Big Data, Small Machine

Data Science Singapore - 20160616

Adam Drake

@aadrake

http://aadrake.com

@aadrake #hacker

@aadrake #thoughtleader

Claims:

RAM is growing faster than data

Many techniques for dealing with *Big Data*

One machine is fine for ML

The Mega Trend is creating Analytics Cluster Sprawl & Siloed Big Data Application and Data Clusters

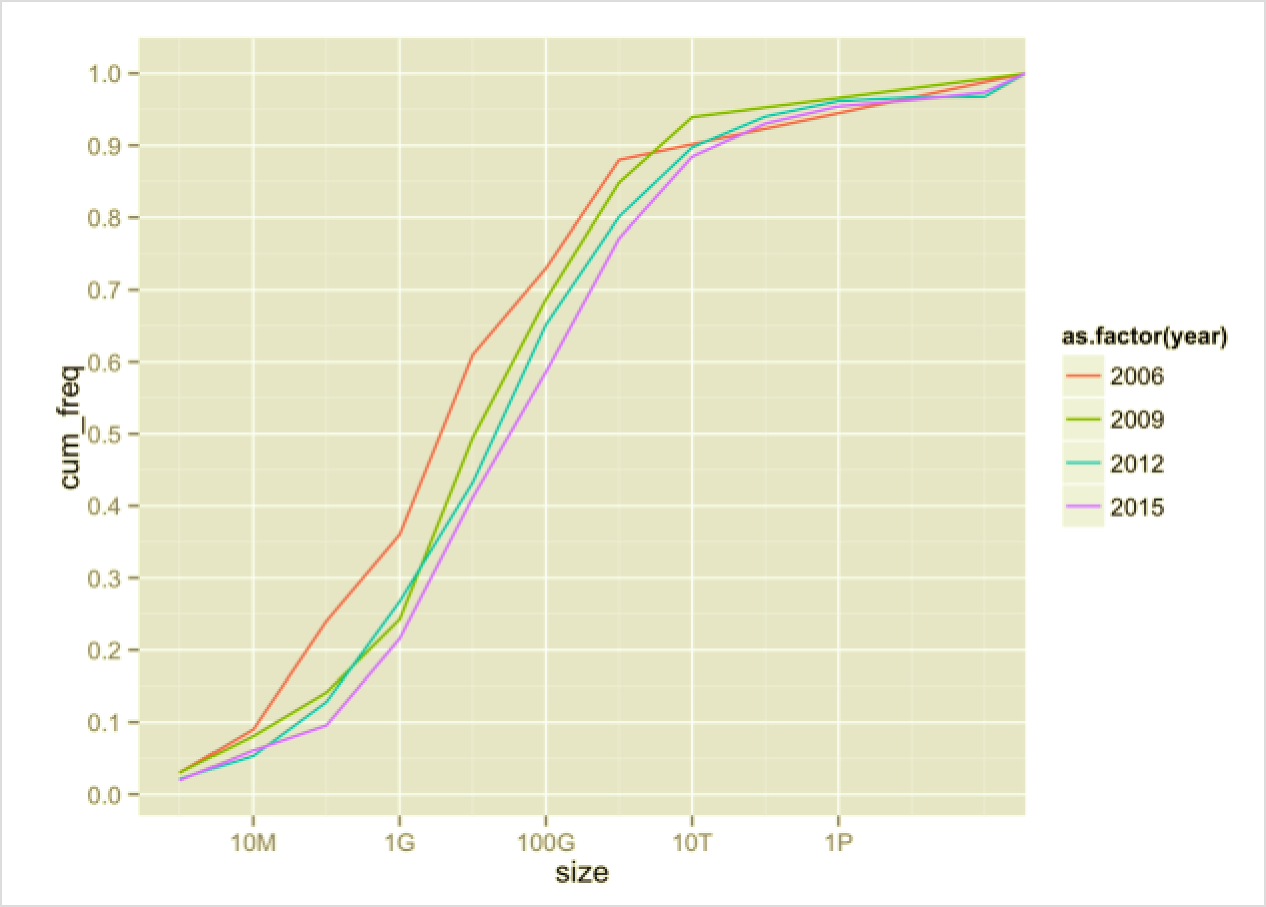
Analytics Pipeline

- *Multiple steps of analytics processing*
- *Each with different computing characteristics*
- *Requiring separate cluster for each step*

