

ADAPT Introductory Modules
for
Data Science and Machine Learning

John Urbanic

Parallel Computing Scientist
Distinguished Service Professor
Pittsburgh Supercomputing Center



*Some
Introductory
Comments*

Development Status

- Our intention way, way back in April, was that these would be 2 completely defined 4-class modules. Today would have been a detailed morning walk-through ("teach the teacher") of *Module 1*, and an afternoon walkthrough of *Module 2*.
- We came to realize that this community has other summer session priorities, and our development could not proceed in a vacuum, without your expertise and feedback.
- We have adopted a more flexible approach to today's launch. We are presenting the content as a whole, and welcoming your adoption as you see fit.
- This is not a generally scalable approach. We will have to standardize the module content. As soon as next semester.
- As such, your use and feedback this semester will be very influential in the shape that this program assumes.

The Plan

- Our original plan was to start with these "beginner" modules. Which we have done.
- This was actually the most effort-intensive content to develop as it was a gap in our existing materials. We generally start just beyond these subjects, as many of our prior audiences either know this material, or find it less exciting.
- But this is the foundation for the machine learning that immediately follows. We would be remiss to give this short shrift.
- This content is also the least challenging computationally. If this was our final destination, we might just do this with Jupyter notebooks on laptops.
- But after covering this ground, you and your students can seamlessly tread into the leading edge, using supercomputers and GPUs to attack interesting problems in big data and AI, adding stand-out experience to their resumes and applications.

Prereqs

- Our goal is that each module is stand-alone, and the only formal pre-requisite is a basic knowledge of Python.
- There is a logical ordering, and eventually using these in series could comprise an entire course.
- We are intentionally avoiding the Linux command line almost entirely. We do not want this to be an implicit pre-requisite, nor do we have time for detours in our modules.
- However, this is a very useful topic, and it could become a mini-module if there is demand.

Grading, ChatGPT and the future...

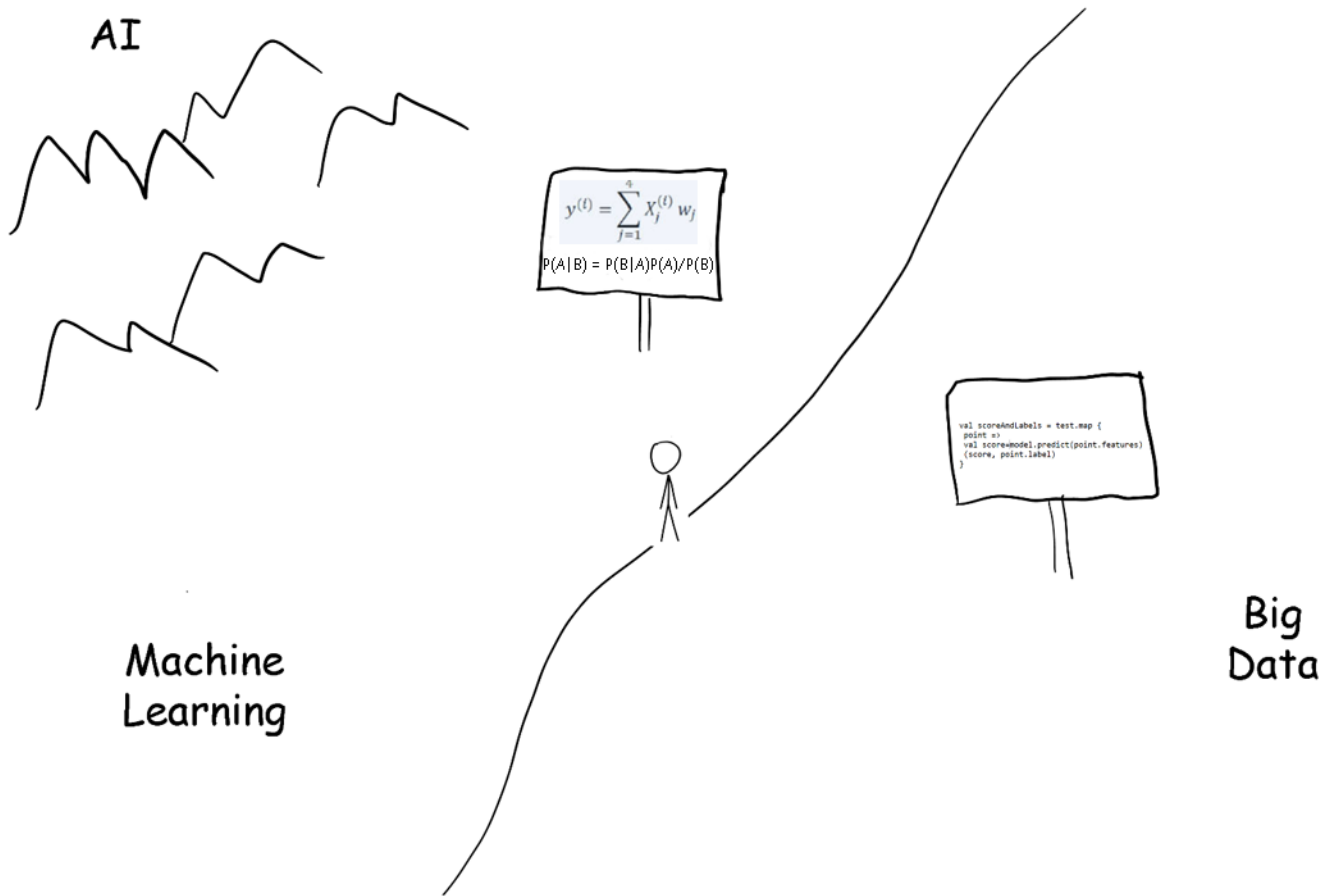
- We recognize that AI tools, like ChatGPT are both a boon and a hindrance for an educator.
- But, they are the current reality, and only likely to become more so.
- We have embraced this reality in two substantial ways:
 - We acknowledge that using these tools to write code is becoming an accepted practice. One immediate effect is that exhaustive coverage of all of the commands or routines in any given software toolkit is no longer an efficient way to teach a subject. We instead present a survey with the expectation that the student can find a particular command at the time of need.
 - We have attempted to develop our assignments in such a way that they are not trivial to solve with chatbots. Details as we get there. However, this is undoubtedly an ongoing challenge.

And now...

Data Science Modules 1 & 2



The
landscape
your
students
are
facing.



As the Data Scientist wanders across the ill-defined boundary between Data Science and Machine Learning, in search of the fabled land of Artificial Intelligence, they find that the language changes from programming to a creole of linear algebra and probability and statistics.

Data Science Today

- Basic Data

- Pandas



- "Serious" Data Science

- SQL



- Big Data

- Spark



Pandas

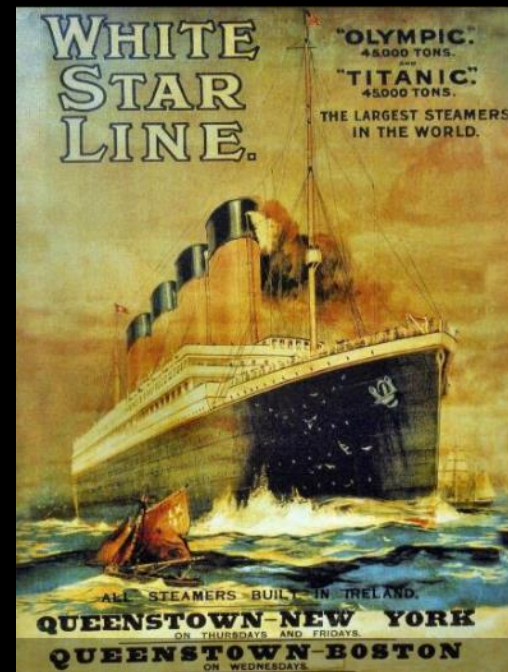


- Pandas has become the standard Python way to input, manipulate and write basic data.
- It also integrates well with other tools, like visualizing with Matplotlib.
- It has limitations, which is why SQL and big data techniques are essential for many tasks, but for quick-and-dirty, or limited applications it is very efficient.
- In many Python environments, it is there by default. If not, it is easy to add. In this course, if you start a python shell, it will be there.

Our First Dataset

We will begin our exploration of Pandas using a well known dataset drawn from the infamous Titanic disaster.

It has a variety of data on each of 891 passengers.



Amongst the typical demographic data is included their survival. It enables an interesting, if somewhat morbid, analysis to determine the foremost factors in survival. Women and children first? Or, save the rich?

Getting Started with Pandas

This "pd" is very standard

Smart, understands "csv"

```
import pandas as pd
```

```
titanic = pd.read_csv("titanic.csv")
```

```
titanic
```

	PassengerId	Survived	Pclass
0	1	0	3
4	5	0	3
5	6	0	3
6	7	0	1
7	8	0	3
...
883	884	0	2
884	885	0	3
886	887	0	2
889	890	1	1
890	891	0	3

[577 rows x 12 columns]

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

Fare	Cabin	Embarked
2500	NaN	S
0500	NaN	S
4583	NaN	Q
8625	E46	S
0750	NaN	S
...
5000	NaN	S
0500	NaN	S
0000	NaN	S
0000	C148	C
7500	NaN	Q

Survived	Survived (0 = No; 1 = Yes)
Pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
Name	Name
Sex	Sex
Age	Age
SibSp	Number of Siblings/Spouses Aboard
Parch	Number of Parents/Children Aboard
Ticket	Ticket Number
Fare	Fare (British pound)
Cabin	Cabin number
Embarked	Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

DataFrame Queries

```
titanic["Name"]
```

```
0          Braund, Mr. Owen Harris
1  Cumings, Mrs. John Bradley (Florence Briggs Th...
2          Heikkinen, Miss. Laina
3  Futrelle, Mrs. Jacques Heath (Lily May Peel)
4          Allen, Mr. William Henry
...
886         Montvila, Rev. Juozas
887         Graham, Miss. Margaret Edith
888  Johnston, Miss. Catherine Helen "Carrie"
889         Behr, Mr. Karl Howell
890         Dooley, Mr. Patrick
```

DataFrame Queries

```
titanic[["Age","Sex"]]
```

	<i>Age</i>	<i>Sex</i>
<i>0</i>	<i>22.0</i>	<i>male</i>
<i>1</i>	<i>38.0</i>	<i>female</i>
<i>2</i>	<i>26.0</i>	<i>female</i>
<i>3</i>	<i>35.0</i>	<i>female</i>
<i>4</i>	<i>35.0</i>	<i>male</i>
<i>...</i>	<i>...</i>	<i>...</i>
<i>886</i>	<i>27.0</i>	<i>male</i>
<i>887</i>	<i>19.0</i>	<i>female</i>
<i>888</i>	<i>NaN</i>	<i>female</i>
<i>889</i>	<i>26.0</i>	<i>male</i>
<i>890</i>	<i>32.0</i>	<i>male</i>

DataFrame Conditional Queries

```
titanic[titanic["Age"]>30]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
11	12	1	1	Bonnell, Miss. Elizabeth	female	58.0	0	0	113783	26.5500	C103	S
..
873	874	0	3	Vander Cruyssen, Mr. Victor	male	47.0	0	0	345765	9.0000	NaN	S
879	880	1	1	Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0	0	1	11767	83.1583	C50	C
881	882	0	3	Markun, Mr. Johann	male	33.0	0	0	349257	7.8958	NaN	S
885	886	0	3	Rice, Mrs. William (Margaret Norton)	female	39.0	0	5	382652	29.1250	NaN	Q
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

DataFrame Sorting

```
titanic.sort_values(by="Age")[["Name", "Age"]]
```

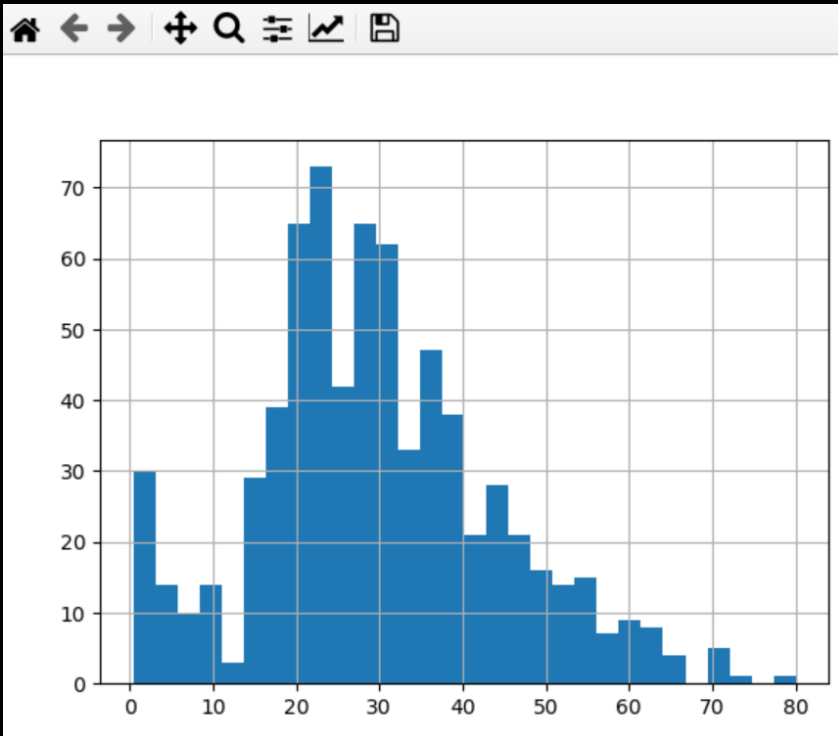
```
      Name  Age
803 Thomas, Master. Assad Alexander 0.42
755 Hamalainen, Master. Viljo 0.67
644 Bacchini, Miss. Eugenie 0.75
469 Bacchini, Miss. Helene Barbara 0.75
78 Caldwell, Master. Alden Gates 0.83
..      ..
859 Razi, Mr. Raihed NaN
863 Sage, Miss. Dorothy Edith "Dolly" NaN
868 van Melkebeke, Mr. Philemon NaN
878 Laleff, Mr. Kristo NaN
888 Johnston, Miss. Catherine Helen "Carrie" NaN
```

```
titanic.sort_values(by="Age")[["Name", "Age"]][0:10]
```

```
      Name  Age
803 Thomas, Master. Assad Alexander 0.42
755 Hamalainen, Master. Viljo 0.67
644 Bacchini, Miss. Eugenie 0.75
469 Bacchini, Miss. Helene Barbara 0.75
78 Caldwell, Master. Alden Gates 0.83
..      ..
859 Razi, Mr. Raihed NaN
863 Sage, Miss. Dorothy Edith "Dolly" NaN
868 van Melkebeke, Mr. Philemon NaN
878 Laleff, Mr. Kristo NaN
888 Johnston, Miss. Catherine Helen "Carrie" NaN
```


If you like pictures (matplotlib)

```
import matplotlib.pyplot as plt
titanic["Age"].hist(bins=30)
plt.show()
```



This assumes you have an X server running on your laptop.

Which we do.

*First Assignment:
Find a survival factor*



Assignment: Can we find a significant survival variable?

Can you find a significant factor in the data which could be used to predict survival rates?

I will suggest you focus on one variable at a time.

Note that there are many possible answers. Going from a simple hypothesis ("Maybe people from Cherbourg are unlucky?") to a more complex formula incorporating multiple variables - with the goal of a more accurate prediction - is the path of data analysis. This is our first step on that journey.



Connect to adaptpa.psc.edu, and start "python".

Find a meaningful factor and send me your full analysis (maybe just a few lines) and results.

Email to whatever@wherever.edu

This is due by **January 1st, 2045.**

The background of the slide features a horizontal line representing the horizon. Above the horizon, the sky is dark with some light, wispy clouds. Below the horizon, the ocean is a deep, dark blue, fading into black at the bottom of the frame. The text is centered in the upper half of the image.

***Titanic
Assignment
Review***

Getting Started with Titanic

```
import pandas as pd
titanic = pd.read_csv("titanic.csv")
```

```
males = titanic[titanic["Sex"]=="male"]
```

<i>PassengerId</i>	<i>Survived</i>	<i>Pclass</i>	<i>Name</i>	<i>Sex</i>	<i>Age</i>	<i>SibSp</i>	<i>Parch</i>	<i>Ticket</i>	<i>Fare</i>	<i>Cabin</i>	<i>Embarked</i>	
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0	0	0	17463	51.8625	E46	S
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S
...
883	884	0	2	Banfield, Mr. Frederick James	male	28.0	0	0	C.A./SOTON 34068	10.5000	NaN	S
884	885	0	3	Sutehall, Mr. Henry Jr	male	25.0	0	0	SOTON/OQ 392076	7.0500	NaN	S
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

[577 rows x 12 columns]

```
males.shape
```

(577, 12)

```
males[males["Survived"]==1].shape
```

(109, 12)

109/577

0.18890814558058924

19% Survival Rate for Males

How did the women fare?

```
titanic[titanic["Sex"]=="female"].shape
```

(314, 12)

```
titanic[ (titanic["Sex"]=="female") & (titanic["Survived"]==1) ].shape
```

(233, 12)

233/314

0.7420382165605095

74% Survival Rate for Females

Hypothesis confirmed: chivalry not dead.

But Jack Dawson is.

Women and children first!?

```
men = titanic[ (titanic["Sex"]=="male") & (titanic["Age"]>15) ]
```

```
men.shape
```

```
(413, 12)
```

```
men[ men["Survived"]==1 ].shape
```

```
(72, 12)
```

```
72/413
```

```
0.17433414043583534233/314
```

17% Survival Rate for Men

NaNs are everywhere!

```
women_and_children = titanic[ (titanic["Sex"]=="female") | (titanic["Age"]<16) ]  
women_and_children.shape
```

```
(354, 12)
```

```
#Seems like some people are missing...
```

```
titanic[titanic["Age"].isna()]
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
17	18	1	2	Williams, Mr. Charles Eugene	male	NaN	0	0	244373	13.0000	NaN	S
19	20	1	3	Masselmani, Mrs. Fatima	female	NaN	0	0	2649	7.2250	NaN	C
26	27	0	3	Emir, Mr. Farred Chehab	male	NaN	0	0	2631	7.2250	NaN	C
28	29	1	3	O'Dwyer, Miss. Ellen "Nellie"	female	NaN	0	0	330959	7.8792	NaN	Q
..
859	860	0	3	Razi, Mr. Raihed	male	NaN	0	0	2629	7.2292	NaN	C
863	864	0	3	Sage, Miss. Dorothy Edith "Dolly"	female	NaN	8	2	CA. 2343	69.5500	NaN	S
868	869	0	3	van Melkebeke, Mr. Philemon	male	NaN	0	0	345777	9.5000	NaN	S
878	879	0	3	Laleff, Mr. Kristo	male	NaN	0	0	349217	7.8958	NaN	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	w./C. 6607	23.4500	NaN	S

```
[177 rows x 12 columns]
```

```
413+354+177
```

```
944
```

This is bigger than the total passenger list (891).
But makes sense as we have double counted some
females with Age=NaN in our logic.

Women and children first!

```
women_and_children[ women_and_children["Survived"]==1 ].shape
```

```
(254, 12)
```

```
254/354
```

```
0.7175141242937854
```

72% Survival Rate for
Women & Children

How did Thurston Howell III make out?

Another obvious question we might ask is how did the wealthier, 1st class, passengers do versus the underclasses?

We could continue with our basic tools and separate out the various passenger classes, and perform some math to get at an answer.

However, we are now starting to ask questions that can utilize more sophisticated tools like:

- Joins (called Merges in Pandas)
- Grouping
- Pivot tables

Pandas has these capabilities. However, more complex data manipulation like this can often benefit from the more powerful capabilities of a Structured Query Language (SQL) database. Certainly at scale.

So we will preview the power of these operations with one last look at this problem, and then we will move on to SQL.

After you have learned SQL, you will easily be able to employ these operations in Pandas when you wish.



Grouping

Grouping typically performs 3 steps:

- Splits the data into groups base on some criteria:
- Applies a function to each group separately:
- Combine the results into a new table

Pclass
Survival Rate

That is one way to get directly at our answer. It becomes this simple:

```
titanic[['Pclass', 'Survived']].groupby('Pclass').mean()
```

```
Pclass  Survived  
1       0.629630  
2       0.472826  
3       0.242363
```

That is a pretty brutal curve.
I believe it speaks for itself.

SQL

The image features a serene landscape of a sunset or sunrise over a body of water. The sky is a deep, dark blue, transitioning into a bright orange and yellow glow at the horizon. The water below is a calm, dark blue, reflecting the light from the sky. The word "SQL" is written in a light green, italicized font, centered in the upper half of the image.

Big Data

The background of the slide features a horizontal horizon line. Above the horizon, the sky is dark with some wispy clouds. Below the horizon, there is a bright, glowing blue band that transitions into a darker blue at the bottom of the frame.

Big data is a broad term for data sets so large or complex that traditional data processing applications are inadequate.

—*Wikipedia*

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....



Less sophisticated is sometimes better...



“Chronologically” or “geologically” organized.
Familiar to some of you at tax time.

Get all articles from 2007.

Get all papers on “fault tolerance”
– grumble and cough

Indexing will determine your individual performance.
Teamwork can scale that up.



The culmination of centuries...



Find books on Modern Physics (DD# 539)

Find books by Wheeler

where he isn't the first author – grumble...



Your only hope...



Then data started to grow.

1956 IBM Model 350



5 MB of data!

But still pricey. \$

Better think about what
you want to save.

