No More Sad Pandas: optimizing Pandas code for performance

Sofia Heisler

Lead Data Scientist, **Upside**

Download these slides at:
bit.ly/2rCVVUD

What's Pandas?

- Open-source library that offers data structure support and a great set of tools for data analysis

- Makes Python a formidable competitor to R and other data science tools

- Widely used in everything from simple data manipulation to complex machine learning

bit.ly/2rCVVUD

Why optimize Pandas?

- Pandas is built on top of NumPy and Cython, making it very fast when used correctly

- Correct optimizations can make the difference between minutes and milliseconds

bit.ly/2rCVVUD

Benchmarking
(A.k.a. why is my code so slow?)

Our working dataset

All hotels in New York state sold by Expedia

```python
import pandas as pd
import numpy as np
from math import *
```

```python
df = pd.read_csv('new_york_hotels.csv', encoding='cp1252')
```

```python
df.head()
```

	ean_hotel_id	name	address1	city	state_province	postal_code	latitude	longitude	star_rating	high_rate	low_rate
0	269955	Hilton Garden Inn Albany/SUNY Area	1389 Washington Ave	Albany	NY	12206	42.68751	-73.81643	3.0	154.0272	124.0216
1	113431	Courtyard by Marriott Albany Thruway	1455 Washington Avenue	Albany	NY	12206	42.68971	-73.82021	3.0	179.0100	134.0000
2	108151	Radisson Hotel Albany	205 Wolf Rd	Albany	NY	12205	42.72410	-73.79822	3.0	134.1700	84.1600
3	254756	Hilton Garden Inn Albany Medical Center	62 New Scotland Ave	Albany	NY	12208	42.65157	-73.77638	3.0	308.2807	228.4597
4	198232	CrestHill Suites SUNY University Albany	1415 Washington Avenue	Albany	NY	12206	42.68873	-73.81854	3.0	169.3900	89.3900

Source: http://developer.ean.com/database/property-data

Our example function

```python
def normalize(df, pd_series):
    pd_series = pd_series.astype(float)

    # Find upper and lower bound for outliers
    avg = np.mean(pd_series)
    sd  = np.std(pd_series)
    lower_bound = avg - 2*sd
    upper_bound = avg + 2*sd

    # Collapse in the outliers
    df.loc[pd_series < lower_bound , "cutoff_rate" ] = lower_bound
    df.loc[pd_series > upper_bound , "cutoff_rate" ] = upper_bound

    # Finally, take the log
    normalized_price = np.log(df["cutoff_rate"].astype(float))

    return normalized_price
```

Magic commands

- "Magic" commands available through Jupyter/ IPython notebooks provide additional functionality on top of Python code to make it that much more awesome

- Magic commands start with **%** (executed on just the line) or **%%** (executed on the entire cell)

Timing functions with **%timeit**

- Use IPython's **%timeit** command

- Re-runs a function repeatedly and shows the average and standard deviation of runtime obtained

- Can serve as a benchmark for further optimization

bit.ly/2rCVVUD

Timing functions with **%timeit**

```
%timeit df['hr_norm'] = normalize(df, df['high_rate'])

2.84 ms ± 180 µs per loop (mean ± std. dev. of 7 runs,
100 loops each)
```

Profiling with **line_profiler**

```
%load_ext line_profiler
%lprun -f normalize df['hr_norm'] =\
                    normalize(df, df['high_rate'])
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def normalize(df, pd_series):
     2         1          639    639.0      4.0      pd_series = pd_series.astype(float)
     3
     4                                              # Find upper and lower bound for ou
     5         1         1004   1004.0      6.3      avg = np.mean(pd_series)
     6         1          925    925.0      5.8      sd  = np.std(pd_series)
     7         1            4      4.0      0.0      lower_bound = avg - 2*sd
     8         1            2      2.0      0.0      upper_bound = avg + 2*sd
     9
    10                                              # Collapse in the outliers
    11         1         8567   8567.0     54.0      df.loc[pd_series < lower_bound , "c
    12         1         3959   3959.0     24.9      df.loc[pd_series > upper_bound , "c
    13
    14                                              # Finally, take the log
    15         1          776    776.0      4.9      normalized_price = np.log(df["cutof
    16
    17         1            2      2.0      0.0      return normalized_price
```

bit.ly/2rCVVUD

Slow Pandas: Looping

Our practice function: Haversine distance

```python
def haversine(lat1, lon1, lat2, lon2):
    miles_constant = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad,\
                            [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) *\
        np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    mi = miles_constant * c
    return mi
```

bit.ly/2rCVVUD

Crude iteration, or what not to do

- Rookie mistake: "I just wanna loop over all the rows!"

