Distributed Machine Learning with Dask

Vitalii Duk

Lead Data Scientist
EMLC

follow this presentation online

https://github.com/root-ua/dask-intro

Python - scientific ecosystem

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis

scikit-learn
Powerful Machine Learning library

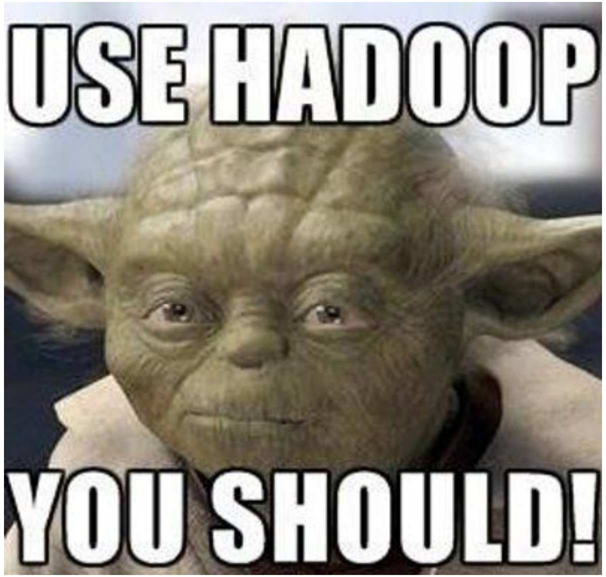
… and a lot more, but most of those tools are mostly designed to run on 1 machine.

Big Data?

- NYC Taxi Data
  - available on http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml

- nearly 1.5 billion records
- more than 500GB in size

Big Data!

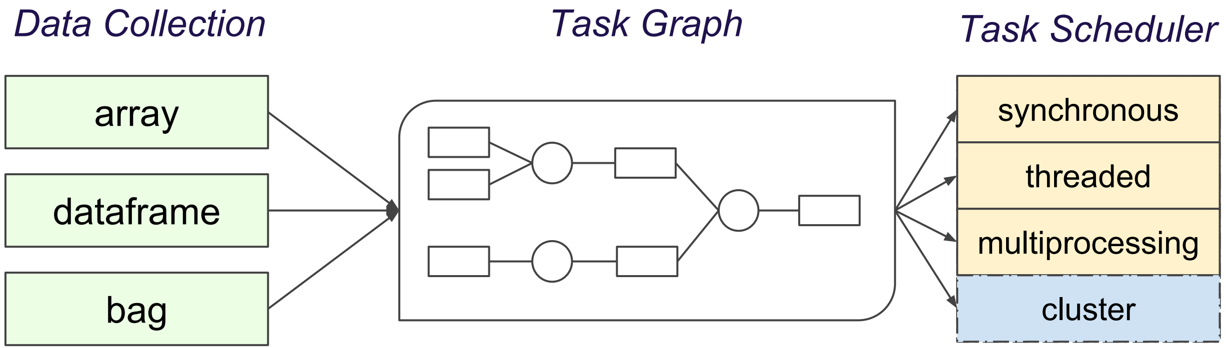


dask - overview

- Distributed computing framework, written in pure **Python**
- Supports *out-of-core* computations for datasets that don't fit RAM
- Provides a parallelized version of **NumPy** arrays and **Pandas** DataFrames
- Use ***lazy*** evaluation

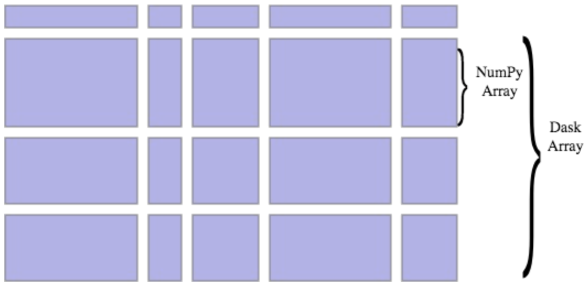
- Consists of three main components:
  - Distributed/Parallel Data Collections: **Arrays, DataFrames, Bags** (lists)
  - Task Graph
  - Dynamic Task Scheduler