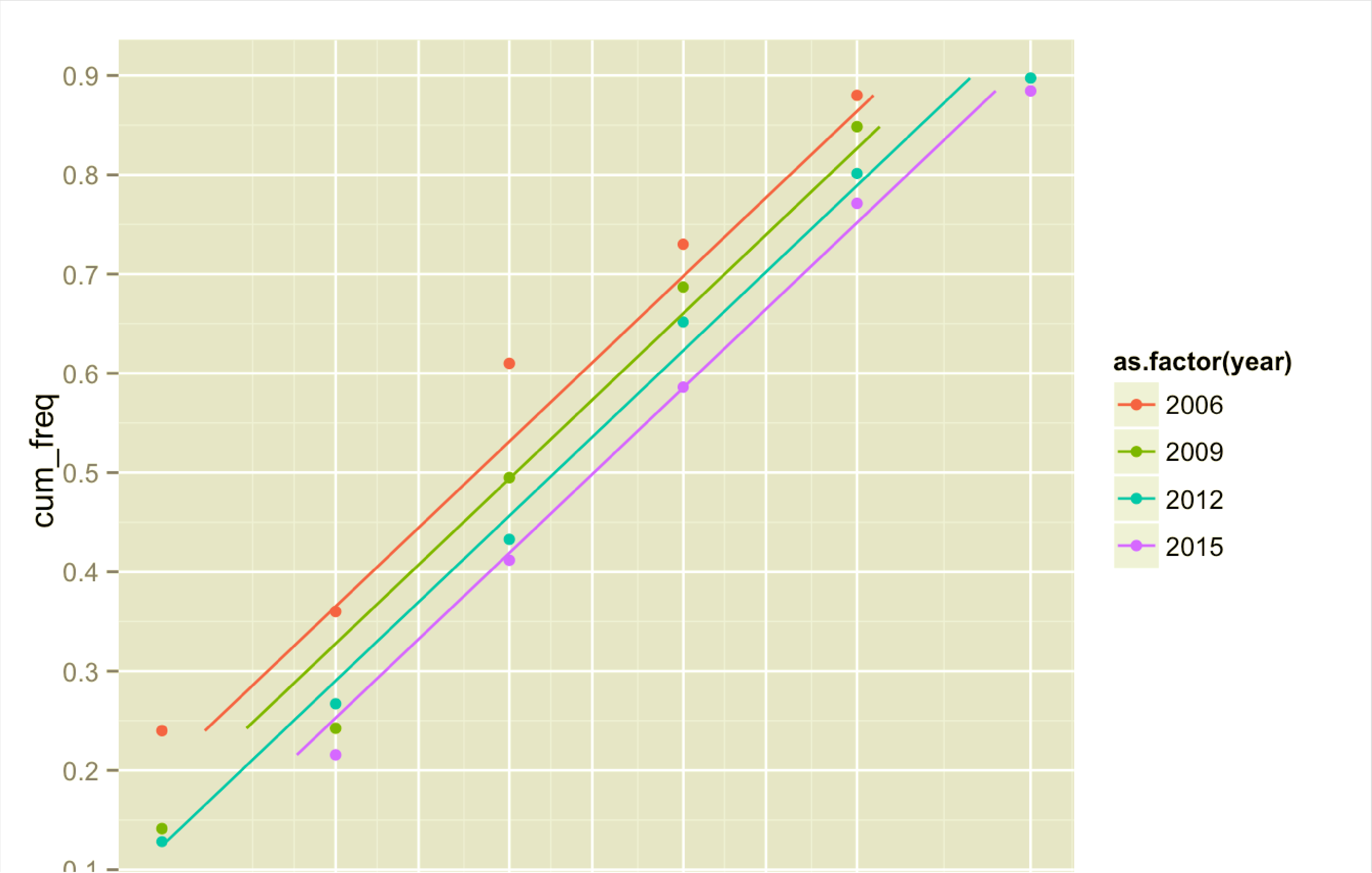
High TCO

- *Large datacenter footprint*
- *High management cost*
- *Significant data movement*

Step 0: More RAM?



Source: http://datascience.la/big-ram-is-eating-big-data-size-of-datasets-used-for-analytics


cum_freq
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
as.factor(year)
2006
2009
2012
2015

Big RAM is eating big data

Big EC2 instance RAM size increase by **50%** y/y

Year	Type	RAM (GiB)
2007	m1.xlarge	15
2009	m2.4xlarge	68
2012	hs1.8xlarge	117
2014	r3.8xlarge	244
2016	x1.32xlarge	1952

⇒ single node in-memory analytics forever!?

