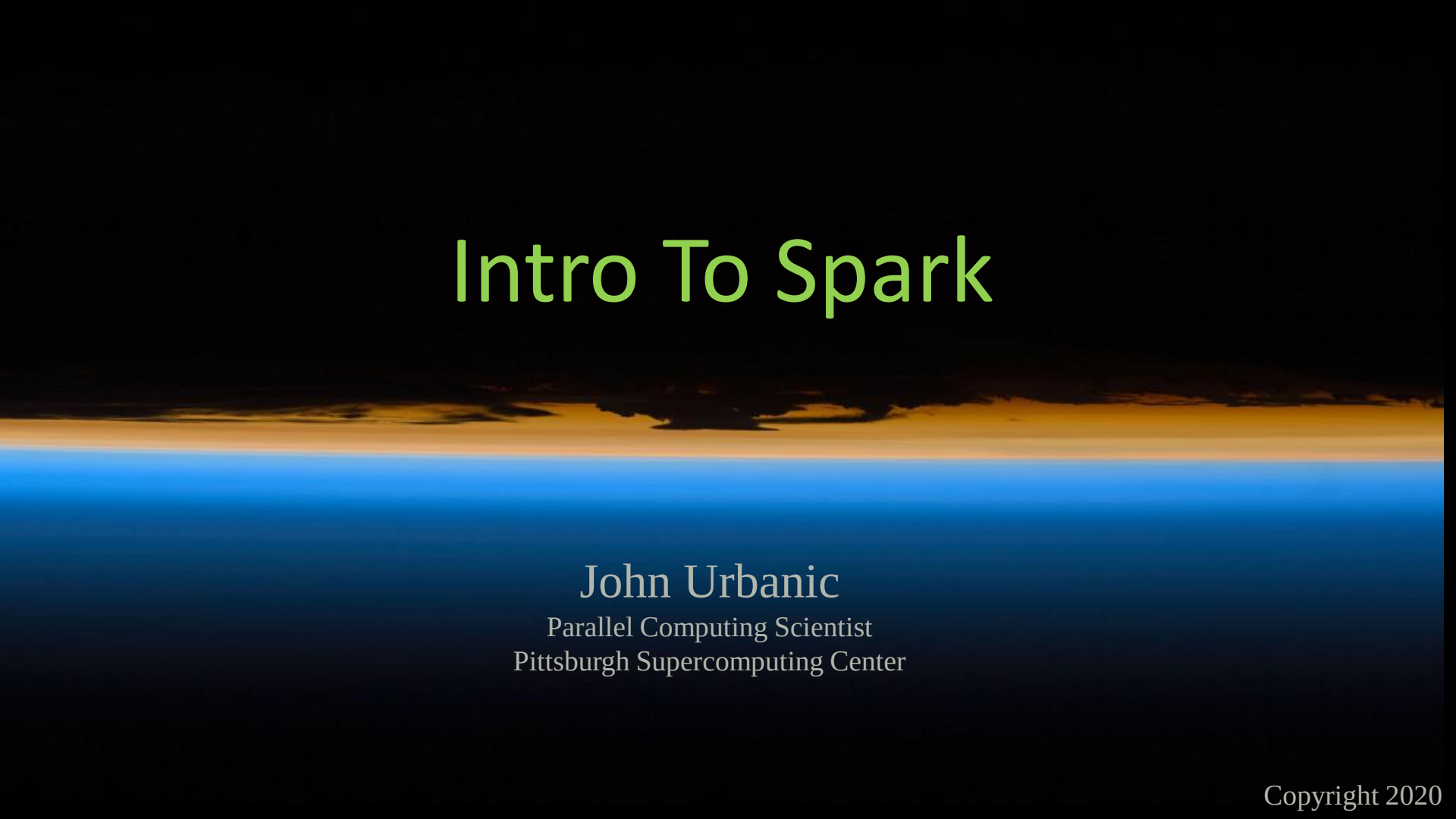


Intro To Spark

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

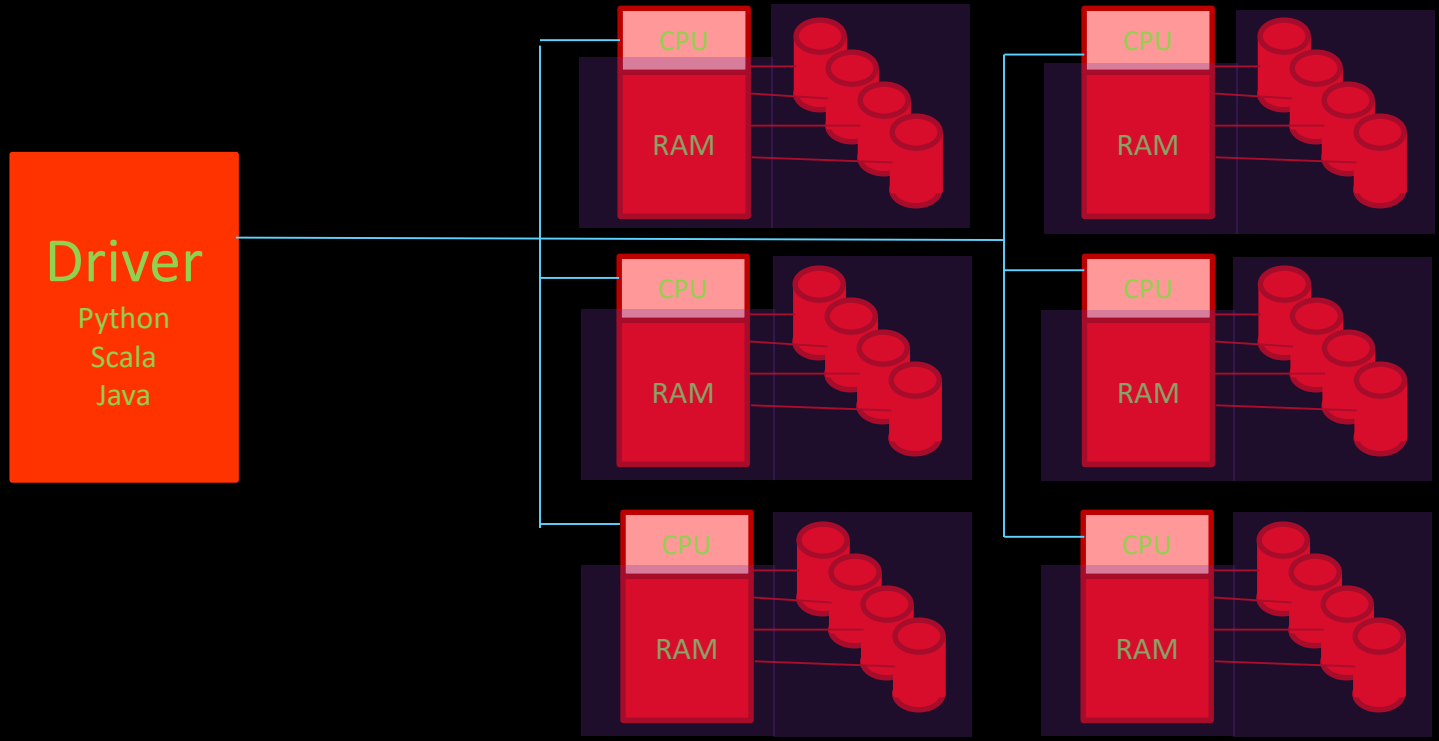
Spark Capabilities

(i.e. Hadoop shortcomings)

- Performance
  - First, use RAM
  - Also, be smarter
- Ease of Use
  - Python, Scala, Java first class citizens
- New Paradigms
  - SparkSQL
  - Streaming
  - MLib
  - GraphX
  - …more

But using Hadoop as the backing store is a common and sensible option.

Same Idea (improved)



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‘
```

Lambdas

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

Action	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 *all* of 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.

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.

Action	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

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

Joins Are Quite Useful

Any database designer can tell you how common joins are. Let's look at a simple example. We have (here we create it) an RDD of our top purchasing customers.

And an RDD with all of our customers' addresses.

To create a mailing list of special coupons for those favored customers we can use a join on the two datasets.

```
>>> best_customers_rdd = sc.parallelize([("Joe", "$103"), ("Alice", "$2000"), ("Bob", "$1200")])

>>> customer_addresses_rdd = sc.parallelize([("Joe", "23 State St."), ("Frank", "555 Timer Lane"), ("Sally", "44
Forest Rd."), ("Alice", "3 Elm Road"), ("Bob", "88 West Oak")])

>>> promotion_mail_rdd = best_customers_rdd.join(customer_addresses_rdd)

>>> promotion_mail_rdd.collect()
[('Bob', ('$1200', '88 West Oak')), ('Joe', ('$103', '23 State St.')), ('Alice', ('$2000', '3 Elm Road'))]
```

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.

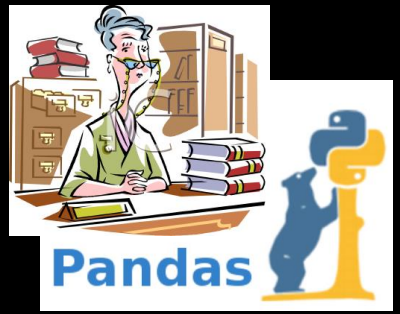
Who needs this Spark stuff?

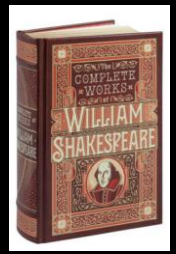
As we do our first Spark exercises, you might think of several ways to accomplish these tasks that you already know. For example, Python *Pandas* is a fine way to do our following problem, and it will probably work on your laptop reasonably well.