And finally 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

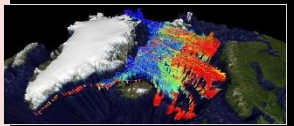
= Facebook
= Deep Learning
= IoT
= Science... **10 TB** *



Pan-STARRS



Genome sequencers
(Wikipedia Commons)



Horniman museum:
<http://www.horniman.ac.uk>
[get_involved/blog/bioblitz-insects-reviewed](http://www.horniman.ac.uk/get_involved/blog/bioblitz-insects-reviewed)

Wikipedia Commons

mentions a more accurate 208TB, and in

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

**Actually, a silly estimate. The original reference in*
2013 the digital collection alone was 3PB.



A better sense of biggish

Size

- 1000 Genomes Project
 - AWS hosted
 - 260TB
- Common Crawl
 - Hosted on Bridges
 - 300-800TB+

Throughput

- Square Kilometer Array
 - Building now
 - Exabyte of raw data/day – compressed to 10PB
- Internet of Things (IoT) / motes
 - Endless streaming

Records

- GDEL (Global Database of Events, Language, and Tone) (also soon to be hosted on Bridges)
 - Only about 2.5TB per year, but...
 - 250M rows and 59 fields (BigTable)
 - *“during periods with relatively little content, maximal translation accuracy can be achieved, with accuracy linearly degraded as needed to cope with increases in volume in order to ensure that translation always finishes within the 15 minute window.... and prioritizes the highest quality material, accepting that lower-quality material may have a lower-quality translation to stay within the available time window.”*

3 V's of Big Data

- Volume
- Velocity
- Variety

Good Ol' SQL couldn't keep up.



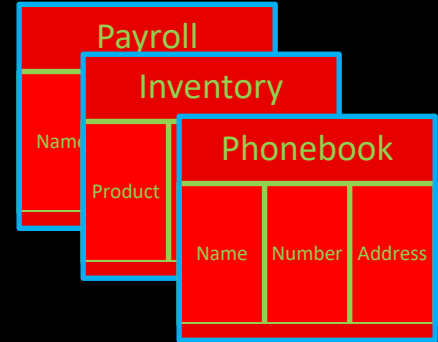
Oracle



```
SELECT NAME, NUMBER, FROM PHONEBOOK
```

Why it *wasn't* fashionable:

- Schemas set in stone:
 - Need to define before we can add data
 - Not a fit for *agile development*
"What do you mean we didn't plan to keep logs of everyone's heartbeat?"
- Queries often require accessing multiple indexes and joining and sorting multiple tables
- Sharding isn't trivial
- Caching is tough
 - ACID (Atomicity, Consistency, Isolation, Durability) in a *transaction* is costly.



So we gave up: Key-Value

Redis, Memcached, Amazon DynamoDB, Riak, Ehcache

```
GET foo
```

- Certainly agile (no schema)
- Certainly scalable (linear in most ways: hardware, storage, cost)
- Good hash might deliver fast lookup
- Sharding, backup, etc. could be simple
- Often used for “session” information: online games, shopping carts

```
GET cart:joe:15~4~7~0723
```

foo	bar
2	fast
6	0
9	0
0	9
text	pic
1055	stuff
bar	foo

How does a pile of unorganized data solve our problems?

Sure, giving up ACID buys us a lot performance, but doesn't our crude organization cost us something? Yes, but remember these guys?



This is what they look like today.



Document



GET foo

- Value must be an object the DB can understand
- Common are: XML, JSON, Binary JSON and nested thereof
- This allows server side operations on the data

GET plant=daisy

- Can be quite complex: Linq query, JavaScript function
- Different DB's have different update/staleness paradigms

foo	
2	<pre><CATALOG> <PLANT> <COMMON>Bloodroot</COMMON> <BOTANICAL>Sanguinaria canadensis</BOTANICAL> <ZONE>4</ZONE> <LIGHT>Mostly Shady</LIGHT> <PRICE>\$2.44</PRICE> <AVAILABILITY>031599</AVAILABILITY> </PLANT> <PLANT> <COMMON>Columbine</COMMON> <BOTANICAL>Aquilegia canadensis</BOTANICAL> <ZONE>3</ZONE> <LIGHT>Mostly Shady</LIGHT> <PRICE>\$9.37</PRICE> <AVAILABILITY>030699</AVAILABILITY> </PLANT> </pre>
6	JSON
9	XML
0	Binary JSON
bar	JSON XML
12	XML XML

Wide Column Stores



```
SELECT Name, Occupation FROM People WHERE key IN (199, 200, 207);
```

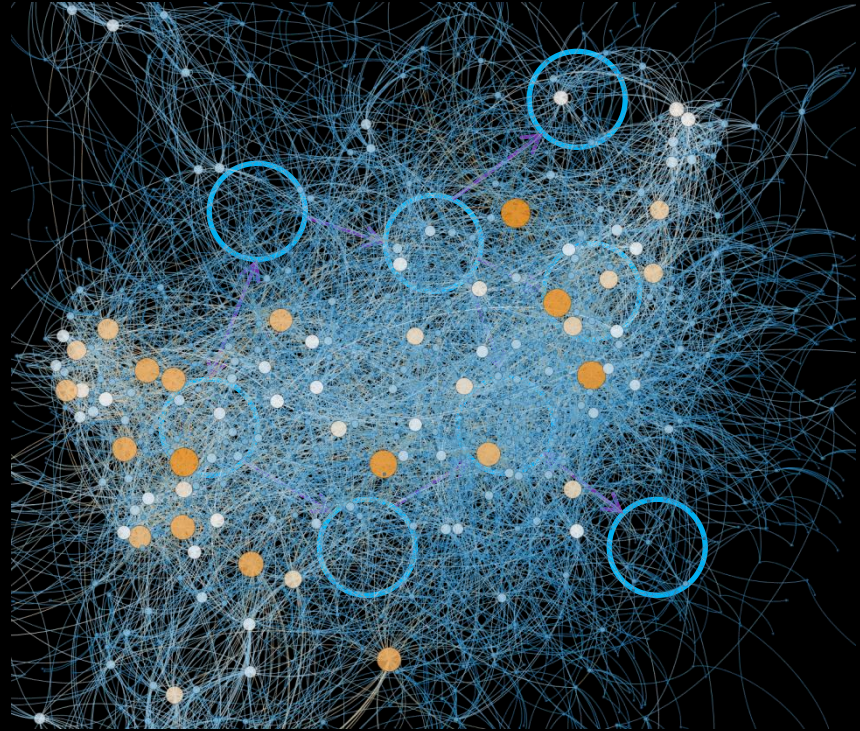
- No predefined schema
- Can think of this as a 2-D key-value store: the value may be a key-value store itself
- Different databases aggregate data differently on disk with different optimizations

Key			
Joe	Email: joe@gmail	Web: www.joe.com	
Fred	Phone: 412-555-3412	Email: fred@yahoo.com	Address: 200 S. Main Street
Julia	Email: julia@apple.com		
Mac	Phone: 214-555-5847		

Graph

Neo4j Titan, GEMS

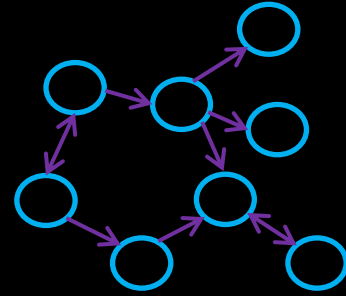
- Great for semantic web
- Great for graphs 😊
- Can be hard to visualize
- Serialization can be difficult
- Queries more complicated



From [PDX Graph Meetup](#)

Queries

SPARQL, Cypher



SPARQL (W3C Standard)

- Uses Resource Description Framework format
 - triple store
- RDF Limitations
 - No named graphs
 - No quantifiers or general statements
 - “Every page was created by some author”
 - “Cats meow”
- Requires a schema or *ontology* (RDFS) to define rules
 - "The object of 'homepage' must be a Document."
 - "Link from an actor to a movie must connect an object of type Person to an object of type Movie."

```
SELECT ?name ?email
WHERE {
    ?person a foaf:Person.
    ?person foaf:name ?name.
    ?person foaf:mbox ?email. }
```

Cypher (Neo4J only)

- No longer proprietary
- Stores whole graph, not just triples
- Allows for named graphs
- ...and general Property Graphs (edges and nodes may have values)


```
SMATCH (Jack:Person
        { name:'Jack Nicolson' })-[[:ACTED_IN]]-(movie:Movie)
RETURN movie
```

Graph Databases

- These are not curiosities, but are behind some of the most high-profile pieces of Web infrastructure.
- They are definitely *big* data.

Microsoft Bing Knowledge Graph	Search and conversations.	~2 billion primary entries ~55 billion facts
Facebook		~50 million primary entries ~500 million assertions
Google Knowledge Graph	Search and conversations.	~1 billion entries ~55 billion facts
LinkedIn graph		590 million members 30 million companies

Hadoop & Spark



What kind
of databases
are they?

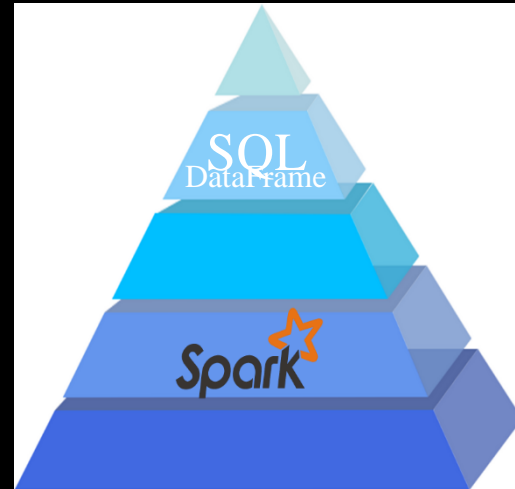
Frameworks for Data

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



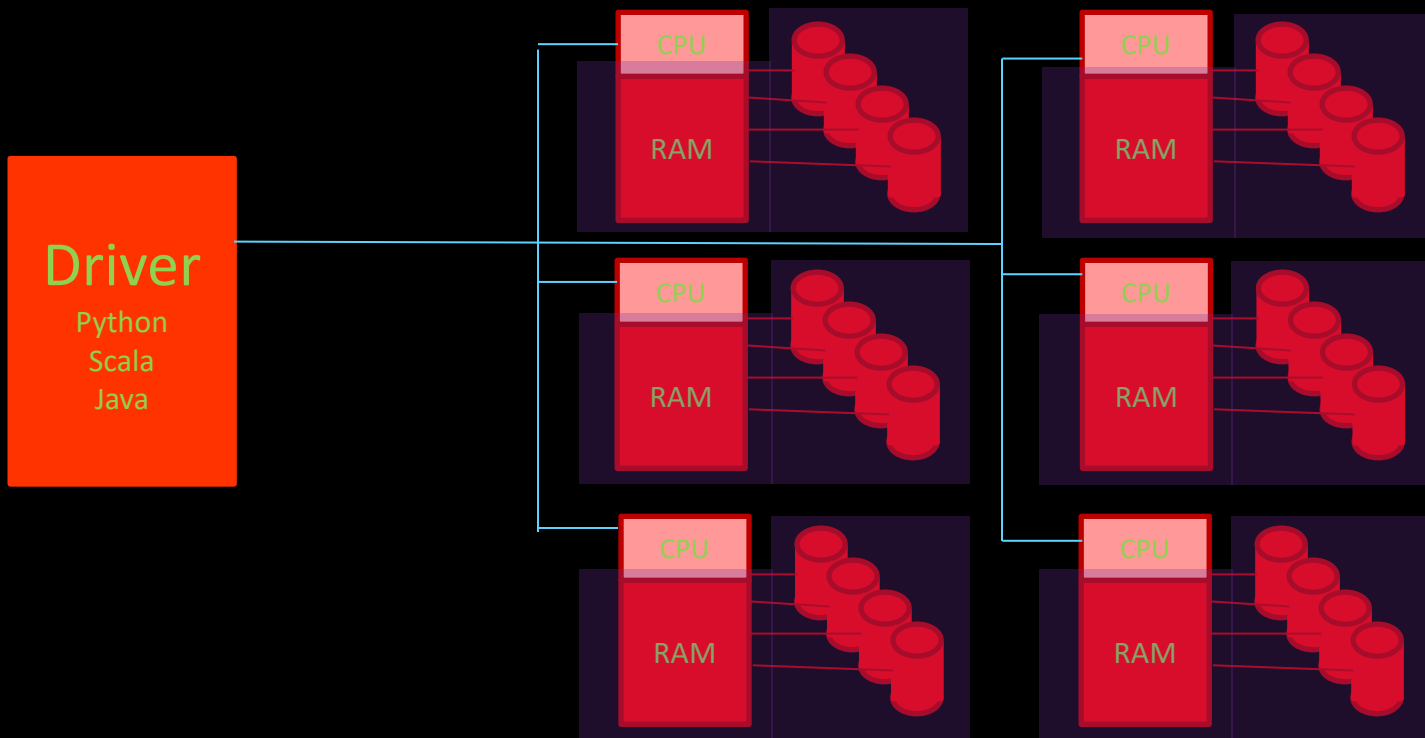
Spark Capabilities

(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_serverlog_20190404.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_serverlog_20190404.tsv")
>>> HubbleLines_rdd = lines_rdd.filter(lambda line: "Hubble" in line)
>>> HubbleLines_rdd.count()
47
>>> HubbleLines_rdd.first()
'www.nasa.gov\shuttle/missions/61-c/Hubble.gif'
```

 **Read into RDD**

 **Transform**

 **Actions**

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 <i>func</i> .	Same Size
filter(func)	Return a new RDD by selecting the elements for which <i>func</i> returns true.	Fewer Elements
flatMap(func)	<i>func</i> can return multiple items, and generate a sequence, allowing us to “flatten” nested entries (JSON) into a list.	More Elements
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).	

Common Actions

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

Transformations vs. Actions

Transformations go from one RDD to another¹.

Actions bring some data back from the RDD.

Transformations are where the Spark machinery can do its magic with lazy evaluation and clever algorithms to minimize communication and parallelize the processing. You want to keep your data in the RDDs as much as possible.

Actions are mostly used either at the end of the analysis when the data has been distilled down (*collect*), or along the way to "peek" at the process (*count*, *take*).

¹ Yes, some of them also create an RDD (parallelize), but you get the idea.

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
<code>reduceByKey(func)</code>	Reduce values using <i>func</i> , but on a key by key basis. That is, combine values with the same key.
<code>groupByKey()</code>	Combine values with same key. Each key ends up with a list.
<code>sortByKey()</code>	Return an RDD sorted by key.
<code>mapValues(func)</code>	Use <i>func</i> to change values, but not key.
<code>keys()</code>	Return an RDD of only keys.
<code>values()</code>	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
<code>countByKey()</code>	Count the number of elements for each key.
<code>lookup(key)</code>	Return all the values for this key.

Two Pair RDD Transformations

Transformation	Result
<code>subtractByKey(otherRDD)</code>	Remove elements with a key present in other RDD.
<code>join(otherRDD)</code>	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.
<code>leftOuterJoin(otherRDD)</code>	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 .
<code>rightOuterJoin(otherRDD)</code>	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 .
<code>cogroup(otherRDD)</code>	Group data from both RDDs by key.

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

And an RDD with all of our customers' addresses

To create a mailing list of special coupons for

join on the two datasets.

If you are coming from a Pandas DataFrame background, *joins* are congruent with the *Merge* functions. If you've used them, you may have noticed that they can take some time with even small datasets. They do not scale well.

```
>>> 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), or which word makes Macbeth so creepy ("the", yes) 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.

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 it useful to have <http://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html#pyspark.RDD> 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, 'precious-princely': 1, 'christenings?': 1, 'empire': 11, 'vaunts': 2, 'Lubber's': 1,
'poet.': 2, 'Toad!': 1, 'leaden': 15, 'captains': 1, 'leaf': 9, 'Barnes,': 1, 'lead': 101, 'Hell':
1, 'wheat,': 3, 'lean': 28, 'Toad,': 1, 'trencher!': 2, '1.F.2.': 1, 'leas': 2, '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

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.

```
>>> 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
>>>
>>> key_value_rdd = words_rdd.map(lambda x: (x,1))
>>>
>>> key_value_rdd.take(5)
[('The', 1), ('Project', 1), ('Gutenberg', 1), ('EBook', 1), ('of', 1)]
>>>
>>> word_counts_rdd = key_value_rdd.reduceByKey(lambda x,y: x+y)
>>> word_counts_rdd.take(5)
[('fawn', 11), ('considered-', 1), ('Fame,', 3), ('mustachio', 1), ('protested,', 1)]
>>>
>>> flipped_rdd = word_counts_rdd.map(lambda x: (x[1],x[0]))
>>> flipped_rdd.take(5)
[(11, 'fawn'), (1, 'considered-'), (3, 'Fame, '), (1, 'mustachio'), (1, 'protested,')]
>>>
>>> results_rdd = flipped_rdd.sortByKey(False)
>>> results_rdd.take(5)
[(23407, 'the'), (19540, 'I'), (18358, 'and'), (15682, 'to'), (15649, 'of')]
>>>
```

```
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)
```


Spark Anti-Patterns

Here are a couple code clues that you are not working with Spark, but probably against it.

```
for loops, collect in middle of analysis, large data structures
```

```
...  
intermediate_results = data_rdd.collect()  
python_data = []  
for datapoint in intermediate_results:  
    python_data.append(modify_datapoint(datapoint))  
next_rdd = sc.parallelize(python_data)  
...
```

Ask yourself, "would this work with billions of elements?". And likely anything you are doing with a for is something that Spark will gladly parallelize for you, if you let it.

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) )
```

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. But they do not scale 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 actually a parallel databank that could scale to PBs.



* See Panda's creator Wes McKinney's "10 Things I Hate About Pandas" at <https://wesmckinney.com/blog/apache-arrow-pandas-internals/>

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)
['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 (Hadoop 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 t
- 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.

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 importantly, 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. 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 require a little more code to line up the types properly. But some of the latest features use them.

And while they would just complicate our basic examples, they are often simpler for real research problems. So don't shy away from using 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"] )
```

Just Spark DataFrames making life easier..

Data from <https://github.com/spark-examples/pyspark-examples/raw/master/resources/zipcodes.json>

```
{"RecordNumber":1,"Zipcode":704,"ZipCodeType":"STANDARD","City":"PARC PARQUE","State":"PR","LocationType":"NOT ACCEPTABLE","Lat":17.96,"Long":-66.22,"Xaxis":0.38,"Yaxis":-0.87,"Zaxis":0.3,"WorldRegion":...}
{"RecordNumber":2,"Zipcode":704,"ZipCodeType":"STANDARD","City":"PASEO COSTA DEL SUR","State":"PR","LocationType":"NOT ACCEPTABLE","Lat":17.96,"Long":-66.22,"Xaxis":0.38,"Yaxis":-0.87,"Zaxis":0.3,"WorldRegion":...}
{"RecordNumber":10,"Zipcode":709,"ZipCodeType":"STANDARD","City":"BDA SAN LUIS","State":"PR","LocationType":"NOT ACCEPTABLE","Lat":18.14,"Long":-66.26,"Xaxis":0.38,"Yaxis":-0.86,"Zaxis":0.31,"WorldRegion":...}
```

```
>>> df = spark.read.json("zipcodes.json")
>>> df.printSchema()
root
 |-- City: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Decommissioned: boolean (nullable = true)
 |-- EstimatedPopulation: long (nullable = true)
 |-- Lat: double (nullable = true)
 |-- Location: string (nullable = true)
 |-- LocationText: string (nullable = true)
 |-- LocationType: string (nullable = true)
 |-- Long: double (nullable = true)
 |-- Notes: string (nullable = true)
 |-- RecordNumber: long (nullable = true)
 |-- State: string (nullable = true)
 |-- TaxReturnsFiled: long (nullable = true)
 |-- TotalWages: long (nullable = true)
 |-- worldRegion: string (nullable = true)
 |-- Xaxis: double (nullable = true)
 |-- Yaxis: double (nullable = true)
 |-- Zaxis: double (nullable = true)
 |-- ZipCodeType: string (nullable = true)
 |-- Zipcode: long (nullable = true)
```

```
>>> df.show()
```

City	Country	Decommissioned	EstimatedPopulation	Lat	Location
PARC PARQUE	US	false	null	17.96	NA-US-PR-PARC PARQUE
PASEO COSTA DEL SUR	US	false	null	17.96	NA-US-PR-PASEO CO...
BDA SAN LUIS	US	false	null	18.14	NA-US-PR-BDA SAN ...
CINGULAR WIRELESS	US	false	null	32.72	NA-US-TX-CINGULAR...
FORT WORTH	US	false	4053	32.75	NA-US-TX-FORT WORTH
FT WORTH	US	false	4053	32.75	NA-US-TX-FT WORTH
URB EUGENE RICE	US	false	null	17.96	NA-US-PR-URB EUGE...
MESA	US	false	26883	33.37	NA-US-AZ-MESA
MESA	US	false	25446	33.38	NA-US-AZ-MESA
HILLIARD	US	false	7443	30.69	NA-US-FL-HILLIARD
HOLDER	US	false	null	28.96	NA-US-FL-HOLDER
HOLT	US	false	2190	30.72	NA-US-FL-HOLT
HOMOSASSA	US	false	null	28.78	NA-US-FL-HOMOSASSA
BDA SAN LUIS	US	false	null	18.14	NA-US-PR-BDA SAN ...
SECT LANAUSSÉ	US	false	null	17.96	NA-US-PR-SECT LAN...
SPRING GARDEN	US	false	null	33.97	NA-US-AL-SPRING G...
SPRINGVILLE	US	false	7845	33.77	NA-US-AL-SPRINGVILLE
SPRUCE PINE	US	false	1209	34.37	NA-US-AL-SPRUCE PINE
ASH HILL	US	false	1666	36.4	NA-US-NC-ASH HILL
ASHEBORO	US	false	15228	35.71	NA-US-NC-ASHEBORO

And Sometime DataFrames Are Limiting

DataFrames are not as flexible as plain RDDs, and it isn't uncommon to find yourself fighting to do something that would be simple with a map, for example. In that case, don't hesitate to flip back into a plain RDD.

```
>>> 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"] )  
  
>>> another_row_rdd = aDataFrameFromRDD.rdd
```

Notice that this is not even a method, it is just a property. This is a clue that behind the scenes we are always working with RDDs.

A minor technicality here is that the returned object is actually a "Row" type. You may not care. If you want it be the original tuple type then

```
>>> tuple_rdd = aDataFrameFromRDD.rdd.map(tuple)
```

Note that when our map function is a function that already exists, there is no need for a lambda.

