NumPy/Pandas Speed

CMPT 732, Fall 2018

I gave my undergrad class a Pandas entity resolution problem and was surprised when students started asking “is it okay that my program takes two minutes?” when mine took a second or two to run. Here's what I learned…

Why So Slow?

Here's a reduced version of the problem: we'll create this DataFrame:

```
n = 100000000
df = pd.DataFrame({
    'a': np.random.randn(n),
    'b': np.random.randn(n),
    'c': np.random.randn(n),
})
```

… and (for some reason) we want to calculate

$$\sin(a-1)+1\,.$$

The underlying problem: DataFrames are an abstraction of what's really going on. Underneath, there's some memory being moved around and computation happening.

The abstraction is leaky, as they all are.

Having a sense of what's happening behind-the-scenes will help us use the tools effectively.

One fact to notice: each Pandas `Series` is stored as a NumPy array.

i.e. this is an array that *is already in memory*, so refering to it is basically free.

```
df['col'].values
```

This isn't in memory (in this form) and must be constructed:

```
df.iloc[0] # a row object
```

So, any time we operate on a **Pandas series** as a unit, it's probably going to be fast.

Pandas is *column-oriented*: the thing that it stores in contiguous memory is a column.

NumPy Expression

The solution I was hoping for:

```
def do_work_numpy(a):
    return np.sin(a - 1) + 1

result = do_work_numpy(df['a'])
```

The arithmetic is done as single operations on NumPy arrays.

The `np.sin` and the `+`/`-` operations are done by NumPy at C speeds (with SSE/AVX instructions in my installation).

Running time: 1.96 s.

Applying to a Series

Can apply a function to each value in a series:

```
def do_work(a):
    return math.sin(a - 1) + 1

result = df['a'].apply(do_work)
```

The `do_work` function gets called n times (once for each element in the series). Arithmetic done in Python.

Running time: 35.9 s.

Vectorizing

Or something that looks like a NumPy vector operations, because that's what you call it:

```
def do_work(a):
    return math.sin(a - 1) + 1
do_work_vector = np.vectorize(do_work, otypes=[np.float])

result = do_work_vector(df['a'])
```

The `do_work` function *still* gets called n times, but it's hidden by `vectorize`, which makes it *look* like a NumPy function. Arithmetic still done in Python.

Running time: 29.9 s.

Applying By Row

Applying over the rows of the DataFrame:

```
def do_work_row(row):
    return math.sin(row['a'] - 1) + 1

result = df.apply(do_work_row, axis=1)
```

This is a by-row application: `do_work_row` is called on *every row* in the DataFrame. But the rows don't exist in memory, so they must be constructed. Then the function called, and arithmetic done in Python.

Running time: 977 s.

Using Python

Every assignment in that course had a “no loops” restriction, which prevented:

```
def do_work_python(a):
    result = np.empty(a.shape)
    for i in range(a.size):
        result[i] = math.sin(a[i] - 1) + 1
    return result

result = do_work_python(df['a'])
```

The loop is done in Python; the arithmetic is done in Python.

Running time: 1370 s.

With `numexpr`

Let's look again at the best-so-far version:

```
def do_work_numpy(a):
    return np.sin(a - 1) + 1

result = do_work_numpy(df['a'])
```

NumPy has to calculate and store each intermediate result, which creates overhead. This is a limitation of Python: it asks NumPy to calculate `a-1`, then calls `np.sin` on the result, then adds to that result.

The `numexpr` package overcomes this: has its own expression syntax that gets compiled internally. Then you can apply that expression (to the local variables in scope).

```
import numexpr
def do_work_numexpr(a):
    expr = 'sin(a - 1) + 1'
    return numexpr.evaluate(expr)

result = do_work_numexpr(df['a'])
```

This way, the whole expression can be calculated (on each element, in some C code somewhere, multi-threaded), and the result stored in a new array.

Running time: 0.405 s.

Can also access `numexpr` functionality as `pd.eval(expr, engine='numexpr')`.

Summary

Method	Time	Relative Time
NumPy expression	1.96 s	1.00
Series.apply	35.9 s	18.37
Vectorized	29.9 s	15.26
DataFrame.apply	977 s	499.21
Python loop	1370 s	700.38
numexpr	0.405 s	0.21

Lessons:

- The abstractions you're using need to be in the back of your head somewhere.
- Moving data around in memory is expensive.
- Python is still slow, but NumPy (and friends) do a good job insulating us from that.

Don't believe me? [Notebook with the code](#).

Don't believe me even more? [A Beginner’s Guide to Optimizing Pandas Code for Speed](#). [I did it first, I swear!]

We saw the same kind of thing with Spark DataFrames (and UDFs):

- Using whole-DataFrame operations was an order of magnitude faster.
- Trying to work with individual rows came with a cost of speed and code readability.
- Working at the right level of abstraction made everything nicer.