However we are learning how to leverage scalable techniques that work on very big data. Shortly, we will encounter problems that are considerable in size, and you will leave this workshop knowing how to harness very large resources.

Searching the *Complete Works of William Shakespeare* for patterns is a lot different from searching the entire Web (perhaps as the 800TB *Common Crawl* dataset).

So everywhere you see an RDD, realize that it is a actually a parallel databank that could scale to PBs.





Some Simple Problems

We have an input file, Complete _Shakespeare.txt, that you can also find at http://www.gutenberg.org/ebooks/100.
You might find http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD useful to have in a browser window.

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()*? It is an *action* that gives us 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 we reach our final results. So, no.

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) )
```

Optimizations

We said one of the advantages of Spark is that we can control things for better performance. There are a multitude of optimization, performance, tuning and programmatic features to enable better control. We quickly look at a few of the most important.

- Persistence
- Partitioning
- Parallel Programming Capabilities
- Performance and Debugging Tools

Persistence

- Lazy evaluation implies by default that all the RDD dependencies will be computed when we call an action on that RDD.
- If we intend to use that data multiple times (say we are filtering some log, then dumping the results, but we will analyze it further) we can tell Spark to persist the data.
- We can specify different levels of persistence: *MEMORY_ONLY, MEMORY_ONLY_SER, MEMORY_AND_DISK, MEMORY_AND_DISK_SER, DISK_ONLY*

```
>>> lines_rdd = sc.textFile("nasa_19950801.tsv")
>>> stanfordLines_rdd = lines.filter(lambda line: "stanford" in line)
>>> stanfordLines_rdd.persist(StorageLevel.MEMORY_AND_DISK)
>>> stanfordLines_rdd.count()
47

>>> stanfordLines_rdd.first(1)
[u.glim.stanford.edu\t-\t807258394\tGET\t/shuttle/…/orbiters-logo.gif\t200\t1932\t\t']
.
.
.
>>> stanfordLines.unpersist()
```

Do before first action.

Actions

Otherwise will just get evicted when out of memory (which is fine).

