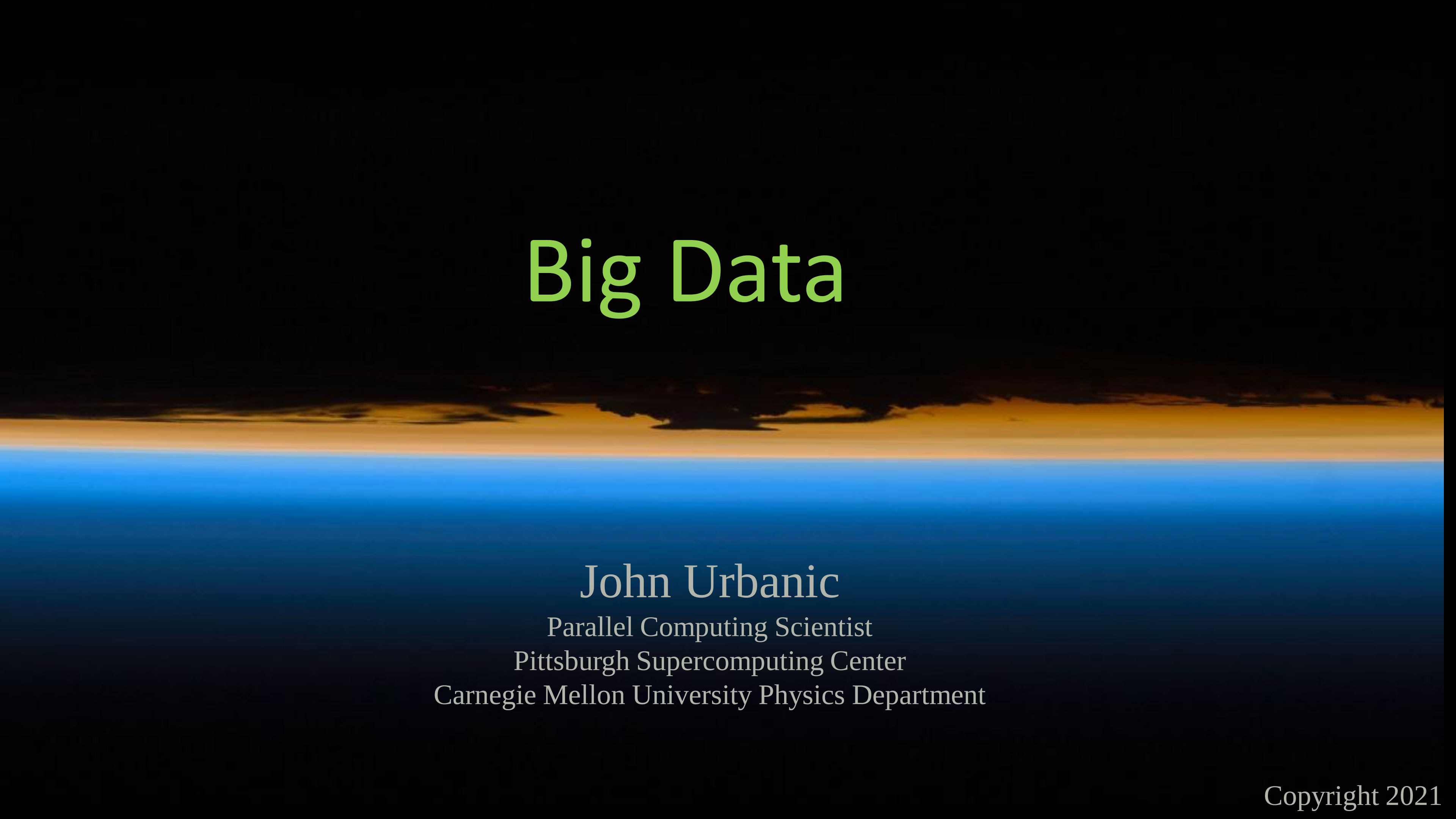


Big Data

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center
Carnegie Mellon University Physics Department

Preliminary Exercise

Let’s get the boring stuff out of the way now.

