QSS20: Modern Statistical Computing

Unit 10: APIs

Goals for today

- Where we're headed
- Recap of text as data
- API: terminology and basics
- Example 1: API with no credentials and no wrapper (NAEP data API)
- Example 2: API with credentials and no wrapper (Yelp API)

Goals for today

- **Where we're headed**
- Recap of text as data
- API: terminology and basics
- Example 1: API with no credentials and no wrapper (NAEP data API)
- Example 2: API with credentials and no wrapper (Yelp API)

Where we're headed

DataCamp deadlines:

- **Today (02/13):** Chapter on importing data with APIs
- **Wednesday 02/15:** Chapter on intro to HTML (essential for web-scraping)
- **Next week:** Course on supervised machine learning with `scikit-learn`

TITLE	ASSIGNEES	STATUS	DUE BY	C	A	CR	DETAILS
Intermediate Importing Data in Python **Interacting with APIs to import data from the web** Chapter	Organization	DUE SOON	Feb 13, 15:30 EST	17	30	56%	View
Web Scraping in Python **Introduction to HTML** Chapter	Organization	DUE SOON	Feb 15, 15:30 EST	4	30	13%	View
Supervised Learning with scikit-learn Course	Individuals	Active	Feb 20, 15:30 EST	0	1	0%	View

Class and problem sets:

- **Next class:** Intro to web-scraping for data collection & cleaning
- **Next week:** Intro to supervised machine learning (using text data)
- **Grades for pset 1 makeup:** Part of your Canvas pset 1 grades (good job!)
- **Grades for pset 3:** No later than Thursday
- **Grades for milestone 1:** By end of the week
- **Deadline this Sunday 02/19, 11:59 PM:** pset 4 due via GitHub Issue

Why no Twitter API?

Link to this tweet

Why no Twitter API?

Choose level of usage

	Total Requests PER MONTH ⓘ	Month-to-month PRICE PER MONTH ⓘ
Paid		
	Up to 500	$149.00
	Up to 1000	$289.00
	Up to 2,500	**$699.00**
	Up to 5,000	$1,299.00
	Up to 10,000	$2,499.00

Twitter API docs

Where we are

- Where we're headed
- **Recap of text as data**
- API: terminology and basics
- Example 1: API with no credentials and no wrapper (NAEP data API)
- Example 2: API with credentials and no wrapper (Yelp API)

Recap of text as data

What do you remember?

Review of text as data

- *Natural Language Processing (NLP)*: Science of processing language to extract signal, find patterns, connect with social world
- *Tokens*: Units of language, e.g., words, bigrams, phrases, punctuation

  `"I like Dartmouth" (string) -> ["I", "like", "Dartmouth"] (list of str)`
- *Supervised methods*: Guided by pre-existing knowledge, often a list of words or doc labels
- *Unsupervised methods*: Inductively discover patterns in text data, e.g. topics, key words
- *Part of Speech (POS) tagging*:

```
tokens = word_tokenize(text) # Tokenize
tokens_pos = pos_tag(tokens) # get POS tags
```

- *Named Entity Recognition (NER)*: NLP pipeline for identifying named entities, e.g. people, places

```
spacy_obj = nlp(text) # run NER pipeline
[entity.label_ for entity in spacy_obj.ents] # get tags
```

- *Sentiment analysis*: Use dictionary to score positive/negative "feel" at document level

  `SentimentIntensityAnalyzer().polarity_scores(text)`
- *Document-Term Matrix (DTM)*: each row is a doc, each col is a term, values are 0, 1, ... n for # of times that term occurs in that doc
- *Latent Dirichlet Allocation (LDA)*: Generative model of language based on idea that language comes from "topics". LDA estimates topics (mixture of words) and their distro over docs (mixture of topics).