- Pandas is built on NumPy, designed for vector manipulation - loops are inefficient

- The Pandas **iterrows** method will provide a tuple of (Index, Series) that you can loop through - but it's quite slow

Running function with iterrows

```
%%timeit
haversine_series = []
for index, row in df.iterrows():
    haversine_series.append(haversine(40.671, -73.985,\
                    row['latitude'], row['longitude']))
df['distance'] = haversine_series
```

```
184 ms ± 6.65 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

bit.ly/2rCVVUD

Nicer looping: using apply

- **apply** applies a function along a specified axis (rows or columns)

- More efficient than **iterrows**, but still requires looping through rows

- Best used only when there is no way to vectorize a function

Timing looping with apply

```
%%timeit
df['distance'] =\
df.apply(lambda row: haversine(40.671, -73.985,\
         row['latitude'], row['longitude']), axis=1)
```

```
78.1 ms ± 7.55 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping with iterrows	184.00	
Looping with apply	**78.10**	**2.4x**

Apply is doing a lot of repetitive steps

```
%lprun -f haversine \
df.apply(lambda row: haversine(40.671, -73.985,\
         row['latitude'], row['longitude']), axis=1)
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
1					def haversine(lat1, l
2	1631	1429	0.9	3.1	miles_constant =
3	1631	17035	10.4	36.7	lat1, lon1, lat2,
4	1631	1669	1.0	3.6	dlat = lat2 - lat
5	1631	1143	0.7	2.5	dlon = lon2 - lon
6	1631	16049	9.8	34.6	a = np.sin(dlat/2
7	1631	6474	4.0	13.9	c = 2 * np.arcsin
8	1631	1586	1.0	3.4	mi = miles_consta
9	1631	1050	0.6	2.3	return mi

bit.ly/2rCVVUD

Vectorization

Doing it the pandorable way: vectorize

- The basic units of Pandas are arrays:

  - **Series** is a one-dimensional array with axis labels

  - **DataFrame** is a 2-dimensional array with labeled axes (rows and columns)

- **Vectorization** is the process of performing the operations on arrays rather than scalars

Why vectorize?

- Many built-in Pandas functions are built to operate directly on arrays (e.g. aggregations, string functions, etc.)

- Vectorized functions in Pandas are inherently much faster than looping functions

Vectorizing significantly improves performance

```
%%timeit
df['distance'] = haversine(40.671, -73.985,\
                    df['latitude'], df['longitude'])
```

```
1.79 ms ± 230 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The function is no longer looping

```
%lprun -f haversine haversine(40.671, -73.985,\
                              df['latitude'], df['longitude'])
```

Line #	Hits	Time	Per Hit	% Time	Line Contents
1					def haversine(lat1, lon
2	1	2	2.0	0.0	miles_constant = 39
3	1	529	529.0	8.8	lat1, lon1, lat2, lo
4	1	362	362.0	6.0	dlat = lat2 - lat1
5	1	232	232.0	3.9	dlon = lon2 - lon1
6	1	3511	3511.0	58.5	a = np.sin(dlat/2)*
7	1	869	869.0	14.5	c = 2 * np.arcsin(n
8	1	494	494.0	8.2	mi = miles_constant
9	1	2	2.0	0.0	return mi

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping with iterrows	184.00	
Looping with apply	78.10	2.4x
Vectorization with Pandas series	**1.79**	**43.6x**

bit.ly/2rCVVUD

Vectorization with NumPy arrays

Why NumPy?

- NumPy is a "fundamental package for scientific computing in Python"

- NumPy operations are executed "under the hood" in optimized, pre-compiled C code on **ndarray**s

- Cuts out a lot of the overhead incurred by operations on Pandas series in Python (indexing, data type checking, etc.)

Converting code to operate on NumPy arrays instead of Pandas series

```
%%timeit
df['distance'] = haversine(40.671, -73.985,\
        df['latitude'].values, df['longitude'].values)
```

370 μs ± 18 μs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Optimizing with NumPy arrays

- We've gotten our runtime down from 184 ms to 370 µs

- That's more than 500-fold improvement!