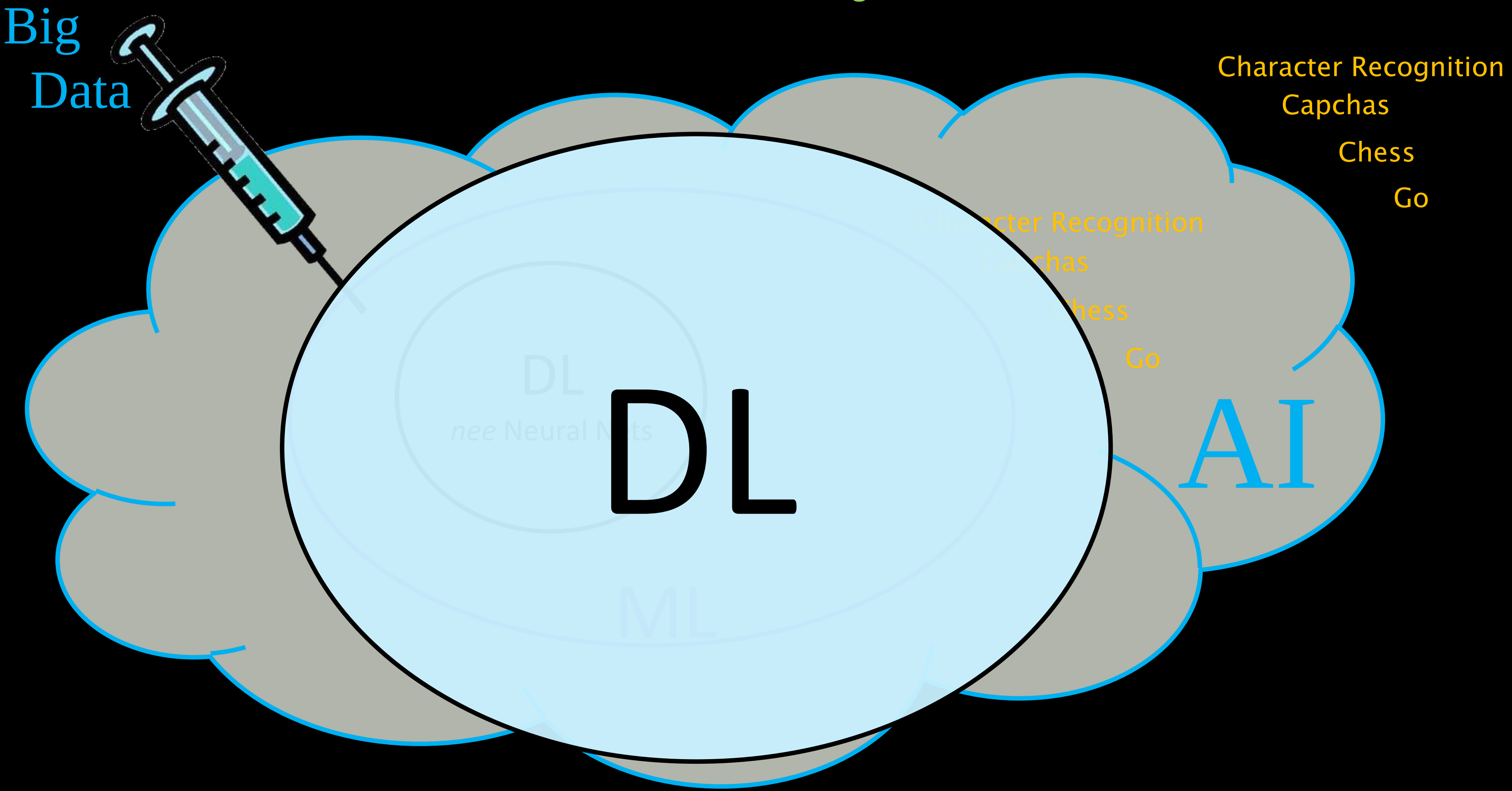
- Run the Big Data setup script:

```
~training/Setup
```

- Logout and then login again, so that the proper modules are in effect.
- Start an interactive session.

```
interact
```

How does all this fit together?



> 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 *

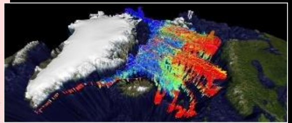
Pan-STARRS

Genome sequencers

(Wikipedia Commons)

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

Wikipedia Commons



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

**Actually, a silly estimate. The original refere[illegible]ntions a more accurate 208TB, and 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

- GDELT (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.”*

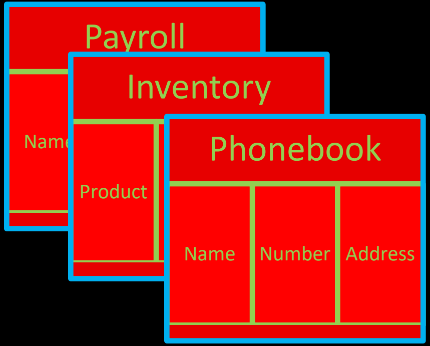
3 V's of Big Data

- Volume
- Velocity
- Variety

Good Ol’ SQL couldn't keep up.

MySQL PostgreSQL Oracle SQLite

```
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.

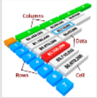
Document

```
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	<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> .
6	. JSON
9	XML
0	Binary JSON
bar	JSON XML
12	XML XML

Wide Column Stores

Cassandra Google BigTable APACHE HBASE

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

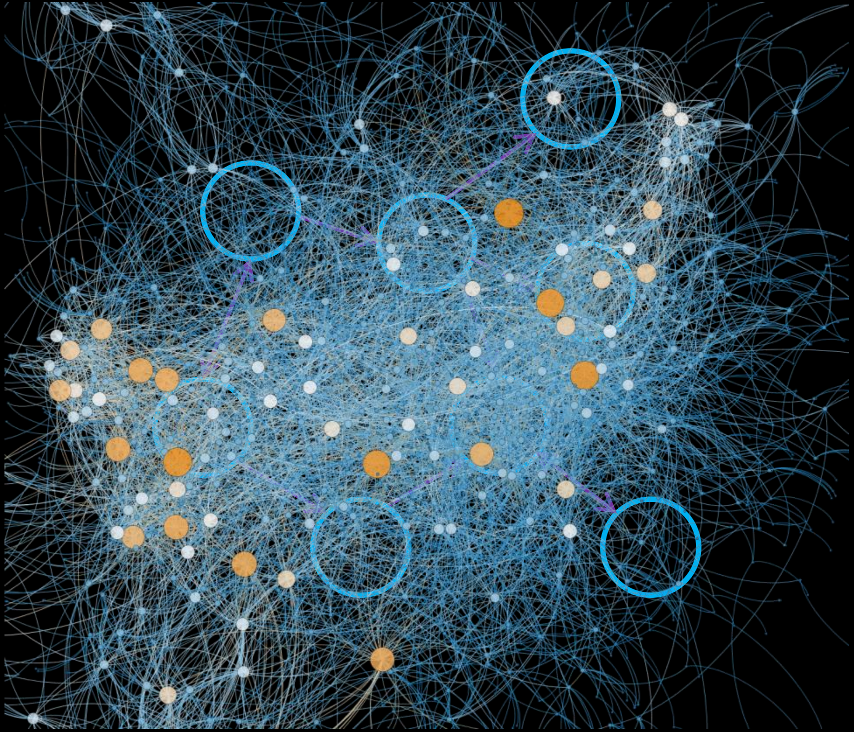
- No predefined schema
- Can think of this as a 2-D key-value store: the value may be a key-value store itself
- Different databases aggregate data differently on disk with different optimizations

