Intro To Spark

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Firing Up The Hands-On

Let’s make sure all is well with our hands-on environment first.

Grab a node

```
interact
```

We have hundreds of packages on Bridges. They each have many paths and variables that need to be set for their own proper environment, and they are often conflicting. We shield you from this with the wonderful modules command. You can load the two packages we will be using as

Spark

```
module load spark
```

Tensorflow (Not now! But later)

```
module load tensorflow/1.5_gpu
```

Copy over our exercise data, and move into an interesting starting point

```
cp -r ~training/BigData .
cd BigData/Shakespear
```

Start up Spark

```
pyspark
```

You may see a lot of noise (warning! Warning!) which you should ignore as long as you get a nice “Wecome to Spark” message at the end.

Our Setup For This Workshop

After you copy the files from the training directory**, you will have:**

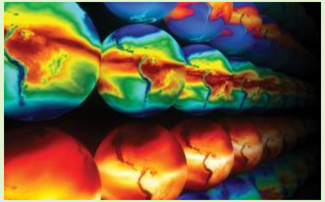
```
/BigData
        /Clustering
        /MNIST
        /Recommender
        /Shakespeare
```

Datasets, and also cut and paste code samples are in here.

The Shift to Big Data

Pan-STARRS telescope
http://pan-starrs.ifa.hawaii.edu/public/

Genome sequencers
(Wikipedia Commons)



NOAA climate modeling
http://www.ornl.gov/info/ornlreview/v42_3_09/article02.shtml

New Emphases

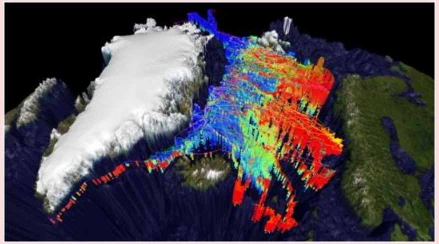
Social networks and the Internet

Video
Wikipedia Commons

Library of Congress stacks
https://www.flickr.com/photos/danlem2001/6922113091/

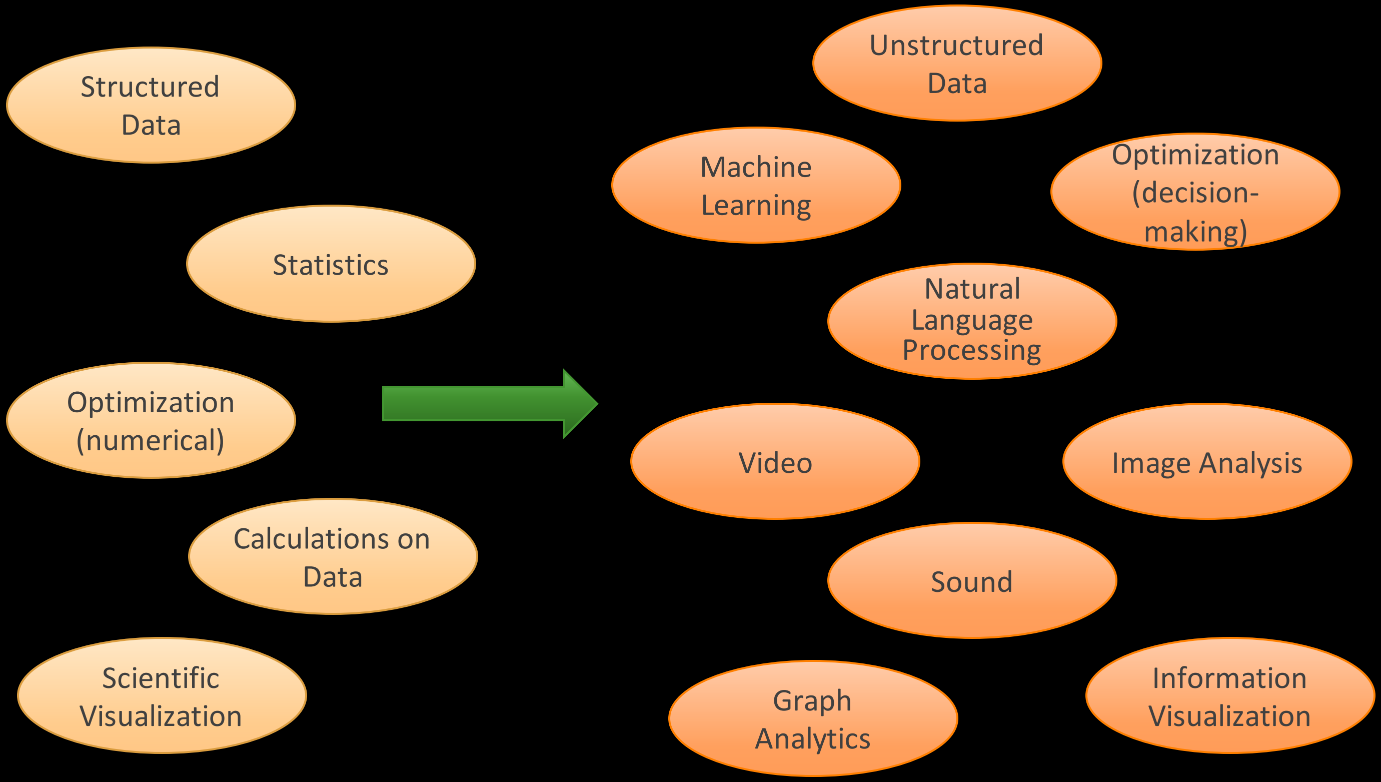
Collections
Horniman museum: http://www.horniman.ac.uk/get_involved/blog/bioblitz-insects-reviewed

Legacy documents
Wikipedia Commons



Environmental sensors: Water temperature profiles from tagged hooded seals
http://www.arctic.noaa.gov/report11/biodiv_whales_walrus.html

Challenges and Software are Co-Evolving



Structured Data

Statistics

Optimization (numerical)

Calculations on Data

Scientific Visualization

Unstructured Data

Machine Learning

Optimization (decision-making)

Natural Language Processing

Video

Image Analysis

Sound

Graph Analytics

Information Visualization

Once there was only small data...