Note: Tyan FT76-B7922 has 6TB RAM

Source: https://github.com/ogrisel/decks/tree/master/2016_pydata_berlin

Step 1: Sampling

Online advertising: 0.17% CTR

or...

~ 20 clicks per 10,000 views

Source: http://www.smartinsights.com/internet-advertising/internet-advertising-analytics/display-advertising-clickthrough-rates/

We need

- Data source
- Stateless feature extraction
- Model which supports incremental learning

Data Source

```
def getRecord(path, numFeatures):
    count = 0
    for i, line in enumerate(open(path)):
        if i == 0:
            # do whatever you want at initialization
            x = [0] * numFeatures # So we don't need to create a new x every time
            continue
        for t, feat  in enumerate(line.strip().split(',')):
            if t == 0:
                y = feat # assuming first position in record is some kind of label
            else:
                # do something with the features
                x[m] = feat
        yield (count, x, y)
```

Or...

```
reader = pd.read_csv('blah.csv', chunksize=10000)

for chunk in reader:
    doSomething(chunk)
```

Stateless Feature Extraction

Hello hashing trick...

Assume data like

```
fname,lname,location
Adam,Drake,Singapore
```

```python
features = ['fnameAdam', 'lnameDrake', 'locationSingapore']

maxWeights = 2**25

def hashedFeatures(list):
    hashes = [hash(x) for x in features]
    return [x % maxWeights for x in hashes]

print(hashedFeatures(features))
# [18445008, 8643786, 20445187]
```

These are the indices in the weights array

Incremental learning

Just use any model in sklearn which has a `partial_fit()` method

- Classification
  - sklearn.naive_bayes.MultinomialNB
  - sklearn.naive_bayes.BernoulliNB
  - sklearn.linear_model.Perceptron
  - sklearn.linear_model.SGDClassifier
  - sklearn.linear_model.PassiveAggressiveClassifier
- Regression
  - sklearn.linear_model.SGDRegressor
  - sklearn.linear_model.PassiveAggressiveRegressor
- Clustering

Incremental learning contd.

Not all models can handle stateless features and will need to know the classes in advance. Check the documentation and presence of `classes` argument for `partial_fit()`

Or...

Just stick with `SGDClassifier` and `SGDRegressor`.

Or write your own...

```python
# Turn the record into a list of hash values
x = [0]  # 0 is the index of the bias term
for key, value in record.items():
    index = int(value + key[1:], 16) % D  # weakest hash ever ;)
    x.append(index)

# Get the prediction for the given record (now transformed to hash values)
wTx = 0.
for i in x:  # do wTx
    wTx += w[i] # w[i] * x[i], but if i in x we got x[i] = 1.
p = 1. / (1. + exp(-max(min(wTx, 20.), -20.)))  # bounded sigmoid

# Update the loss
p = max(min(p, 1. - 10e-12), 10e-12)
loss += -log(p) if y == 1. else -log(1. - p)

# Update the weights
for i in x:
    # alpha / (sqrt(n) + 1) is the adaptive learning rate heuristic
    # (p - y) * x[i] is the current gradient
    # note that in our case, if i in x then x[i] = 1
    w[i] -= (p - y) * alpha / (sqrt(n[i]) + 1.)
    n[i] += 1.
```

Current logloss: 0.463056

Run time 1h06m36s

Power up: Multicore

Use all cores

Lock shared memory to prevent non-atomic modifications

But GIL...

