx.dfs_successors(k_graph, node)
    nodes = []
    for k, v in edges.items():
        nodes.extend([k])
        nodes.extend(v)
    subgraph = k_graph.subgraph(nodes)
    layout = (nx.random_layout(k_graph))
    nx.draw_networkx(
        subgraph,
        node_size=1000,
        arrowsize=20,
        linewidths=1.5,
        pos=layout,
        edge_color='red',
        edgecolors='black',
        node_color='white'
        )
    labels = dict(zip((list(zip(pairs.subject, pairs.object))),
                    pairs['relation'].tolist()))
    edges= tuple(subgraph.out_edges(data=False))
    sublabels ={k: labels[k] for k in edges}
    nx.draw_networkx_edge_labels(subgraph, pos=layout, edge_labels=sublabels,
                                font_color='red')
    plt.axis('off')
    plt.show()
```



```
filter_graph(pairs, 'Congress')
```



Knowledge Graphs at Scale

To effectively use the entire corpus of **~800** Wikipedia pages for our topic, use the columns created in the *wiki_scrape* function to add properties to each node, then you can track which pages and categories each node lies in.

I recommend using **multiprocessing** or **parallel processing** to reduce execution time.

Knowledge graphs on a large scale are at the frontier of AI research. Alas, real-world knowledge is not structured neatly into a schema but rather unstructured, messy, and organic.