Large Data Computation in Python

Yujia Deng and Annie Qu

University of Illinois at Urbana-Champaign

Out-of-memory Data: Pandas+SQL

Out-of-memory Data: Pandas+SQL

- NYC's 311 complaints since 2010 (Updated daily): https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9
- About 10GB, 19.6M records, 41 columns

```
display(pd.read_csv('311_Service_Requests_from_2010_to_Present.csv', nrows=2))
```

	Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	Location Type	Incident Zip	Incident Address
0	34887213	11/29/2016 11:17:07 PM	11/30/2016 02:32:46 AM	NYPD	New York City Police Department	Noise - Residential	Loud Music/Party	Residential Building/House	10472	1237 ELDER AVENUE
1	34887215	11/29/2016 07:41:03 AM	12/01/2016 11:38:17 AM	HPD	Department of Housing Preservation and Develop...	WATER LEAK	SLOW LEAK	RESIDENTIAL BUILDING	11201	86 BERGEN STREET

2 rows × 41 columns

- MemoryError if we load the whole data at once

```
%timeit dt311 = pd.read_csv('311_Service_Requests_from_2010_to_Present.csv') # memory error
```

- Create a database and connect using SQL

```
disk_engine = create_engine('sqlite:///NYC_311.db')
start = dt.datetime.now()
chunksize = 100
j = 0
index_start = 1
```

Read by crunk and subset

- Subset and append to the database chunk by chunk

```
for df in pd.read_csv('311_Service_Requests_from_2010_to_Present.csv', chunksize=chunksize, iterator=T
rue, encoding='utf-8', header=0):# head=0 specifies the number of the row where the header is located

    df = df.rename(columns={c: c.replace(' ', '') for c in df.columns}) # Remove spaces from columns

    df['CreatedDate'] = pd.to_datetime(df['CreatedDate'], errors='coerce', infer_datetime_format=True)
 # Convert to datetimes
    df['ClosedDate'] = pd.to_datetime(df['ClosedDate'], errors='coerce', infer_datetime_format=True)

    df.index += index_start

    # Remove the un-interesting columns
    columns = ['Agency', 'CreatedDate', 'ClosedDate', 'ComplaintType', 'Descriptor',
               'CreatedDate', 'ClosedDate', 'TimeToCompletion', 'City']

    for c in df.columns:
        if c not in columns:
            df = df.drop(c, axis=1)

    j+=1
    if j%12000 == 0:
        print('{} seconds: completed {} rows'.format((dt.datetime.now() - start).seconds, j*chunksize
))

    df.to_sql('data', disk_engine, if_exists='append') # Connect the trunk to the database
    index_start = df.index[-1] + 1
```

- Even so, the reading procedure takes more than 4 hours to complete.

Use SQL to explore the data

- Check the total number of records

```
pd.read_sql_query('SELECT COUNT(*) as `num_of_records`'
                  'FROM data ', disk_engine)
```

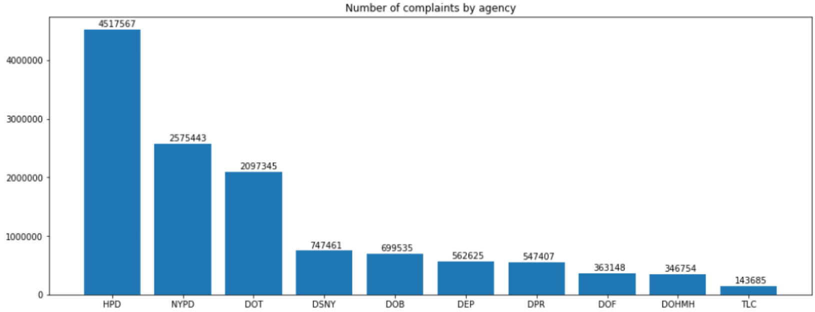
	num_of_records
0	12995000

- Investigate which department receives the most complaints

```
complaints_by_agency = pd.read_sql_query('SELECT Agency, COUNT(*) as `num_complaints`'
                       'FROM data '
                       'GROUP BY Agency '
                       'ORDER BY -num_complaints ' # in a descending order
                       'LIMIT 10', disk_engine) # notice there is space at the end of each line of SQL
 command
```

Visualization using `matplotlib`

```
import matplotlib.pyplot as plt
ind = range(complaints_by_agency.shape[0])
plt.figure(figsize=(16,6))
plt.bar(x=ind,height=complaints_by_agency['num_complaints'])
for k in ind:
    plt.text(k-0.2, complaints_by_agency['num_complaints'][k]+50000,str(complaints_by_agency['num_comp
laints'][k]))
plt.title('Number of complaints by agency')
plt.xticks(ind,complaints_by_agency['Agency'])
plt.show()
```