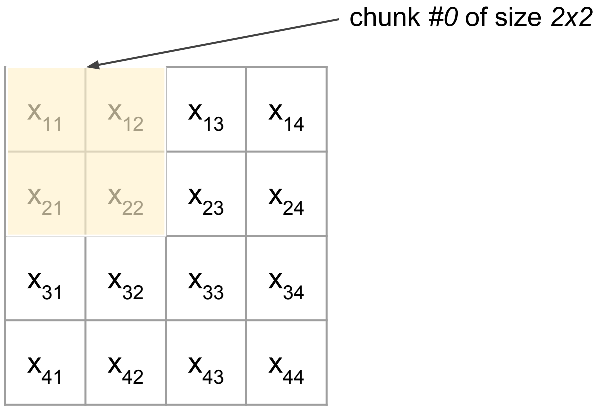
dask - architecture



dask.array

```
In [24]: import dask.dataframe as dd

In [25]: X = da.random.random((4, 4), chunks=(2,2))

In [26]: X

Out[26]: dask.array<da.random.random_sample, shape=(4, 4), dtype=float64, chunksize=(2, 2)>

In [27]: X.npartitions

Out[27]: 4
```





dask.array - dot product

NumPy example

In [1]:
```python
import numpy as np
```

In [2]:
```python
a = np.random.random((3, 3))
b = np.random.random((3, 3))
```

In [3]:
```python
a
```

Out[3]:
```
array([[0.87213254, 0.68547721, 0.50865265],
       [0.87727514, 0.09465553, 0.88171158],
       [0.8388894 , 0.86770436, 0.53854953]])
```

In [4]:
```python
b
```

Out[4]:
```
array([[0.02044914, 0.6765649 , 0.95178781],
       [0.72093418, 0.27771347, 0.46948846],
       [0.91204126, 0.80118586, 0.68841198]])
```

In [5]:
```python
a.dot(b)
```

Out[5]:
```
array([[0.97593052, 1.18794584, 1.50207134],
       [0.89033726, 1.32623553, 1.48640027],
       [1.13389169, 1.24001458, 1.57656583]])
```

dask.array - dot product

DASK

NumPy example

```
In [1]: import numpy as np

In [2]: a = np.random.random((3, 3))
        b = np.random.random((3, 3))

In [3]: a

Out[3]: array([[0.87213254, 0.68547721, 0.50865265],
               [0.87727514, 0.09465553, 0.88171158],
               [0.8388894 , 0.86770436, 0.53854953]])

In [4]: b

Out[4]: array([[0.02044914, 0.6765649 , 0.95178781],
               [0.72093418, 0.27771347, 0.46948846],
               [0.91204126, 0.80118586, 0.68841198]])

In [5]: a.dot(b)

Out[5]: array([[0.97593052, 1.18794584, 1.50207134],
               [0.89033726, 1.32623553, 1.48640027],
               [1.13389169, 1.24001458, 1.57656583]])
```

Dask Array example

split original array into parallel collections

```
In [1]: import dask.array as da

In [2]: a = da.random.random((3, 3), chunks=(1, 1))
        b = da.random.random((3, 3), chunks=(1, 1))

In [3]: a

Out[3]: dask.array<da.random.random_sample, shape=(3, 3), dtype=float64, chunksize=(1, 1)>

In [4]: b

Out[4]: dask.array<da.random.random_sample, shape=(3, 3), dtype=float64, chunksize=(1, 1)>

In [5]: a.dot(b)

Out[5]: dask.array<sum-aggregate, shape=(3, 3), dtype=float64, chunksize=(1, 1)>

In [6]: a.dot(b).compute()

Out[6]: array([[0.80716022, 0.51040152, 1.22022674],
               [0.42065195, 0.23507697, 0.52136113],
               [1.23128921, 0.62149032, 0.73284065]])
```

create task graph

initialize computations

dask.array - distributed dot product

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw + by & ax + bz \\ cw + dx & cx + dz \end{bmatrix}$$

x_{11}	x_{12}	x_{13}	x_{14}
x_{21}	x_{22}	x_{23}	x_{24}
x_{31}	x_{32}	x_{33}	x_{34}
x_{41}	x_{42}	x_{43}	x_{44}

y_{11}	y_{12}	y_{13}	y_{14}
y_{21}	y_{22}	y_{23}	y_{24}
y_{31}	y_{32}	y_{33}	y_{34}
y_{41}	y_{42}	y_{43}	y_{44}