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](#).

Other Scalable Alternatives: Dask



Of the many alternatives to play with data on your laptop, there are only a few that aspire to scale up to big data. The only one, besides Spark, that seems to have any traction is Dask.

It attempts to retain more of the "laptop feel" of your toy codes, making for an easier port. The tradeoff is that the scalability is a lot more mysterious. If it doesn't work - or someone hasn't scaled the piece you need - your options are limited.

At this time, I'd say it is riskier, but academic projects can often entertain more risk than industry.

Numpy like operations

```
import dask.array as da
a = da.random.random(size=(10000, 10000),
                     chunks=(1000, 1000))
a + a.T - a.mean(axis=0)
```

Dataframes implement Pandas

```
import dask.dataframe as dd
df = dd.read_csv('/.../2020-**-*.csv')
df.groupby(df.account_id).balance.sum()
```

Pieces of Scikit-Learn

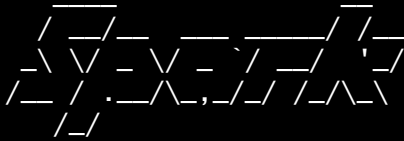
```
from dask_ml.linear_model import \
LogisticRegression
lr = LogisticRegression()
lr.fit(train, test)
```

Drill Down?

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.

Machine Learning

The background of the slide features a horizontal split. The top half is a dark, almost black sky with a thin, wispy layer of clouds just above the horizon. The bottom half is a deep blue gradient, representing the surface of water, which transitions from a lighter blue near the horizon to a darker blue at the bottom.

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.

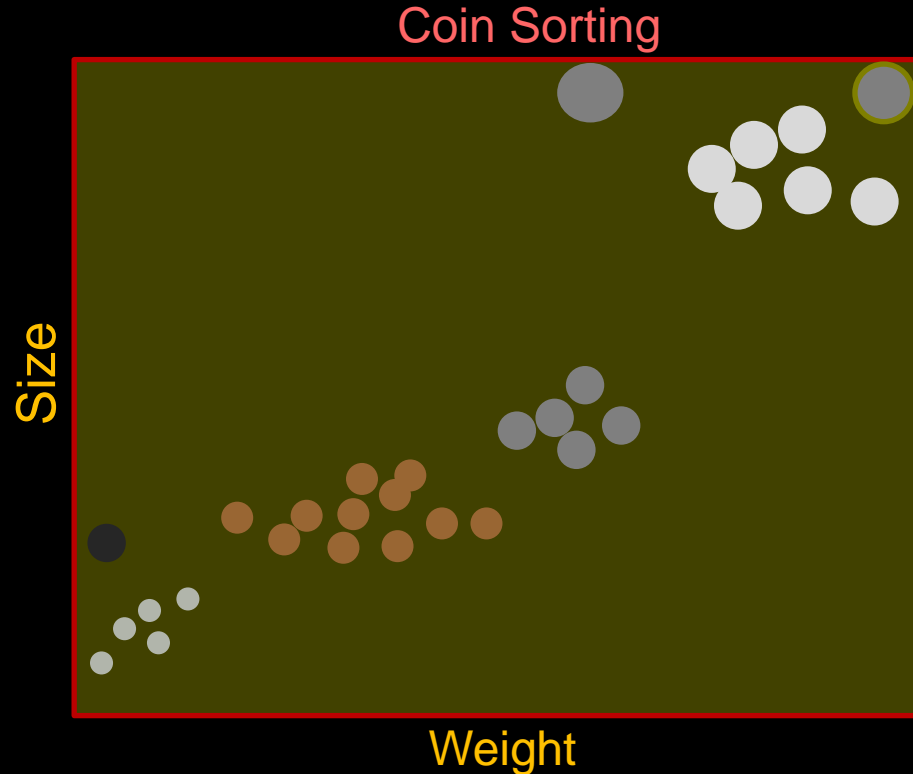
There are good example documents for the clustering routine we are using, as well as alternative clustering algorithms, here:

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

I suggest you use these pages for 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.



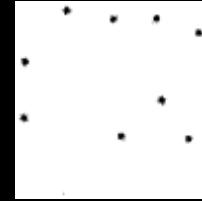
Clustering

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

You might think

high-dimensional spaces.

But it can get

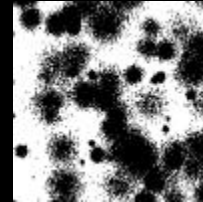


Sometimes you

start with. Often you don't.

How hard can

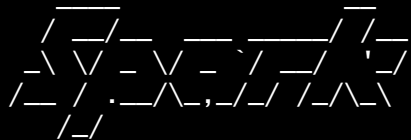
there?



From 1900 until 1956 humans were considered to have 48 chromosomes, instead of 46, based upon the interpretation of this camera lucida image.

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
SparkContext available

```
>>>  
>>> rdd1 = sc.textFile(  
>>>  
>>> rdd2 = rdd1.map(  
>>> rdd3 = rdd2.map(  
>>>
```

```
br06% interact
```

```
...  
r288%
```

```
r288%
```

```
r288% module load spark
```

```
r288% pyspark
```

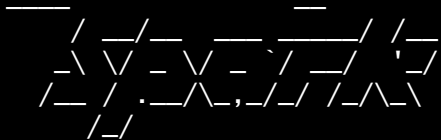
Make sure you are in the directory with the data file. Otherwise, Spark is dangerously quiet when you `textFile()` a file that does not exist. It is "lazy" and you won't find out that you have missing data until a later error.

and integers

Finding Our Way

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

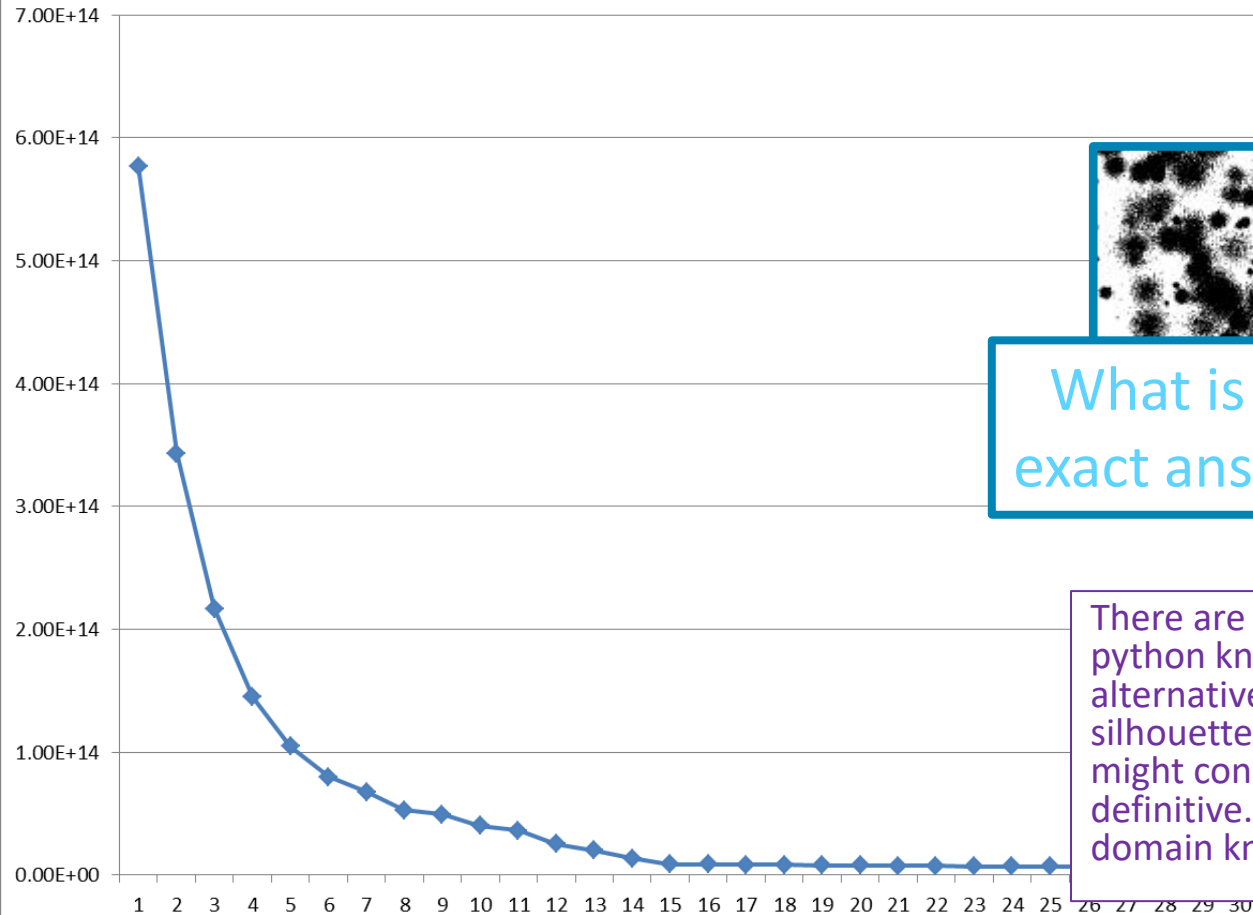
New in version 0.9.0.

```
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|| initializationMode. (default: None)

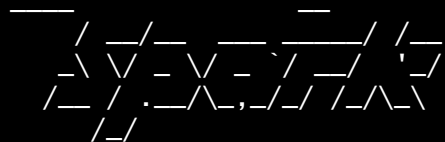
Finding Clusters



What is the exact answer?

There are helper algorithms (the python kneed package) or alternative metrics, such as the silhouette coefficient, that you might consider. None are definitive. Judgement and domain knowledge are critical.

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
```

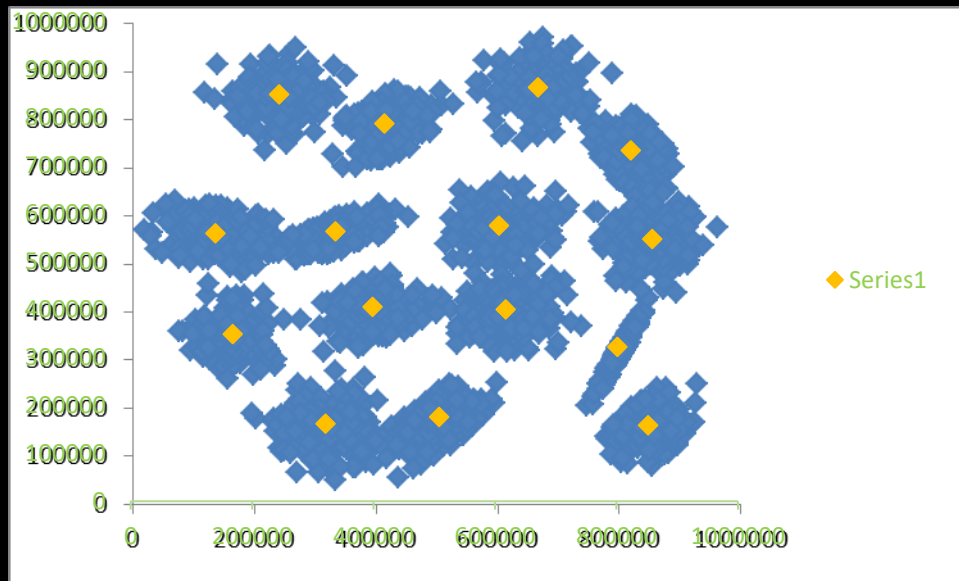
```
12 2.5187054064e+13  
13 1.83498739266e+13  
14 1.96076943156e+13  
15 1.41725666214e+13  
16 1.41986217172e+13  
17 8.46755159547e+12
```

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

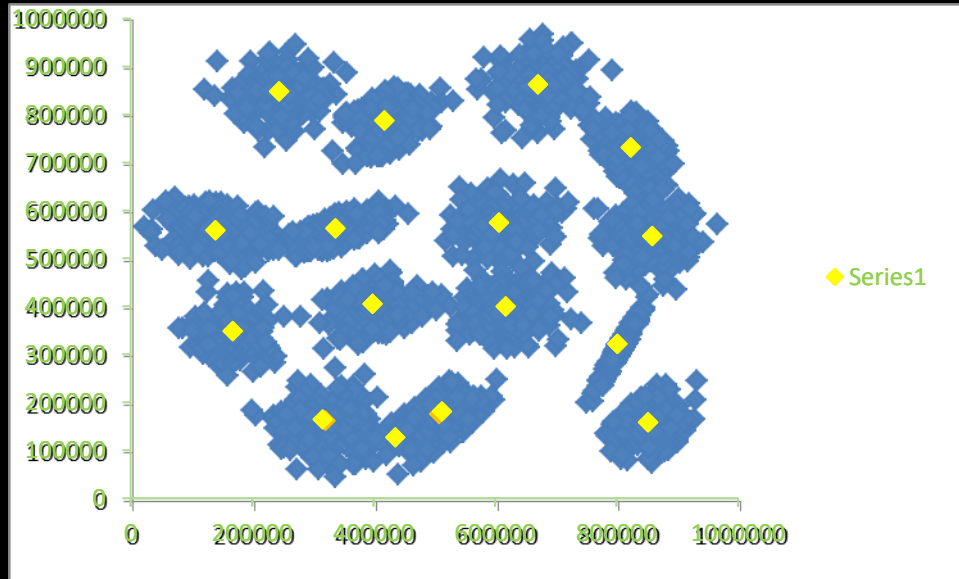
```
12 2.5187054064e+13  
13 2.04031903147e+13  
14 1.95213876047e+13  
15 1.93000628589e+13  
16 2.07670831868e+13  
17 8.47797102908e+12
```

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


Fit?



16 Clusters



We are closer to leading edge science than you might think.

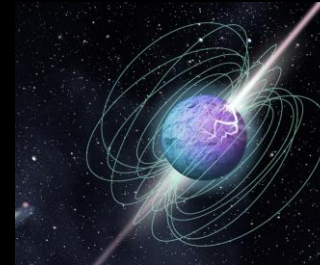


The LIGO gravitational wave detector was able to confirm the collision of two neutron stars with both a gamma ray satellite and optical and other electromagnetic spectrum telescopes. For these transient events, it requires rapid real-time signal analysis to steer other instruments to the proper celestial coordinates. The 2 second gamma-ray burst was detected 1.7 seconds after the GW merger signal. 70 observatories were able to mine signatures in the following days. Even so, the refined location alert took a long time, and much improvement lies ahead.



In April 2020, astronomers picked up some bursts of activity, in the X-ray band of the spectrum, a “run-of-the-mill” magnetar. But the team found that, shortly after the magnetar burst in the X-ray band, CHIME picked up two sharp staccato peaks in the radio band, within several milliseconds of each other, signaling a fast radio burst. The researchers were able to track the radio bursts to a point in the sky that was within a fraction of a degree of SGR 1935+2154 — the same magnetar that was blasting out X-rays around the same time. The team used calibration data from other astrophysical sources to estimate the magnetar’s brightness. They calculated that the magnetar, in the fraction of a second that the FRB flashed, was 3,000 times brighter than any other magnetar radio signal that has yet been observed. Happening in our own galaxy, thousands of times brighter than any other pulse we’ve ever seen.

This rapid processing requirement will only become more extreme as the Square Kilometer Array comes fully on-line. It will generate over an Exabyte of data a day. It will require extreme real-time processing to classify and compress this data down to an archivable size.



Strange, repeating radio signal near the center of the Milky Way has scientists stumped



This article (www.livescience.com/strange-radio-source-milky-way-center) is the summary of the paper (arxiv.org/pdf/2109.00652.pdf) that looks an awful lot like what we are doing.

Assignment: Using Spark to mine astro signals

Q: Can you find the repeating cosmological signal (pulsar?) in the captured data?

In `~/urbanic/Advanced_Computational_Physics/Spark/pulsar.dat` on Bridges you will find the data with a series of signals stored as:

```
ascension (degrees), declination (degrees), time (seconds), frequency (MHz)
```

These are radiofrequency signals captured by an array of instruments scanning a solid angle of the sky.

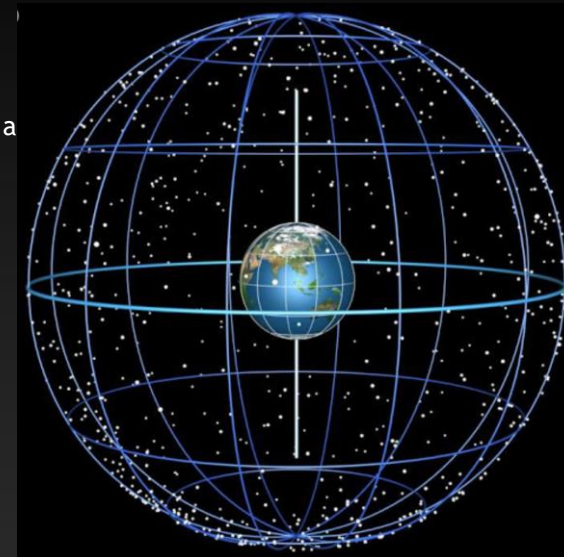
The data is, like almost all real data, a little noisy and has sources of errors. In our case the angular coordinates have 0.1 degrees error, the signal frequency has 0.1 MHz error and the timebase/period error is $< 0.01s$ (that is one STD or standard deviation).

Your job is to find the most regular temporarily repeating RF source.

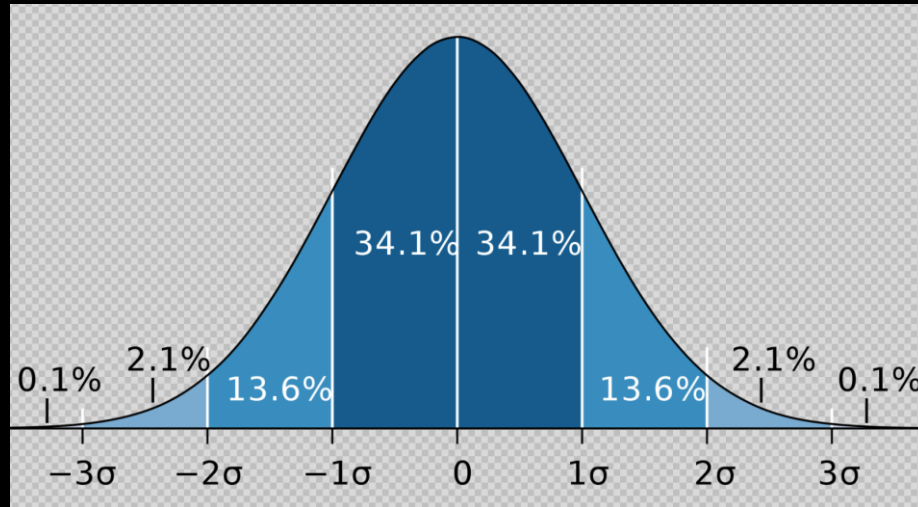
Your target will be found in the same location (within error) of the sky, on the same frequency (within error) chirping for the most blips, regularly spaced in time during that active period. So...

.....blip...blip...blip...blip...blip...blip.....

Again, you are looking for the signal with the most blips.

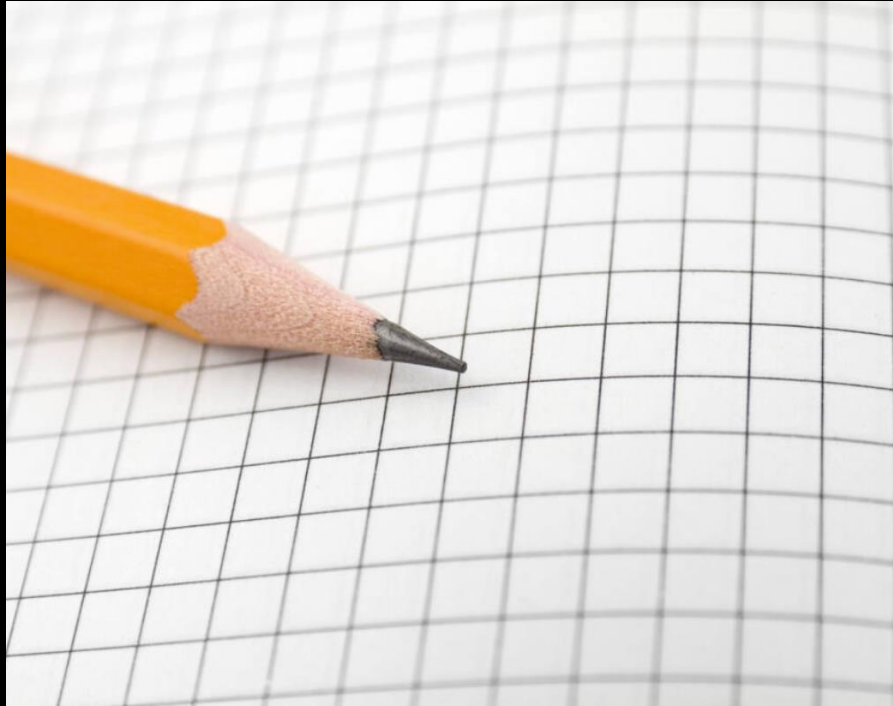


A brief word about errors...



For a **normal** or **gaussian distribution**, the standard deviation indicates the region where about 2/3 of the points will fall. And for small sample sizes, this may be noisy.

How to approach an algorithmic problem...



Think about how you, without a computer, would attempt a tiny version of the problem. Once you think about how you would do it, an algorithm often becomes clear. If you can't think about how you would do it, writing lines of code isn't going to make a solution materialize.

Assignment: Using Spark to mine astro signals

There are multiple approaches to this problem that will work. I can think of several that would be dead painful (hints in lecture). In particular, **clustering is not needed!** We happen to be covering this now, and it seems like an intuitively useful tool for this task, but it is not necessary at all, and I think it is much harder than other simpler methods.


Biggest Hint: I am a nice guy! While the data could have nasty surprises, it does not. Use common sense and experiment around and you will find an clear answer.