Partitions

- Spark distributes the data of your RDDs across its resources. It tries to do some obvious things.
- With key/value pairs we can help keep that data grouped efficiently.
- We can create custom partitioners that beat the default (which is probably a hash or maybe range).
- Use persist() if you have partitioned your data in some smart way. Otherwise it will keep getting re-partitioned.

Parallel Programming Features

Spark has several parallel programming features that make it easier and more efficient to do operations in parallel in a more explicit way.

Accumulators are variables that allow many copies of a variable to exist on the separate worker nodes.

It is also possible to have replicated data that we would like all the workers to have access to. Perhaps a lookup table of IP addresses to country codes so that each worker can transform or filter on such information. Maybe we want to exclude all non-US IP entries in our logs. You might think of ways you could do this just by passing variables, but they would likely be expensive in actual operation (usually requiring multiple sends). The solution in Spark is to send an (immutable, read only) broadcast variable

Accumulators

```
log = sc.textFile("logs")
blanks = sc.accumulator(0)

def tokenizeLog(line)
            global blanks      # write-only variable
            if (line =="")
                        blanks += 1
            return line.split(" ")

entries = log.flatMap(tokenizeLog)
entries.saveAsTextFile("parsedlogs.txt")
print "Blank entries: %d" blanks.value
```

Broadcast Variables

```
log = sc.textFile("log.txt")

IPtable = sc.broadcast(loadIPTable())

def countryFilter(IPentry, IPtable)
            return (IPentry.prefix() in IPTable)
USentries = log.filter(countryFilter)
```

Performance & Debugging

We will give unfortunately short shrift to performance and debugging, which are both important. Mostly, this is because they are very configuration and application dependent.

Here are a few things to at least be aware of:

- SparkConf() class. A lot of options can be tweaked here.
- Spark Web UI. A very friendly way to explore all of these issues.

IO Formats

Spark has an impressive, and growing, list of input/output formats it supports. Some important ones:

- Text
- CSV
- SQL type Query/Load
  - JSON (can infer schema)
  - Parquet
  - Hive
  - XML
  - Sequence (Hadoopy key/value)
  - Databases: JDBC, Cassandra, HBase, MongoDB, etc.
- Compression (gzip…)

And it can interface directly with a variety of filesystems: local, HDFS, Lustre, Amazon S3,...

Spark Streaming

Spark addresses the need for streaming processing of data with a API that divides the data into batches, which are then processed as RDDs.

There are features to enable:

- Fast recovery from f
- Load balancing
- Integration with sta
- Integration with oth

15% of the "global datasphere" (quantification of the amount of data created, captured, and replicated across the world) is currently real-time. That number is growing quickly both in absolute terms and as a percentage.

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…

Using MLlib

One of the reasons we use spark is for easy access to powerful data analysis tools. The MLlib library gives us a machine learning library that is easy to use and utilizes the scalability of the Spark system.

It has supported APIs for Python (with NumPy), R, Java and Scala.

We will use the Python version in a generic manner that looks very similar to any of the above implementations.

There are good example documents for the clustering routine we are using here:

http://spark.apache.org/docs/latest/mllib-clustering.html

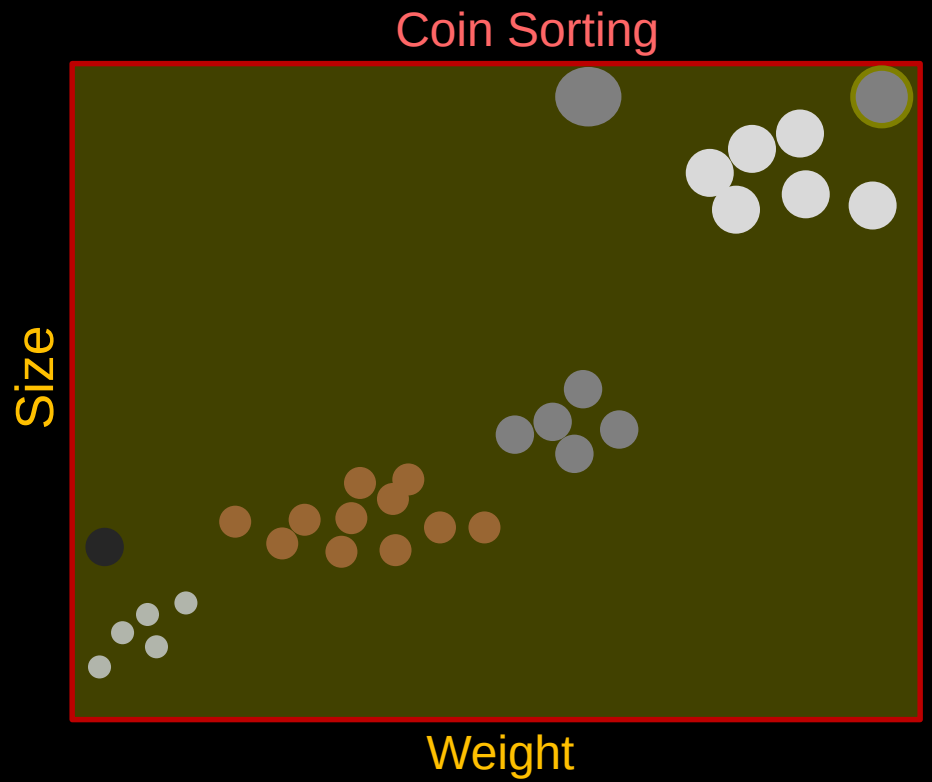
And an excellent API reference document here:

http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.clustering.KMeans

I suggest you use these pages for all your Spark work.

Clustering

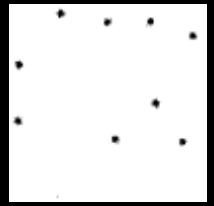
Clustering is a very common operation for finding grouping in data and has countless applications. This is a very simple example, but you will find yourself reaching for a clustering algorithm frequently in pursuing many diverse machine learning objectives, sometimes as one part of a pipeline.