A classic amount of “small” data

Find a tasty appetizer – Easy!

Find something to use up these oranges – grumble...

What if....

...then data got **BIG**.

8TB for $130

Whys:

Storage got cheap

So why not keep it all?

Today data is a hot commodity $

And we got better at generating it

= facebook
IoT
Science...

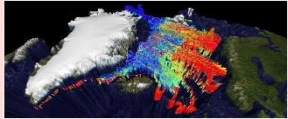
10 TB *

Pan-STARRS

Genome sequencers
(Wikipedia Commons)

Horniman museum:
http://www.horniman.ac.uk
get_involved/blog/bioblitz-insects-reviewed

Wikipedia
Commons



http://www.arctic.noaa.gov/report11/biodiv_whales_walrus.html

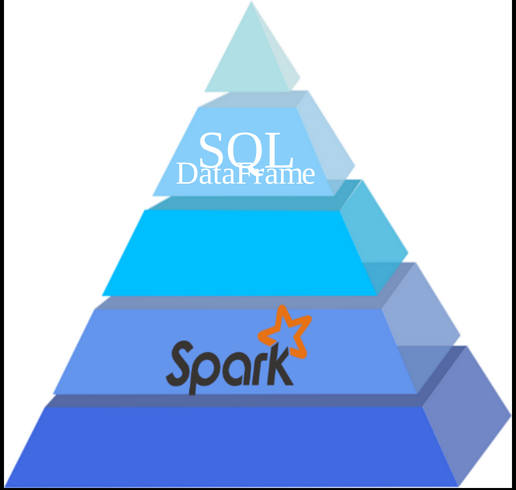
**Actually, a silly estimate. The original refere entions a more accurate 208TB, and in 2013 the digital collection alone was 3PB.*

Parallel Frameworks for Data

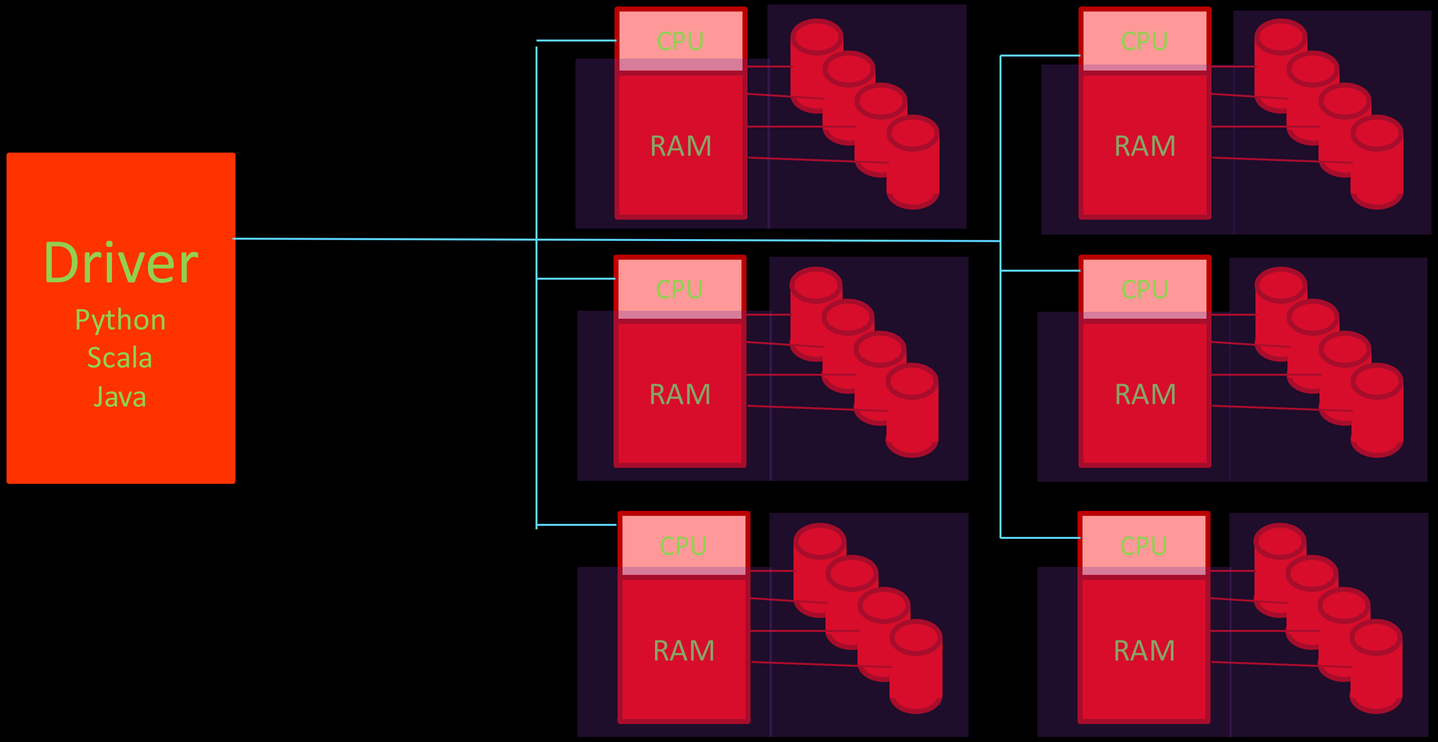
These are both frameworks for distributing and retrieving data. Hadoop is focused on disk based data and a basic map-reduce scheme, and Spark evolves that in several directions that we will get in to. Both can accommodate multiple types of databases and *achieve their performance gains by using parallel workers*.