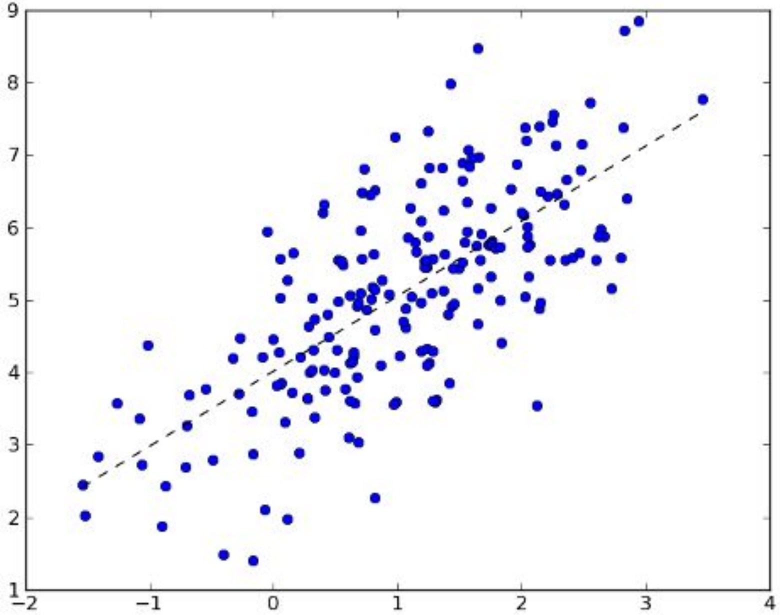
$x_{11}y_{11} + x_{12}y_{21} \longrightarrow$ *reduce(sum)* $\longrightarrow$ $\mathbf{r_{11}}$

$x_{11}y_{12} + x_{12}y_{22} \longrightarrow$ *reduce(sum)* $\longrightarrow$ $\mathbf{r_{12}}$

$x_{21}y_{11} + x_{22}y_{21} \longrightarrow$ *reduce(sum)* $\longrightarrow$ $\mathbf{r_{21}}$

$x_{21}y_{12} + x_{22}y_{22} \longrightarrow$ *reduce(sum)* $\longrightarrow$ $\mathbf{r_{22}}$