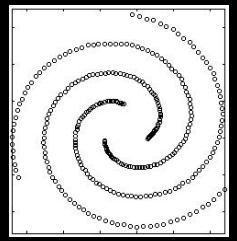
Clustering

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

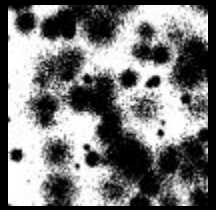
You might think this is trivial to implement in lower dimensional spaces.



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

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   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.textFil
>>>
>>> rdd2 = rdd1.map(l
>>> rdd3 = rdd2.map(l
>>>
```

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

to RDD

rm to words and integers

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.

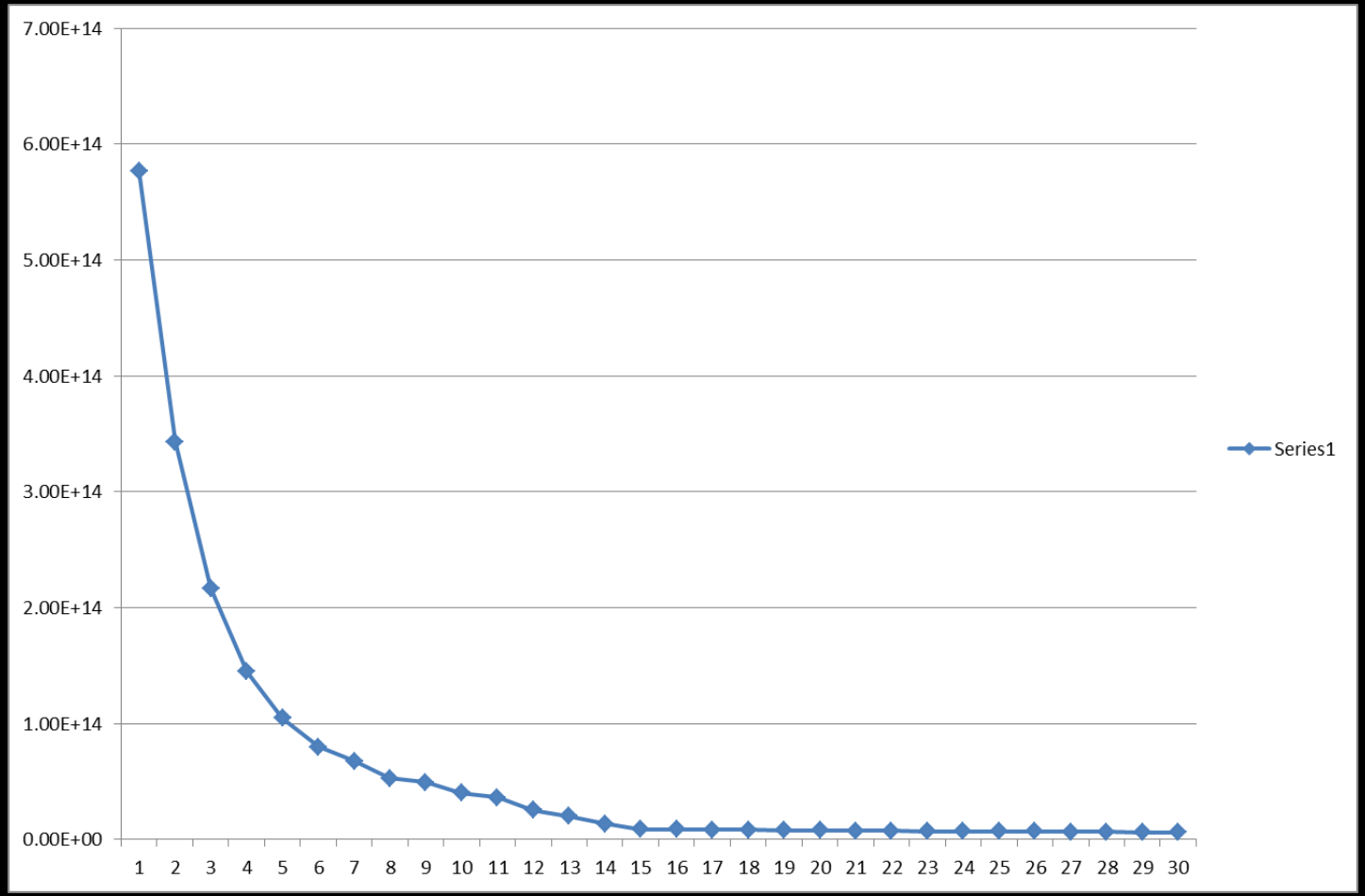
classmethod **train**(*rdd, k, maxIterations=100, runs=1, initializationMode='k-means||', seed=None, initializationSteps=5, epsilon=0.0001, initialModel=None*) ¶

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
```