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

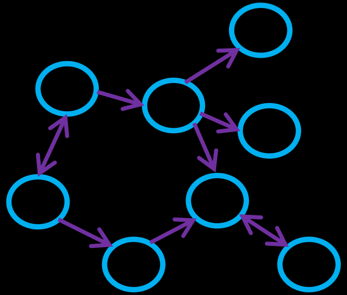
- Great for semantic web
- Great for graphs
- Can be hard to visualize
- Serialization can be difficult
- Queries more complicated



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

Noy, Goa, Jain. Communications of the ACM, August 2019

Hadoop & Spark

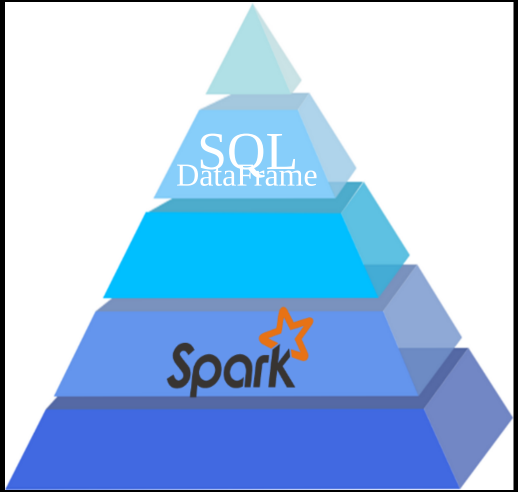
~~What kind of databases are they?~~

Frameworks for Data

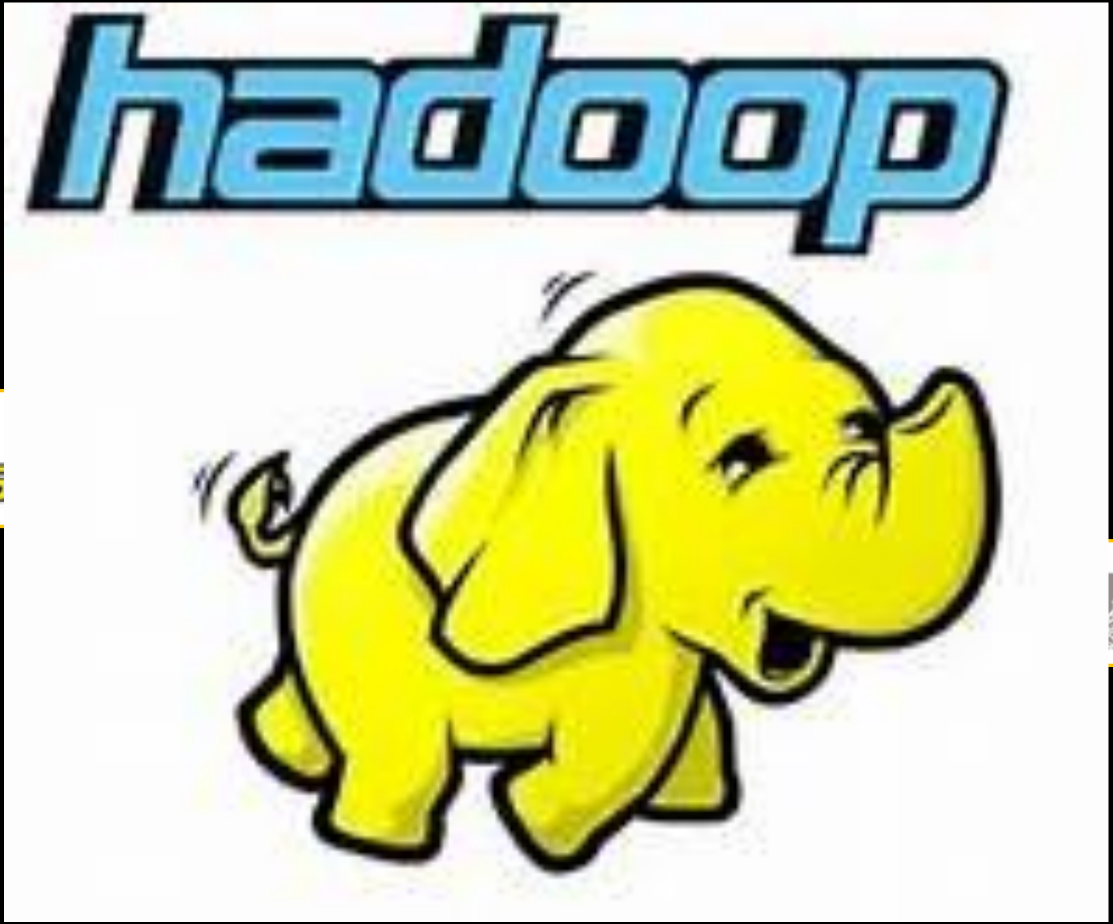
These are both frameworks for distributing and retrieving data. Hadoop is focused on disk based data and a basic map-reduce scheme, and Spark evolves that in several directions that we will get in to. Both can accommodate multiple types of databases and *achieve their performance gains by using parallel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



Hadoop Ecosystem Lives On



And lots more...