The mother of Hadoop was necessity. It is trendy to ridicule its primitive design, but it was the first step.

We have repurposed many of these blocks to build a better framework.



Spark Idea



Driver

Python

Scala

Java

CPU

RAM

CPU

RAM

CPU

RAM

CPU

RAM

CPU

RAM

CPU

RAM

RDD

Resilient Distributed Dataset

Spark Formula

1. Create/Load RDD

   Webpage visitor IP address log

2. *Transform* RDD

   ”Filter out all non-U.S. IPs”

3. But don’t do anything yet!

   Wait until data is actually needed
   Maybe apply more transforms (“distinct IPs)

4. Perform *Actions* that return data

   Count “How many unique U.S. visitors?”

Simple Example

```
>>> lines_rdd = sc.textFile("nasa_19950801.tsv")
```

Read into RDD

Spark Context

The first thing a Spark program requires is a context, which interfaces with some kind of cluster to use. Our pyspark shell provides us with a convenient *sc*, using the local filesystem, to start. Your standalone programs will have to specify one:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("Test_App")
sc = SparkContext(conf = conf)
```

You would typically run these scripts like so:

```
spark-submit Test_App.py
```

Simple Example

```
>>> lines_rdd = sc.textFile("nasa_19950801.tsv")

>>> stanfordLines_rdd = lines_rdd.filter(lambda line: "stanford" in line)

>>> stanfordLines_rdd.count()
47

>>> stanfordLines_rdd.first()
u'glim.stanford.edu\t-\t807258357\tGET\t/shuttle/missions/61-c/61-c-patch-small.gif\t‘
```

Read into RDD

Transform

Actions

<u>Lambdas</u>

We’ll see a lot of these. A lambda is simply a function that is too simple to deserve its own subroutine. Anywhere we have a lambda we could also just name a real subroutine that could go off and do anything.

When all you want to do is see if “*given an input variable line, is “stanford” in there?*”, it isn’t worth the digression.

Most modern languages have adopted this nicety.

Common Transformations

Transformation	Result
map(func)	Return a new RDD by passing each element through *func*.
filter(func)	Return a new RDD by selecting the elements for which *func* returns true.
flatMap(func)	*func* can return multiple items, and generate a sequence, allowing us to “flatten” nested entries (JSON) into a list.
distinct()	Return an RDD with only distinct entries.
sample(…)	Various options to create a subset of the RDD.
union(RDD)	Return a union of the RDDs.
intersection(RDD)	Return an intersection of the RDDs.
subtract(RDD)	Remove argument RDD from other.
cartesian(RDD)	Cartesian product of the RDDs.
parallelize(list)	Create an RDD from this (Python) list (using a spark context).

Same Size

Fewer Elements

More Elements

Full list at http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Common Actions

Transformation	Result
collect()	Return all the elements from the RDD.
count()	Number of elements in RDD.
countByValue()	List of times each value occurs in the RDD.
reduce(func)	Aggregate the elements of the RDD by providing a function which combines any two into one (sum, min, max, …).
first(), take(n)	Return the first, or first n elements.
top(n)	Return the n highest valued elements of the RDDs.
takeSample(…)	Various options to return a subset of the RDD..
saveAsTextFile(path)	Write the elements as a text file.
foreach(func)	Run the *func* on each element. Used for side-effects (updating accumulator variables) or interacting with external systems.

Full list at http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Pair RDDs

- Key/Value organization is a simple, but often very efficient schema, as we mentioned in our NoSQL discussion.

- Spark provides special operations on RDDs that contain key/value pairs. They are similar to the general ones that we have seen.

- On the language (Python, Scala, Java) side key/values are simply tuples. If you have an RDD whose elements happen to be tuples of two items, it is a Pair RDD and you can use the key/value operations that follow.

Pair RDD Transformations

Transformation	Result
reduceByKey(func)	Reduce values using *func*, but on a key by key basis. That is, combine values with the same key.
groupByKey()	Combine values with same key. Each key ends up with a list.
sortByKey()	Return an RDD sorted by key.
mapValues(func)	Use *func* to change values, but not key.
keys()	Return an RDD of only keys.
values()	Return an RDD of only values.

Note that all of the regular transformations are available as well.

Two Pair RDD Transformations

Transformation	Result
subtractByKey(otherRDD)	Remove elements with a key present in other RDD.
join(otherRDD)	Inner join: Return an RDD containing all pairs of elements with matching keys in self and other. Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in self and (k, v2) is in other.
leftOuterJoin(otherRDD)	For each element (k, v) in self, the resulting RDD will either contain all pairs (k, (v, w)) for w in other, or the pair (k, (v, None)) if no elements in other have key k.
rightOuterJoin(otherRDD)	For each element (k, w) in other, the resulting RDD will either contain all pairs (k, (v, w)) for v in this, or the pair (k, (None, w)) if no elements in self have key k.
cogroup(otherRDD)	Group data from both RDDs by key.

Full list at http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

Simple Example

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])

>>> y = sc.parallelize([("a", 2), ("a", 3)])

>>> z = x.join(y)

>>> z.collect()
[('a', (1, 2)), ('a', (1, 3))]
```

Pair RDD Actions

As with transformations, all of the regular actions are available to Pair RDDs, and there are some additional ones that can take advantage of key/value structure.