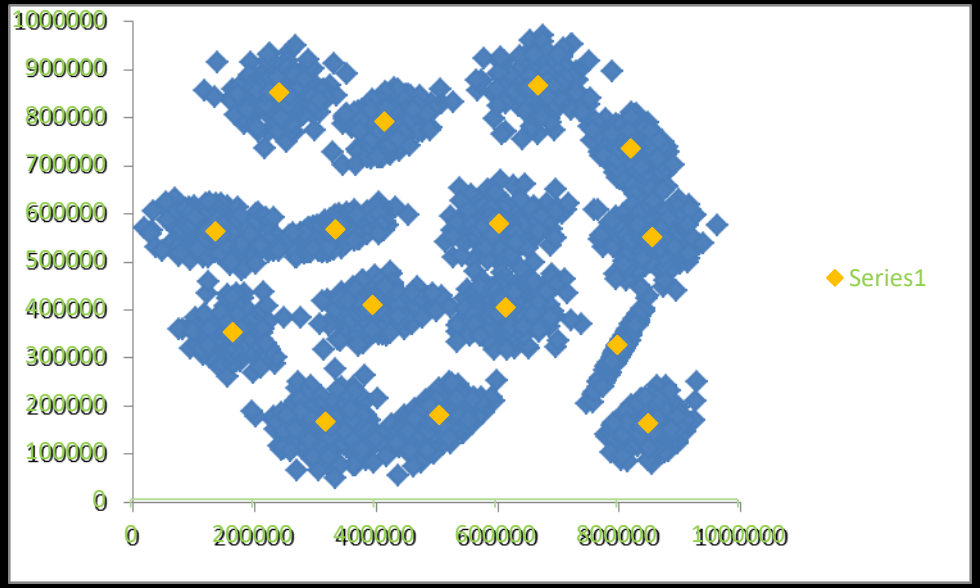
```
12 2.31466520037e+13
13 1.91856542103e+13
14 1.49332023312e+13
15 1.3506302755e+13
16 8.7757678836e+12
17 1.60075548613e+13

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

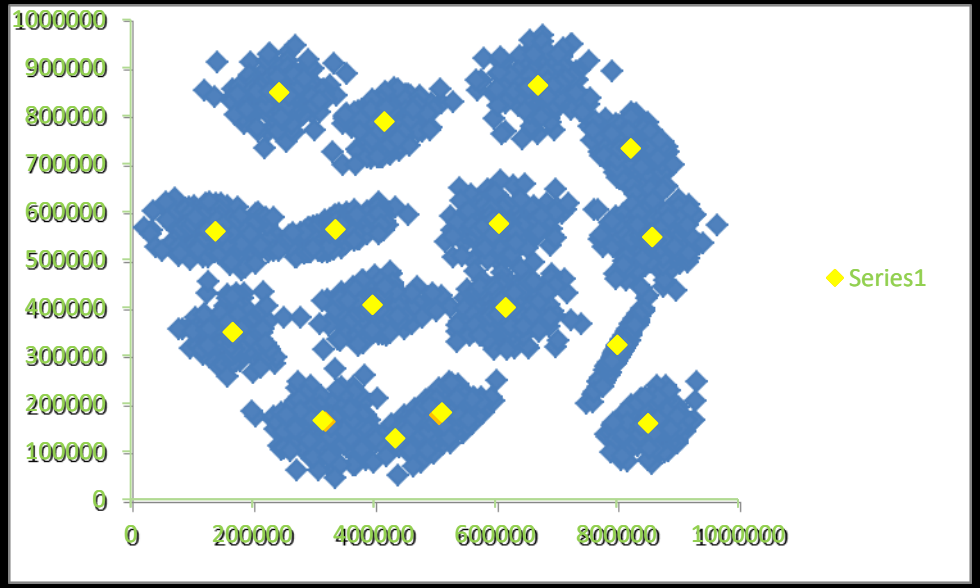
```
>>> for trials in range(10):                    #Try ten times to find best result
...     for clusters in range(12, 16):          #Only look in interesting range
...         model = KMeans.train(rdd3, clusters)
...         cost = model.computeCost(rdd3)
...         centers = model.clusterCenters      #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

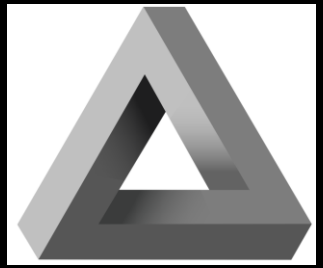
Dimensionality Reduction

We are going to find a recurring theme throughout machine learning:

- Our data naturally resides in higher dimensions
- Reducing the dimensionality makes the problem more tractable
- And simultaneously provides us with insight

This last two bullets highlight the principle that "learning" is often finding an effective compressed representation.

As we return to this theme, we will highlight these slides with our Dimensionality Reduction badge so that you can follow this thread and appreciate how fundamental it is.



Why all these dimensions?

The problems we are going to address, as well as the ones you are likely to encounter, are naturally highly dimensional. If you are new to this concept, lets look at an intuitive example to make it less abstract.

Category	Purchase Total ($)
Children's Clothing	$800
Pet Supplies	$0
Cameras (Dash, Security, Baby)	$450
Containers (Storage)	$350
Romance Book	$0
Remodeling Books	$80
Sporting Goods	$25
Children's Toys	$378
Power Tools	$0
Computers	$0
Garden	$0
Children's Books	$180

< 2900 Categories >

This is a 2900 dimensional vector.

Why all these dimensions?

If we apply our newfound clustering expertise, we might find we have 80 clusters (with an acceptable error).

People spending on “child’s toys “ and “children’s clothing” might cluster with “child’s books” and, less obvious, "cameras (Dashcams, baby monitors and security cams)", because they buy new cars and are safety conscious. We might label this cluster "Young Parents". We also might not feel obligated to label the clusters at all. We can now represent any customer by their distance from these 80 clusters.

80 dimensional vector.

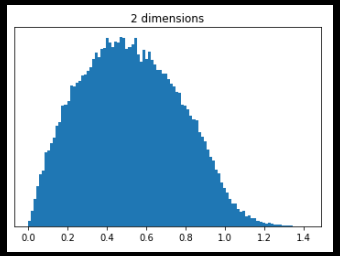
Customer Representation									
Cluster	Young Parents	College Athlete	Auto Enthusiast	Knitter	Steelers Fan	Shakespeare Reader	Sci-Fi Fan	Plumber	...
Distance	0.02	2.3	1.4	8.4	2.2	14.9	3.3	0.8	...

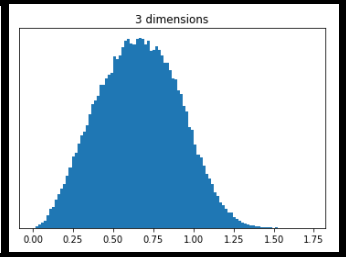
We have now accomplished two things:

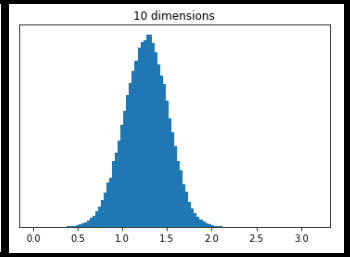
- we have compressed our data
- learned something about our customers (who to send a dashcam promo to).