This is doable interactively. You can explore the data within PySpark using operations that are reasonably quick. You do not need to create long running scripts to get to the answer.

To recap:

1. log on
2. cp the datafile to where you want to work with your pyspark session
3. get an interactive node
4. load the spark module
5. start a pyspark session
6. load the datafile
7. and use transforms to wring out the answer: the coordinates, the frequency and the period (rounded to integers is fine).

Use Spark commands (RDDs) to derive your answer. Yes, with a dataset this size you could use python directly, but a real dataset would be far too large. Use Spark transforms to boil down your data until you have a modest amount (screenful) of data to inspect. Use python at that point, if you wish, or just observe your answer. Ask if you don't understand this point. You could lose credit otherwise.

Submit your answer along with the exact sequence of spark commands that got you there. *This means a single email.* An attachment of the  commands with your answer in the body of the mail, and any explanation you care to provide, is the required format.

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, let's 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.

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	...

80 dimensional vector.

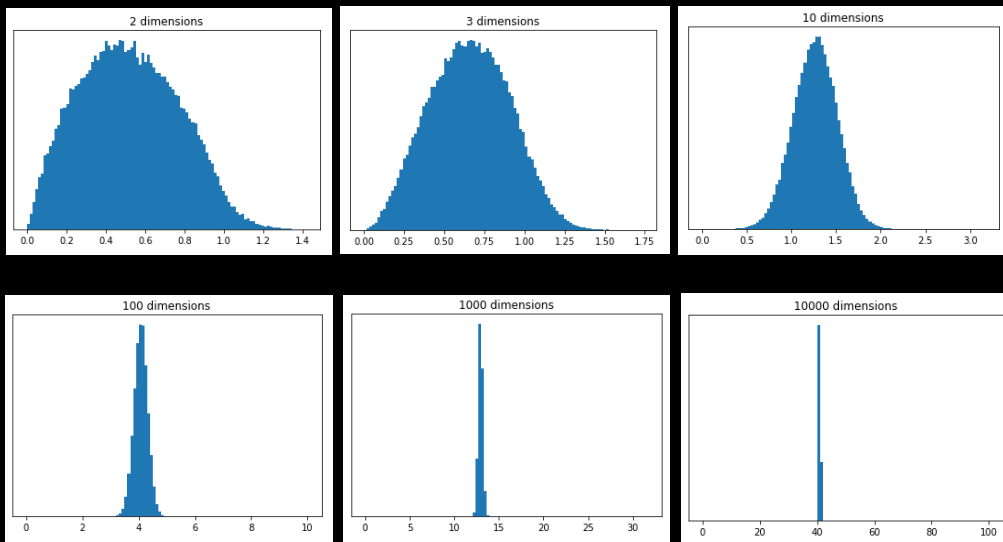
We have now accomplished two things:

- 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.

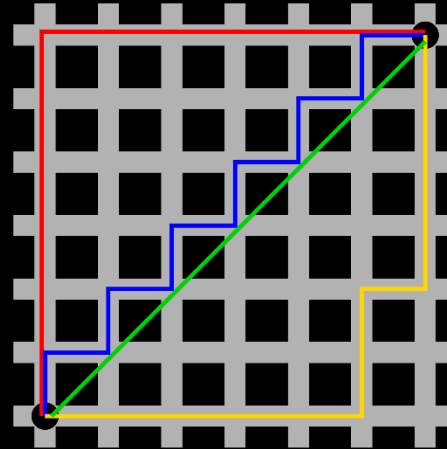


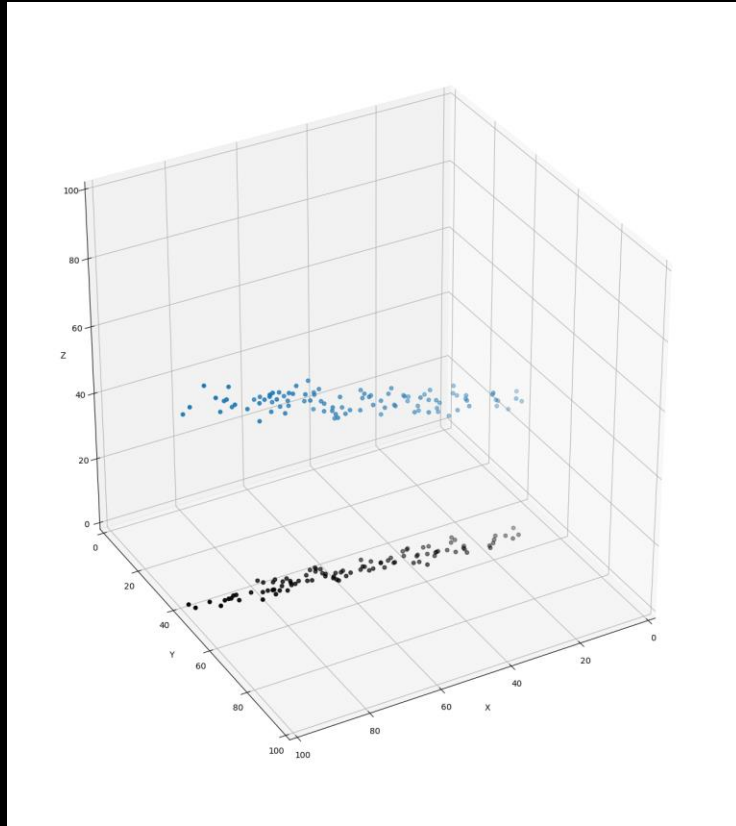
Image Source: Wikipedia

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.

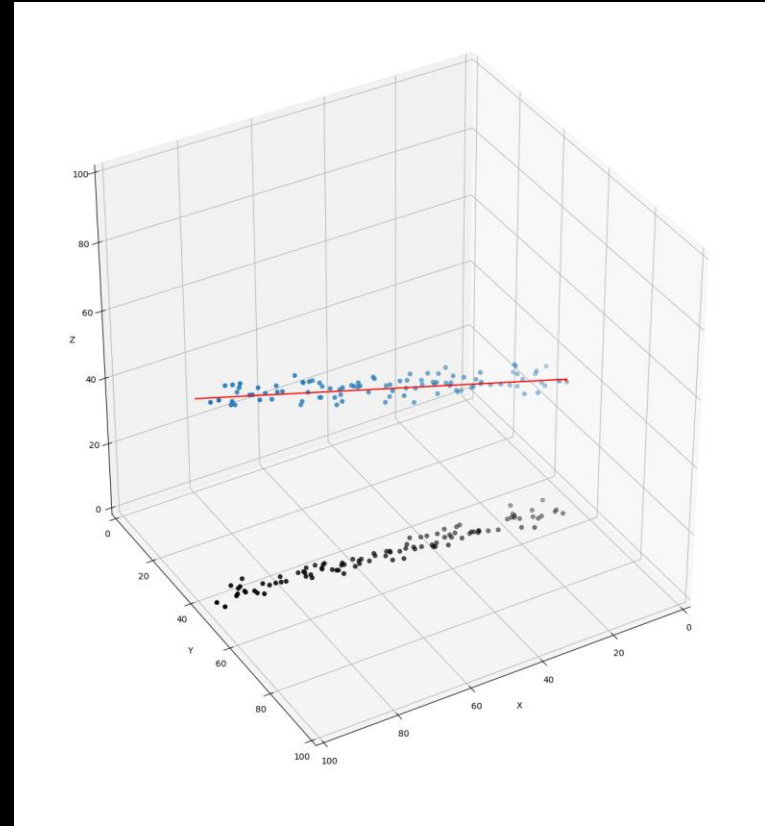
For genomics, you might care more about string sequences.

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

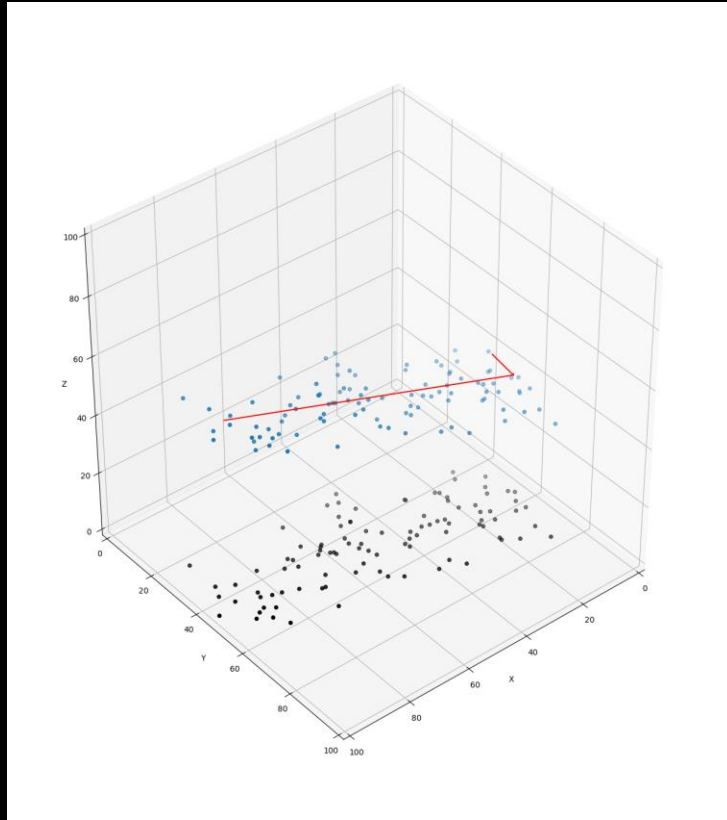


3D Data Set

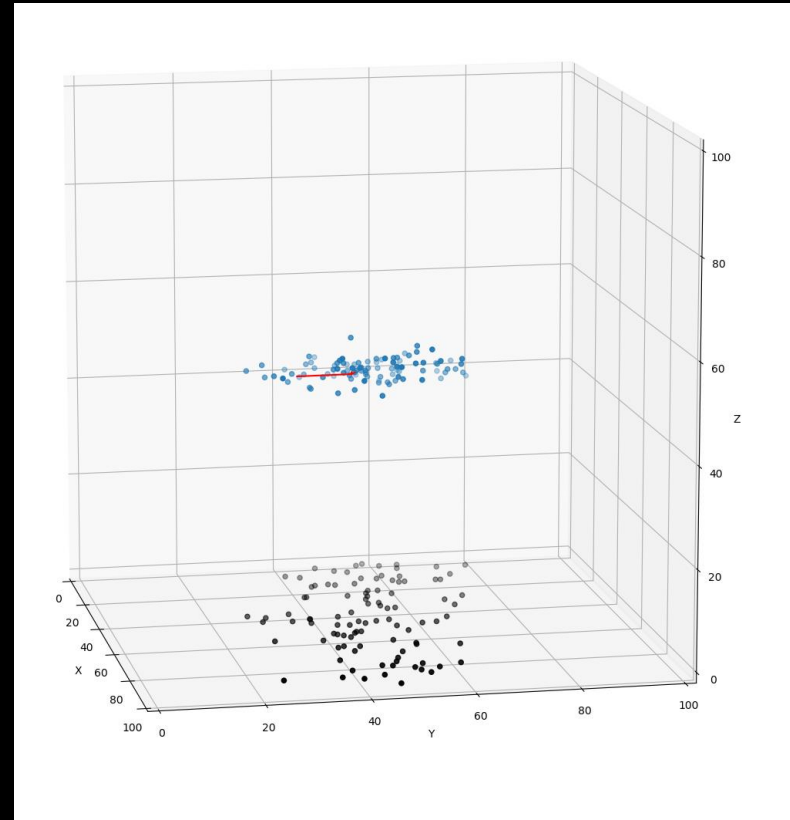


Maybe mostly 1D!

Alternative DR: Principal Component Analysis



Flatter 2D-ish Data Set



View down the 1st Princ. Comp.

Why So Many Alternatives?

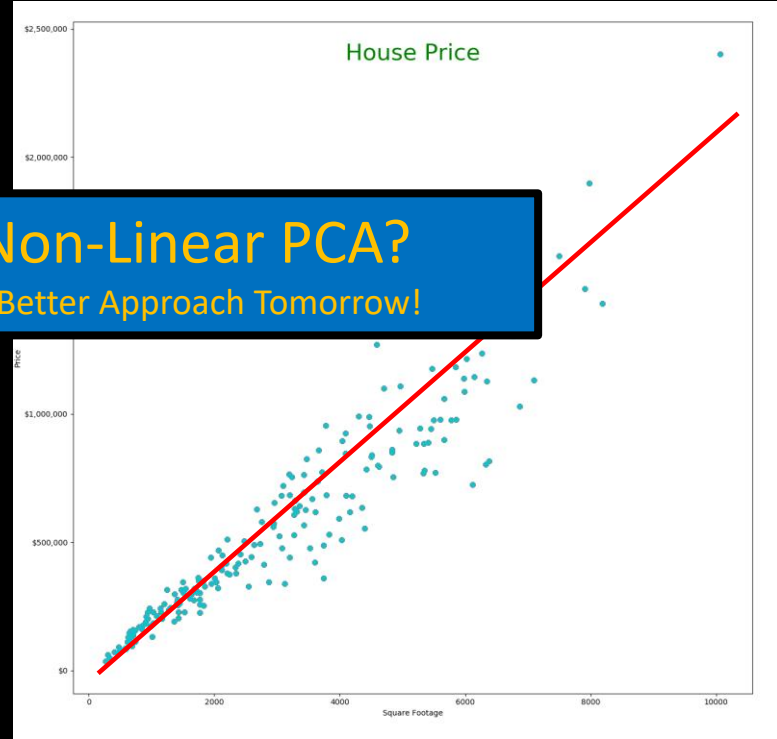


Let's look at one more example today. Suppose we are trying to do a Zillow type of analysis and predict home values based upon available factors. We may have an entry (vector) for each home that captures this kind of data:

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



Non-Linear PCA?
A Better Approach Tomorrow!

1st Component Off
Data Not Very Linear

D x W Is Not Linear
But (DxW) Fits Well

Why the fascination with linear techniques?



The Streetlight Effect

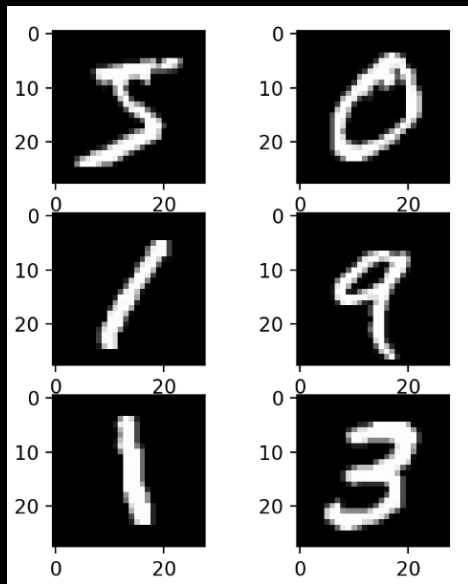
This is a very real and powerful force throughout the sciences.

It is not because practitioners are dumb.

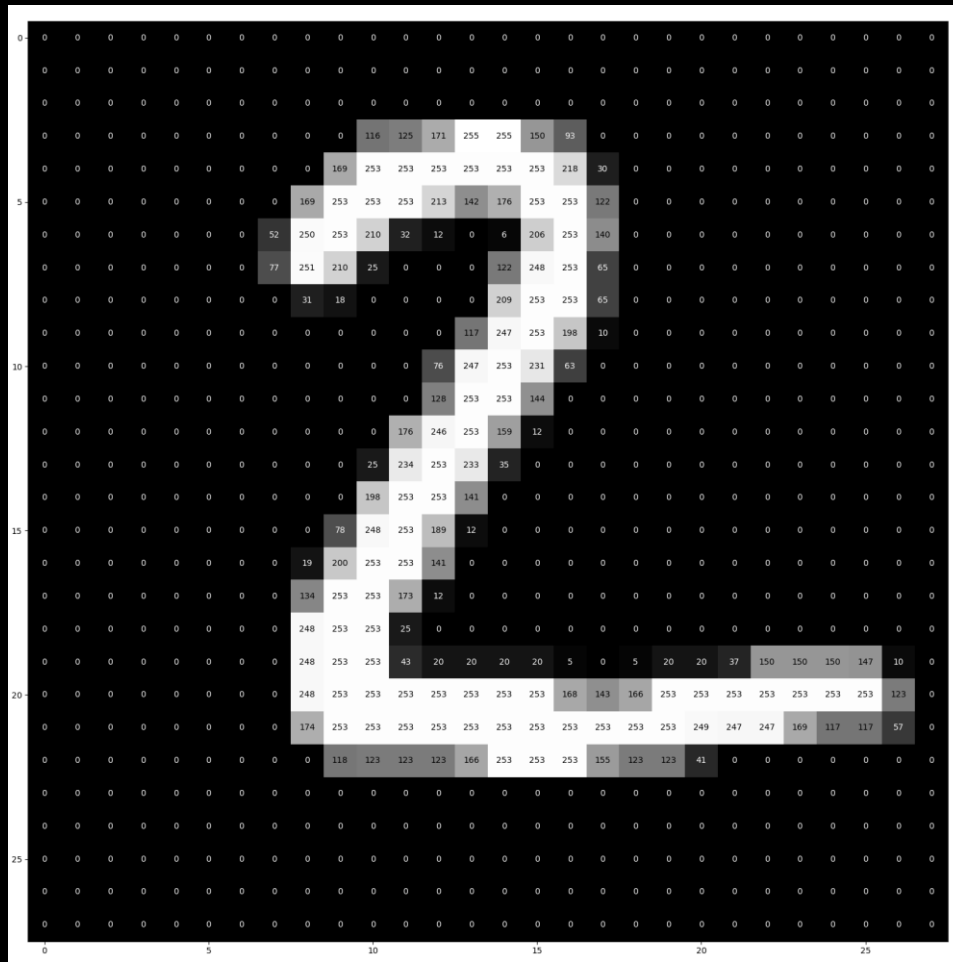
But, it is also very often neither explained nor justified.

Which leads to great confusion.

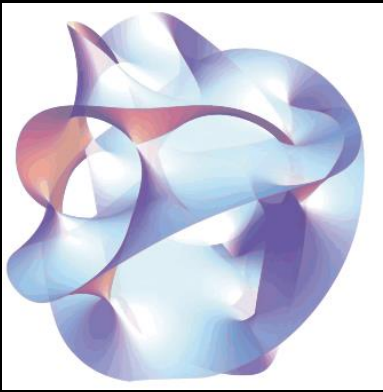
Why Would An Image Have 784 Dimensions?



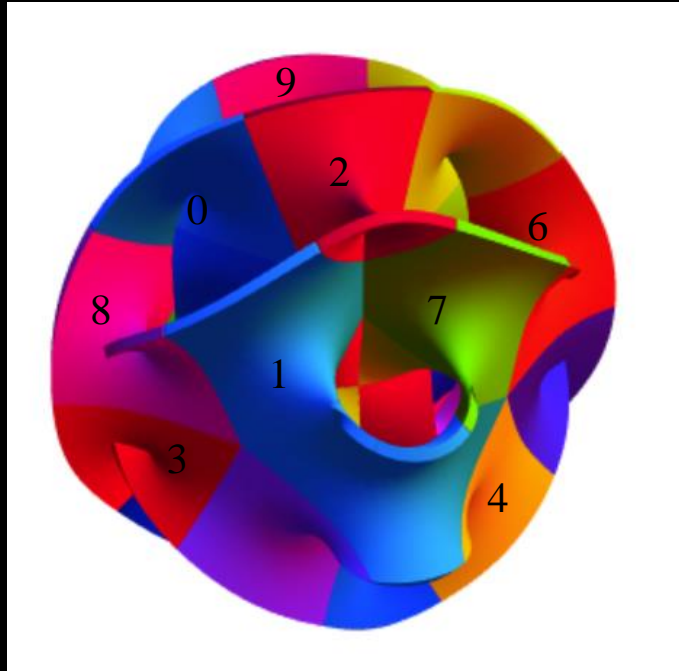
MNIST 28x28
greyscale images



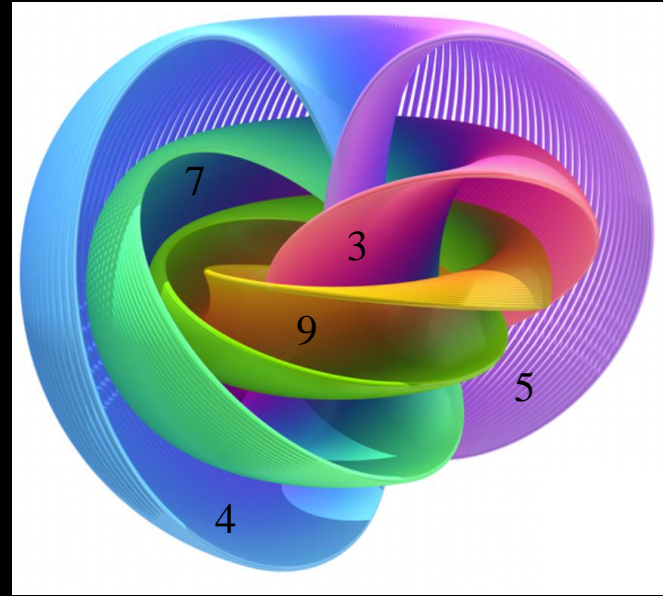
Central Hypothesis of Modern DL



Data Lives On
A Lower Dimensional
Manifold



Maybe Very Contiguous



Maybe A Small Set
Of Disconnected

Testing These Ideas With Scikit-learn



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import (datasets, decomposition, manifold, random_projection)
```