  `ldamodel.LdaModel(corpus_word_map, num_topics=5, id2word=text_raw_dict)`

Steps for topic modeling using gensim: preprocessing

```
## Step 1: tokenize documents and store in list
text_raw_tokens = [wordpunct_tokenize(one_text)
                   for one_text in ab_small.name_lower]
## Step 2: use gensim create dictionary - gets all unique
    words across documents
text_raw_dict = corpora.Dictionary(text_raw_tokens)
## Step 3: filter out very rare and very common words
    from dictionary; feeding it n docs as lower and upper
    bounds
text_raw_dict.filter_extremes(no_below = lower_bound,
                              no_above = upper_bound)
## Step 4: map words in dictionary to words in each
    document in the corpus
corpus_fromdict = [text_raw_dict.doc2bow(one_text)
                   for one_text in text_raw_tokens]
```

Steps for topic modeling using gensim: estimation

See documentation for many parameters you can vary!:

https://radimrehurek.com/gensim/models/ldamodel.html

```
## Step 5: estimate a model by feeding it: (1) the corpus
## mapped to the dictionary, (2) the dictionary itself,
## (3) number of topics (in notation, k),
## and assorted other arguments
ldamod = gensim.models.ldamodel.LdaModel(corpus_fromdict,
                                         num_topics = 5,
                                         id2word=text_raw_dict,
                                         passes=6,
                                         alpha = 'auto',
                                         per_word_topics = True)
```

Returns a trained ldamodel class with various methods/attributes

Where we are

- Where we're headed
- Recap of text as data
- **API: terminology and basics**
- Example 1: API with no credentials and no wrapper (NAEP data API)
- Example 2: API with credentials and no wrapper (Yelp API)

Terminology

- **API:** application programming interface; way to ask an app or website for something and get something in return
- **Call the API**: sending a request for something to the API
- **Response**: can think of this as a message back telling us *whether* we got something back or whether the call returned an error
- **JSON:** if we get something back, oftentimes it'll be stored in `json` format, which is basically a text string with a particular structure that is similar to the *data structure* of a dictionary; can pretty easily convert to a `pandas` dataframe
- **Wrapper:** a language-specific module or package that helps simplify the process of calling an API with code written in a particular language (e.g., `tweepy`, a Python wrapper for the Twitter API; there are also R wrappers for the Twitter API)

Main use in our context: data acquisition

Three general routes to acquiring data:

Exists already:
Great first step to check out, can save a lot of time if matches your research question; examples include the csv/excel data we've been working with for problem sets

API:
A "front door" to a website, where the developers provide easy access to content but also set limits (e.g., what content or how much); most relevant for "high-velocity" data that changes frequently (e.g., tweets; job postings) and also for using code rather than point/click to get data

Scraping:
A "back door" to web content for when there's no API or when we need content beyond the structured fields the API returns; can be time-consuming and code can break if website structure changes (can happen often)

Where we are

- Where we're headed
- Recap of text as data
- API: terminology and basics
- **Example 1: API with no credentials and no wrapper (NAEP data API)**
- Example 2: API with credentials and no wrapper (Yelp API)