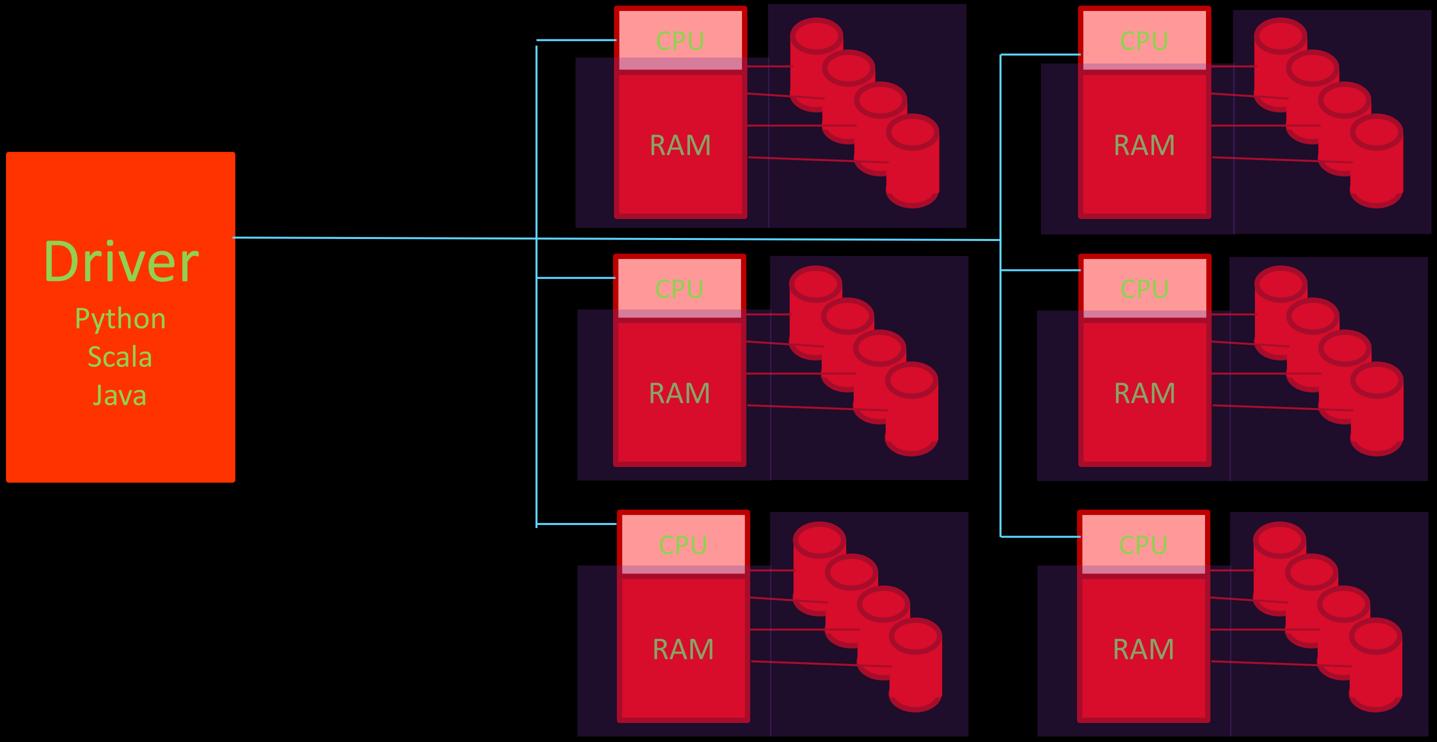
Spark Capabilities

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

Same Size

Fewer Elements

More Elements

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

Common Actions

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

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

Transformations vs. Actions

Transformations go from one RDD to another[1].

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

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

Pair RDD Actions

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

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

Two Pair RDD Transformations

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

Joins Are Quite Useful

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

And an RDD with all of our customers' addresses.

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

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

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

>>> promotion_mail_rdd = best_customers_rdd.join(customer_addresses_rdd)

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

Shakespeare, a Data Analytics Favorite

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

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

Who needs this Spark stuff?

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

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

```
>>> lines_rdd = sc.textFile("Complete_Shakespeare.txt")
>>>
>>> lines_rdd.count()
124787
>>>
>>> words_rdd = lines_rdd.flatMap(lambda x:
>>> words_rdd.count()
904061
>>>
>>> words_rdd.distinct().count()
67779
>>>
>>> key_value_rdd = words_rdd.map(lambda x: (x,1))
>>>
>>> key_value_rdd.take(5)
[('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')]
>>>
```

Things data scientists do.