```
def draw(X, title):
    plt.figure()
    plt.xlim(X.min(0)[0], X.max(0)[0]); plt.ylim(X.min(0)[1], X.max(0)[1])
    plt.xticks([]); plt.yticks([])
    plt.title(title)
    for i in range(X.shape[0]):
        plt.text(X[i, 0], X[i, 1], str(y[i]), color=plt.cm.Set1(y[i] / 10.))
```

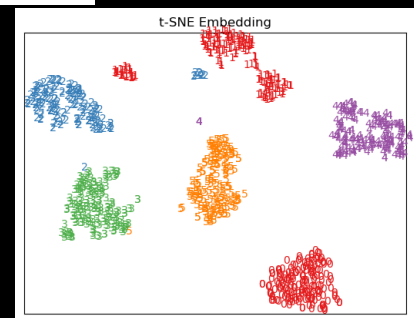
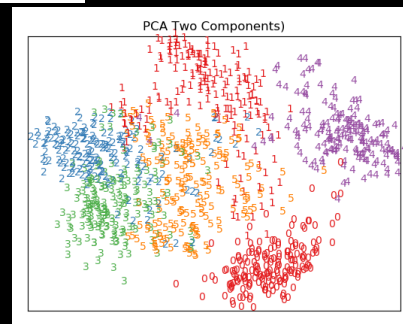
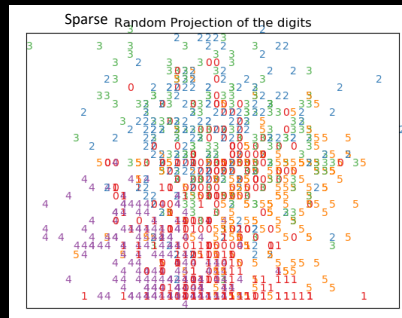
```
digits = datasets.load_digits(n_class=6)
X = digits.data
y = digits.target
```

```
rp = random_projection.SparseRandomProjection(n_components=2, random_state=42)
X_projected = rp.fit_transform(X)
draw(X_projected, "Sparse Random Projection of the digits")
```

```
X_pca = decomposition.PCA(n_components=2).fit_transform(X)
draw(X_pca, "PCA (Two Components)")
```

```
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
X_tsne = tsne.fit_transform(X)
draw(X_tsne, "t-SNE Embedding")
```

```
plt.show()
```



Sample of 64-dimensional digits dataset

0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	
5	5	0	4	1	3	5	1	0	0	2	2	2	0	1	4	3	3	3	3	3	3	3	3	3
4	4	1	5	0	5	1	4	0	0	1	3	2	1	4	1	3	1	7	1	4	1	4	1	4
3	1	4	0	5	7	1	5	6	4	2	2	5	5	6	0	0	1	1	1	1	1	1	1	1
2	7	4	5	0	4	2	3	4	1	0	4	2	3	4	0	0	5	5	5	5	5	5	5	5
0	4	1	3	5	1	0	0	2	1	1	0	1	1	3	3	3	3	4	4	4	4	4	4	4
4	5	0	5	2	1	0	0	4	3	1	4	3	1	3	4	4	4	4	4	4	4	4	4	4
0	5	7	4	5	4	4	1	1	3	5	4	4	0	0	0	0	0	0	0	0	0	0	0	0
0	5	7	4	5	4	4	1	1	3	5	4	4	0	0	0	0	0	0	0	0	0	0	0	0
3	5	1	0	0	2	2	2	0	4	2	3	3	3	3	4	4	4	4	4	4	4	4	4	4
5	5	2	2	0	0	1	3	2	4	6	3	4	3	1	4	3	1	6	5	5	5	5	5	5
3	1	5	4	2	2	2	5	7	4	0	3	0	1	2	3	4	5	5	5	5	5	5	5	5
0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0
5	5	1	0	0	1	2	0	1	1	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4
2	1	0	0	1	3	1	4	4	3	1	4	3	1	4	3	1	4	3	1	4	3	1	4	3
4	5	4	4	2	1	4	5	6	4	4	0	1	2	3	4	5	0	1	2	3	4	5	0	1
1	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2
0	0	0	1	2	0	1	1	3	3	3	3	4	4	5	0	1	2	3	4	5	0	1	2	3
0	0	1	2	1	4	3	1	4	3	1	4	3	1	4	3	1	4	3	1	4	3	1	4	3
4	4	2	1	5	6	0	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5

How does all this fit together?

Big
Data



DL
Deep Neural Nets

DL

Paintings of
Monkeys Piloting Jets

Angsty Poetry

Character Recognition
Captcha

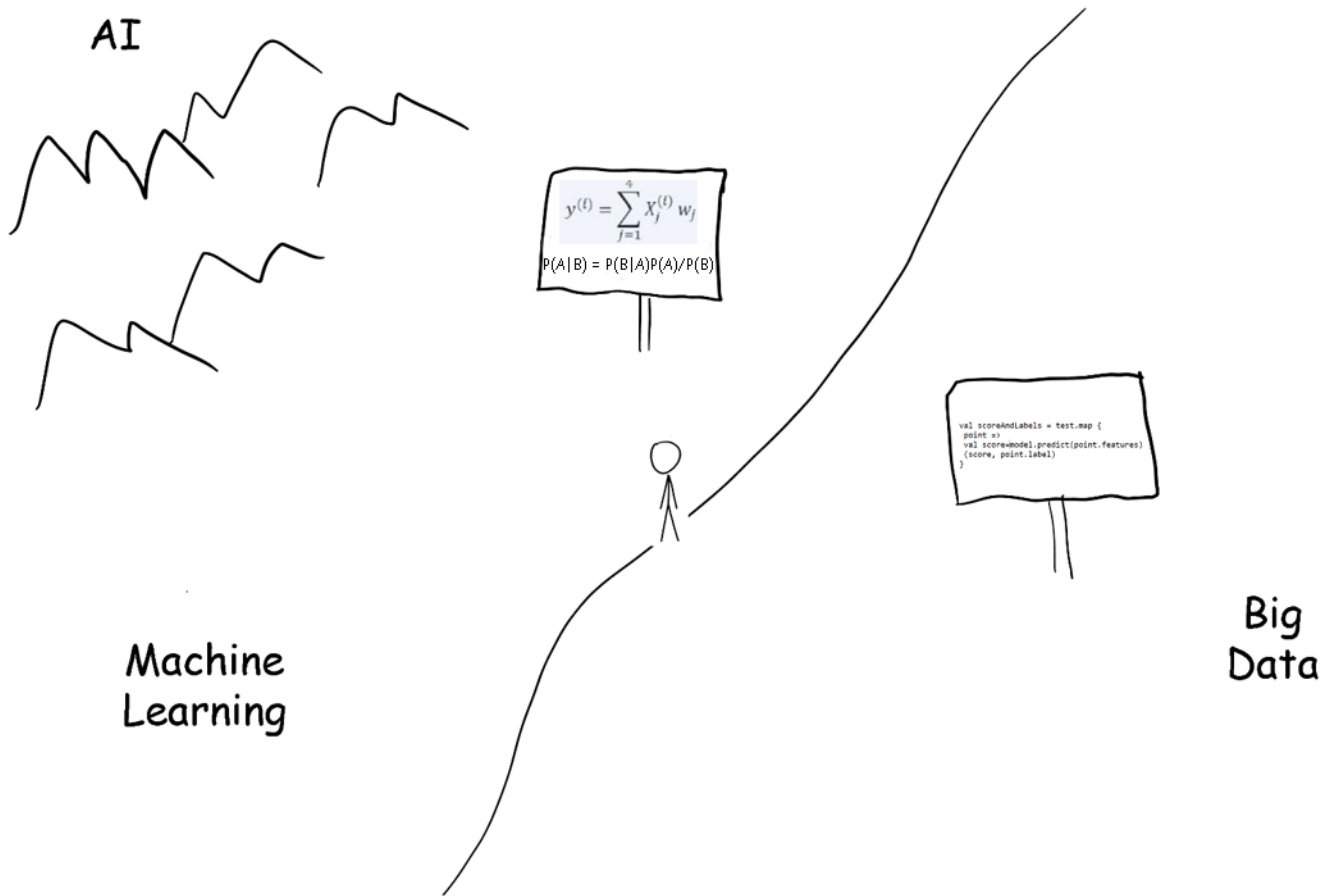
Chess

Go

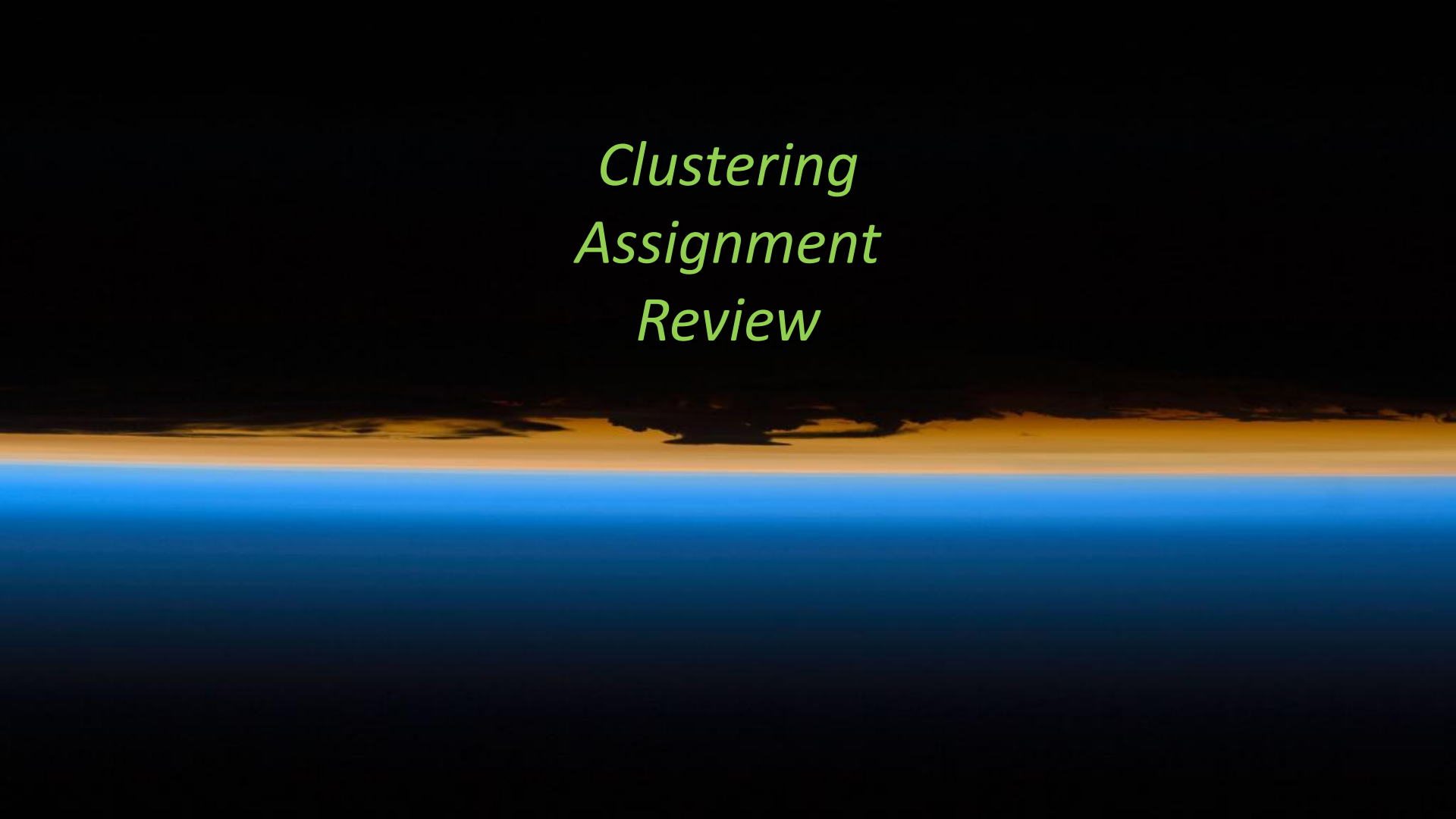
ML

AI

The Journey Ahead



As the Data Scientist wanders across the ill-defined boundary between Data Science and Machine Learning, in search of the fabled land of Artificial Intelligence, they find that the language changes from programming to a creole of linear algebra and probability and statistics.

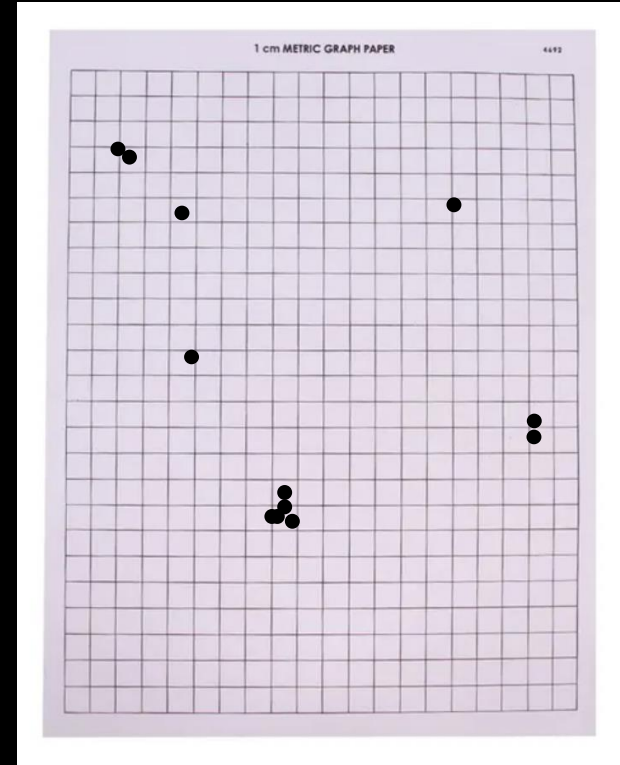


*Clustering
Assignment
Review*

Our particular problem.

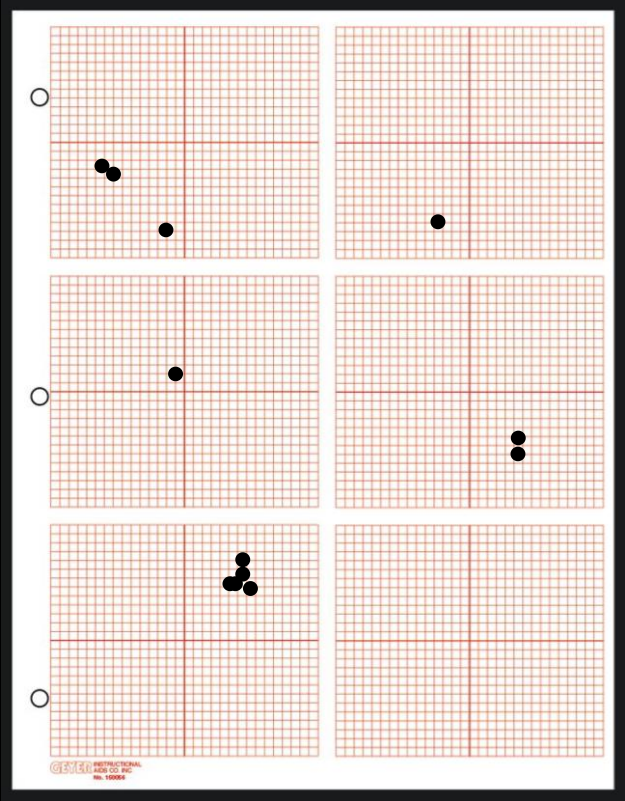
In this case, I might plot the points out on graph paper and see if any pattern emerges.

If this seems like a reasonable approach, I could think about how to implement this in Spark.



Spark will buy us scalability.

And if we do this with RDDs, we will naturally obtain scalability. Our helpers can all work on a small portion of the overall problem.



First Approach: Sort Coordinates Into Bins

```
# Just read stuff in and round off
raw = sc.textFile('pulsar.dat')
tokens = raw.map(lambda x: x.split())
floats = tokens.map(lambda x: (float(x[0]),float(x[1]),float(x[2]),float(x[3])) )
rounded = floats.map(lambda x: (round(x[0]),round(x[1]),round(x[2]),round(x[3])) )
```

```
rounded.take(5)
[(94, 108, 1766, 6310),
 (68, 93, 2370, 5114),
 (80, 101, 1324, 5299),
 (86, 91, 1386, 4572),
 (95, 77, 2000, 2841)]
```

```
keyed = rounded.map(lambda x: ( tuple([x[0],x[1]]), [[x[2],x[3]]] ) )
```

```
keyed.take(3)
[((94, 108), [[1766, 6310]]),
 ((68, 93), [[2370, 5114]]),
 ((80, 101), [[1324, 5299]])]
```

```
reduced = keyed.reduceByKey(lambda x,y: x+y)
```

```
reduced.take(2)
[((94, 108), [[1766, 6310], [1768, 6310]]),
 ((68, 93), [[2370, 5114], [2367, 5113], [594, 3576], [3074, 1572]])]
```


```
repeats = reduced.map(lambda x: ( x[0],len(x[1])) )
repeats.take(4)
out[14]: [((94, 108), 2), ((68, 93), 4), ((80, 101), 4), ((86, 91), 3)]
```

groupByKey() functions similarly here. It leaves you with a Spark *Iterable* type, which is fine but requires some further processing to see the result.

First Approach: Sort Coordinates Into Bins (contd.)

```
sorted = repeats.takeOrdered(30, key = lambda x: -x[1])
```

```
sorted  
[((73, 100), 14),  
 ((108, 101), 13),  
 ((101, 105), 12),  
 ((63, 107), 11),  
 ((63, 114), 11),  
 ((84, 107), 10),  
 ((75, 87), 10),  
 ((116, 119), 10),  
 ((108, 119), 10),  
 ((72, 100), 10)]
```



Insight: there are 14 points in at least this coordinate patch. But, they could be from different sources. Better look at all the data.

First Approach: Sort Coordinates Into Bins (contd.)

```
sorted_with_data = reduced.takeOrdered(4, key = lambda x: -len(x[1]))
```

```
sorted_with_data
```

```
[[((73, 100),  
  [[2471, 6973],  
   [2590, 3717],  
   [2442, 6973],  
   [2446, 6973],  
   [2140, 6112],  
   [2487, 6973],  
   [2512, 6973],  
   [2483, 6972],  
   [2458, 6972],  
   [2479, 6973],  
   [1455, 5587],  
   [1463, 5587],  
   [2499, 6972],  
   [2462, 6972]])],  
 ((108, 101),  
  [[1065, 1174],  
   [1896, 1706],  
   [1934, 1706],  
   [1904, 1706],  
   [1973, 1706],  
   [1942, 1706],  
   [1950, 1706],  
   [1911, 1706],  
   [1980, 1706],  
   [1073, 1174],  
   [1919, 1706],  
   [1965, 1706],  
   [1927, 1706]])],  
 ((101, 105),  
  [[40, 3284],  
   [2841, 1419],  
   [2839, 1419],  
   [2752, 7860],  
   [2651, 5529],  
   [1604, 5641],  
   [2901, 6707],  
   [2748, 7860],  
   [631, 3779],  
   [2648, 5529],  
   [1596, 5641],  
   [634, 3779]])],  
 ((63, 107),  
  [[1614, 1258],  
   [930, 2720],  
   [1615, 1259],  
   [1617, 1259],  
   [2173, 4264],  
   [1305, 4608],  
   [1618, 1259],  
   [2164, 4264],  
   [1313, 4608],  
   [971, 6670],  
   [935, 2720]])]
```

Insight: 73,100 has 10 signals
around 6973 MHz and 108,101
has 11 of 1706 MHz.

We should check neighboring
bins for both to see if there are
associated points there.

Or, maybe we can look for the
suspect frequencies and see
where they are.

Second Approach: Let's key on frequency.

```
# Use same RDD's as prior approach to get to "rounded"
```

```
freq_keys = rounded.map(lambda x: ( x[3], [[x[0],x[1],x[2]]] ) )
```

```
reduced_freq = freq_keys.reduceByKey(lambda x,y: x+y)
```

```
freq_sorted = reduced_freq.takeOrdered(5, key = lambda x: -len(x[1]))
```

```
freq_sorted
```

```
[(5182,  
  [[119, 97, 3206],  
   [77, 78, 1623],  
   [119, 96, 3221],  
   [119, 96, 3217],  
   [112, 82, 583],  
   [84, 103, 1981],  
   [64, 115, 96],  
   [81, 100, 3551],  
   [119, 96, 3210],  
   [119, 97, 3202],  
   [77, 78, 1616],  
   [119, 96, 3213]]),  
(1706,  
  [[108, 101, 1896],  
   [108, 101, 1934],  
   [109, 101, 1957],  
   [108, 101, 1904],  
   [108, 101, 1973],  
   [108, 101, 1942],  
   [108, 101, 1950],  
   [108, 101, 1911],  
   [108, 101, 1980],  
   [108, 101, 1919],  
   [108, 101, 1965],  
   [108, 101, 1927]]),
```

```
(5402,  
  [[82, 110, 1726],  
   [96, 92, 3137],  
   [82, 110, 1712],  
   [82, 110, 1721],  
   [76, 110, 2253],  
   [109, 106, 1633],  
   [82, 110, 1717],  
   [82, 110, 1735],  
   [82, 110, 1731],  
   [109, 106, 1625]]),  
(6973,  
  [[73, 100, 2471],  
   [73, 100, 2442],  
   [72, 100, 2450],  
   [73, 100, 2446],  
   [73, 100, 2487],  
   [73, 100, 2512],  
   [72, 100, 2467],  
   [72, 100, 2475],  
   [73, 100, 2479],  
   [72, 100, 2508]]),  
(7592,  
  [[66, 83, 1161],  
   [66, 102, 3363],  
   [90, 75, 2767],  
   [90, 75, 2774],  
   [73, 94, 3422],  
   [68, 109, 410],  
   [68, 109, 403],  
   [66, 83, 1169],  
   [66, 102, 3365]])
```

Both 1706 MHz and 6973 MHz look like strong candidates. And 6973 does indeed bleed into a neighboring box.

Let's focus on just those two frequency bands.

Second Approach: Let's key on frequency. (contd.)