Transformation	Result
countByKey()	Count the number of elements for each key.
lookup(key)	Return all the values for this key.

Full list at http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

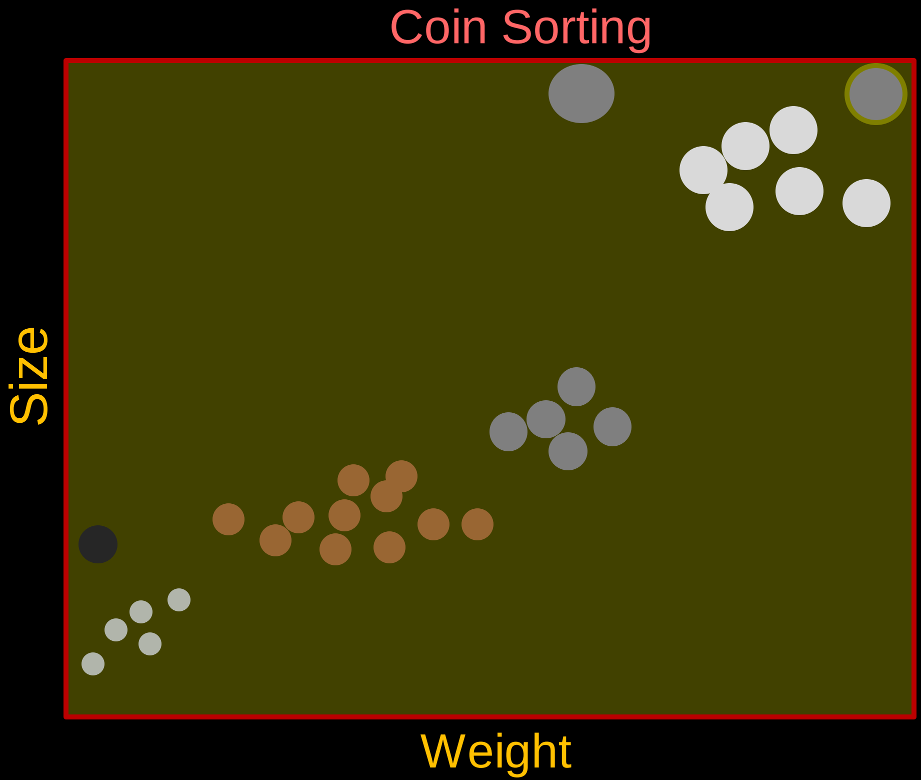
MLib

MLib rolls in a lot of classic machine learning algorithms. We barely have time to touch upon this interesting topic today, but they include:

- Useful data types
- Basic Statistics
- Classification (including SVMs, Random Forests)
- Regression
- Dimensionality Reduction (Princ. Comp. Anal., Sing. Val. Decomp.)
- Algorithms (SGD,…)
- Clustering…

Clustering

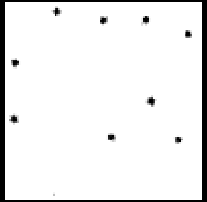
Clustering is a very common operation for finding grouping in data and has countless applications. This is a very simple example, but you will find yourself reaching for a clustering algorithm frequently in pursuing many diverse machine learning objectives, sometimes as one part of a pipeline.



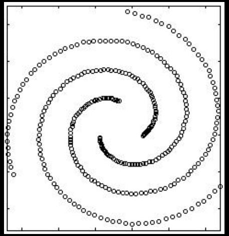
Clustering

As intuitive as clustering is, it presents challenges to implement in an efficient and robust manner.

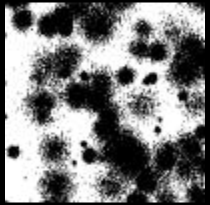
You might think this is trivial to implement in lower dimensional spaces.



But it can get tricky even there.



Sometimes you know how many clusters you have to start with. Often you don't.
How hard can it be to count clusters? How many are here?



We will start with 5000 2D points. We want to figure out how many clusters there are, and their centers. Let's fire up pyspark and get to it…

Finding Clusters

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\
      /_/

Using Python version 2.7.5
SparkContext available as
>>>
>>> rdd1 = sc.textFile("50
>>>
>>> rdd2 = rdd1.map(lambda
>>> rdd3 = rdd2.map(lambda
>>>
```

```
br06% interact
...
r288%
r288% module load spark
r288% pyspark
```

DD

words and integers

is around

*RDD map() takes a function to apply to the elements. We can certainly create our own separate function, but lambdas are a way many languages allow us to define trivial functions “in place”.

Finding Our Way