Turn these into k/v pairs

Reduce to get words counts

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

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

Some Homework Problems

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

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

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

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

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

IO Formats

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

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

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

Spark Streaming

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

There are features to enable:

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

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

A Few Words About DataFrames

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

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

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

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,"WorldR
{"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)
 |-- Decommisioned: 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|Decommisioned|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 LANAUSSE|     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
+-------------------+-------+-------------+-------------------+-----+--------------------
```

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?

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.

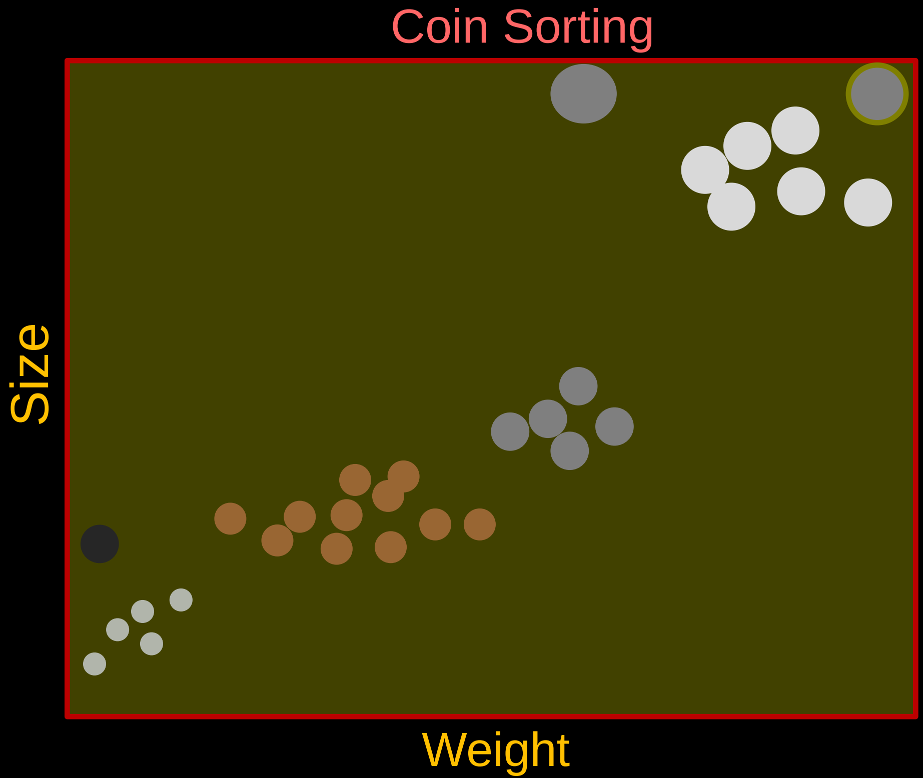
There are good example documents for the clustering routine we are using, as well as alternative clustering algorithms, here:

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

I suggest you use these pages for your Spark work.

Clustering

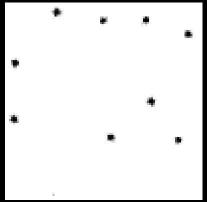
Clustering is a very common operation for finding grouping in data and has countless applications. This is a very simple example, but you will find yourself reaching for a clustering algorithm frequently in pursuing many diverse machine learning objectives, sometimes as one part of a pipeline.



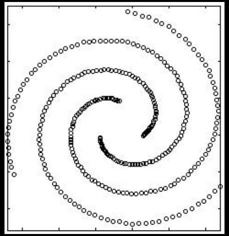
Clustering

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

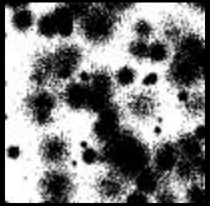
You might think this is trivial to implement in lower dimensional spaces.



But it can get tricky even there.



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



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

Finding Clusters

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

Using Python version 2.7.5 (default, Nov 20 2015 02:00:19)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
>>> rdd1 = sc.textFil
>>>
>>> rdd2 = rdd1.map(l
>>> rdd3 = rdd2.map(l
>>>
```

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

to RDD

rm to words and integers

Finding Our Way

```
>>> rdd1 = sc.textFile("5000_points.txt")
>>> rdd1.count()
5000
>>> rdd1.take(4)
['    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]]
>>>
```