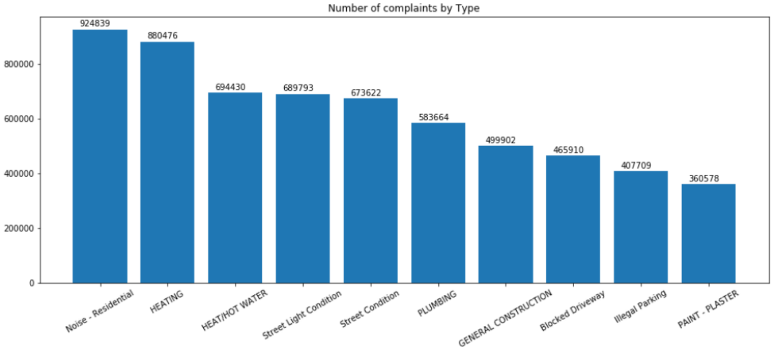


Department of Housing Preservation and Development (HPD)

Most common complaints

- Similarly, we can investigate what are the most common complaints

```
most_common_complaints = pd.read_sql_query('SELECT ComplaintType, COUNT(*) as `num_complaints` '
                        'FROM data '
                        'GROUP BY `ComplaintType` '
                        'ORDER BY -num_complaints LIMIT 10', disk_engine)
```



Within-memory Data: How to save space

Save the memory by specifying the datatype

- Airline data: 3GB, 7009728 observations, 29 columns
- Overview of the datatype

```
DT_panda.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7009728 entries, 0 to 7009727
Data columns (total 29 columns):
Year                  int64
Month                 int64
DayofMonth            int64
DayOfWeek             int64
DepTime               float64
CRSDepTime            int64
ArrTime               float64
CRSArrTime            int64
UniqueCarrier         object
FlightNum             int64
TailNum               object
ActualElapsedTime     float64
```

Compress the numerical data

- `int64`: $(-2^{64}$ to $2^{64}-1)$ and uses 8 bytes
- `int16`: $(-2^{16}$ to $2^{16}-1)$ or (-32,768 to +32,767) and uses only 2 bytes.
- `pd.to_numeric(, downcast=)` determines the smallest possible type automatically

> **downcast** : {'integer', 'signed', 'unsigned', 'float'} , default None If not None, and if the data has been successfully cast to a numerical dtype (or if the data was numeric to begin with), downcast that resulting data to the smallest numerical dtype possible according to the following rules:
>
> - 'integer' or 'signed': smallest signed int dtype (min.: np.int8)
>   - 'unsigned': smallest unsigned int dtype (min.: np.uint8)
>   - 'float': smallest float dtype (min.: np.float32)

int64 to unsigned

- Check data size change

```
DT_int = DT_panda.select_dtypes(include=['int64'])
converted_int = DT_int.apply(pd.to_numeric, downcast='unsigned')

print(mem_usage(DT_int))
print(mem_usage(converted_int))
```

```
534.80 MB
100.28 MB
```

- Before vs After

	before	after
Year	int64	uint16
Month	int64	uint8
DayofMonth	int64	uint8
DayOfWeek	int64	uint8
CRSDepTime	int64	uint16
CRSArrTime	int64	uint16
FlightNum	int64	uint16
Distance	int64	uint16
Cancelled	int64	uint8

float to float

- For `float` type

```
DT_float = DT_panda.select_dtypes(include=['float64'])
converted_float = DT_float.apply(pd.to_numeric, downcast='float')

print(mem_usage(DT_float))
print(mem_usage(converted_float))

compare_float = pd.concat([DT_float.dtypes, converted_float.dtypes], axis=1)
compare_float.columns = ['before','after']
display(compare_float)
compare_float.apply(pd.Series.value_counts)
```

```
748.72 MB
374.36 MB
```

	before	after
DepTime	float64	float32
ArrTime	float64	float32
ActualElapsedTime	float64	float32
CRSElapsedTime	float64	float32
AirTime	float64	float32
ArrDelay	float64	float32
DepDelay	float64	float32
TaxiIn	float64	float32
TaxiOut	float64	float32

object to category

- `object` : save strings

```
DT_obj = DT_panda.select_dtypes(include='object').copy()
DT_obj.describe()
```