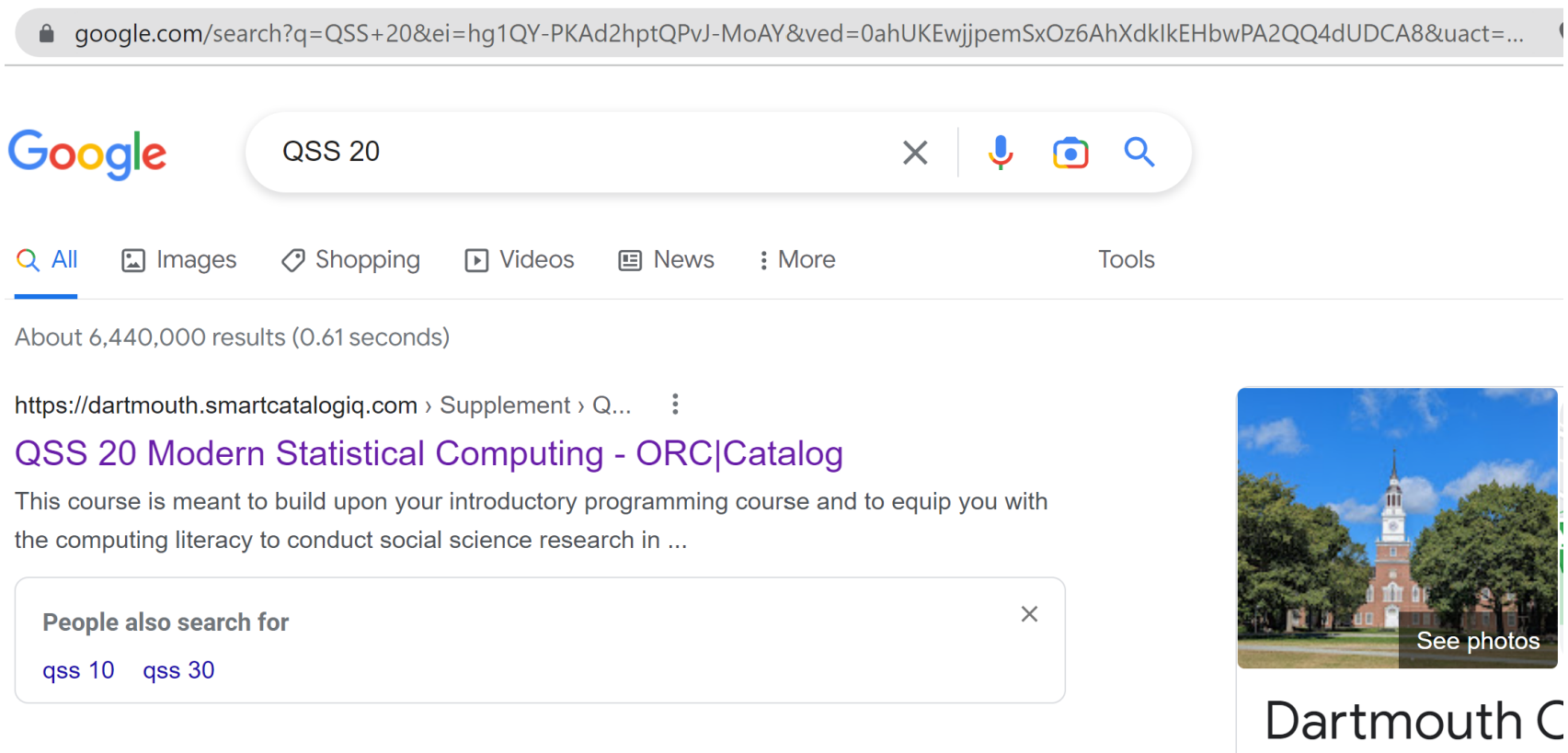
High-level overview of steps: APIs that don't need credentials

1. Construct a query that tells the API what we want to pull
2. Use `requests.get(querystring)` to call the API (same query for other languages, too!)
3. Examine the response: message from the API telling us whether it returned something
4. If the response returned something, extract the content of the response and make it usable

Example query: googling ‘QSS 20’

Here’s the call to google search API my browser makes for ‘QSS 20’:

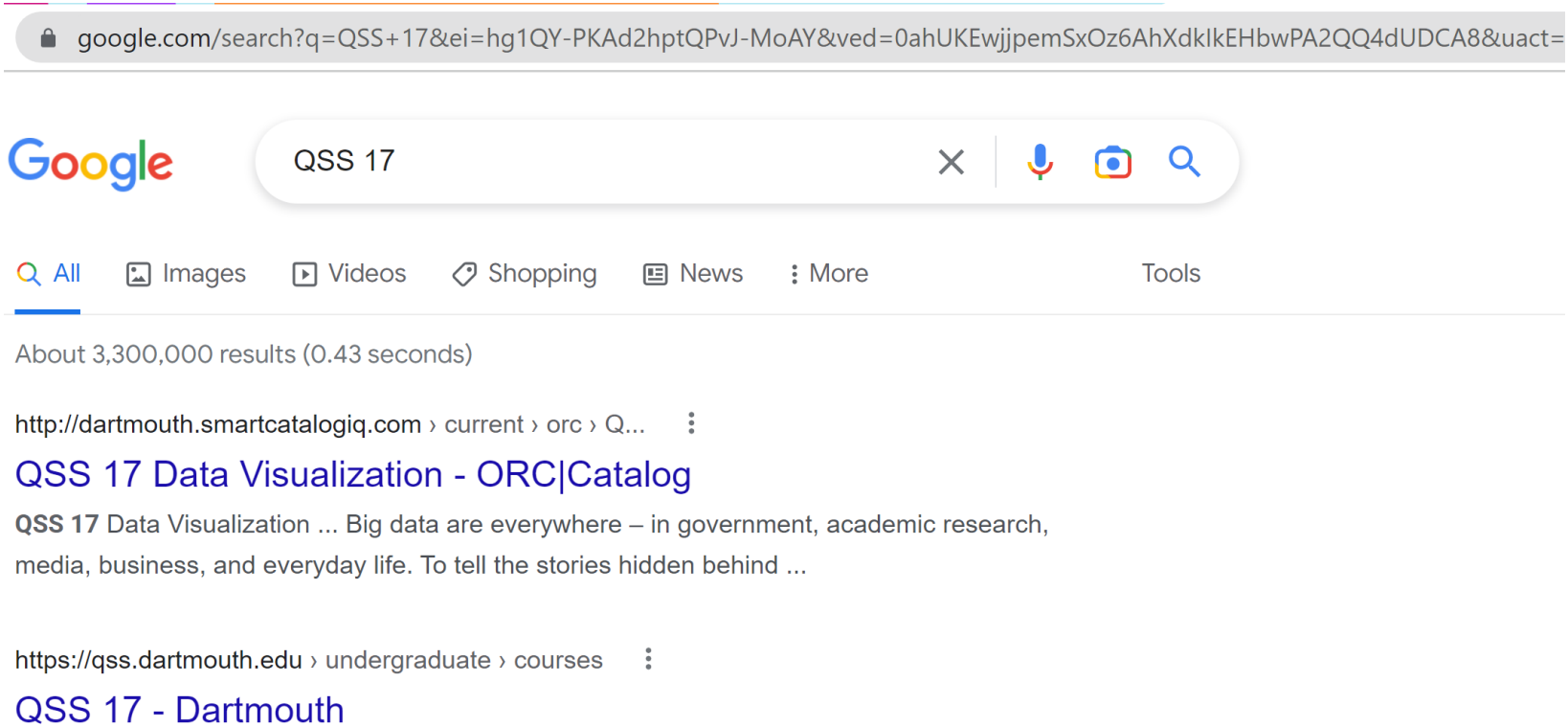
```
('https://www.google.com/search?'
'q=QSS+20&source=hp&'
'ei=bA1QY6HmMb2IptQP4bOEsA8&'
'... sclient=gws-wiz')
```



Example query: googling ‘QSS 17’

I can easily change this into a google search for ‘QSS 17’ by updating the ‘q’ (for ‘query’) parameter of this API call:

```
('https://www.google.com/search?'
'q=QSS+17&source=hp&'
'ei=bA1QY6HmMb2IptQP4bOEsA8&'
'... sclient=gws-wiz')
```



Step 1: construct a query

- Generic example:
  "https://baseurl.com/onething=something&anotherthing=somethingelse"
- Specific NAEP example (use the '()' syntax to split across lines)

```
example_naep_query = (
'https://www.nationsreportcard.gov/'
'Dataservice/GetAdhocData.aspx?'
'type=data&subject=writing&grade=8&'
'subscale=WRIRP&variable=GENDER&',
'jurisdiction=NP&stattype=MN:MN&',
'Year=2011')
```

- Breaking things down:
  - `nationsreportcard`: this is the "base url" we're using for the API call and what we add parameters to
  - `subject:` type of parameter
  - `subject=writing`: specific value for that parameter (error if we feed it a subject that doesn't exist)
  - And so on...