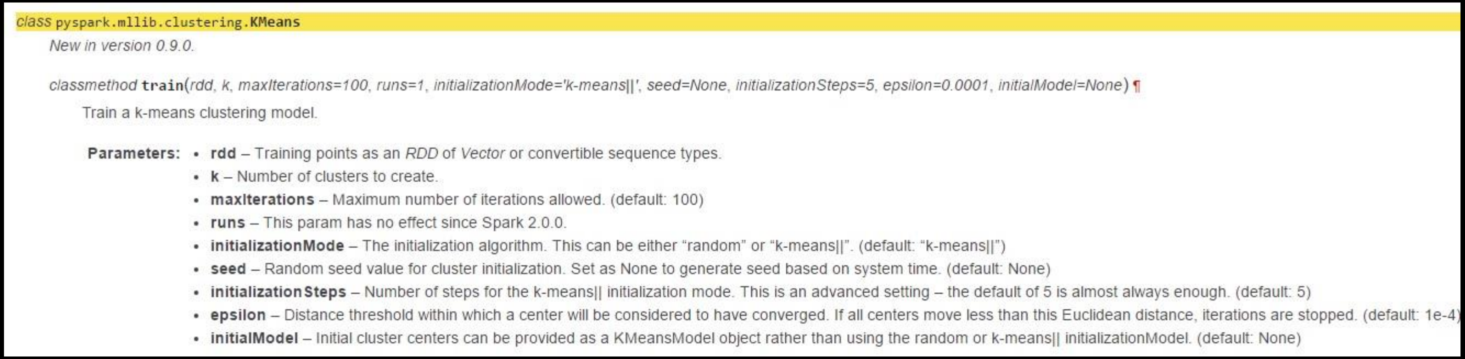
Finding Clusters

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

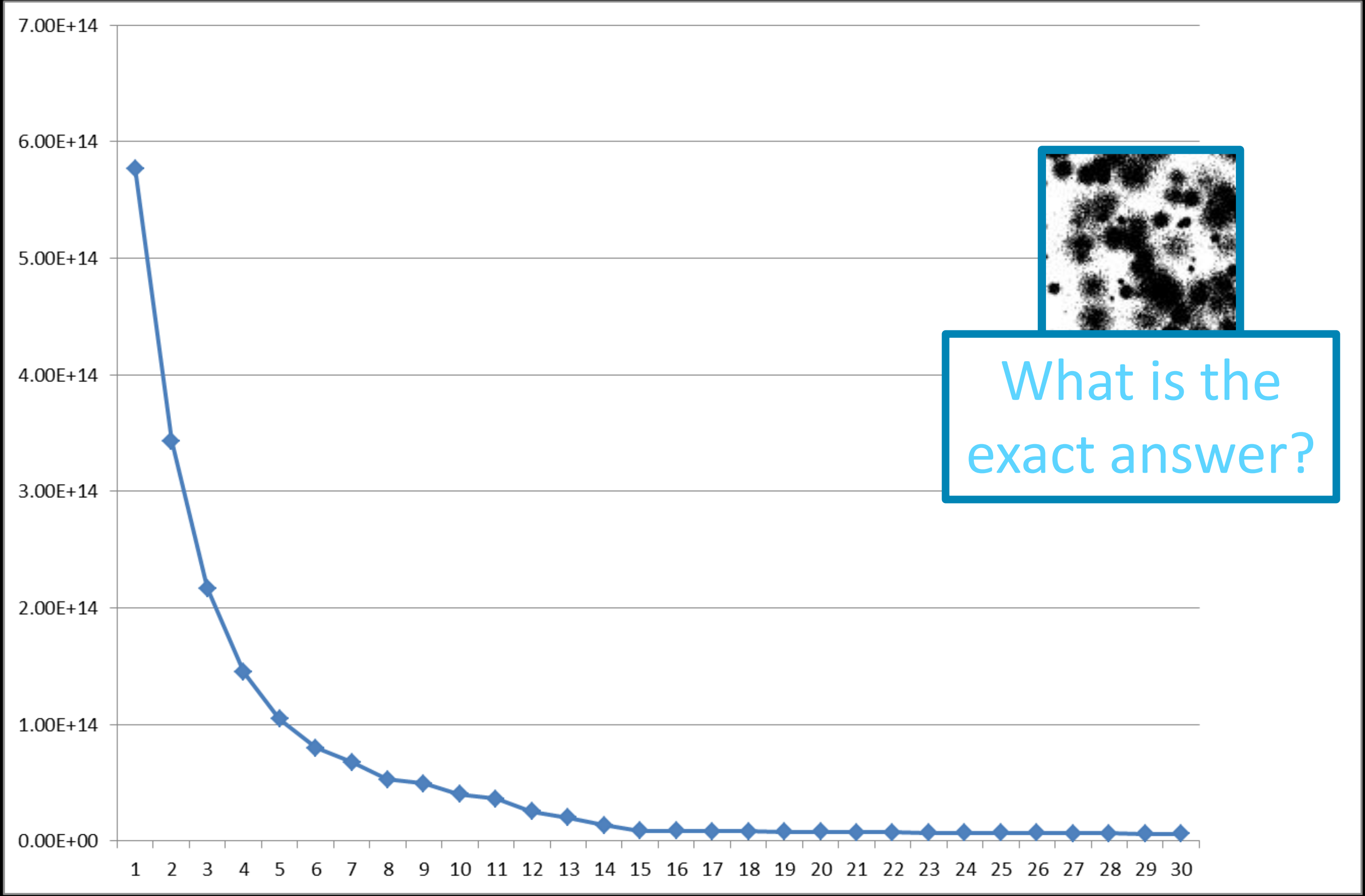
classmethod **train**(*rdd, k, maxIterations=100, runs=1, initializationMode='k-means||', seed=None, initializationSteps=5, epsilon=0.0001, initialModel=None*) ¶

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|| initializationModel. (default: None)

Finding Clusters



What is the exact answer?

Finding Clusters

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

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

Let’s see results for 1-30 cluster tries

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

Right Answer?