Methodology	Avg. single run time	Marginal performance improvement
Looping with iterrows	184.00	
Looping with apply	78.10	2.4x
Vectorization with Pandas series	1.79	43.6x
Vectorization with NumPy arrays	**0.37**	**4.8x**

bit.ly/2rCVVUD

Okay, but I really wanted to use a loop…

Okay, but I really want to use a loop…

- There are a few reasons why you might actually want to use a loop:

  - Your function is complex and does not yield itself easily to vectorization

  - Trying to vectorize your function would result in significant memory overhead

  - You're just plain stubborn

bit.ly/2rCVVUD

Using Cython to speed up loops

Speeding up code with Cython

- Cython language is a superset of Python that additionally supports calling C functions and declaring C types

- Almost any piece of Python code is also valid Cython code

- Cython compiler will convert Python code into C code which makes equivalent calls to the Python/C API.

bit.ly/2rCVVUD

Re-defining the function in the Cython compiler

```python
%load_ext cython

%%cython
cpdef haversine_cy(lat1, lon1, lat2, lon2):
    miles_constant = 3959
    lat1, lon1, lat2, lon2 = map(np.deg2rad,\
                                [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) *\
        np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arcsin(np.sqrt(a))
    mi = miles_constant * c
    return mi
```

Re-defining the function in the Cython compiler

```
%%timeit
df['distance'] =\
    df.apply(lambda row: haversine_cy(40.671, -73.985,\
        row['latitude'], row['longitude']), axis=1)


76.5 ms ± 6.42 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```

Scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping with iterrows	184.00	
Looping with apply	78.10	2.4x
Running row-wise function through Cython compiler	**76.50**	**1.0x**
Vectorization with Pandas series	1.79	43.6x
Vectorization with NumPy arrays	0.37	4.8x

bit.ly/2rCVVUD

Evaluating results of conversion to Cython

Adding the **-a** option to **%%cython** magic command shows how much of the code has *not* actually been converted to C by default… and it's a lot!

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
 01:
 02: # Haversine cythonized (no other edits)
+03: import numpy as np
+04: cpdef haversine_cy(lat1, lon1, lat2, lon2):
+05:     miles_constant = 3959
+06:     lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
+07:     dlat = lat2 - lat1
+08:     dlon = lon2 - lon1
+09:     a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
+10:     c = 2 * np.arcsin(np.sqrt(a))
+11:     mi = miles_constant * c
+12:     return mi
```

bit.ly/2rCVVUD

Speeding up code with Cython

- As long as Cython is still using Python methods, we won't see a significant improvement

- Make the function more Cython-friendly:

  - Add explicit typing to the function

  - Replace Python/NumPy libraries with C-specific math libraries

Better cythonizing through static typing and C libraries

```python
%%cython -a
from libc.math cimport sin, cos, acos, asin, sqrt

cdef deg2rad_cy(float deg):
    cdef float rad
    rad = 0.01745329252*deg
    return rad


cpdef haversine_cy_dtyped(float lat1, float lon1, float lat2, float lon2):
    cdef:
        float dlon
        float dlat
        float a
        float c
        float mi

    lat1, lon1, lat2, lon2 = map(deg2rad_cy, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    mi = 3959 * c
    return mi
```

bit.ly/2rCVVUD

Timing the cythonized function

```
%%timeit
df['distance'] =\
df.apply(lambda row: haversine_cy_dtyped(40.671, -73.985,\
         row['latitude'], row['longitude']), axis=1)
```

```
50.1 ms ± 2.74 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

Scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping with iterrows	184.00	
Looping with apply	78.10	2.4x
Running row-wise function through Cython compiler	76.50	1.0x
Looping with Cythoninzed function	**50.10**	**1.6x**
Vectorization with Pandas series	1.79	28x
Vectorization with NumPy arrays	0.37	4.8x

bit.ly/2rCVVUD

Our code is looking a lot more Cythonized, too

Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