Steps 2-4: call the API, examine the response, and if response indicates something usable, extract content

```
## use requests.get to call the API
naep_resp = requests.get(example_naep_query)

## we got usable response, so get json of status and
    result
naep_resp_j = naep_resp.json()

## extract contents in `result' key
## and convert to dataframe
naep_resp_d = pd.DataFrame(naep_resp_j['result'])
```

Activity 1: practice pulling data using the NAEP API

Notebook: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/06_apis_blank.ipynb

- Example of executing a query that doesn't have errors
- Example of executing a query that returns nothing
- Working together to write a function to do multiple calls to the API

Where we are

- Where we're headed
- Recap of text as data
- API: terminology and basics
- Example 1: API with no credentials and no wrapper (NAEP data API)
- **Example 2: API with credentials and no wrapper (Yelp API)**

What changes about the general steps?

1. Acquire credentials for the API: these may be an API key (single string) or a client ID and secret (strings; can store in a `yaml` creds file that I'll outline)
2. Construct a query that tells the API what we want to pull
3. Feed API your credentials/authenticate
4. Examine the response: message from the API telling us whether it returned something
5. If the response returned something, extract the content of the response and make it usable

Step 1: acquire credentials

- Varies across APIs, but general involves going to the "developer's portal," creating an account, and obtaining credentials
- Examples:
  - Google developer's console (things like google geocoding API; maps API): https://console.cloud.google.com
  - Facebook: https://developers.facebook.com/docs/development
  - Twitter (via Tweepy wrapper): https://docs.tweepy.org/en/latest/auth_tutorial.html
  - Yelp: https://www.yelp.com/developers/documentation/v3/authentication
- Note weird-ish terminology for social science applications since you often set up "an application" in order to get credentials (but we're often doing a one-way pull of data and not developing an app. that repeatedly calls it)

Step 1: store those credentials somewhere

- Your key or client ID/secret are meant to be unique to you like a password, so you shouldn't generally print in code
- Can use any text editor to make a yaml file (structured similar to a dictionary); screenshot below from Sublime text with fake credentials

Step 1: load the file with credentials

```
## imports
import yaml

## load creds
with open("../../private_data/my_cred.yaml", 'r') as
    stream:
     try:
          creds = yaml.safe_load(stream)
     except yaml.YAMLError as exc:
          print(exc)

## can then get the relevant key
creds['yelp_api']['api_key']
```

Step 2: construct a query

Same exact process as before; here focusing on **Yelp Fusion API**; API has different endpoints shown in the screenshot; we'll initially focus on Business Search, since that returns a Yelp-specific ID (https://www.yelp.com/developers/documentation/v3/get_started)

Name	Path	Description
Business Search	/businesses/search	Search for businesses by keyword, category, location, price level, etc.
Phone Search	/businesses/search/phone	Search for businesses by phone number.
Transaction Search	/transactions/{transaction_type}/search	Search for businesses which support food delivery transactions.
Business Details	/businesses/{id}	Get rich business data, such as name, address, phone number, photos, Yelp rating, price levels and hours of operation.
Business Match	/businesses/matches	Find the Yelp business that matches an exact input location. Use this to match business data from other sources with Yelp businesses.
Reviews	/businesses/{id}/reviews	Get up to three review excerpts for a business.
Autocomplete	/autocomplete	Provide autocomplete suggestions for businesses, search keywords and categories.