```
>>> rdd1 = sc.textFile("5000_points.txt")
>>> rdd1.count()
5000
>>> rdd1.take(4)
[u'    664159    550946', u'    665845    557965', u'    597173    575538', u'    618600    551446']
>>> rdd2 = rdd1.map(lambda x:x.split())
>>> rdd2.take(4)
[[u'664159', u'550946'], [u'665845', u'557965'], [u'597173', u'575538'], [u'618600', u'551446']]
>>> rdd3 = rdd2.map(lambda x: [int(x[0]),int(x[1])])
>>> rdd3.take(4)
[[664159, 550946], [665845, 557965], [597173, 575538], [618600, 551446]]
>>>
```

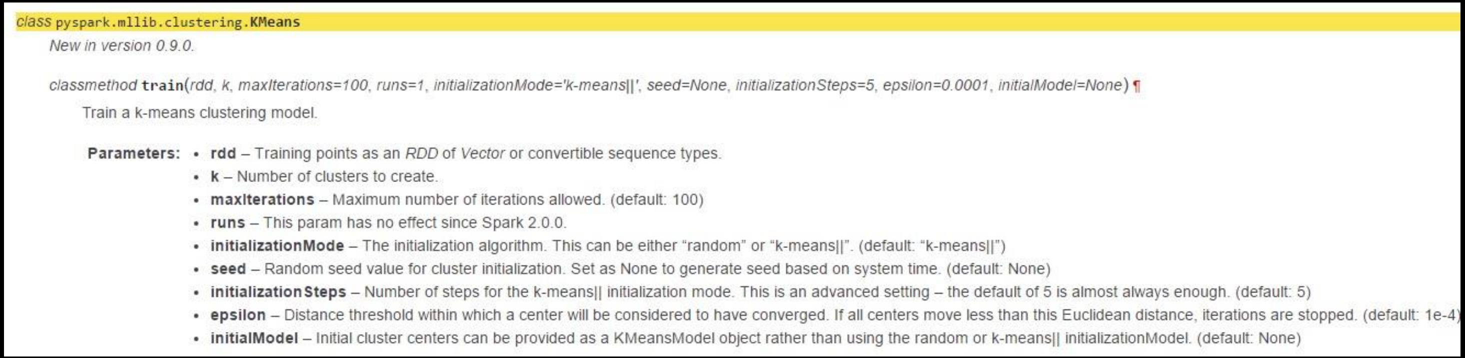
Finding Clusters

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 2.7.5 (default, Nov 20 2015 02:00:19)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
>>> rdd1 = sc.textFile("5000_points.txt")
>>>
>>> rdd2 = rdd1.map(lambda x:x.split())
>>> rdd3 = rdd2.map(lambda x: [int(x[0]),int(x[1])])
>>>
>>>
>>> from pyspark.mllib.clustering import KMeans
```

Read into RDD

Transform

Import Kmeans



class pyspark.mllib.clustering.**KMeans**

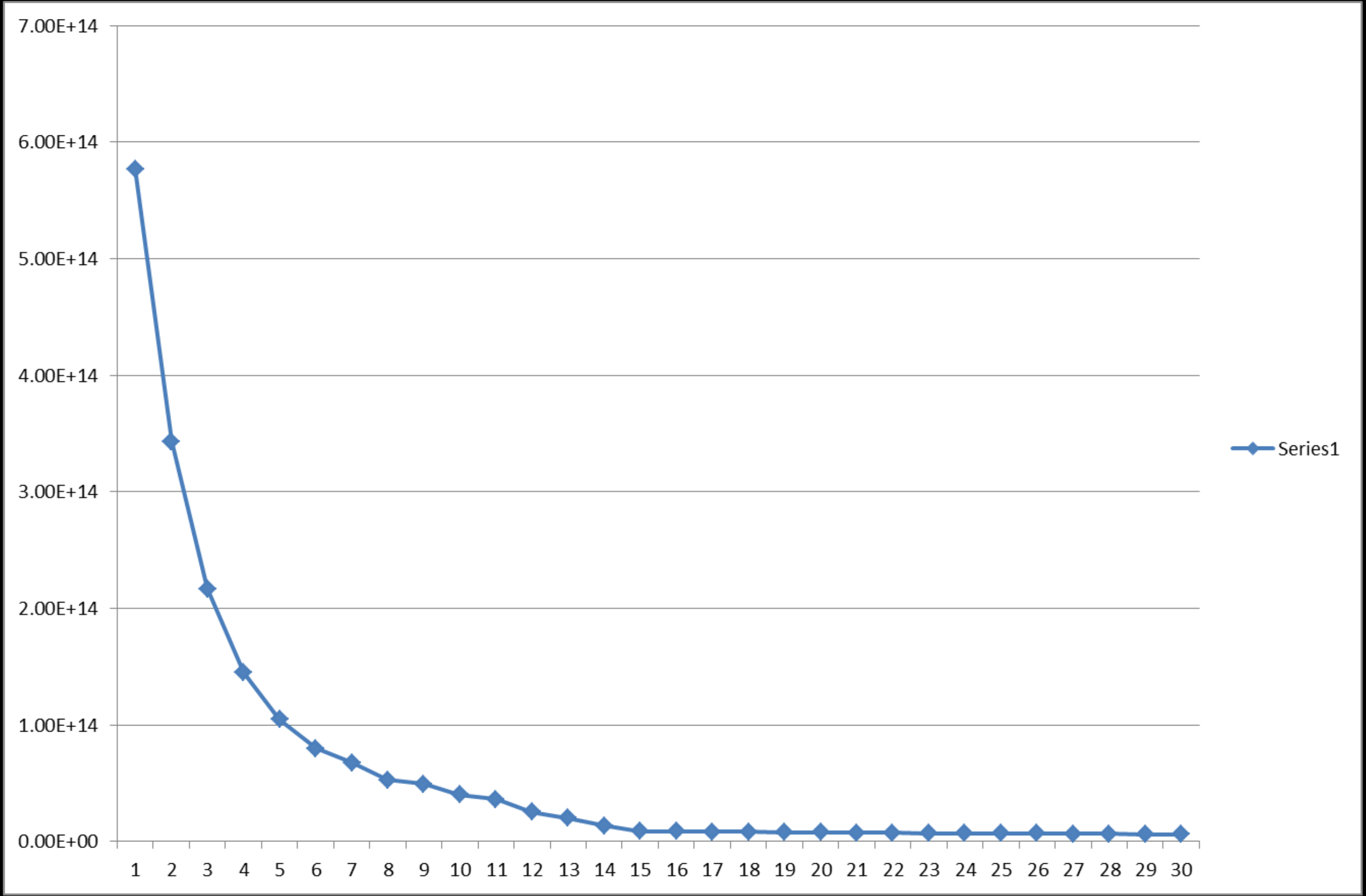
New in version 0.9.0.

classmethod **train**(*rdd, k, maxIterations=100, runs=1, initializationMode='k-means||', seed=None, initializationSteps=5, epsilon=0.0001, initialModel=None*) ¶

Train a k-means clustering model.