	UniqueCarrier	TailNum	Origin	Dest	CancellationCode
count	7009728	6926363	7009728	7009728	137434
unique	20	5373	303	304	4
top	WN	N476HA	ATL	ATL	B
freq	1201754	4701	414513	414521	54904

- `category` type codes the string into category numbers (similar as factor type in R)
    - From: `array([nan, 'A', 'C', 'B', 'D'], dtype=object)`
    - To: `array([-1, 0, 1, 2, 3], dtype=int8)`

object to category

- Not economic if there are too many unique values
- Check if unique value $< 50\%$

```
converted_obj = pd.DataFrame()
for col in DT_obj.columns:
    num_unique_values = len(DT_obj[col].unique())
    num_total_values = len(DT_obj[col])
    if num_unique_values/ num_total_values < 0.5:
        converted_obj.loc[:,col] = DT_obj[col].astype('category')
    else:
        converted_obj.loc[:,col] = DT_obj[col]
```

- Memory saving

```
print(mem_usage(DT_obj))
print(mem_usage(converted_obj))
```

```
1833.13 MB
54.02 MB
```

Prespecify the datatype before loading

- Define the datatype

```
predetermined_dtypes = {'Year': 'uint16',
 'Month': 'uint8',
 'DayofMonth': 'uint8',
 'DayOfWeek': 'uint8',
 'CRSDepTime': 'uint16',
 'CRSArrTime': 'uint16',
 'FlightNum': 'uint16',
 'Distance': 'uint16',
```

... ...

```
DT_optimized = pd.read_csv('Airline2008.csv', dtype= predetermined_dtypes)
```

- 528.65 MB vs 3116.65 MB, 83% saved!

Parallel Computing using `joblib`

joblib

- `joblib` is based on `multiprocess` but provides more powerful and flexible methods.
- Basic syntax

```
Parallel(n_jobs=4)(delayed(np.power)(i,2) for i in range(10))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Compared with

```
[np.power(i,2) for i in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Multiple inputs and multiple outputs

```
num_pair = [(1,2), (2,3), (0,5)]
def foo(a,b):
    return a**2, b**2
Parallel(n_jobs=4)(delayed(foo)(i,j) for i,j in num_pair)
```

```
[(1, 4), (4, 9), (0, 25)]
```

Example: bootstrap on Iris data

- We use Python to repeat the bootstrap sample on Iris data
- Module sklearn contains many machine learning methods and commonly used dataset
- Load the data from sklearn module

```
from sklearn.datasets import load_iris
data = load_iris()
subset = data.target != 0
X = data.data[subset,0] # sepal length
Y = data.target[subset]
```

- Run glm from sklearn

```
from sklearn.linear_model import LogisticRegression
def booti(i, X, Y):
    np.random.seed(i)
    ind = np.random.choice(range(X.shape[0]), 100, replace=True)
    clf = LogisticRegression(random_state=i, solver='lbfgs').fit(X[ind].res
hape([-1,1]), Y[ind])
    return np.vstack((clf.intercept_, clf.coef_))
```

Example: bootstrap on Iris data

- Sequential way

```
import time
n_sim = 10000
tic = time.time()
res_loop = [booti(i, X, Y) for i in range(n_sim)]
toc = time.time()
print("Sequential bootstraping for %d times costs %2.3f seconds\n" %(n_sim,
 toc-tic))
np.hstack((res_loop[:3]))
```

```
Sequential bootstraping for 10000 times costs 19.935 seconds


array([[-12.62095792,  -9.43895749, -12.75926784],
       [  1.99107085,   1.47326895,   2.07603253]])
```

- Parallel way

```
n_jobs = 4
tic = time.time()
res_par = Parallel(n_jobs, verbose=0)(delayed(booti)(i, X, Y) for i in rang
e(n_sim))
toc = time.time()
print("Parrellel bootstraping for %d times on %d workers costs %2.3f second
s\n" %(n_sim, n_jobs,toc-tic))
np.hstack((res_par[:3]))
```

```
Parrellel bootstraping for 10000 times on 4 workers costs 8.729 seconds


array([[-12.62095792,  -9.43895749, -12.75926784],
       [  1.99107085,   1.47326895,   2.07603253]])
```

‘overhead’ and memory mapped file

- Not exactly 4 times faster because of ‘overhead’ (extra time on copying data for each worker, communication etc.)
- Memory mapped file: a chunk of memory whose bytes are accessible by more than one process
  - Save space: avoid copying dataset repeatedly
  - Save time : reduce "overhead"
- `dump` and `load`

```
import os
from joblib import load, dump
temp_folder = './joblib_memmap'
try:
    os.mkdir(temp_folder)
except FileExistsError:
    pass
disk_X = os.path.join(temp_folder, 'X.mmap')
dump(X, disk_X)
disk_Y = os.path.join(temp_folder, 'Y.mmap')
dump(Y, disk_Y)
```