```
 01:  # Haversine cythonized
 02:  from libc.math cimport sin, cos, acos, asin, sqrt
 03:
+04:  cpdef deg2rad_cy(float deg):
 05:      cdef float rad
+06:      rad = 0.01745329252*deg
+07:      return rad
 08:
+09:  cpdef haversine_cy_dtyped(float lat1, float lon1, float lat2, float lon2):
 10:      cdef:
 11:          float dlon
 12:          float dlat
 13:          float a
 14:          float c
 15:          float mi
 16:
+17:      lat1, lon1, lat2, lon2 = map(deg2rad_cy, [lat1, lon1, lat2, lon2])
+18:      dlat = lat2 - lat1
+19:      dlon = lon2 - lon1
+20:      a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
+21:      c = 2 * asin(sqrt(a))
+22:      mi = 3959 * c
+23:      return mi
```

bit.ly/2rCVVUD

Summing it up

bit.ly/2rCVVUD

The scoreboard

Methodology	Avg. single run time (ms)	Marginal performance improvement
Looping with iterrows	184.00	
Looping with apply	78.10	2.4x
Looping with Cython	50.10	1.6x
Vectorization with Pandas series	1.79	28x
Vectorization with NumPy arrays	0.37	4.8x

bit.ly/2rCVVUD

The zen of Pandas optimization

- Avoid loops

- If you must loop, use apply, not iteration functions

- If you must apply, use Cython to make it faster

- Vectorization is usually better than scalar operations

- Vector operations on NumPy arrays are more efficient than on native Pandas series

bit.ly/2rCVVUD

A word of warning…

Source: https://xkcd.com/1691/
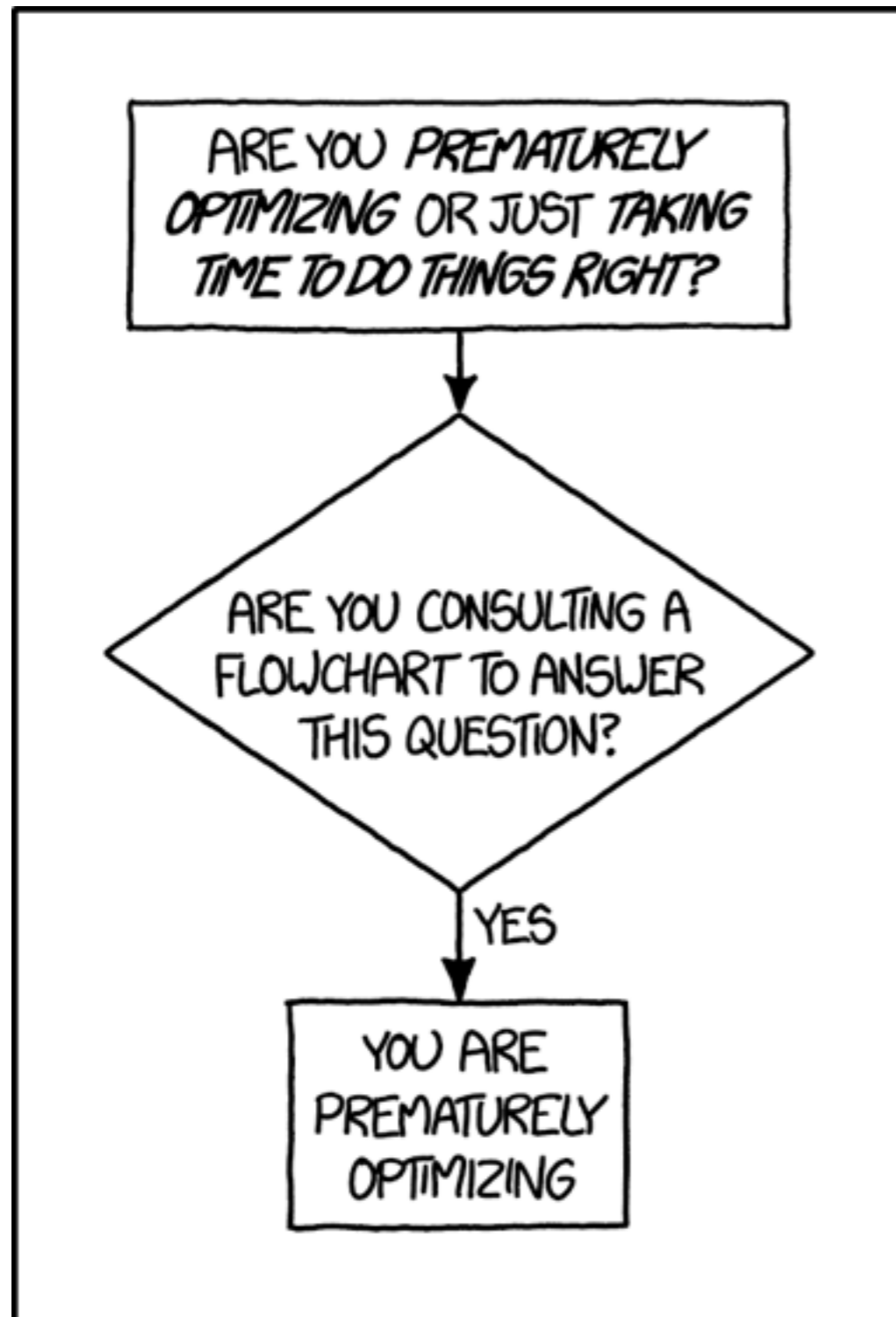
"Premature optimization is the root of all evil"

Bonus pitch…

- We're hiring!

- Check us out at **upside.com** or come talk to me!

References

- http://cython.readthedocs.io/en/latest/

- http://cython.org/

- http://pandas.pydata.org/pandas-docs/stable/

- http://www.nongnu.org/avr-libc/user-manual/group__avr__math.html

- https://docs.python.org/2/library/profile.html

- https://docs.scipy.org/doc/numpy/user/whatisnumpy.html

- https://ipython.org/notebook.html

- https://penandpants.com/2014/09/05/performance-of-pandas-series-vs-numpy-arrays/

- https://www.datascience.com/blog/straightening-loops-how-to-vectorize-data-aggregation-with-pandas-and-numpy/