```
reduced_freq.lookup(6973)
```

```
[[[73, 100, 2471],  
  [73, 100, 2442],  
  [72, 100, 2450],  
  [73, 100, 2446],  
  [73, 100, 2487],  
  [73, 100, 2512],  
  [72, 100, 2467],  
  [72, 100, 2475],  
  [73, 100, 2479],  
  [72, 100, 2508]]]
```

```
reduced_freq.lookup(6972)
```

```
[[[72, 100, 2491],  
  [73, 100, 2483],  
  [73, 100, 2458],  
  [72, 100, 2495],  
  [72, 100, 2504],  
  [73, 100, 2499],  
  [72, 100, 2454],  
  [73, 100, 2462]]]
```

```
reduced_freq.lookup(6974)
```

```
[[[97, 78, 2139], [97, 78, 2148]]]
```

18 correlated sources here.

```
reduced_freq.lookup(1706)
```

```
[[[108, 101, 1896],  
  [108, 101, 1934],  
  [109, 101, 1957],  
  [108, 101, 1904],  
  [108, 101, 1973],  
  [108, 101, 1942],  
  [108, 101, 1950],  
  [108, 101, 1911],  
  [108, 101, 1980],  
  [108, 101, 1919],  
  [108, 101, 1965],  
  [108, 101, 1927]]]
```

```
reduced_freq.lookup(1707)
```

```
[[[92, 67, 135], [92, 89, 377]]]
```

```
reduced_freq.lookup(1705)
```

```
[[[94, 95, 2506]]]
```

Only 12 here. We have a v

Don't Panic!

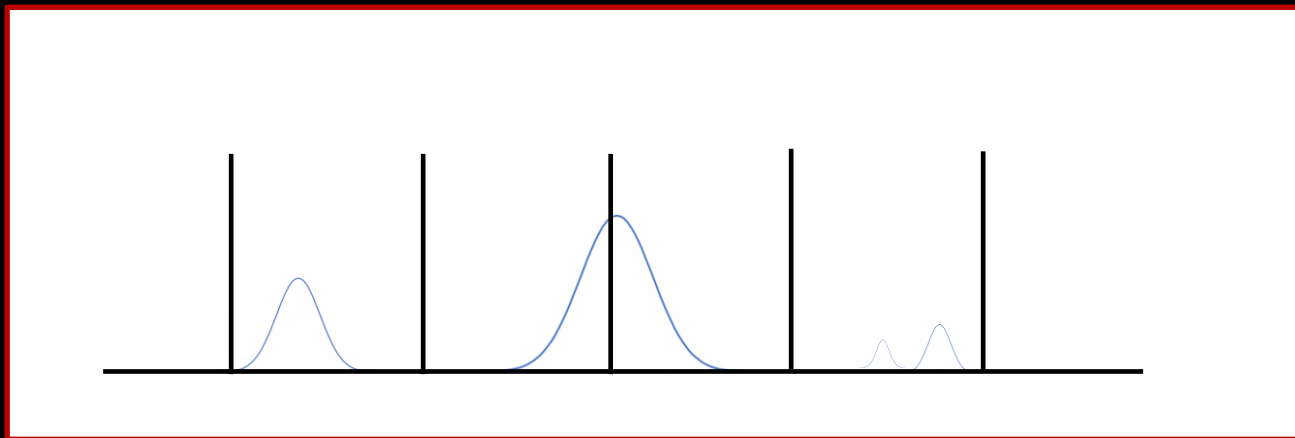
This isn't a pass/fail assignment. Maybe you didn't find the strongest source? As long as you made a considered attempt, there is plenty of credit to go around. Hopefully, you gave me enough code/comments to let me give you as much as possible.

More Rigorous Solutions

I gave you permission to be a little casual about locating the source. And I gave you a dataset that wasn't intentionally devious. But you can see that we could need to be more diligent about dealing with sources that get rounded or sorted in odd ways.

If we wanted to pursue the analysis techniques that I am using here, we could scale up (normalize) our dataset depending on the std, and consider more carefully what happens when a signal is on a bin boundary.

Let's use Spark to see how we can be a little more (although not completely) thorough.



Third Approach: Shifting data around before reducing.

```
# Use same RDD's as prior approach to get to the "floats" RDD.
```

```
def shift_bin (x):
```

```
  std = 0.1
```

```
  up_shift = (x[0]+std,x[1],x[2],x[3])
```

```
  down_shift = (x[0]-std,x[1],x[2],x[3])
```

```
  return (x,up_shift,down_shift)
```

```
freq_first = floats.map(lambda x: (x[3],x[0],x[1],x[2]) )
```

```
freq_smeared = freq_first.flatMap(shift_bin)
```

```
freq_rounded = freq_smeared.map(lambda x: ( round(x[0]), x[1], x[2], x[3]) )
```

```
freq_rounded.take(3)
```

```
[(5079, 98.98274915340335, 80.51828454605635, 5340.876552864837),
```

```
 (5079, 98.98274915340335, 80.51828454605635, 5340.876552864837),
```

```
 (5079, 98.98274915340335, 80.51828454605635, 5340.876552864837)]
```

```
freq_distinct = freq_rounded.distinct() # Dangerous to do with floats! Should really round properly first
```

```
freq_key_value = freq_distinct.map(lambda x: ( x[0], [[x[1],x[2],x[3]]] ) )
```

```
reduced_freq = freq_key_value.reduceByKey(lambda x,y: x+y)
```

```
freq_sorted = reduced_freq.takeOrdered(10, key = lambda x: -len(x[1]))
```

Third Approach: Shifting data around before reducing. (contd.)

```
freq_sorted
[(5182,
 [[118.52976392074571, 96.54201070629634, 3206.1214714999783],
 [77.45460061630328, 78.07530545083995, 1622.7858796279986],
 [118.64689982756818, 96.44132078278089, 3220.7666267843792],
 [118.59482658216085, 96.44941929769246, 3217.1068733712573],
 [111.8910509747487, 82.11533278823124, 583.0978357887793],
 [84.28463367457108, 103.23625952005631, 1980.5177436710767],
 [64.22176401688995, 114.73331621175502, 96.17267725692972],
 [81.26205876660073, 100.49647521225083, 3550.781584237549],
 [118.64551263852323, 96.49666570792286, 3209.784326499607],
 [118.70400080192714, 96.55672285267048, 3202.462710244625],
 [77.34895154230571, 77.89713080431697, 1616.348257398497],
 [118.56628035253055, 96.40234392252746, 3213.445682114889]]],
 (6973,
 [[72.50191364575532, 99.92189255143647, 2470.699753412948],
 [72.73381667361762, 99.83371460919332, 2442.003948596213],
 [72.39253672571188, 99.81561711859166, 2450.194875456176],
 [72.58351485796673, 99.75978902115273, 2446.099585222095],
 [72.55759922126946, 100.0294214129675, 2487.100038667084],
 [72.57307064374842, 99.85451048898713, 2511.701509335189],
 [72.51428256839976, 99.72397545952776, 2482.998216754375],
 [72.34625610801322, 100.0444800094792, 2466.598744544187],
 [72.13653068287205, 99.87661917278416, 2474.799710524056],
 [72.64006627072443, 99.78935257630796, 2478.899786929185],
 [72.40932961147622, 99.94220119655255, 2454.295945248991],
 [72.48122227881026, 99.84273302955303, 2507.600661605101]]],
 (1706,
 [[108.3443282919675, 100.84273369075228, 1895.8400028984754],
 [108.36077607279039, 100.8823430099223, 1934.2533854297046],
 [108.50283974674959, 100.79203616247979, 1957.3002332427275],
 [108.32695404645516, 100.92893729274489, 1903.5232083185826],
 [108.33282596888398, 100.68171905697531, 1972.6684081408775],
 [108.27555113633017, 100.84239054725518, 1941.9362817726542],
 [108.32603028445537, 100.98607041292506, 1949.6191849036916],
 [108.43945026646126, 100.80429909066194, 1911.20545694204],
 [108.41938821659426, 100.786398653622, 1980.3506962812596],
 [108.33984731223752, 100.88363740996039, 1918.888494813312],
 [108.32525890645205, 100.68360289978423, 1964.984398622469],
 [108.31696912684708, 100.69812697897636, 1926.571082107931]]],
 ...
 ...]
```

It's a little messy, but you can see that we have an interesting signal in the 6973 MHz band.

Third Approach: Shifting data around before reducing. (contd.)

#take a step back and redo a few steps with rounded values. Just to make it easier to see.

```
freq_key_value_rounded = freq_distinct.map(lambda x: ( x[0], [[round(x[1]),round(x[2]),round(x[3])]]) ) )
reduced_freq_key_value_rounded = freq_key_value_rounded.reduceByKey(lambda x,y: x+y)
freq_rounded_sorted = reduced_freq_key_value_rounded.takeOrdered(4, key = lambda x: -len(x[1]))
```

```
freq_rounded_sorted
[(5182,
 [[119, 97, 3206],
 [77, 78, 1623],
 [119, 96, 3221],
 [119, 96, 3217],
 [112, 82, 583],
 [84, 103, 1981],
 [64, 115, 96],
 [81, 100, 3551],
 [119, 96, 3210],
 [119, 97, 3202],
 [77, 78, 1616],
 [119, 96, 3213]]],
 (6973,
 [[73, 100, 2471],
 [73, 100, 2442],
 [72, 100, 2450],
 [73, 100, 2446],
 [73, 100, 2487],
 [73, 100, 2512],
 [73, 100, 2483],
 [72, 100, 2467],
 [72, 100, 2475],
 [73, 100, 2479],
 [72, 100, 2454],
 [72, 100, 2508]]],
 (1706,
 [[108, 101, 1896],
 [108, 101, 1934],
 [109, 101, 1957],
 [108, 101, 1904],
 [108, 101, 1973],
 [108, 101, 1942],
 [108, 101, 1950],
 [108, 101, 1911],
 [108, 101, 1980],
 [108, 101, 1919],
 [108, 101, 1965],
 [108, 101, 1927]]],
 (6972,
 [[73, 100, 2442],
 [72, 100, 2491],
 [73, 100, 2483],
 [73, 100, 2458],
 [73, 100, 2479],
 [72, 100, 2495],
 [72, 100, 2504],
 [73, 100, 2499],
 [72, 100, 2454],
 [72, 100, 2508],
 [73, 100, 2462]]])
```

Third Approach: Shifting data around before reducing. (contd.)

#Now that I have the answer, I'll just do a quick python `sort on the timebase` to clean it up.

```
data = freq_rounded_sorted[1][1]
```

```
data.sort(key = lambda x: x[2])
```

```
data
[[73, 100, 2442],
 [73, 100, 2446],
 [72, 100, 2450],
 [72, 100, 2454],
 [72, 100, 2467],
 [73, 100, 2471],
 [72, 100, 2475],
 [73, 100, 2479],
 [73, 100, 2483],
 [73, 100, 2487],
 [72, 100, 2508],
 [73, 100, 2512]]
```



Once again we find that coords 73,100 really do have a single source of frequency 6792 MHz. With a cycle time of ~4s.

Notice we are missing a few points from our other analysis. If we go back and look and the floating point values, they fell just outside our 1 STD. We could relax that in our shift function.

Creating a Reusable Solution

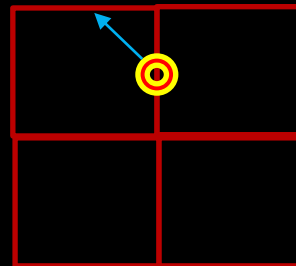
We did this in an exploratory mode, which is a wonderful capability of Spark. You can play around with real datasets, not just toy problems.

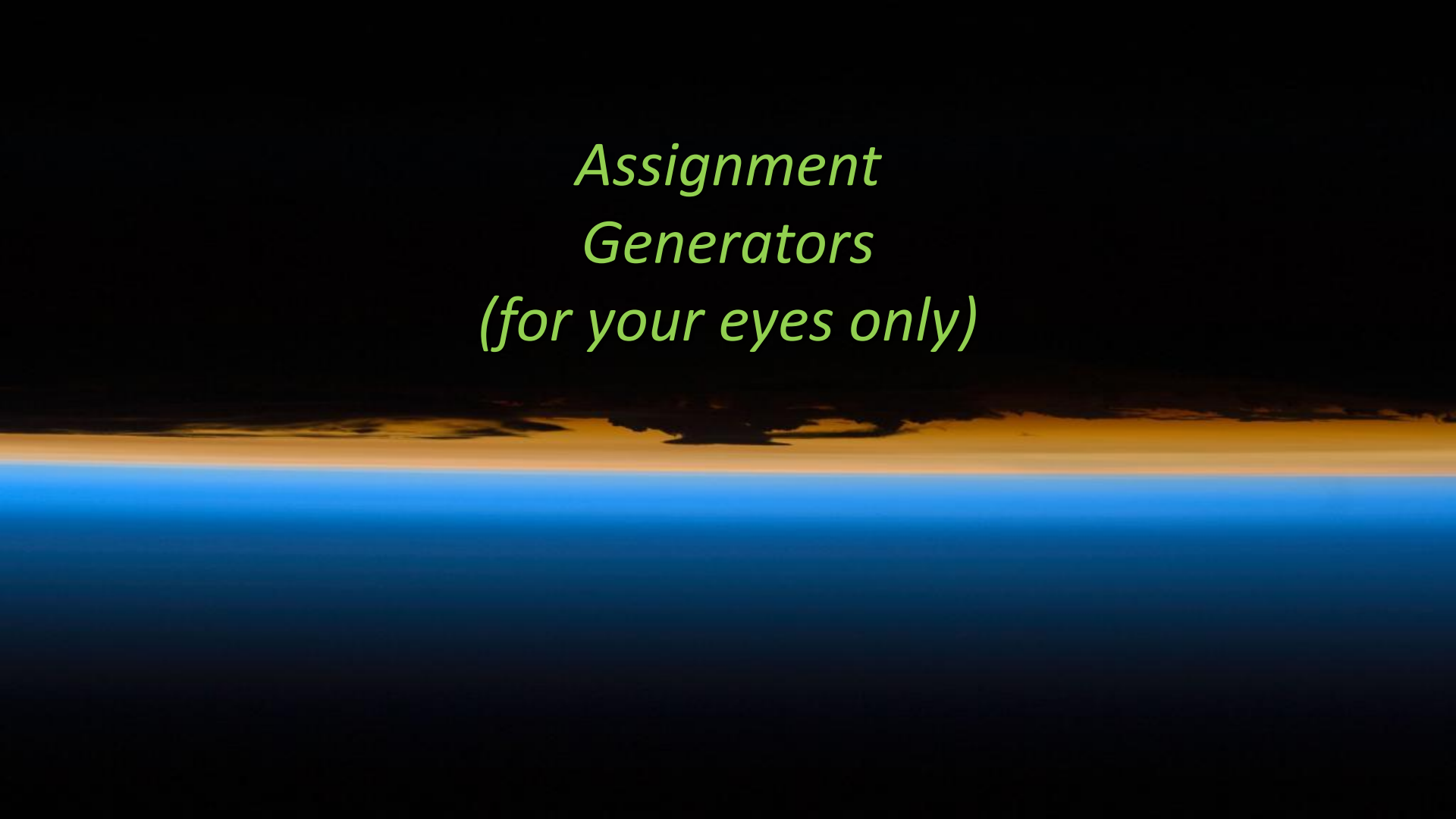
However we might want to pull these ideas together and create a rigorous code. For our spatial approach, we can use the same shift technique, but now in two dimensions. If we think about the worst case scenario in which our target cluster is centered on a border, we would realize that no single shift is assured to not land on another border. We would actually need two slides just to be sure.

We could also make the (rounded) frequency part of the key to deal with distinguishing frequencies at the same time. We would also need to do a similar bin shift trick there as well.

If you are worried about how well this scales, especially as we consider we may want more boxes as our resolution goes up, realize that this technique scales proportionate to the number of sources, not the number of bins of the size of the sky.

And of course there are alternate techniques as well.



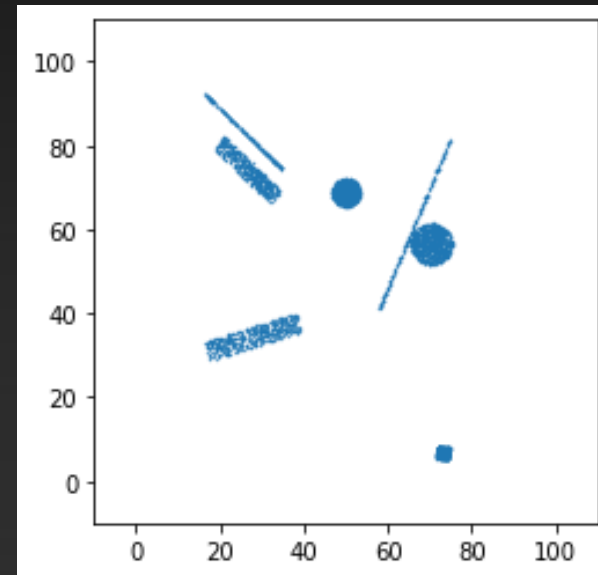
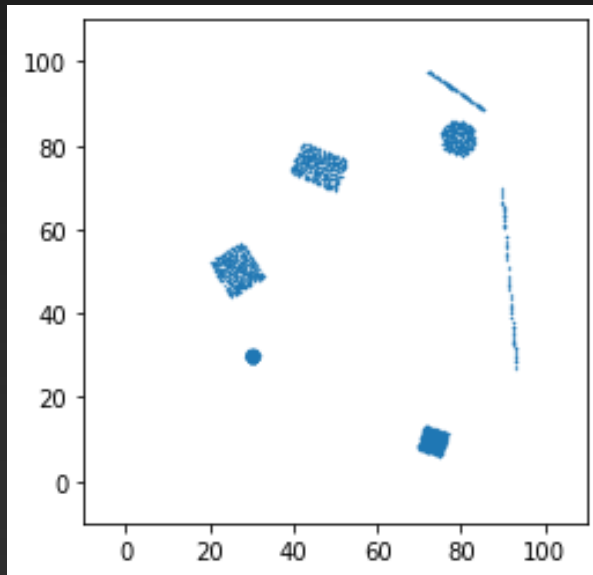


*Assignment
Generators
(for your eyes only)*

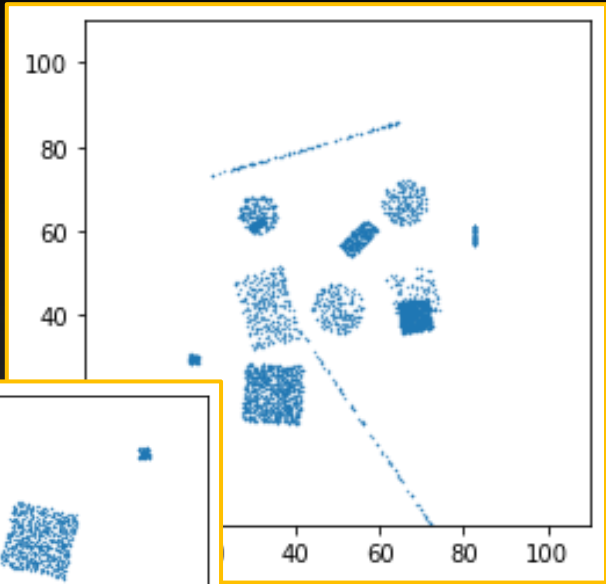
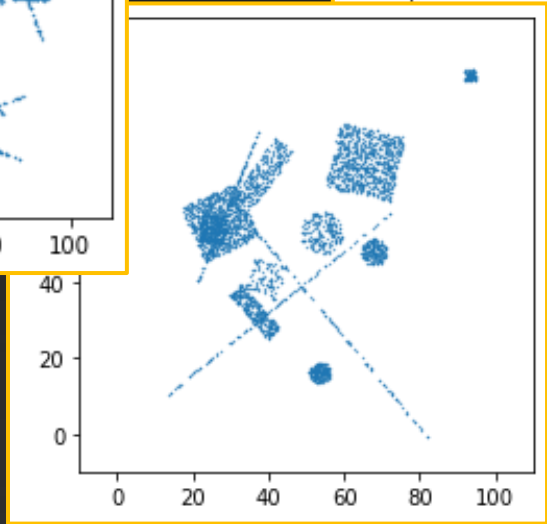
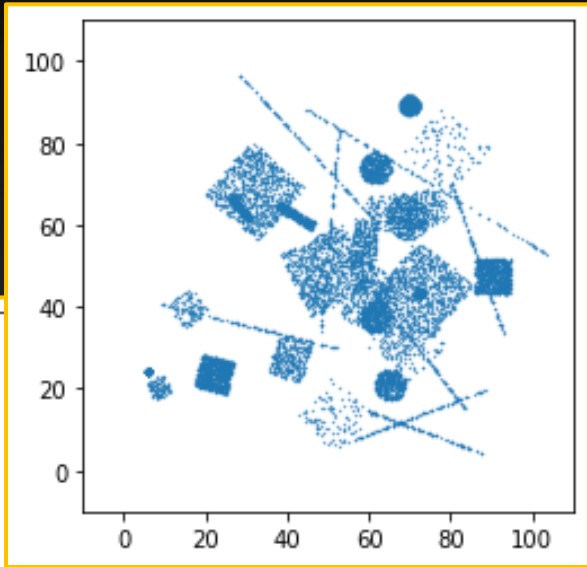
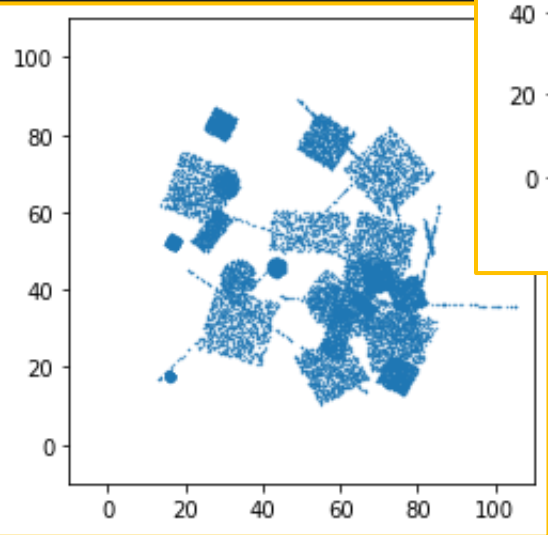
Assignment Dataset Generators

- We provide the tools for you to generate your own problem sets.
- This avoids cheating (to some extent).
- And allow you to choose difficulty.

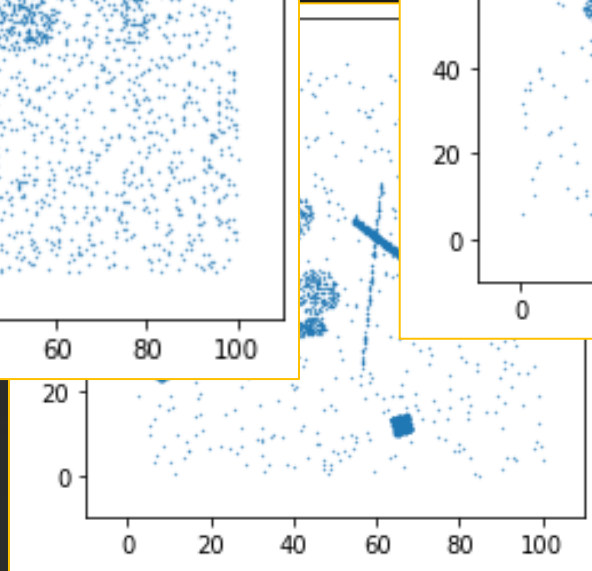
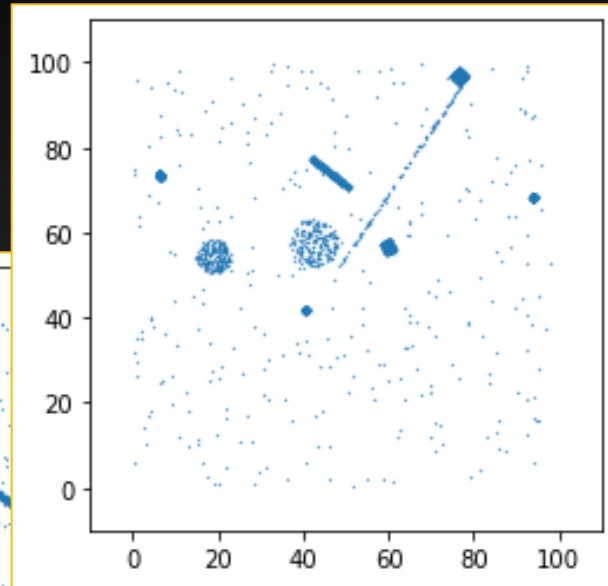
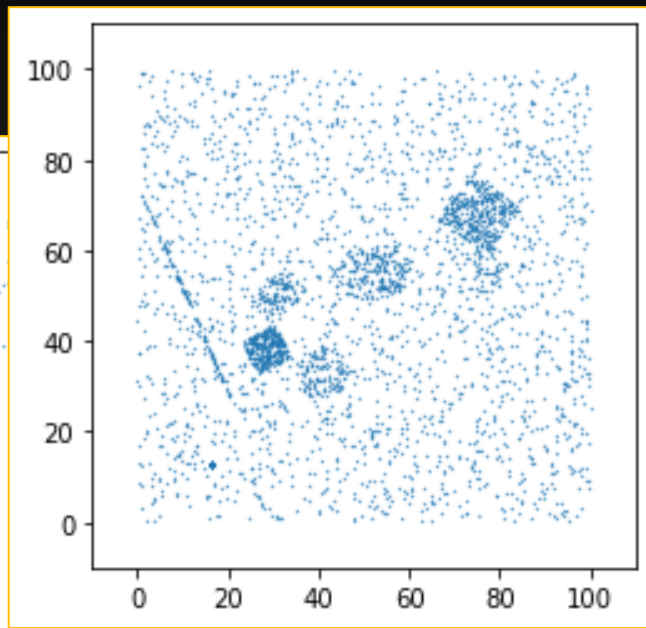
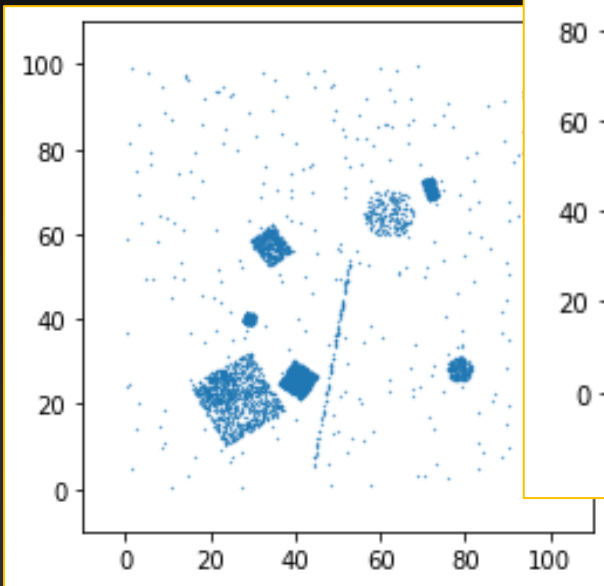
Here are some example distributions. They may be simple and few:



Or they may be more complex:



They might even have noise:





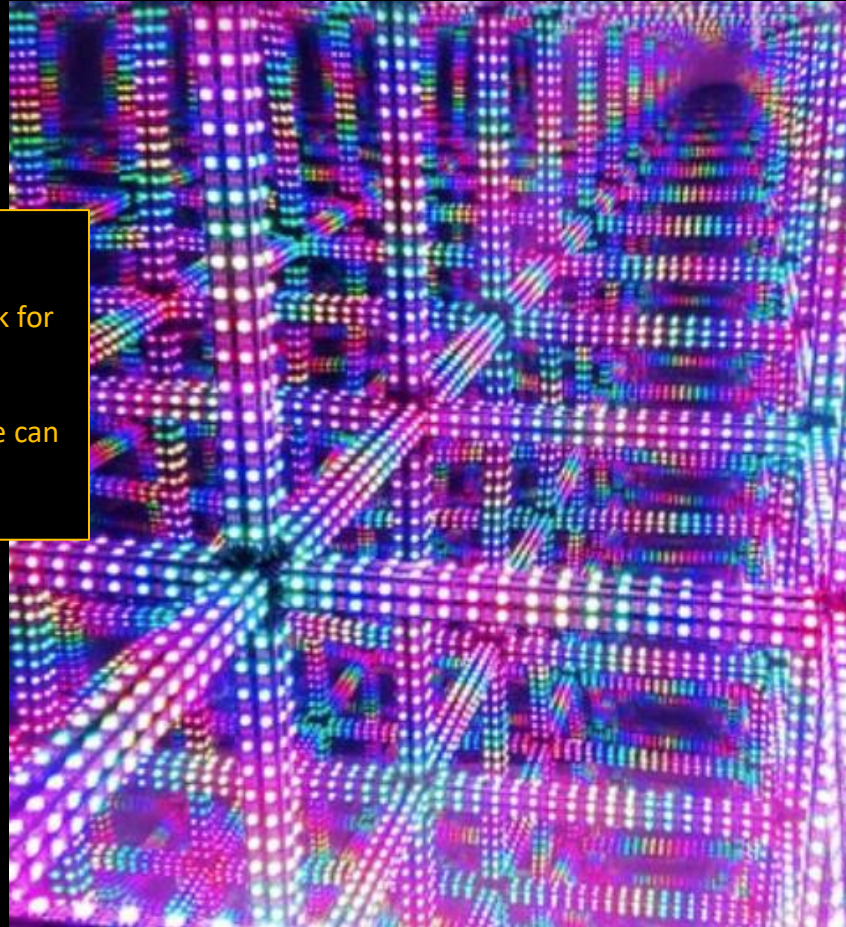
***Advanced
Data Analysis
Assignment Review***

The Big Reveal: What was in our data?

Oh yeah, it's 6 dimensional.

We are going to have to work for it.

Let's dive in and see what we can peel apart.



First dive: How many "things" are there?