Step 2: construct a query

```
## defining inputs
base_url = "https://api.yelp.com/v3/businesses/search?"
my_name = "restaurants"
my_location = "Hanover,NH,03755"

## combining them into a query
yelp_genquery = (
    f'{base_url}'
    'term={my_name}'
    '&location={my_location}')
```

Step 3: authenticate and call the API

For Yelp, we feed a dictionary with our key directly into the `get` call via the optional `header` parameter; for other APIs, we sometimes authenticate in a separate step

```
## construct my http header dict
header = {'Authorization': f'Bearer {API_KEY}'}

## call the API
yelp_genresp = requests.get(yelp_genquery, headers =
    header)
```

Step 3: output of successful and unsuccessful call

- Successful call:

```
<Response [200]>
```

- Unsuccessful call (put Hanover,WY,09999 as the location, which doesn't exist)

```
<Response [400]>

{'error': {'code': 'LOCATION_NOT_FOUND',
  'description': 'Could not execute search, try specifying a more exact location.'}}
```

Step 4: if output successful, make results usable

See that 'businesses' key of json file has a dictionary for each business, but some nesting to deal with variable lengths (e.g., within 'location', 'address1', 'address2', etc.) that might produce odd things when we concat. to a df:


```
{'id': '8ybF6YyRldtZmU9jil4xlg',
 'alias': 'mollys-restaurant-and-bar-hanover',
 'name': "Molly's Restaurant & Bar",
 'image_url': 'https://s3-media2.fl.yelpcdn.com/bphoto/1YkJFic4Czt9b2FsZyOrwQ/o.jpg',
 'is_closed': False,
 'url': 'https://www.yelp.com/biz/mollys-restaurant-and-bar-hanover?adjust_creative=A
gn=yelp_api_v3&utm_medium=api_v3_business_search&utm_source=ABQTB3e9fTiSiyqs0c-3Bg',
 'review_count': 403,
 'categories': [{'alias': 'tradamerican', 'title': 'American (Traditional)'},
  {'alias': 'burgers', 'title': 'Burgers'},
  {'alias': 'pizza', 'title': 'Pizza'}],
 'rating': 4.0,
 'coordinates': {'latitude': 43.701144, 'longitude': -72.2894249},
 'transactions': ['delivery'],
 'price': '$$',
 'location': {'address1': '43 South Main St',
  'address2': '',
  'address3': '',
  'city': 'Hanover',
  'zip_code': '03755',
  'country': 'US',
  'state': 'NH',
  'display_address': ['43 South Main St', 'Hanover, NH 03755']},
 'phone': '+16036432570',
 'display_phone': '(603) 643-2570',
 'distance': 250.8301601841674}
```

Approach 1 to step 4: more automatic `pd.DataFrame` that leaves those as lists

```
1 yelp_gendf = pd.DataFrame(yelp_genjson['businesses'])
```

Approach 2 to step 4: only retaining columns that are already strings

```
def clean_yelp_json(one_biz):

    ## restrict to str cols
    d_str = {key:value for key, value in one_biz.items()
    if type(value) == str}

    df_str = pd.DataFrame(d_str, index = [d_str['id']])
    return(df_str)

yelp_stronly = [clean_yelp_json(one_b)
        for one_b in yelp_genjson['businesses']]
yelp_stronly_df = pd.concat(yelp_stronly)
```

Activity 2: practice with the Yelp API

Same url: https://github.com/jhaber-zz/QSS20_public/blob/main/activities/06_apis_blank.ipynb

- Try running a business search query for your hometown or another place by constructing a query similar to ‘yelp_genquery‘ but changing the location parameter
- Other endpoints require feeding what’s called the business’ fusion id into the API. Take an id from ‘yelp_stronly.id‘ and use the documentation here to pull the reviews for that business: https://www.yelp.com/developers/documentation/v3/business_reviews
- **Challenge**: generalize the previous step by writing a function that (1) takes a list of ids as an input, (2) calls the reviews API for each id, (3) returns the results, and (4) rowbinds all results