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

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

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

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

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

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

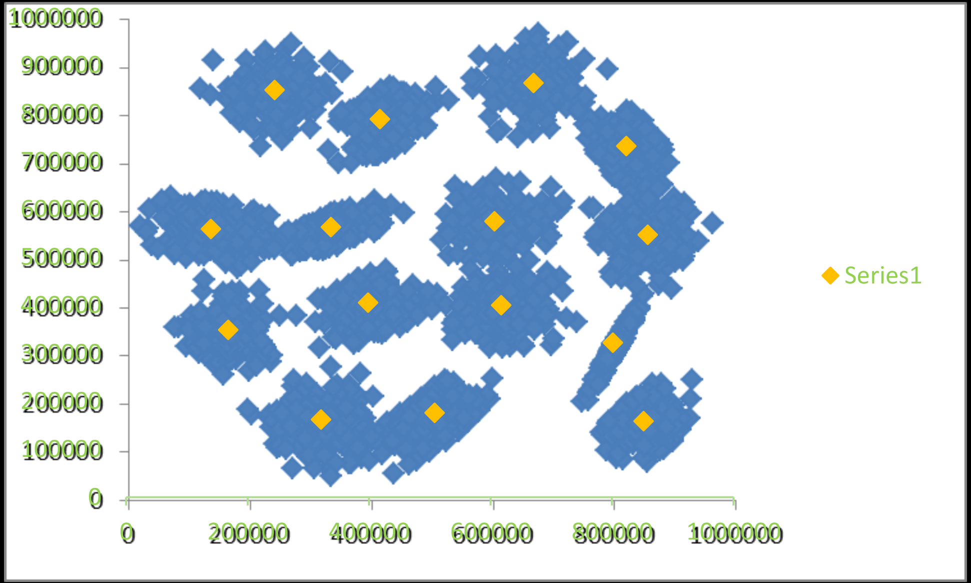
```
12 2.31466520037e+13
13 1.91856542103e+13
14 1.4933202312e+13
15 1.3506302755e+13
16 8.7757678836e+12
17 1.60075548613e+13

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

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

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

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

Find the Centers

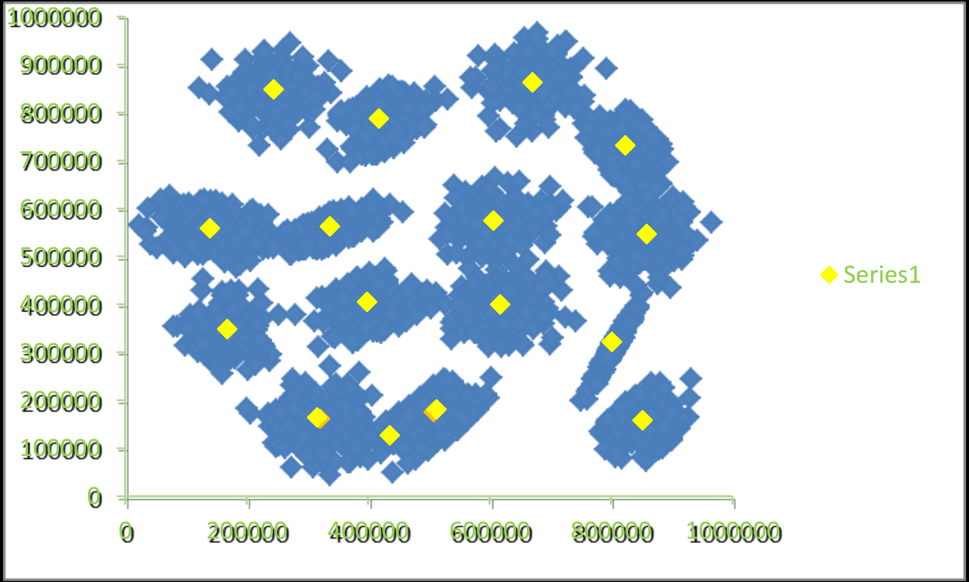
```
>>> for trials in range(10):                          #Try ten times to find best result
...    for clusters in range(12, 16):                 #Only look in interesting range
...        model = KMeans.train(rdd3, clusters)
...        cost = model.computeCost(rdd3)
...        centers = model.clusterCenters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

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

Fit?


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

16 Clusters


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

Dimensionality Reduction

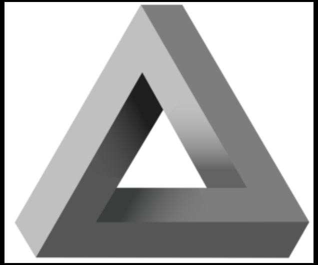
A look ahead to next Thursday

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

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

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

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



Why all these dimensions?

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

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

< 2900 Categories >

This is a 2900 dimensional vector.

Why all these dimensions?

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

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

80 dimensional vector.

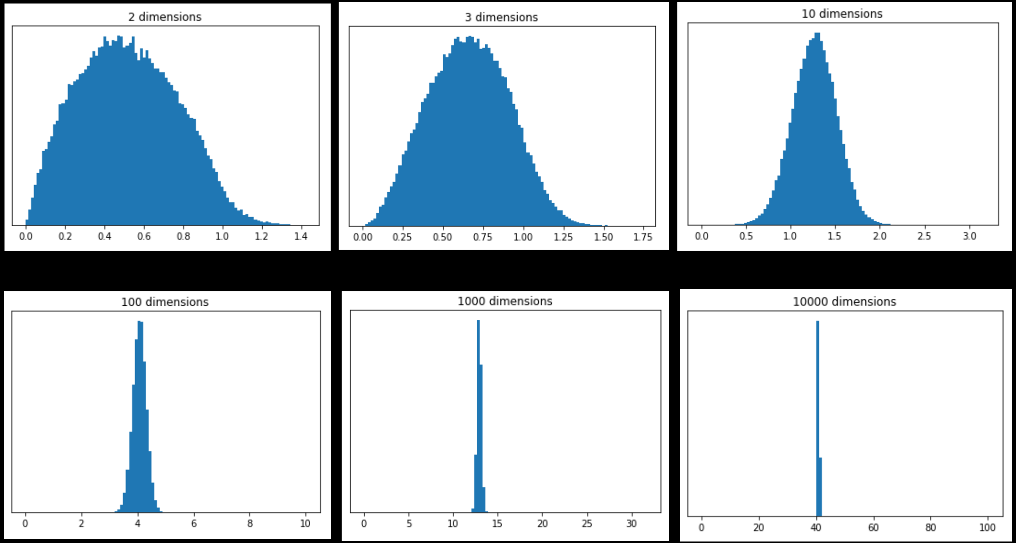
Customer Representation									
Cluster	Young Parents	College Athlete	Auto Enthusiast	Knitter	Steelers Fan	Shakespeare Reader	Sci-Fi Fan	Plumber	...
Distance	0.02	2.3	1.4	8.4	2.2	14.9	3.3	0.8	...