Parameters:
- **rdd** – Training points as an *RDD* of *Vector* or convertible sequence types.
- **k** – Number of clusters to create.
- **maxIterations** – Maximum number of iterations allowed. (default: 100)
- **runs** – This param has no effect since Spark 2.0.0.
- **initializationMode** – The initialization algorithm. This can be either “random” or “k-means||”. (default: “k-means||”)
- **seed** – Random seed value for cluster initialization. Set as None to generate seed based on system time. (default: None)
- **initializationSteps** – Number of steps for the k-means|| initialization mode. This is an advanced setting – the default of 5 is almost always enough. (default: 5)
- **epsilon** – Distance threshold within which a center will be considered to have converged. If all centers move less than this Euclidean distance, iterations are stopped. (default: 1e-4)
- **initialModel** – Initial cluster centers can be provided as a KMeansModel object rather than using the random or k-means|| initializationModel. (default: None)

Finding Clusters



Finding Clusters

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 2.7.5 (default, Nov 20 2015 02:00:19)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
>>> rdd1 = sc.textFile("5000_points.txt")
>>>
>>> rdd2 = rdd1.map(lambda x:x.split())
>>> rdd3 = rdd2.map(lambda x: [int(x[0]),int(x[1])])
>>>
>>> from pyspark.mllib.clustering import KMeans
>>>
>>> for clusters in range(1,30):
...     model = KMeans.train(rdd3, clusters)
...     print clusters, model.computeCost(rdd3)
...
```

Let’s see results for 1-30 cluster tries

```
1 5.76807041184e+14
2 3.43183673951e+14
3 2.23097486536e+14
4 1.64792608443e+14
5 1.19410028576e+14
6 7.97690150116e+13
7 7.16451594344e+13
8 4.81469246295e+13
9 4.23762700793e+13
10 3.65230706654e+13
11 3.16991867996e+13
12 2.94369408304e+13
13 2.04031903147e+13
14 1.37018893034e+13
15 8.91761561687e+12
16 1.31833652006e+13
17 1.39010717893e+13
18 8.22806178508e+12
19 8.22513516563e+12
20 7.79359299283e+12
21 7.79615059172e+12
22 7.70001662709e+12
23 7.24231610447e+12
24 7.21990743993e+12
25 7.09395133944e+12
26 6.92577789424e+12
27 6.53939015776e+12
28 6.57782690833e+12
29 6.37192522244e+12
```

Right Answer?

```
>>> for trials in range(10):
...     print
...     for clusters in range(12,18):
...         model = KMeans.train(rdd3,clusters)
...         print clusters, model.computeCost(rdd3)
```

```
12 2.45472346524e+13
13 2.00175423869e+13
14 1.90313863726e+13
15 1.52746006962e+13
16 8.67526114029e+12
17 8.49571894386e+12

12 2.62619056924e+13
13 2.90031673822e+13
14 1.52308079405e+13
15 8.91765957989e+12
16 8.70736515113e+12
17 8.49616440477e+12

12 2.5524719797e+13
13 2.14332949698e+13
14 2.11070395905e+13
15 1.47792736325e+13
16 1.85736955725e+13
17 8.42795740134e+12

12 2.31466242693e+13
13 2.10129797745e+13
14 1.45400177021e+13
15 1.52115329071e+13
16 1.41347332901e+13
17 1.31314086577e+13

12 2.47927778784e+13
13 2.43404436887e+13
14 2.1522702068e+13
15 8.91765000665e+12
16 1.4580927737e+13
17 8.57823507015e+12
```

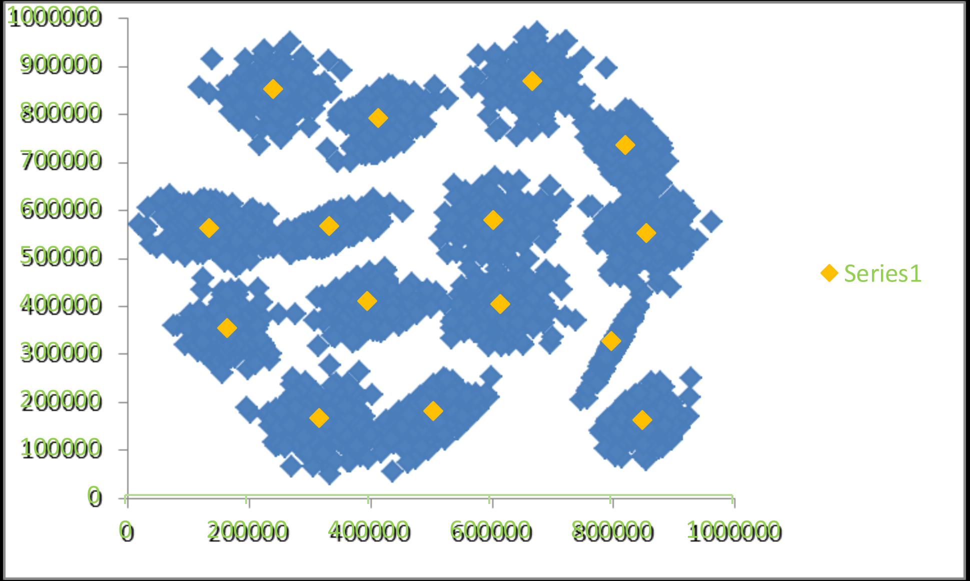
```
12 2.31466520037e+13
13 1.91856542103e+13
14 1.4933202312e+13
15 1.3506302755e+13
16 8.7757678836e+12
17 1.60075548613e+13