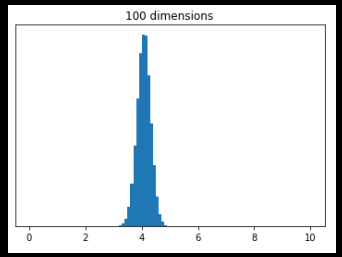
Curse of Dimensionality

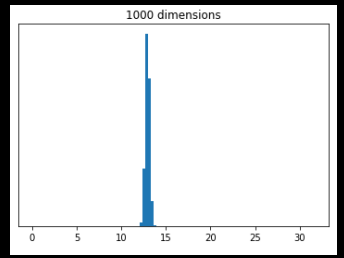
This is a good time to point out how our intuition can lead us astray as we increase the dimensionality of our problems - which we will certainly be doing - and to a great degree. There are several related aspects to this phenomenon, often referred to as the *Curse of Dimensionality*. One root cause of confusion is that our notion of Euclidian distance starts to fail in higher dimensions.

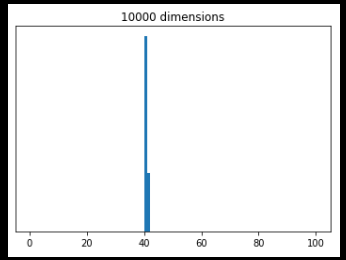












These plots show the distributions of pairwise distances between randomly distributed points within differently dimensioned unit hypercubes. Notice how all the points start to be about the same distance apart.

Once can imagine this makes life harder on a clustering algorithm!

There are other surprising effects: random vectors are almost all orthogonal; the unit sphere takes almost no volume in the unit square. These cause all kinds of problems when generalizing algorithms from our lowly 3D world.

Metrics

Even the definition of distance (the *metric*) can vary based upon application. If you are solving chess problems, you might find the Manhattan distance (or taxicab metric) to be most useful.

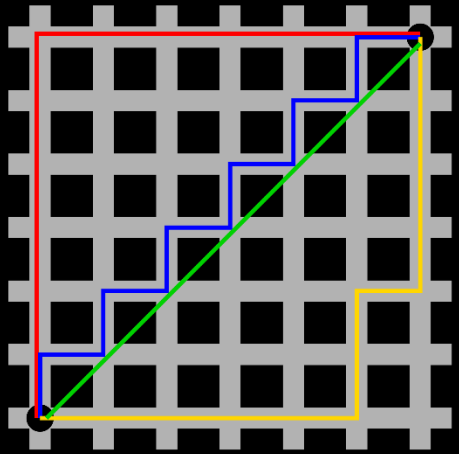


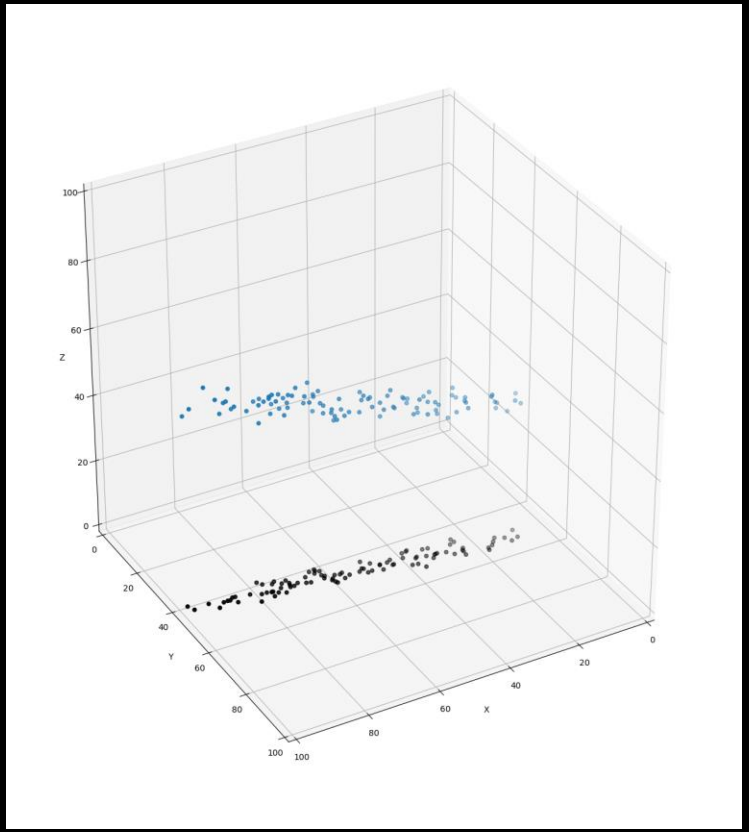
Image Source: Wikipedia

For comparing text strings, we might choose one of dozens of different metrics. For spell checking you might want one that is good for phonetic distance, or maybe edit distance. For natural language processing (NLP), you probably care more about tokens.