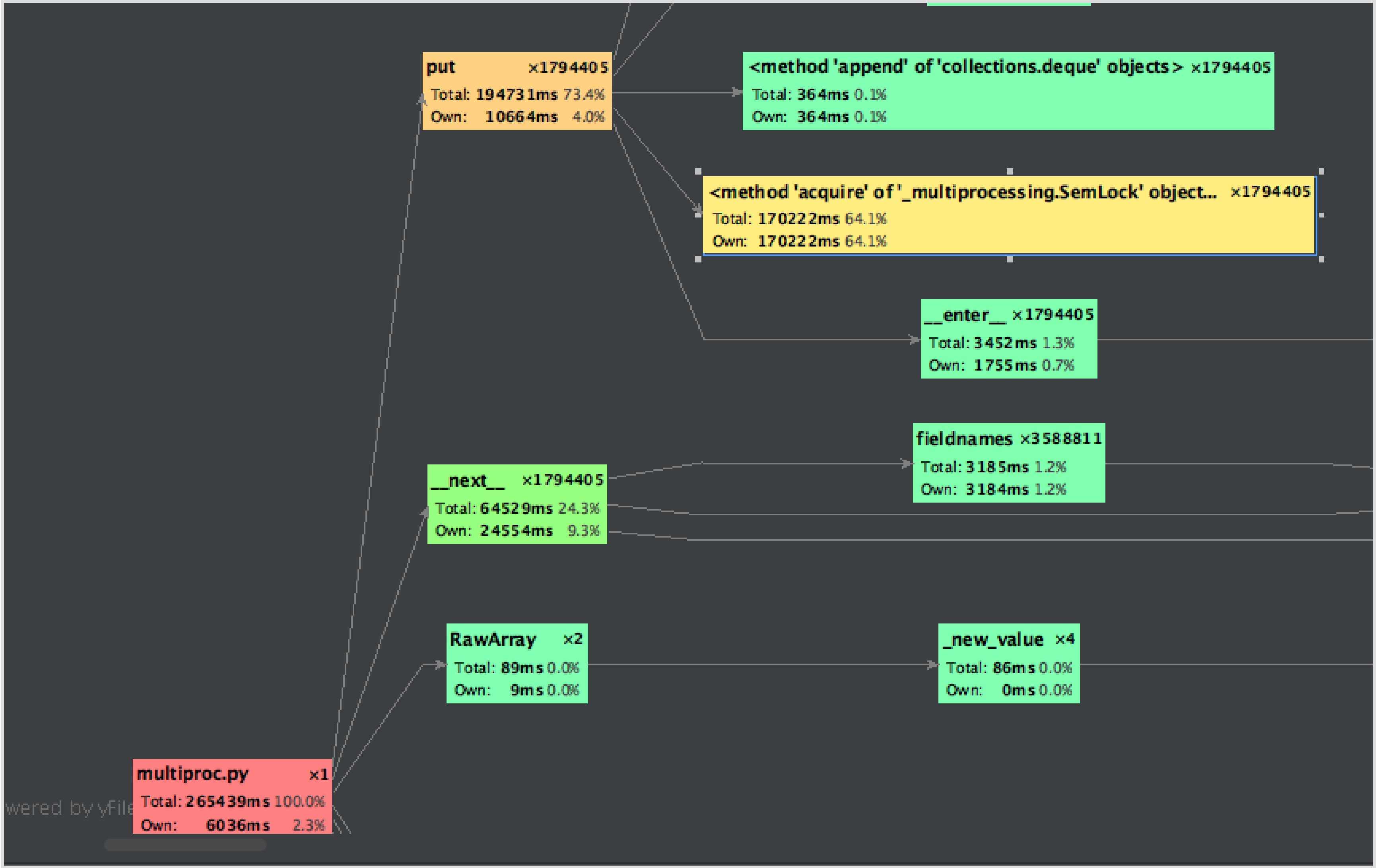
No two **threads** may execute Python bytecode at once

```python
from multiprocessing.sharedctypes import RawArray
from multiprocessing import Process
import time
import random

def incr(arr, i):
    time.sleep(random.randint(1, 4))
    arr[i] += 1
    print(arr[:])

arr = RawArray('d', 10)

procs = [Process(target=incr, args=(arr,i)) for i in range(10)]

for p in procs:
    p.start()
for p in procs:
    p.join()
'''
[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0]
[0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0]
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
'''
```

```python
from multiprocessing import Queue

procs = [Process(target=worker, args=(q, w, n, D, alpha, loss, count,)) \
        for x in range(4)]
for p in procs:
    p.start()

for t, row in enumerate(DictReader(open(train))):
    q.put(row)
```

```python
from multiprocessing import Queue

procs = [Process(target=worker, args=(q, w, n, D, alpha, loss, count,)) \
        for x in range(4)]
for p in procs:
    p.start()

for t, row in enumerate(DictReader(open(train))):
    q.put(row)
```