12 2.5187054064e+13
13 1.83498739266e+13
14 1.96076943156e+13
15 1.41725666214e+13
16 1.41986217172e+13
17 8.46755159547e+12

12 2.38234539188e+13
13 1.85101922046e+13
14 1.91732620477e+13
15 8.91769396968e+12
16 8.64876051004e+12
17 8.54677681587e+12

12 2.5187054064e+13
13 2.04031903147e+13
14 1.95213876047e+13
15 1.93000628589e+13
16 2.07670831868e+13
17 8.47797102908e+12

12 2.39830397362e+13
13 2.00248378195e+13
14 1.34867337672e+13
15 2.09299321238e+13
16 1.32266735736e+13
17 8.50857884943e+12
```

Find the Centers

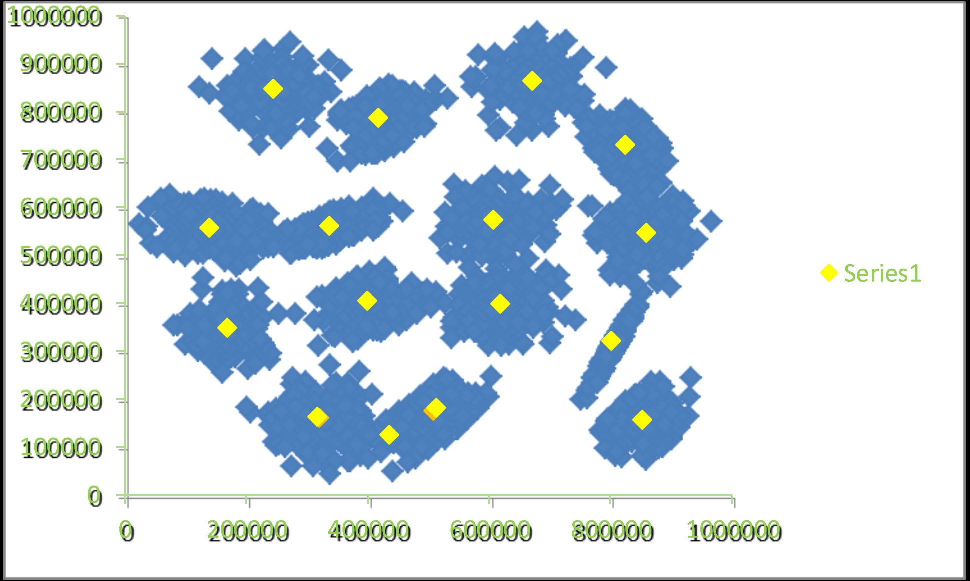
```
>>> for trials in range(10):                          #Try ten times to find best result
...    for clusters in range(12, 16):                 #Only look in interesting range
...        model = KMeans.train(rdd3, clusters)
...        cost = model.computeCost(rdd3)
...        centers = model.clusterCenters             #Let’s grab cluster centers
...        if cost<1e+13:                             #If result is good, print it out
...           print clusters, cost
...           for coords in centers:
...             print int(coords[0]), int(coords[1])
...           break
...
```

```
15 8.91761561687e+12
852058 157685
606574 574455
320602 161521
139395 558143
858947 546259
337264 562123
244654 847642
398870 404924
670929 862765
823421 731145
507818 175610
801616 321123
617926 399415
417799 787001
167856 347812
15 8.91765957989e+12
670929 862765
139395 558143
244654 847642
852058 157685
617601 399504
801616 321123
507818 175610
337264 562123
858947 546259
823421 731145
606574 574455
167856 347812
398555 404855
417799 787001
320602 161521
```

Fit?


1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
200000
400000
600000
800000
1000000
◆ Series1

16 Clusters


1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
200000
400000
600000
800000
1000000
◆ Series1

Run My Programs Or Yours

execfile()

```
[urbanic@r005 Clustering]$ pyspark
Python 2.7.11 (default, Feb 23 2016, 17:47:07)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Python version 2.7.11 (default, Feb 23 2016 17:47:07)
SparkSession available as 'spark'.
>>>
>>>
>>> execfile("clustering.py")
1 5.76807041184e+14
2 3.73234816206e+14
3 2.13508993715e+14
4 1.38250712993e+14
5 1.2632806251e+14
6 7.97690150116e+13
7 7.14156965883e+13
8 5.7815194802e+13
...
...
...
```

If you have another session window open on bridge’s login node, you can edit this file, save it while you remain in the editor, and then run it again in the python shell window with execfile().

You do *not* need this second session to be on a compute node. Do not start another interactive session.

Shakespeare, a Data Analytics Favorite

Applying data analytics to the works of Shakespeare has become all the rage. Whether determining the legitimacy of his authorship (it wasn’t Marlowe) or if Othello is actually a comedy (perhaps), it is amazing how much publishable research has sprung from the recent analysis of 400 year old text.

We’re going to do some exercises here using a text file containing all of his works.

Firing Up The Hands-On

Let’s make sure all is well with our hands-on environment first.

Grab a node