```
input_rdd = sc.textFile('Assignment2.dat').map(lambda x:x.split(',')).map(lambda x:
[float(x[0]),float(x[1]),float(x[2]),float(x[3]),float(x[4]),float(x[5])])
```

```
input_rdd.take(3)
[[57.47974084562296,
  55.44945810334027,
  54.241594907920344,
  52.41360437021964,
  54.93100799212413,
  51.78432087598154],
 [70.46721670551902,
  71.53278329448098,
  ...
```

```
from pyspark.mllib.clustering import KMeans
```

```
for clusters in range(1,10):
    model = KMeans.train(input_rdd, clusters)
    print (model.computeCost(input_rdd))
```

```
22813322.102508288
8222110.121083168
1953379.9003032786
1345930.143051648
248637.68427582923
240944.69344685183
238758.14649268807
152969.2321060632
137938.55840308202
```


Let's verify repeatability/stability

```
for clusters in range(1,10):  
    model = KMeans.train(input_rdd, clusters)  
    print (model.computeCost(input_rdd))
```

```
22813322.102508288  
8222110.121083168  
1953379.9003032786  
856087.4415274587  
248637.68427582923  
174234.43071089557  
166391.823094679  
143271.65116005158  
157528.57192476804
```

What can we say about these five objects?

```
model = KMeans.train(input_rdd, 5)
centers = model.clusterCenters
print(centers)
[array([29.8592193 , 30.00769653, 30.02190367, 80.00416938, 79.97828141, 80.02146926]),
 array([39.99730091, 39.99272142, 40.00456747, 14.99162554, 15.00305203, 15.02544602]),
 array([70.99050247, 71.00221983, 71.00221983, 71.00221983, 71.00221983, 71.00221983]),
 array([54.97439287, 55.02510453, 54.97981585, 54.96350913, 55.04625848, 55.00612376]),
 array([23.95388832, 23.95388832, 23.95388832, 23.95388832, 23.95388832, 23.95388832])]
```

```
pred = model.predict(input_rdd)
clusters = input_rdd.zip(pred)
```

```
clusters.take(3)
[[[57.47974084562296,
  55.44945810334027,
  54.241594907920344,
  52.41360437021964,
  54.93100799212413,
  51.78432087598154],
 3),
 ([70.46721670551902,
  71.53278329448098,
  71.53278329448098,
  71.53278329448098,
  71.53278329448098,
  71.53278329448098],
 2),
 ([54.26327405498564,
  58.49493372842324,
  54.75498592333371,
  55.703180938852945,
  52.83741473654931,
  51.4028551634724],
 3)]
```

These coordinates can't be a coincidence. Once again, what an awfully kind problem creator!

PCA might te

Why DataFrames?

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import PCA

# For each of the labeled clusters: get a count, center cluster and do
for cluster_num in range(5):
    cluster = clusters.filter(lambda x: x[1]==cluster_num).map(lambda
    print("Cluster ",cluster_num," has count ",cluster.count())
    center = [float(x) for x in centers[cluster_num]]
    centered_cluster = cluster.map(lambda x: [x[0]-center[0],x[1]-cent
    df = spark.createDataFrame(centered_cluster)
    assembler = VectorAssembler(inputCols=['_1', '_2','_3','_4','_5','
    ndf = assembler.transform(df)
    pca = PCA(k=6, inputCol="feat_vec", outputCol="pca_features")
    model = pca.fit(ndf)
    print("Cluster: ",cluster_num," explainedVariance: ", model.expla
```

Many of you seem to prefer DataFrames. The DataFrame PCA offered me some useful functionality that I would have had to write myself for the RDD version of PCA, notably the explained variance.

As DataFrame APIs are picky about datatypes, I just went with the VectorAssembler helper routine to create my inputs.

We just include all our columns here, but you should know about this for when you are constructing more selective input features in your own complex applications.

```
Cluster 0 has count 3000
Cluster: 0 explainedVariance: [0.669525536882, 0.1662797338236, 0.16419473782323, 2.45886121026e-16, 5.5166246758226e-17, 2.202681855118e-17]
Cluster 1 has count 1500
Cluster: 1 explainedVariance: [0.517083515243, 0.4829164846848, 1.1240263947e-16, 7.59536857127e-17, 2.2841893262568e-17, 3.756984397667e-18]
Cluster 2 has count 1000
Cluster: 2 explainedVariance: [0.833387789374, 0.1666122103404, 2.4712754937e-32, 1.82764934226e-63, 3.0036689526738e-95, 0.0]
Cluster 3 has count 2500
Cluster: 3 explainedVariance: [0.177078608271, 0.1757344315469, 0.17084112066742, 0.163726169637072, 0.16048767756223278, 0.1521359974611792]
Cluster 4 has count 1000
Cluster: 4 explainedVariance: [0.999999999999, 5.788192129e-17, 2.7961406066e-48, 2.06790561697e-79, 3.398520641703e-111, 0.0]
```

These cluster sizes are a dead giveaway that you are on the right track.

What do we know so far?

```
Cluster 0 has count 3000
Cluster: 0 explainedVariance: [0.669525536882, 0.1662797338236, 0.16419473782323, 2.45886121026e-16, 5.5166246758226e-17, 2.202681855118e-17]
Cluster 1 has count 1500
Cluster: 1 explainedVariance: [0.517083515243, 0.4829164846848, 1.1240263947e-16, 7.59536857127e-17, 2.2841893262568e-17, 3.756984397667e-18]
Cluster 2 has count 1000
Cluster: 2 explainedVariance: [0.833387789374, 0.1666122103404, 2.4712754937e-32, 1.82764934226e-63, 3.0036689526738e-95, 0.0]
Cluster 3 has count 2500
Cluster: 3 explainedVariance: [0.177078608271, 0.1757344315469, 0.17084112066742, 0.163726169637072, 0.16048767756223278, 0.1521359974611792]
Cluster 4 has count 1000
Cluster: 4 explainedVariance: [0.999999999999, 5.788192129e-17, 2.7961406066e-48, 2.06790561697e-79, 3.398520641703e-111, 0.0]
```

Looks like cluster 0 is 3000 points in a 3 dimensional shape, with 2 very similar dimensions and 1 longer.

Looks like cluster 1 is 1500 points in a 2 dimensional shape or roughly equal extents (ball, cube, dodecahedron?).

Looks like cluster 2 is 1000 points also in a 2 dimensional shape, but of seemingly unequal extents (rectangle? The letter "I"? An ellipsoid?).

Looks like cluster 3 is 2500 points in a 6 dimensional shape with fairly equal extents.

Looks like cluster 4 is 1000 points in a very, very straight line.

Easiest first: What is there to say about a line?

```
cluster_num = 4
cluster = clusters.filter(lambda x: x[1]==cluster_num).map(lambda x: x[0])
start0 = cluster.map(lambda x: x[0]).reduce(lambda x,y: min(x,y))
print(start0)
end0 = cluster.map(lambda x: x[0]).reduce(lambda x,y: max(x,y))
print(end0)
20.001463417223093      27.995299694613074
```

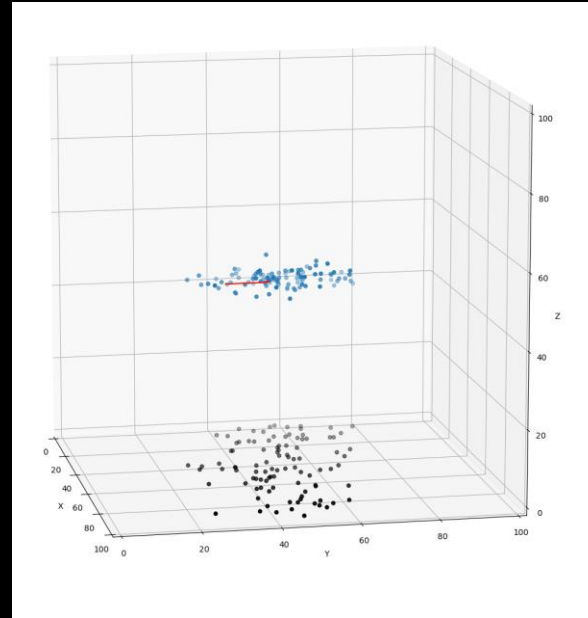
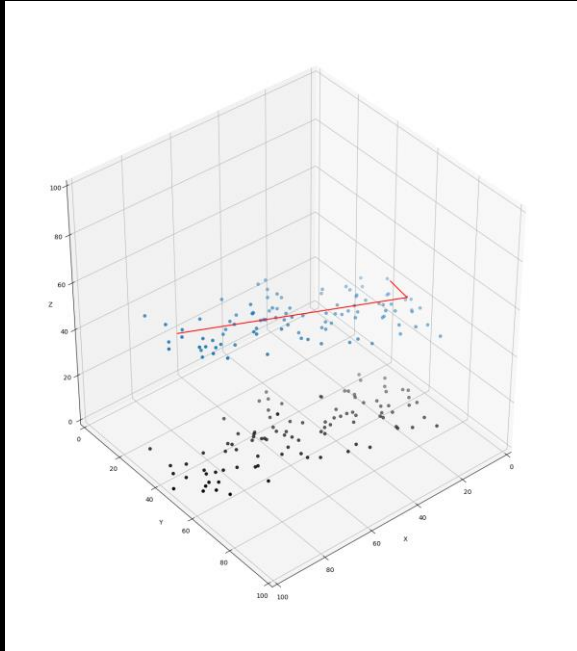
#should do this for all 6 coords...

```
start5 = cluster.map(lambda x: x[5]).reduce(lambda x,y: min(x,y))
print(start5)
end5 = cluster.map(lambda x: x[5]).reduce(lambda x,y: max(x,y))
print(end5)
20.001463417223093      27.995299694613074
```

It appears we have a line that goes from 20,20,20,20,20,20 to 28,28,28,28,28,28

Again, John is playing nice. The round numbers for point counts and coordinates are unlikely to be accidents.

PCA vectors tell us 3 things...

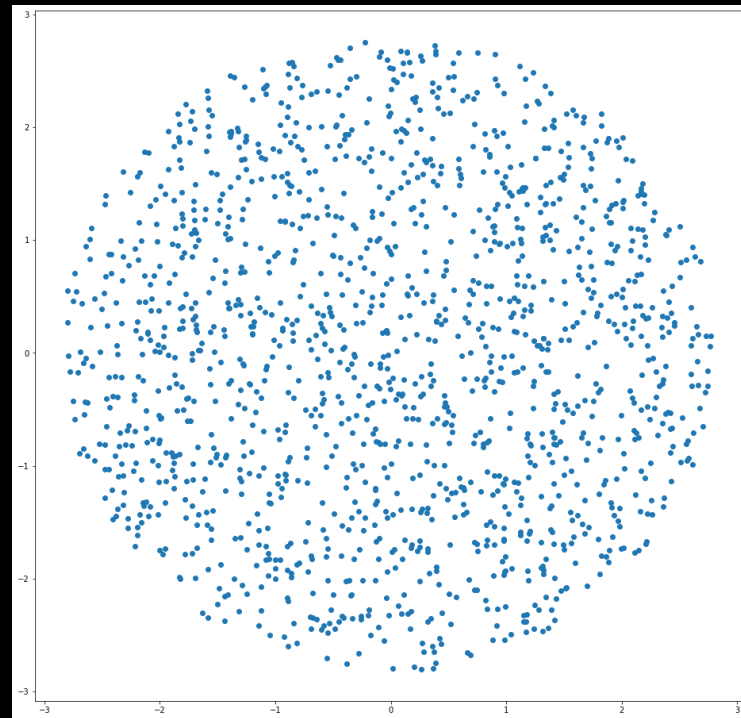


- The significance of each dimension
- The orientation of the data
- The transformation to the lower dimensional space

How about those 2-dimensional shapes?