Linear Regression - recap



$$y = w_0 + w_1 x_1 + \cdots + w_n x_n = \overrightarrow{x^T} \vec{w}$$

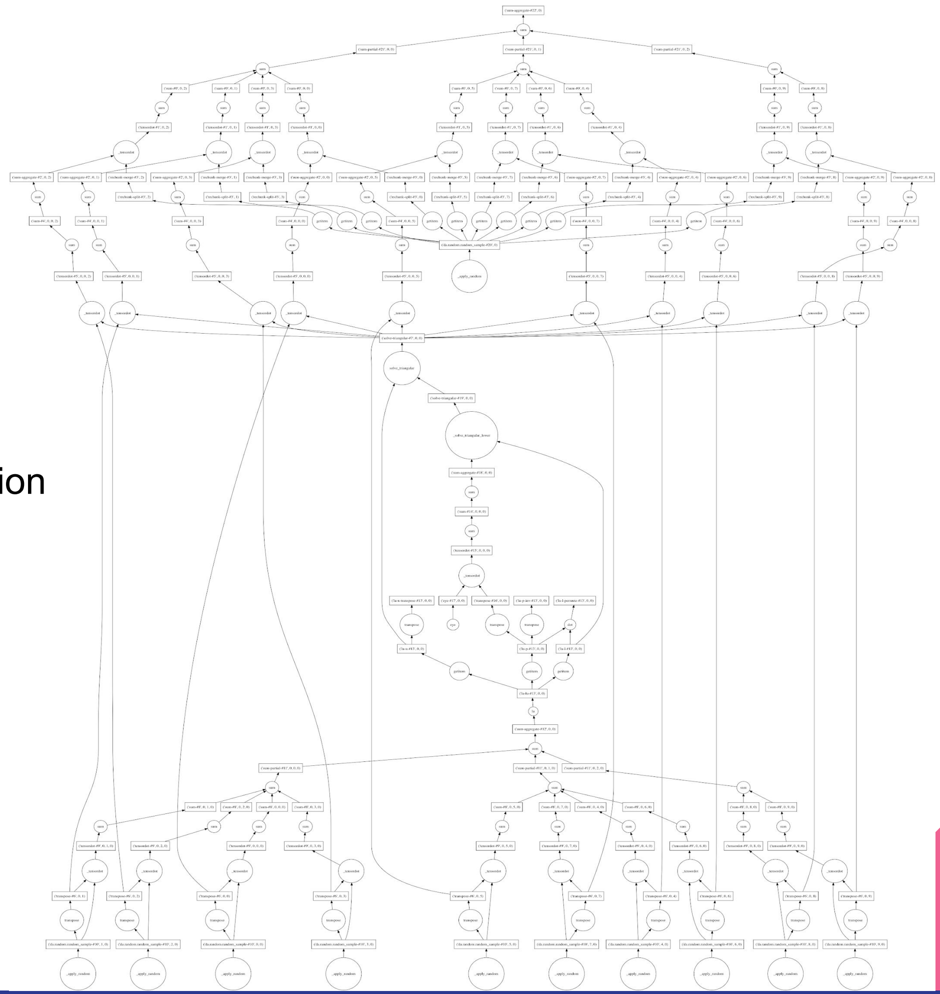
$$\vec{w} = (X^T X)^{-1} X^T \vec{y}$$

dask.array - solving OLS

```
In [4]: X = da.random.random((10000, 20), chunks=(1000, 20))

In [5]: X.shape

Out[5]: (10000, 20)

In [6]: y = da.random.random(10000, chunks=(10000))

In [7]: y.shape

Out[7]: (10000,)

In [8]: w = da.dot(da.dot(da.linalg.inv(da.dot(X.T, X)), X.T), y)

In [9]: w

Out[9]: dask.array<sum-aggregate, shape=(20,), dtype=float64, chunksize=(20,)>

In [10]: w = w.compute()

In [11]: w.shape

Out[11]: (20,)

In [12]: w

Out[12]: array([0.05041846, 0.06319238, 0.06461881, 0.0627598 , 0.06543031,
                0.04299085, 0.05784944, 0.04401282, 0.04984113, 0.0539256 ,
                0.04060992, 0.04105089, 0.03104954, 0.04578884, 0.03814298,
                0.06422944, 0.03839978, 0.02420742, 0.05382565, 0.05480423])
```

$$\vec{w} = (X^T X)^{-1} X^T \vec{y}$$

dask.array



Dask Task Graph visualization
for OLS computations

dask.array - summary

- an interface to a set of NumPy arrays
- using blocked algorithms to perform computations with a distributed NumPy array collection
- allows working with large numerical arrays that don't fit RAM
- coordinate block algorithms using a task graph
- implements almost all typical NumPy array methods and interfaces:
    - arithmetics and scalar operations: `+, *, exp, log, …`
    - reduction functions: `sum(), mean(), var(), std()`
    - matrix transpose: `x.T`
    - array slicing: `x[10:1000, :-2]`
    - linear algebra algorithms: `SVD, QR decompositions, OLS`