```
interact
```

We have hundreds of packages on Bridges. They each have many paths and variables that need to be set for their own proper environment, and they are often conflicting. We shield you from this with the wonderful modules command. You can load the two packages we will be using as

Spark

```
module load spark
```

Tensorflow (Not now! But later)

```
module load tensorflow/1.5_gpu
```

Copy over our exercise data, and move into an interesting starting point

```
cp -r ~training/BigData .
cd BigData/Shakespear
```

Start up Spark

```
pyspark
```

You may see a lot of noise (warning! Warning!) which you should ignore as long as you get a nice “Wecome to Spark” message at the end.

Some Simple Problems

We have an input file, Complete _Shakespeare.txt, that you can also find at http://www.gutenberg.org/ebooks/100.

If you are starting from scratch on the login node:

1) interact 2) cd BigData/Shakespeare 3) module load spark 4) pyspark

```
...
>>> rdd = sc.textFile("Complete_Shakespeare.txt")
```

Let's try a few simple exercises.

1) Count the number of lines

2) Count the number of words (hint: Python "split" is a workhorse)

3) Count unique words

4) Count the occurrence of each word

5) Show the top 5 most frequent words

These last two are a bit more challenging. One approach is to think "key/value". If you go that way, think about which data should be the key and don't be afraid to swap it about with value. This is a very common manipulation when dealing with key/value organized data.

Some Simple Answers

```
>>> lines_rdd = sc.textFile("Complete_Shakespeare.txt")
>>>
>>> lines_rdd.count()
124787
>>>
>>> words_rdd = lines_rdd.flatMap(lambda x: x.split())
>>> words_rdd.count()
904061
>>>
>>> words_rdd.distinct().count()
67779
>>>
```

Next, I know I'd like to end up with a pair RDD of sorted word/count pairs:

```
(23407, 'the'), (19540,'I'), (15682, 'to'), (15649, 'of') ...
```

Why not just *words_rdd.countByValue()*? We get back a massive Python unsorted dictionary of results:

```
 ... 1, u'precious-princely': 1, u'christenings?': 1, 'empire': 11, u'vaunts': 2, u"Lubber's": 1,
u'poet.': 2, u'Toad!': 1, u'leaden': 15, u"captains'": 1, u'leaf': 9, u'Barnes,': 1, u'lead': 101,
 u"'Hell": 1, u'wheat,': 3, u'lean': 28, u'Toad,': 1, u'trencher!': 2, u'1.F.2.': 1, u'leas': 2,
                                    u'leap': 17, ...
```

Where to go next? Sort this in Python or try to get back into an RDD? If this is truly *BIG* data, we want to remain as an RDD until our final results.

Some Harder Answers

```
>>> lines_rdd = sc.textFile("Complete_Shakespeare.txt")
>>>
>>> lines_rdd.count()
124787
>>>
>>> words_rdd = lines_rdd.flatMap(lambda x:
>>> words_rdd.count()
904061
>>>
>>> words_rdd.distinct().count()
67779
>>>
>>> key_value_rdd = words_rdd.map(lambda x: (x,1))
>>>
>>> key_value_rdd.take(5)
[(u'The', 1), (u'Project', 1), (u'Gutenberg', 1), (u'EBook', 1), (u'of', 1)]
>>>
>>> word_counts_rdd = key_value_rdd.reduceByKey(lambda x,y: x+y)
>>> word_counts_rdd.take(5)
[(u'fawn', 11), (u'considered-', 1), (u'Fame,', 3), (u'mustachio', 1), (u'protested,', 1)]
>>>
>>> flipped_rdd = word_counts_rdd.map(lambda x: (x[1],x[0]))
>>> flipped_rdd.take(5)
[(11, u'fawn'), (1, u'considered-'), (3, u'Fame,'), (1, u'mustachio'), (1, u'protested,')]
>>>
>>> results_rdd = flipped_rdd.sortByKey(False)
>>> results_rdd.take(5)
[(23407, u'the'), (19540, u'I'), (18358, u'and'), (15682, u'to'), (15649, u'of')]
>>>
```

Things data scientists do.

Turn these into k/v pairs

Reduce to get words counts

Flip keys and values so we can sort on wordcount instead of words.

```
results_rdd = lines_rdd.flatMap(lambda x: x.split()).map(lambda x: (x,1)).reduceByKey(lambda x,y: x+y).map(lambda x: (x[1],x[0])).sortByKey(False)
```

Some Homework Problems

To do research-level text analysis, we generally want to clean up our input. Here are some of the kinds of things you could do to get a more meaningful distinct word count.

1) Remove punctuation. Often punctuation is just noise, and it is here. Do a Map and/or Filter (some punctuation is attached to words, and some is not) to eliminate all punctuation from our Shakespeare data. Note that if you are familiar with regular expressions, Python has a ready method to use those.

2) Remove stop words. Stop words are common words that are also often uninteresting ("I", "the", "a"). You can remove many obvious stop words with a list of your own, and the *MLlib* that we are about to investigate has a convenient *StopWordsRemover()* method with default lists for various languages.

3) Stemming. Recognizing that various different words share the same root ("run", "running") is important, but not so easy to do simply. Once again, Spark brings powerful libraries into the mix to help. A popular one is the Natural Language Tool Kit. You should look at the docs, but you can give it a quick test quite easily:

```
import nltk
from nltk.stem.porter import  *
stemmer = PorterStemmer()
stems_rdd = words_rdd.map( lambda x: stemmer.stem(x) )
```