Effect of memory mapped file

- With `memmap` file:

```
n_sim = 100000
X_mmap = load(disk_X,mmap_mode='r')
Y_mmap = load(disk_Y,mmap_mode='r')
tic = time.time()
res_par_mmap = Parallel(n_jobs, verbose=0)(delayed(booti)(i, X_mmap, Y_mmap) for i in range(n_sim))
toc = time.time()
print("Parrellel bootstraping for %d times on %d workers costs %2.3f seconds\n" %(n_sim, n_jobs,toc-tic))
np.hstack((res_par_mmap[:3]))
```

```
Parrellel bootstraping for 100000 times on 4 workers costs 74.521 seconds


array([[-12.62095792,  -9.43895749, -12.75926784],
       [  1.99107085,   1.47326895,   2.07603253]])
```

- Without `memmap` file:

```
n_sim = 100000
tic = time.time()
res_par = Parallel(n_jobs, verbose=0)(delayed(booti)(i, X, Y) for i in range(n_sim))
toc = time.time()
print("Parrellel bootstraping for %d times on %d workers costs %2.3f seconds\n" %(n_sim, n_jobs,toc-tic))
np.hstack((res_par_mmap[:3]))
```

```
Parrellel bootstraping for 100000 times on 4 workers costs 82.775 seconds


array([[-12.62095792,  -9.43895749, -12.75926784],
       [  1.99107085,   1.47326895,   2.07603253]])
```

Web scrapping using `Beautiful Soup`

Before you start

- **Do you really need scrapping?** check if the website has public API or dataset available.
- Before scrapping: check the website policy
- Download webpages first to avoid requesting repeatedly

Ending Data Scraping Dispute, Craigslist Reaches $31M Settlement with Instamotor

By **Jeffrey Neuburger** on August 24, 2017
Posted in **Mobile, Online Content, Screen Scraping**

Craigslist has used a variety of technological and legal methods to prevent unauthorized parties from violating its terms of use by scraping, linking to, or accessing user postings for their own commercial purposes. For example, in April, craigslist obtained a $60.5 million judgment against a real estate listings site that had allegedly received scraped craigslist data from another entity. And craigslist recently reached a $31 million settlement and stipulated judgment with Instamotor, an online and app-based used car listing service, over claims that Instamotor scraped craigslist content to create listings on its own service and sent unsolicited emails to craigslist users for promotional purposes. (*Craigslist, Inc. v. Instamotor, Inc.*, No. 17-02449 (Stipulated Judgment and Permanent Injunction Aug. 3, 2017)).

Example: get artist information from national gallery

- National Gallery artist directory: https://web.archive.org/web/20121007172955/https://www.nga.gov/collection/anZ1.htm
- Save first by `requests.get`

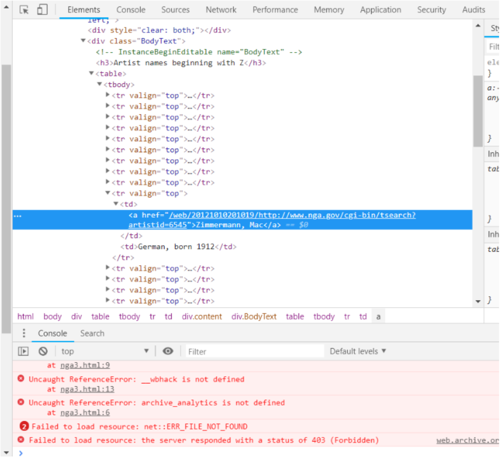
```
import requests
import csv
import os
from bs4 import BeautifulSoup

pages = []
filenames = []
save_path = './webdata'
if not os.path.exists(save_path):
    os.mkdir(save_path)

for i in range(1, 5):
    filename = './webdata/nga' + str(i) + '.html'
    url = 'https://web.archive.org/web/20121007172955/https://www.nga.gov/collection/anZ' + str(i) +
'.htm'
    if os.path.isfile(filename):
        pass
    else:
        print('Fetching the data from %s \n' % url)
        res = requests.get(url)
        c = res.text
        with open(filename,'w') as f:
            f.write(c)
    filenames.append(filename)
```