But it's **SLOWER**


put ×1794405
Total: 194731ms 73.4%
Own: 10664ms 4.0%
<method 'append' of 'collections.deque' objects> ×1794405
Total: 364ms 0.1%
Own: 364ms 0.1%
<method 'acquire' of '_multiprocessing.SemLock' object... ×1794405
Total: 170222ms 64.1%
Own: 170222ms 64.1%
__enter__ ×1794405
Total: 3452ms 1.3%
Own: 1755ms 0.7%
fieldnames ×3588811
Total: 3185ms 1.2%
Own: 3184ms 1.2%
__next__ ×1794405
Total: 64529ms 24.3%
Own: 24554ms 9.3%
RawArray ×2
Total: 89ms 0.0%
Own: 9ms 0.0%
_new_value ×4
Total: 86ms 0.0%
Own: 0ms 0.0%
multiproc.py ×1
Total: 265439ms 100.0%
Own: 6036ms 2.3%
wered by yFile

Go

```go
// Hash the record values
 for i, v := range record {
     hashResult := hash([]byte(fields[i] + v)) % int(D)
     x[i+1] = int(math.Abs(float64(hashResult)))
 }

 // Get the prediction for the given record (now transformed to hash values)
 wTx := 0.0
 for _, v := range x {
     wTx += (*w)[v]
 }
 p := 1.0 / (1.0 + math.Exp(-math.Max(math.Min(wTx, 20.0), -20.0)))

 // Update the loss
 p = math.Max(math.Min(p, 1.-math.Pow(10, -12)), math.Pow(10, -12))
 if y == 1 {
     *loss += -math.Log(p)
 } else {
     *loss += -math.Log(1.0 - p)
 }

 // Update the weights
 for _, v := range x {
     (*w)[v] = (*w)[v] - (p-float64(y))*alpha/(math.Sqrt((*n)[v])+1.0)
     (*n)[v]++
 }
```

```python
# Turn the record into a list of hash values
x = [0]  # 0 is the index of the bias term
for key, value in record.items():
    index = int(value + key[1:], 16) % D  # weakest hash ever ;)
    x.append(index)

# Get the prediction for the given record (now transformed to hash values)
wTx = 0.
for i in x:  # do wTx
    wTx += w[i] # w[i] * x[i], but if i in x we got x[i] = 1.
p = 1. / (1. + exp(-max(min(wTx, 20.), -20.)))  # bounded sigmoid

# Update the loss
p = max(min(p, 1. - 10e-12), 10e-12)
loss += -log(p) if y == 1. else -log(1. - p)

# Update the weights
for i in x:
    # alpha / (sqrt(n) + 1) is the adaptive learning rate heuristic
    # (p - y) * x[i] is the current gradient
    # note that in our case, if i in x then x[i] = 1
    w[i] -= (p - y) * alpha / (sqrt(n[i]) + 1.)
    n[i] += 1.
```

Python

Current logloss: 0.463056

Run time 1h06m36s

Go

Current logloss: 0.459211

Run time 8m22s

Power up: Multicore

Use all cores and lock weights to prevent non-atomic modifications

```
for i := 0; i < 4; i++ {
        wg.Add(1)
        go worker(input, fields, &w, &n, D, alpha, &loss, &count, &wg, mutex)
    }
```

Python

Current logloss: 0.463056

Run time 1h06m36s

Go

Current logloss: 0.459211

Run time 8m22s

Go (4 cores)

Current logloss: 0.459252

Run time 7m3s

Stochastic Gradient Descent

$$\min Q(w) = \sum_{i=1}^{N} Q_i(w)$$

$$Solve : \nabla Q(w) = 0$$

$$w := w - \eta \nabla Q_i(w)$$

```
// Update the weights
for _, v := range x {
  (*w)[v] = (*w)[v] - (p-float64(y))*alpha/(math.Sqrt((*n)[v])+1.0)
  (*n)[v]++
}
```

Source: Robbins and Monro, 1950

More efficient multicore

How about round-robin updates?

Multiple cores compute gradients but only one at a time updates weight vector.

This is a bit faster

Source: http://papers.nips.cc/paper/3888-slow-learners-are-fast.pdf

What if we just ditched the locks?

HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

https://arxiv.org/abs/1106.5730

Python

Current logloss: 0.463056

Run time 1h06m36s

11,471 RPS

Multicore Go (4 cores, no locks)

Current logloss: 0.455223

Run time 3m55s

195,066 RPS

@aadrake

http://aadrake.com

Questions?