dask.dataframe - example

In [1]:
```
%pylab inline
```
```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [3]:
```
from dask.distributed import Client, progress
```

In [4]:
```
client = Client()
# client = Client('127.0.0.1:8786')
client
```

Out[4]:

Client	Cluster
• **Scheduler:** tcp://127.0.0.1:54846 • **Dashboard:** http://127.0.0.1:8787	• **Workers:** 8 • **Cores:** 8 • **Memory:** 17.18 GB

In [5]:
```
import numpy as np
import pandas as pd
import dask.dataframe as dd
```

dask.dataframe - example

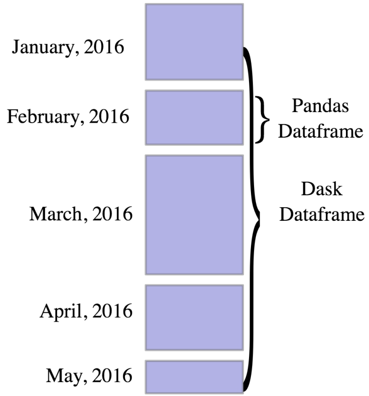
In [6]:
```
%%time

ddf = dd.read_csv('./data/yellow_tripdata_2016-*.csv', assume_missing=True, skip_blank_lines=True, error_bad_lines=False)
```
```
CPU times: user 174 ms, sys: 31.7 ms, total: 206 ms
Wall time: 199 ms
```

In [7]:
```
%%time

ddf.columns
```
```
CPU times: user 12 µs, sys: 1 µs, total: 13 µs
Wall time: 16.2 µs
```
Out[7]:
```
Index(['VendorID', 'tpep_pickup_datetime', 'tpep_dropoff_datetime',
       'passenger_count', 'trip_distance', 'pickup_longitude',
       'pickup_latitude', 'RatecodeID', 'store_and_fwd_flag',
       'dropoff_longitude', 'dropoff_latitude', 'payment_type', 'fare_amount',
       'extra', 'mta_tax', 'tip_amount', 'tolls_amount',
       'improvement_surcharge', 'total_amount'],
      dtype='object')
```

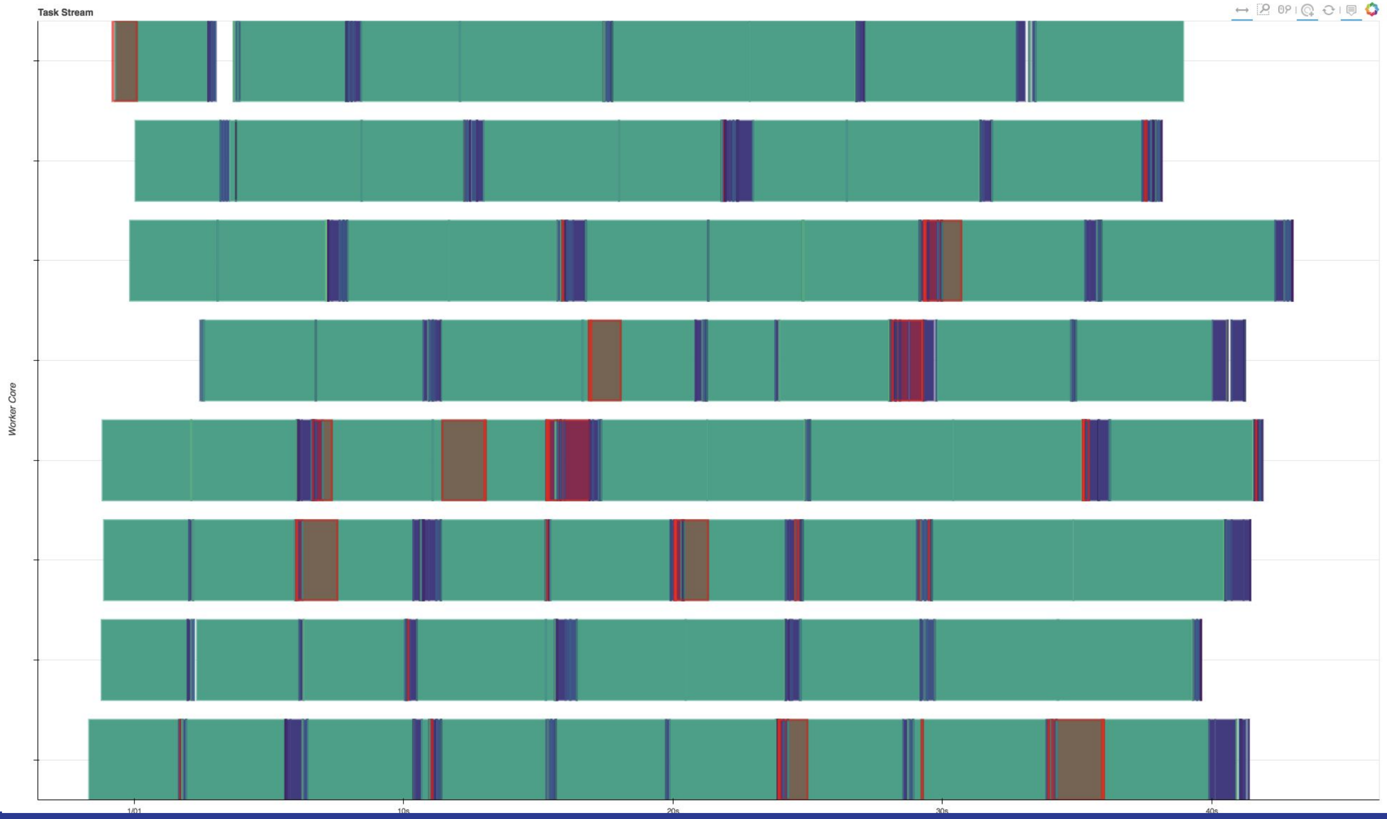
In [8]:
```
%%time