Why think? Just plot!

```
cluster_num = 1
cluster = clusters.filter(lambda x: x[1]==cluster_num).map(lambda x: x[0])
center = [float(x) for x in centers[cluster_num]]
centered_cluster = cluster.map(lambda x: [x[0]-center[0],x[1]-center[1],x[2]-center[2],x[3]-center[3],x[4]-center[4],x[5]-center[5]])
df = spark.createDataFrame(centered_cluster)
assembler = VectorAssembler(inputCols=['_1', '_2', '_3', '_4', '_5', '_6'], outputCol='feat_vec')
ndf = assembler.transform(df)
pca = PCA(k=2, inputCol="feat_vec", outputCol="pca_features")
model = pca.fit(ndf)
x = model.transform(ndf).select('pca_features')
y = x.rdd.map(lambda x: [x[0][0],x[0][1]])
z = y.map(lambda x: str(x[0])+', '+str(x[1]))
z = z.repartition(1)
z.saveAsTextFile('2D data')
```



Just keeping the transformed 2D data and formatting output (boring).

```
x = x.select('pca_features')
```

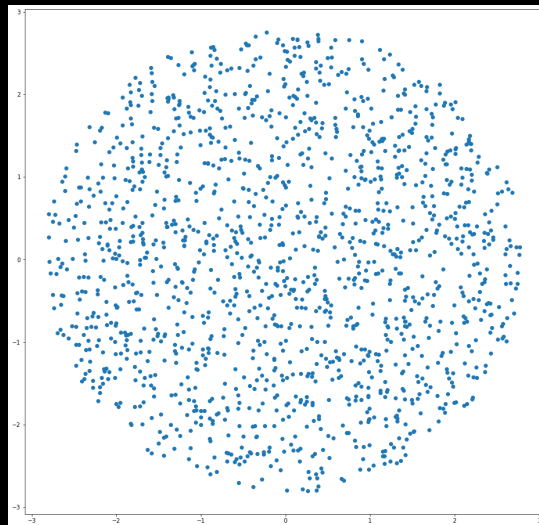
```
x.show(truncate=False)
```

```
+-----+
|pca_features|
+-----+
|[-11.442801230730225, -2.2456486179091795]|
|[8.199412704320977, -4.758034893331184]|
|[-10.244417347787193, 0.5396575347829049]|
|[-11.869736554573121, -1.4845929620214862]|
|[-3.5546269239632644, -4.233601618163395]|
|[2.7393561531172796, 0.6760682876532856]|
|[13.30478751297807, -0.9023265952197561]|
|[-5.988746172900853, -4.015718103211636]|
|[-14.53534496511185, -4.0009984448800555]|
|[4.837789089973879, 4.04799491602994]|
|[8.897186913650424, -2.0837450631466874]|
|[-1.708504262340559, 0.03667661547683404]|
|[-1.1697623781406488, 3.40020825795184]|
|[-1.2026596484091905, -3.8974522565315635]|
|[-3.671529907044845, -4.137095253338099]|
|[-9.267303730165965, 4.855142027927819]|
|[-5.052591326725931, 3.5269543649986637]|
|[12.936260914202773, -2.2082924731603573]|
|[-11.598584911100225, -0.6754460111338405]|
|[7.4124164286566945, -1.0304226290413516]|
+-----+
```

I'm hauling around a lot of unnecessary data in these dataframes, just for illustrative purposes and debugging. I might want to trim that out for efficiency to scale this up.

```
y = x.rdd.map(lambda x: [x[0][0], x[0][1]])
z = y.map(lambda x: str(x[0]) + ', ' + str(x[1]))
z = z.repartition(1)
z.saveAsTextFile('2D.dat')
```


Summary for second object



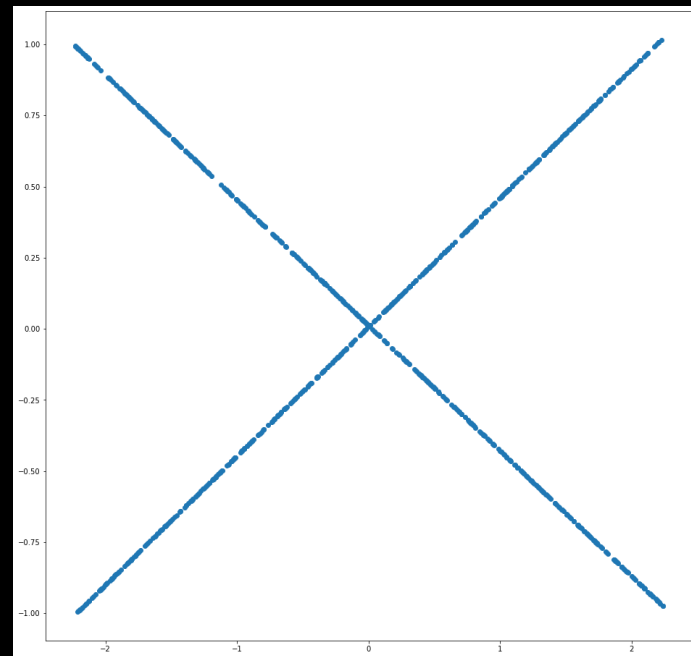
Looks like a circle to me - if you pay attention to the axis scale, that is. So, putting it together, we have:

- 1500 points
- Circle
- Radius about 3
- Centered at 40,40,40,15,15,15

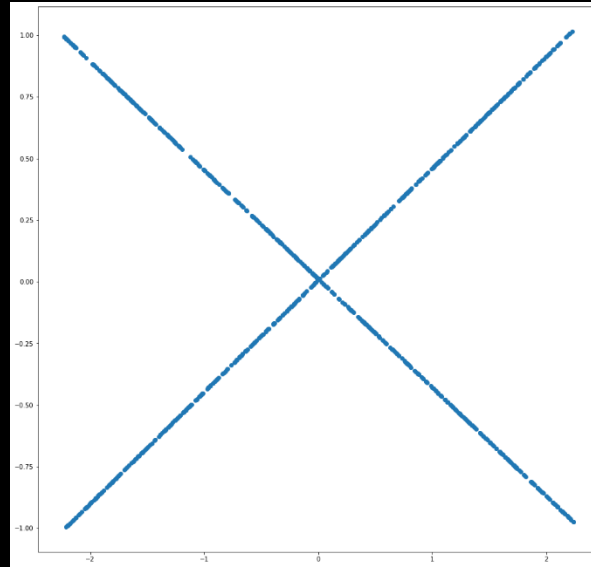
we could even get the orientation from the PCA vectors (but plotting 6D rotation is tough)

Same for other 2 dimensional shape

```
cluster_num = 2
cluster = clusters.filter(lambda x: x[1]==cluster_num).map(lambda x: x[0])
center = [float(x) for x in centers[cluster_num]]
centered_cluster = cluster.map(lambda x: [x[0]-center[0],x[1]-center[1],x[2]-center[2],x[3]-center[3],x[4]-center[4],x[5]-center[5]])
df = spark.createDataFrame(centered_cluster)
assembler = VectorAssembler(inputCols=['_1', '_2', '_3', '_4', '_5', '_6'], outputCol='feat_vec')
ndf = assembler.transform(df)
pca = PCA(k=2, inputCol="feat_vec", outputCol="pca_features")
model = pca.fit(ndf)
x = model.transform(ndf).select('pca_features')
y = x.rdd.map(lambda x: [x[0][0],x[0][1]])
z = y.map(lambda x: str(x[0])+', '+str(x[1]))
z = z.repartition(1)
z.saveAsTextFile('2D data')
```



Summary for other 2D object



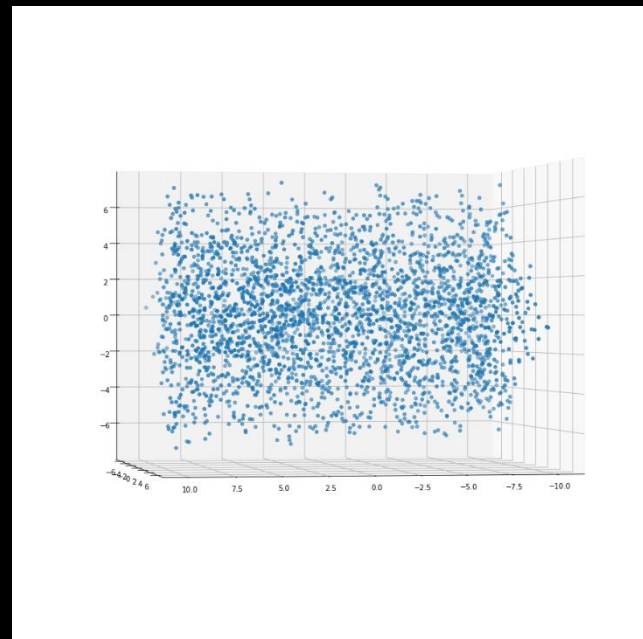
Looks like a cross to me. So, putting it together, we have:

- 1000 points
- X Shape or cross
- Size of $\sim 2 \times 4$ (I'm not picky)
- Centered at 71,71,71,71,71,71

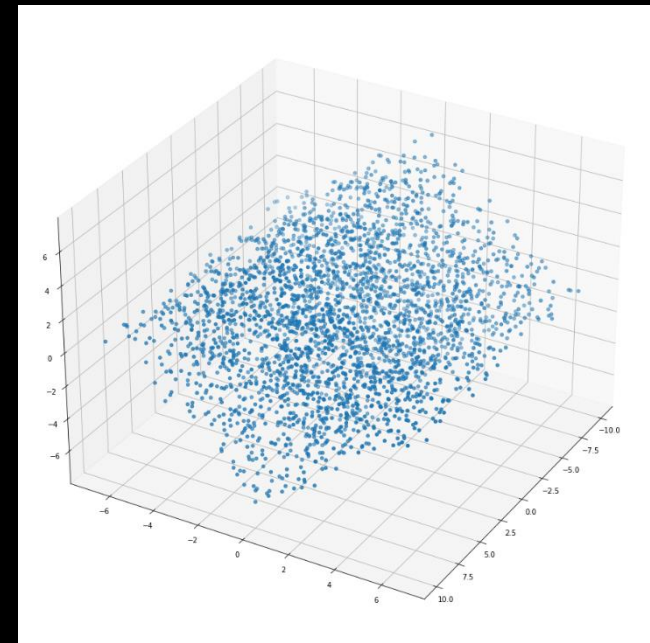
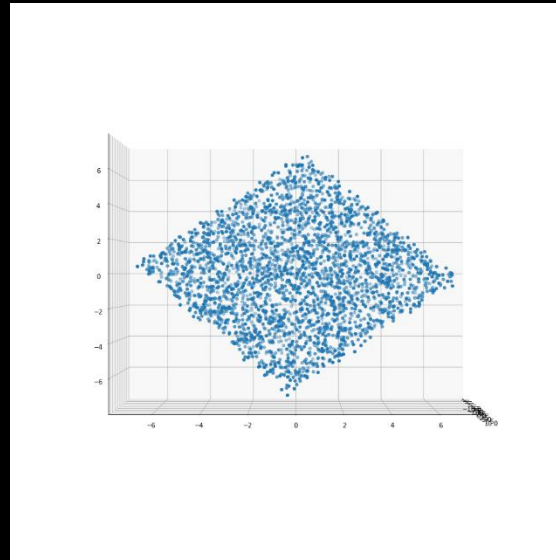
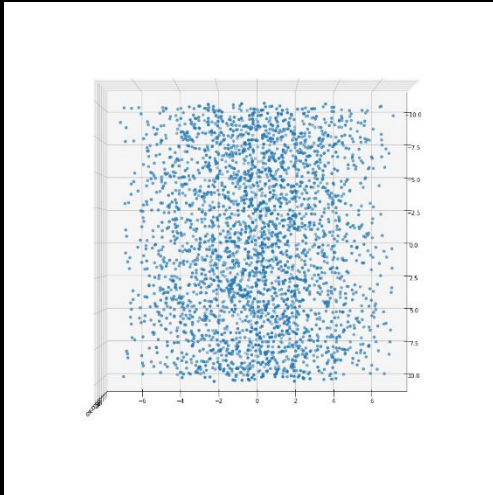
we could again get the orientation from the PCA vectors (but plotting 6D rotation is still tough)

Let's try the same with the 3D cluster.

```
cluster_num = 0
cluster = clusters.filter(lambda x: x[1]==cluster_num).map(lambda x: x[0])
center = [float(x) for x in centers[cluster_num]]
centered_cluster = cluster.map(lambda x: [x[0]-center[0],x[1]-center[1],x[2]-center[2],x[3]-center[3],x[4]-center[4],x[5]-center[5]] )
df = spark.createDataFrame(centered_cluster)
assembler = VectorAssembler(inputCols=['_1', '_2', '_3', '_4', '_5', '_6'], outputCol='feat_vec')
ndf = assembler.transform(df)
pca = PCA(k=3, inputCol="feat_vec", outputCol="pca_features")
model = pca.fit(ndf)
x = model.transform(ndf).select('pca_features')
y = x.rdd.map(lambda x: [x[0][0],x[0][1],x[0][2]])
z = y.map(lambda x: str(x[0])+', '+str(x[1])+', '+str(x[2]))
z = z.repartition(1)
z.saveAsTextFile('3D data.dat')
```



Our 3D Object



This is a rectangular box. I rotated the perspective a bit to convince myself

- 3000 points
- Rectangular box
- About 10x10x20
- Centered at 30, 30, 30, 80, 80, 80

Once again, the orientation (from PCA) tells us its angle in 6D.

```
import csv
My matplotlib

points = []
with open('3d.dat', newline='') as csvfile:
    spamreader = csv.reader(csvfile)
    for row in spamreader:
        points.append(( float(row[0]),float(row[1]),float(row[2]) ))
        print(row)

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(16,16))
ax = fig.add_subplot(projection='3d')
ax.scatter([x[0] for x in points], [x[1] for x in points], [x[2] for x in points])
ax.view_init(elev=0,azim=80)
```

How about that 6-dimensional cluster?

We can't visualize. We can note that the explained variance here

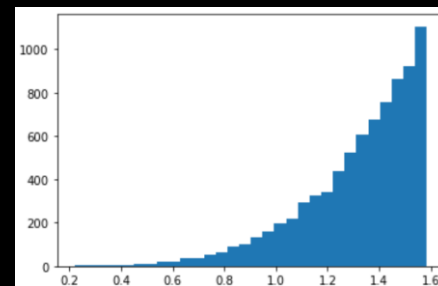
0.177 0.175 0.170 0.163 0.160 0.152

suggests a fairly symmetric shape.

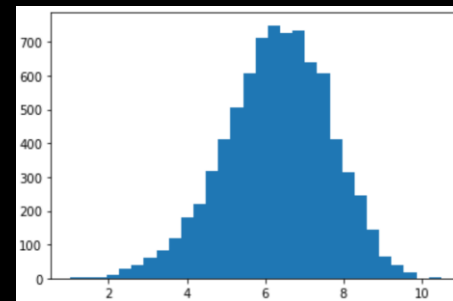
So maybe it is a ball. But could it be a polyhedron? Or maybe a hollow 6D sphere or a gaussian distribution? Or maybe the "curse of dimensionality" has this effect on variance.

I did a histogram of point distances from the cluster center. We might hope that it would reveal the shape. In this case we would find it mostly follows a 5th power curve (which corresponds to a 6D ball), but it does not cutoff as we would expect for a ball. It has a "tail". That does eventually cut off abruptly.

We could develop a few curves for common regular polygons. We would find that this one corresponds to a cube. And the last point gives us the half diagonal length.



6D Ball



6D Cube

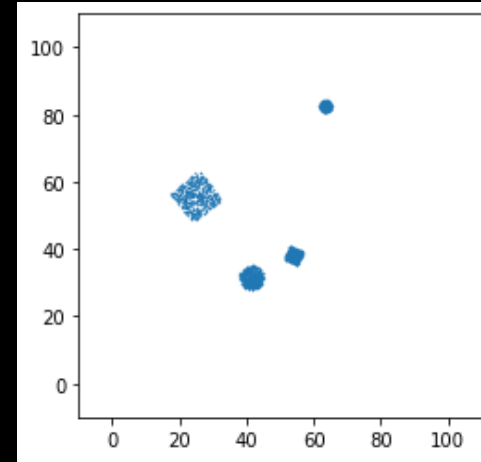
I don't expect that any of you went that far! If you guessed:

- 2500 points
- "Roundish" shape
- Size (side) of ~7
- Centered at 55,55,55,55,55,55

Then congratulations. I think you did very well.

A quick discussion about noise and outliers.

KMeans (and most clustering approaches) have no notion of outliers. Every point gets assigned to a cluster. This introduces issues when faced with noise. Here is a simple dataset.

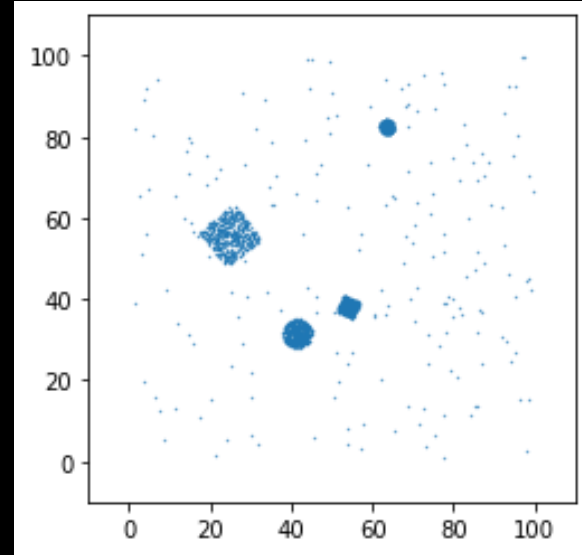


KMeans does well and gives us a sensible clustering.

Cost	Clusters
1212077	(45 , 52)
384888	(63 , 82) (40 , 41)
65392	(47 , 34) (24 , 55) (63 , 82)
15430	(24 , 55) (63 , 82) (53 , 38) (41 , 31)
11707	(53 , 38) (22 , 57) (63 , 82) (41 , 31) (26 , 54)
9401	(25 , 59) (63 , 82) (53 , 38) (27 , 53) (41 , 31) (21 , 55)
8313	(63 , 82) (53 , 38) (23 , 58) (40 , 30) (22 , 53) (28 , 55) (42 , 32)
7412	(22 , 53) (63 , 82) (53 , 38) (39 , 30) (23 , 58) (41 , 33) (28 , 55) (43 , 30)
7081	(63 , 82) (53 , 38) (28 , 56) (39 , 32) (24 , 51) (22 , 57) (43 , 30) (40 , 29) (41 , 33)

A little noise.

Add in a little noise (10%) and it is a little less obvious where our cost "elbow" is. But perhaps we are still OK.

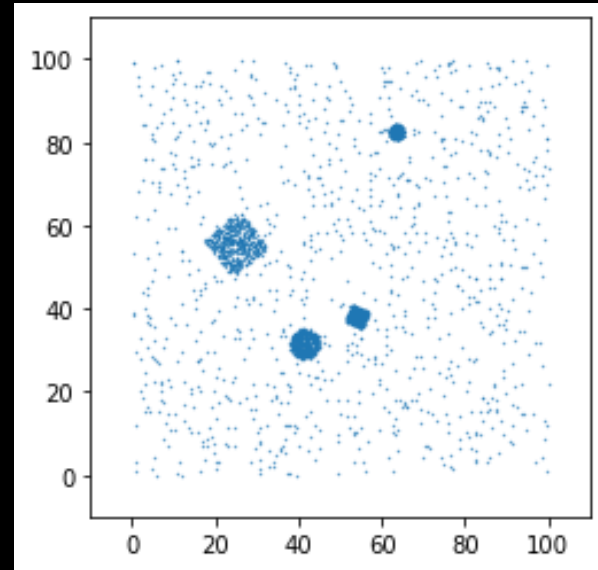


The cluster centers are still reasonable too.

Cost	Clusters
1542624	(46 , 51)
616768	(64 , 81) (40 , 41)
224474	(24 , 56) (48 , 34) (64 , 81)
145791	(40 , 30) (64 , 82) (24 , 56) (56 , 37)
104353	(24 , 56) (64 , 82) (54 , 38) (83 , 32) (40 , 30)
89475	(40 , 30) (63 , 82) (84 , 55) (24 , 56) (54 , 38) (79 , 17)
77963	(24 , 55) (63 , 82) (54 , 38) (40 , 30) (16 , 78) (84 , 55) (78 , 17)
68325	(41 , 31) (63 , 82) (21 , 60) (54 , 38) (25 , 54) (85 , 70) (17 , 16) (83 , 26)
63991	(54 , 38) (63 , 82) (24 , 56) (83 , 40) (17 , 10) (18 , 34) (41 , 31) (74 , 11) (87 , 75)

A little more noise.

If we increase the noise to 33% it becomes hard to be confident about the cluster count. My best guess might be 3...



And the cluster centers are no longer reliable.

Cost	Clusters
2911714	(47 , 51)
1515577	(62 , 80) (40 , 38)
844979	(50 , 31) (66 , 80) (22 , 57)
638983	(45 , 32) (65 , 81) (22 , 58) (81 , 27)
492621	(23 , 60) (65 , 81) (19 , 17) (48 , 34) (81 , 26)
398572	(81 , 26) (66 , 81) (24 , 55) (19 , 85) (20 , 16) (48 , 34)
438669	(45 , 32) (79 , 17) (22 , 52) (65 , 83) (18 , 85) (53 , 62) (86 , 58)
301994	(43 , 11) (62 , 81) (13 , 20) (48 , 35) (87 , 73) (81 , 24) (24 , 55) (18 , 85)
289378	(23 , 60) (61 , 82) (41 , 31) (10 , 29) (82 , 49) (27 , 9) (54 , 38) (88 , 84) (78 , 14)

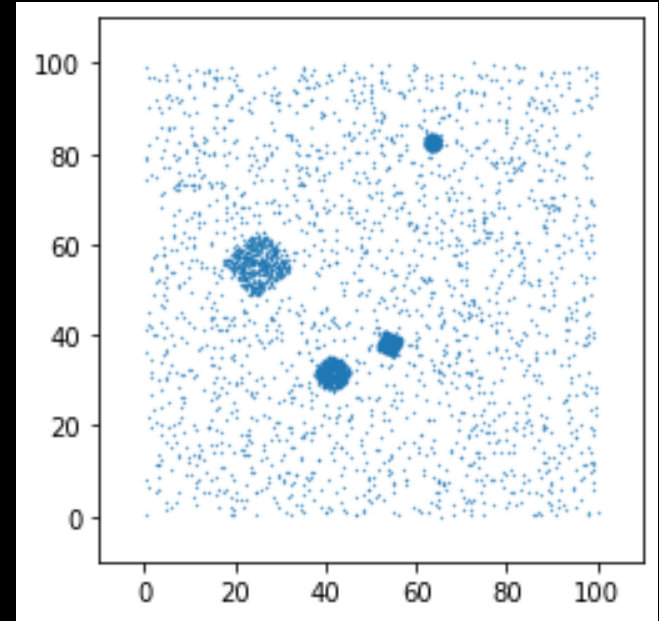
Significant noise.

As the S/N ratio hits 1, it becomes very hard to find any elbow.

The real world often involves noise. Be aware of this limitation.

If there was an easy fix ("import super-k-means"), we would be using it. There are solutions, such as algorithms where you specify the S/N, but no magic bullets.

Even if we cheat and look at 4 clusters, the cluster centers are problematic.



Cost	Clusters
4532592	(48 , 50)
2646052	(42 , 36) (60 , 79)
1553843	(21 , 58) (52 , 29) (67 , 79)
1122661	(22 , 61) (66 , 80) (80 , 26) (43 , 31)
981406	(79 , 54) (22 , 61) (43 , 30) (63 , 83) (79 , 15)
741845	(22 , 62) (80 , 18) (48 , 35) (60 , 82) (20 , 17) (84 , 67)
811270	(63 , 31) (22 , 54) (87 , 67) (35 , 26) (53 , 60) (20 , 84) (63 , 84)
586240	(22 , 54) (86 , 68) (55 , 39) (64 , 82) (12 , 83) (36 , 25) (81 , 17) (40 , 81)
462873	(48 , 35) (15 , 19) (47 , 10) (61 , 82) (83 , 15) (20 , 84) (23 , 55) (87 , 81) (78 , 49)