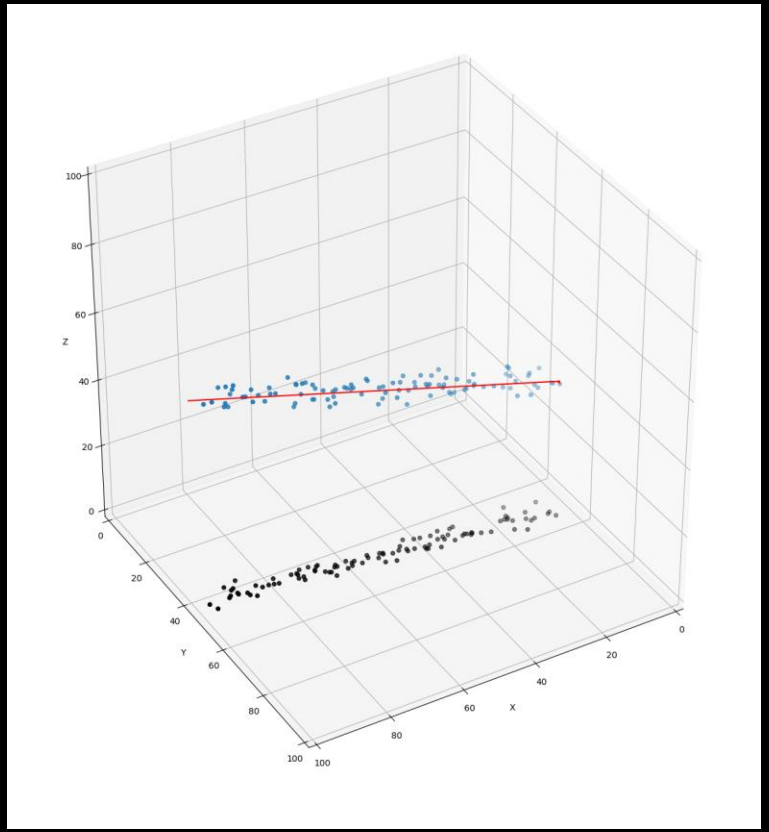
For genomics, you might care more about string sequences.

Some useful measures don't even qualify as metrics (usually because they fail the triangle inequality: $a + b \geq c$).

Alternative DR: Principal Component Analysis

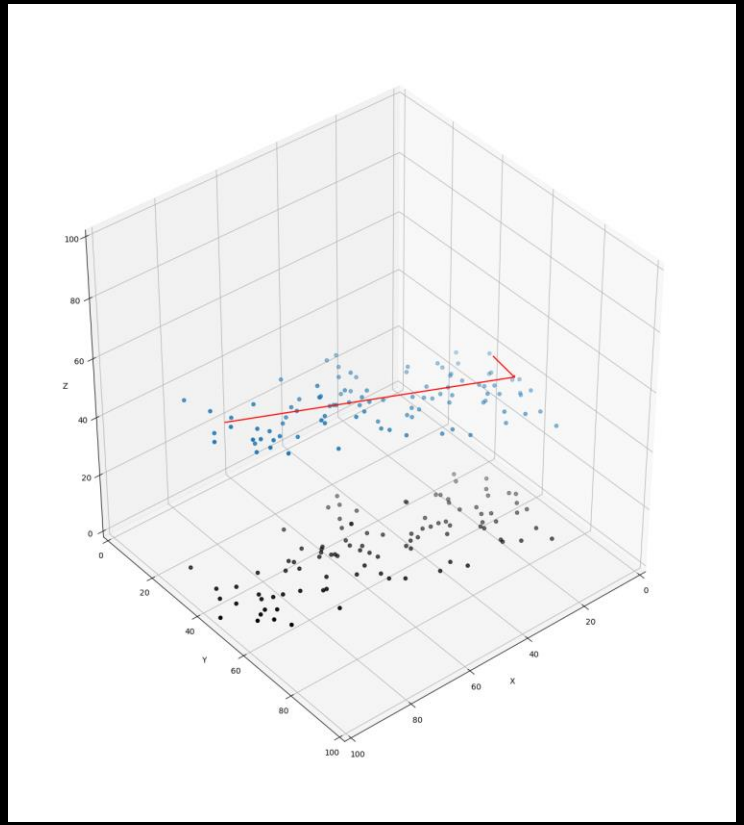


3D Data Set

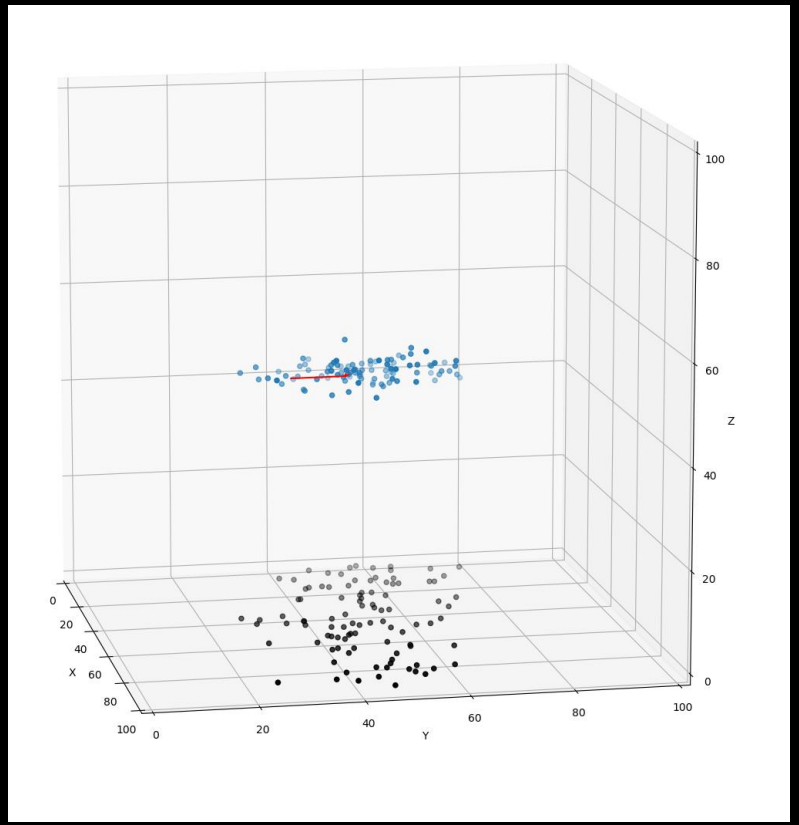


Maybe mostly 1D!

Alternative DR: Principal Component Analysis



Flatter 2D-ish Data Set



View down the 1st Princ. Comp.

Why So Many Alternatives?