ddf['total_amount'] = ddf['total_amount'].astype(float)
```
```
CPU times: user 2.77 ms, sys: 144 µs, total: 2.92 ms
Wall time: 2.83 ms
```

In [9]:
```
%%time

ddf['total_amount'].mean().compute()
```
```
CPU times: user 9.21 s, sys: 11.7 s, total: 20.9 s
Wall time: 1min 17s
```
Out[9]:
```
16.055111934521598
```

In [10]:
```
%%time

ddf.head()
```
```
CPU times: user 215 ms, sys: 352 ms, total: 567 ms
Wall time: 2.19 s
```



dask.dataframe - visualization



dask.delayed - overview

- allows distributedly run code that don’t fit standart Dask data structures - **Arrays** and **DataFrames**
- adds an ability to build **Task Graphs** of any complexity

In [ ]:
```
baseline = 1e6

results = []

for record in records:

    for feature in features:

        size = record[feature]

        if size < baseline:
            f = simple_computational_function(size)
            results.append(f)
        else:
            f = complex_computational_function(size)
            results.append(f)
```

dask.delayed

In [1]:
```
%pylab inline
```
```
Populating the interactive namespace from numpy and matplotlib
```

In [2]:
```
import numpy as np
from time import sleep
from dask import delayed
```

In [3]:
```
def avg(elements):
    sleep(1)
    return sum(elements) / len(elements)
```

In [4]:
```
def add(a, b):
    sleep(1)
    return a + b
```

In [5]:
```
l1 = np.random.random(10)
l2 = np.random.random(20)
```

In [6]:
```
%%time
l1_avg = avg(l1)
l2_avg = avg(l2)
result = add(l1_avg, l2_avg)
```
```
CPU times: user 915 µs, sys: 1.43 ms, total: 2.34 ms
Wall time: 3.01 s
```

In [7]:
```
result
```

Out[7]:
```
0.9784548776155466
```

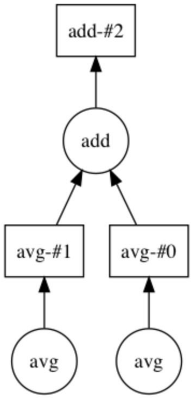
dask.delayed