Know your html

- Inspect the page and locate the field you want to scrape



Search your html

1. Target texts are wrapped in `<div class="BodyText">` and always start with `a`.
2. Use `soup.find` to fetch the data

```
artist_name_list = soup.find(class_='BodyText')
artist_name_list_items = artist_name_list.find_all('a')
```

3. Obtain the name by `artist_name.contents`
4. Obtain the link by `artist_name.get('href')`
5. Exclude the unrelated text

```
last_links = soup.find(class_='AlphaNav')
last_links.decompose()
```

6. Save the data to `.csv` file

Full code

```
for item in filenames:
    with open(item, 'r') as f:
        c = f.read()
    soup = BeautifulSoup(c, 'html.parser')

    last_links = soup.find(class_='AlphaNav')
    last_links.decompose()

    # Create a file to write to, add headers row
    with open('z-artist-names.csv','w') as outfile:
        f = csv.writer(outfile)
        f.writerow(['Name', 'Link'])

        artist_name_list = soup.find(class_='BodyText')
        artist_name_list_items = artist_name_list.find_all('a')

        for artist_name in artist_name_list_items:
            names = artist_name.contents[0]
            links = 'https://web.archive.org' + artist_name.get('href')
            # Add each artist’s name and associated link to a row
            f.writerow([names, links])
```

Result

```
tmp = pd.read_csv('z-artist-names.csv',nrows=5)
display(tmp)
```

	Name	Link
0	Zorzo da Castelfranco	https://web.archive.org/web/20121010201041/htt...
1	Zox, Larry	https://web.archive.org/web/20121010201041/htt...
2	Zsissly	https://web.archive.org/web/20121010201041/htt...
3	Zuccarelli, Francesco	https://web.archive.org/web/20121010201041/htt...
4	Zuccarello, Anthony	https://web.archive.org/web/20121010201041/htt...

Other cool stuffs

- `mmap`: memory-mapping file
- `Spark` and `PySpark`: use "Apache Arrow" to enjoy the familiar pandas syntax and the speed of Spark
- `datatable`: Python version of `data.table` in R. (**No windows versoin available so far**)
- Large data processing benchmark comparison: https://h2oai.github.io/db-benchmark/

Homework1 on Compass

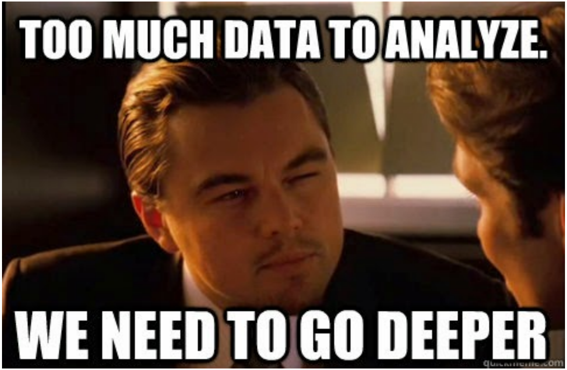
- Due on **Feb.11,11:59pm**
- Scrape data from https://www.isws.illinois.edu/statecli/cuweather/
- **The format of the pages changes over time!**
- Hint:

```
soup = BeautifulSoup(c, 'html.parser')
    trs = soup.find('table').find_all('tr')
    if year == 2014 and month < 10 :
        trs_data = trs[6:(6+day_of_month)]
    elif (year == 2014 and month >= 10 ) or year == 2015:
        trs_data = trs[XX:(XX+day_of_month)]
    elif year == 2016 and month == 12:
        trs_data = trs[XX:(XX+day_of_month)]
    else:
        trs_data = trs[XX:(XX+day_of_month)]

    for row in trs_data:
        tds = row.find_all('td')
        record = [year, month]
        if year == 2014 and month <= 10:
            for col_ind in [X,X,X,X,X,X]:
                record.append(tds[col_ind].text)
        else:
            for col_ind in [X,X,X,X,X,X]:
                record.append(tds[col_ind].text)
        f.writerow(record)
```

Get ready for Deep Learning

- Packages required: `tensorflow` (Python 3.4, 3.5, 3.6), `keras`, `skopt`
- Check installation guide: `pip`, `conda`



Acknowledgment and references

- These tutorial is inspired by the following posts:
  - https://plot.ly/ipython-notebooks/big-data-analytics-with-pandas-and-sqlite/
  - https://www.dataquest.io/blog/pandas-big-data/.
  - https://www.digitalocean.com/community/tutorials/how-to-scrape-web-pages-with-beautiful-soup-and-python-3
- Special thanks to James Balamuta for his advice