Let's look at one more example today. Suppose we are tying to do a Zillow type of analysis and predict home values based upon available factors. We may have an entry (vector) for each home that captures this kind of data:

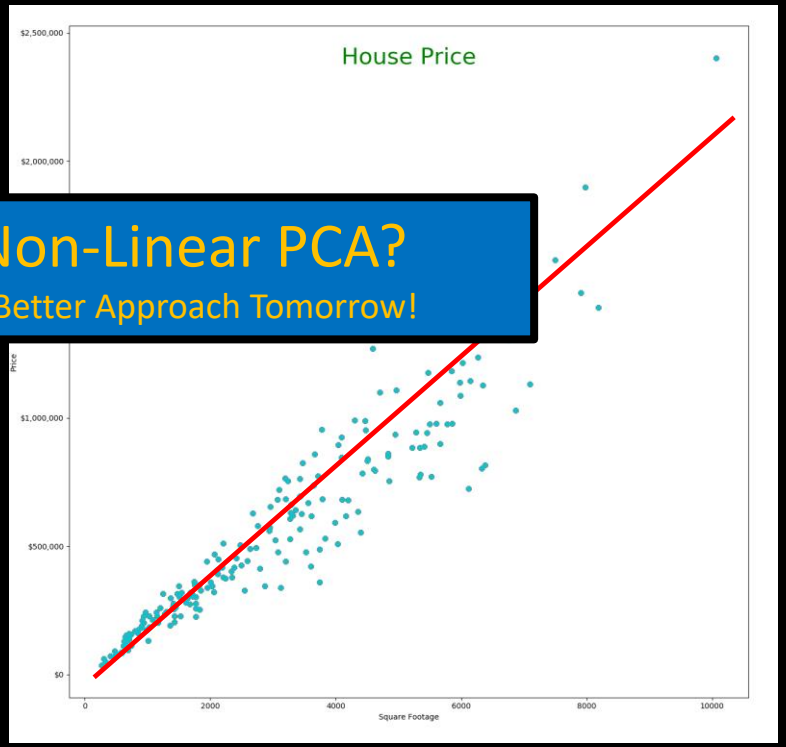
Home Data	
Latitude	4833438 north
Longitude	630084 east
Last Sale Price	$ 480,000
Last Sale Year	1998
Width	62
Depth	40
Floors	3
Bedrooms	3
Bathrooms	2
Garage	2
Yard Width	84
Yard Depth	60
...	...

There may be some opportunities to reduce the dimension of the vector here. Perhaps clustering on the geographical coordinates...

Principal Component Analysis Fail

1st Component Off

Data Not Very Linear



Non-Linear PCA?

A Better Approach Tomorrow!

H x W Is Not Linear

But (HxW) Fits Well

A Few Words About DataFrames

As mentioned earlier, an appreciation for having some defined structure to your data has come back into vogue. For one, because it simply makes sense and naturally emerges in many applications. Often even more important, it can greatly aid optimization, especially with the Java VM that Spark uses.

For both of these reasons, you will see that the newest set of APIs to Spark are DataFrame based. Sound leading-edge? This is simply SQL type columns. Very similar to Python pandas DataFrames (but based on RDDs, so not exactly).

We haven't prioritized them here because they aren't necessary, and some of the pieces aren't mature. But some of the latest features use them.

Creating DataFrames

It is very pretty intuitive to utilize DataFrames. Your elements just have labeled columns.

A *row RDD* is the basic way to go from RDD to DataFrame, and back, if necessary. A "row" is just a tuple.

```
>>> row_rdd = sc.parallelize([ ("Joe","Pine St.","PA",12543), ("Sally","Fir Dr.","WA",78456),
                               ("Jose","Elm Pl.","ND",45698) ])
>>>
>>> aDataFrameFromRDD = spark.createDataFrame( row_rdd, ["name", "street", "state", "zip"] )
>>> aDataFrameFromRDD.show()
+-----+--------+-----+-----+
| name|  street|state|  zip|
+-----+--------+-----+-----+
|  Joe|Pine St.|   PA|12543|
|Sally| Fir Dr.|   WA|78456|
| Jose| Elm Pl.|   ND|45698|
+-----+--------+-----+-----+
```

Creating DataFrames

You will come across DataFrames created without a schema. They get default column names.

```
>>> noSchemaDataFrame = spark.createDataFrame( row_rdd )
>>> noSchemaDataFrame.show()
+-----+--------+---+-----+
|   _1|      _2| _3|   _4|
+-----+--------+---+-----+
|  Joe|Pine St.| PA|12543|
|Sally| Fir Dr.| WA|78456|
| Jose| Elm Pl.| ND|45698|
+-----+--------+---+-----+
```

Datasets

Spark has added a variation (technically a superset) of *DataFrames* called *Datasets*. For compiled languages with strong typing (Java and Scala) these provide static typing and can detect some errors at compile time.

This is not relevant to Python or R.

And you can create them inline as well.

```
>>> directDataFrame = spark.createDataFrame([ ("Joe","Pine St.","PA",12543), ("Sally","Fir Dr.","WA",78456),
                                               ("Jose","Elm Pl.","ND",45698) ],
                                             ["name", "street", "state", "zip"] )
```

Speaking of pandas, or SciPy, or...

Some of you may have experience with the many Python libraries that accomplish some of these tasks. Immediately relevant to today, *pandas* allows us to sort and query data, and *SciPy* provides some nice clustering algorithms. So why not just use them?

The answer is that Spark does these things in the context of having potentially huge, parallel resources at hand. We don't notice it as Spark is also convenient, but behind every Spark call:

- every RDD could be many TB in size
- every transform could use many thousands of cores and TB of memory
- every algorithm could also use those thousands of cores

So don't think of Spark as just a data analytics library because our exercises are modest. You are learning how to cope with Big Data.

Run My Programs Or Yours

exec()

```
[urbanic@r001 ~]$ pyspark
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.0-preview2
      /_/

Using Python version 3.7.4 (default, Aug 13 2019 20:35:49)
SparkSession available as 'spark'
In [1]: exec(open("./clustering.py").read())
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 exec(...).

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

How does all this fit together?