```
In [1]: %pylab inline
Populating the interactive namespace from numpy and matplotlib

In [2]: import numpy as np
        from time import sleep
        from dask import delayed

In [3]: def avg(elements):
            sleep(1)
            return sum(elements) / len(elements)

In [4]: def add(a, b):
            sleep(1)
            return a + b

In [5]: l1 = np.random.random(10)
        l2 = np.random.random(20)

In [6]: %%time
        l1_avg = avg(l1)
        l2_avg = avg(l2)
        result = add(l1_avg, l2_avg)

CPU times: user 915 µs, sys: 1.43 ms, total: 2.34 ms
Wall time: 3.01 s

In [7]: result

Out[7]: 0.9784548776155466
```

```
In [8]: %%time
        l1_avg = delayed(avg)(l1)
        l2_avg = delayed(avg)(l2)
        result = delayed(add)(l1_avg, l2_avg)

CPU times: user 755 µs, sys: 214 µs, total: 969 µs
Wall time: 829 µs
```

lazy evaluation

```
In [9]: result

Out[9]: Delayed('add-1dfe5140-0da1-4d48-b577-7551f4cdcd55')

In [10]: result.visualize()

Out[10]:
```




```
In [11]: %%time
         result.compute()

CPU times: user 6.39 ms, sys: 5.02 ms, total: 11.4 ms
Wall time: 2.01 s

Out[11]: 0.9784548776155466
```

dask-ml - overview

- a wrapper that allows using ***scikit-learn*** on much bigger datasets

- available features:
    - post-fit computations: `predict, predict_proba`
    - preprocessing: `MinMaxScaler, StandardScaler, Categorizer`
    - hyper parameter search: `GridSearchCV, RandomizedSearchCV`
    - parallel meta-estimators: `RandomForest`
    - incremental learning: `PartialSGDRegressor, PartialSGDClassifier`
    - clustering: `KMeans, SpectralClustering`

dask-ml - example

```
In [1]: %pylab inline
```

Populating the interactive namespace from numpy and matplotlib

```
In [2]: from dask.distributed import Client, progress

        client = Client()
        # client = Client('127.0.0.1:8786')
        client
```

Out[2]:

Client	Cluster
• **Scheduler:** tcp://127.0.0.1:56171	• **Workers:** 8
• **Dashboard:** http://127.0.0.1:56172/status	• **Cores:** 8
	• **Memory:** 17.18 GB

```
In [3]: import dask_ml.datasets
        import dask_ml.cluster
        import matplotlib.pyplot as plt
```

```
In [4]: %%time
        X, y = dask_ml.datasets.make_blobs(n_samples=10000000, chunks=1000000, random_state=0, centers=3)
        X = X.persist()
        X
```

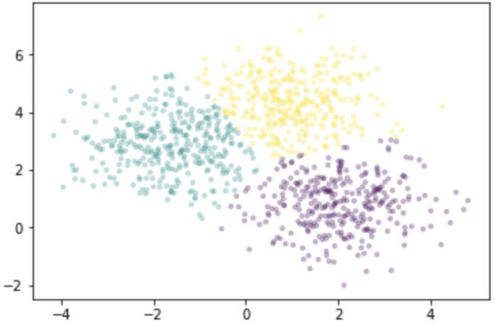
CPU times: user 3.33 s, sys: 613 ms, total: 3.94 s
Wall time: 3.61 s

```
In [5]: %%time
        km = dask_ml.cluster.KMeans(n_clusters=3, init_max_iter=2, oversampling_factor=10)
        km.fit(X)
```

CPU times: user 2.32 s, sys: 1.93 s, total: 4.25 s
Wall time: 11.4 s

```
In [6]: %%time
        fig, ax = plt.subplots()
        ax.scatter(X[::10000, 0], X[::10000, 1], marker='.', c=km.labels_[::10000],
                   cmap='viridis', alpha=0.25);
```

CPU times: user 297 ms, sys: 112 ms, total: 409 ms
Wall time: 1.52 s



Fin! Questions?