We have now accomplished two things:

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

Curse of Dimensionality

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



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

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

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

Metrics

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

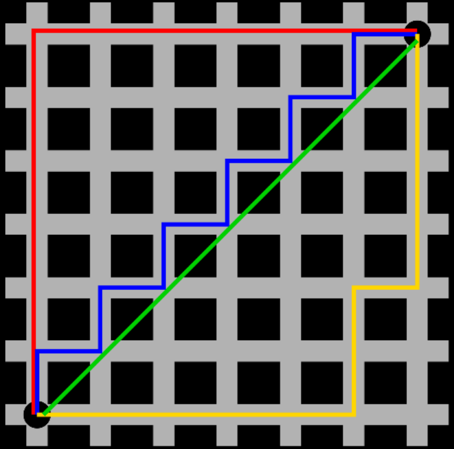


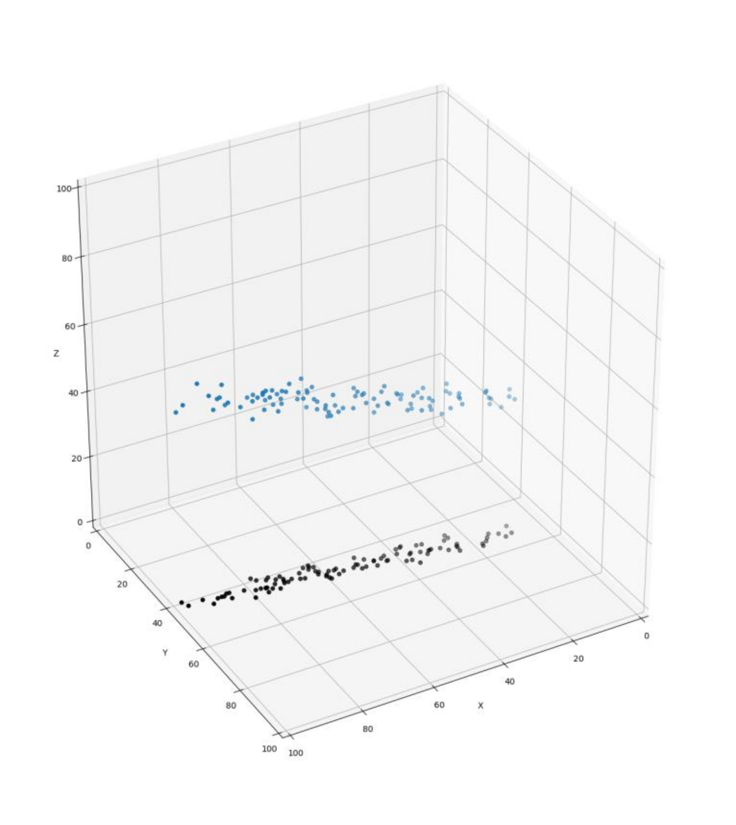
Image Source: Wikipedia

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

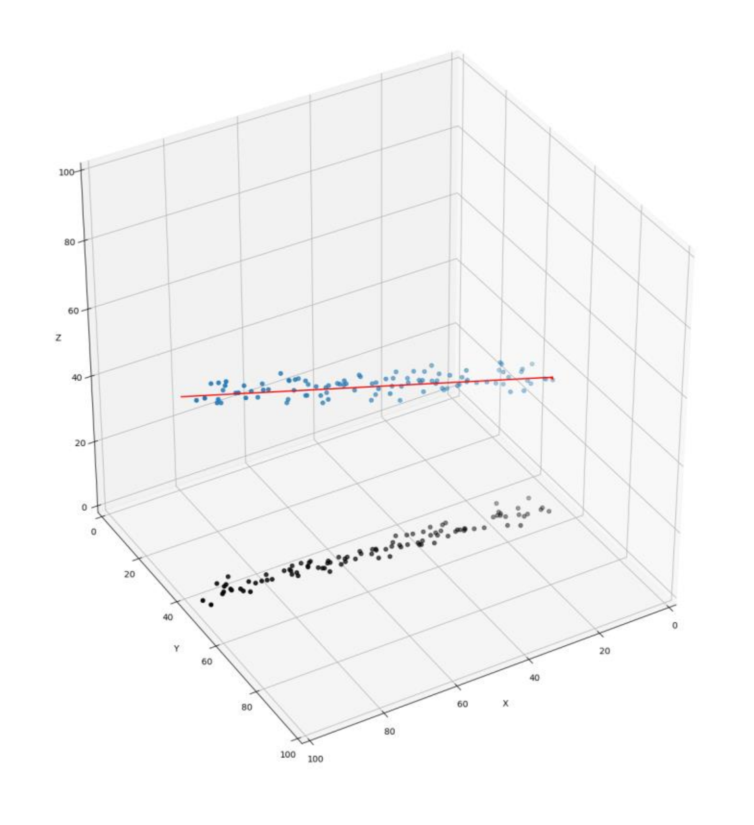
For genomics, you might care more about string sequences.

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

Alternative DR: Principal Component Analysis

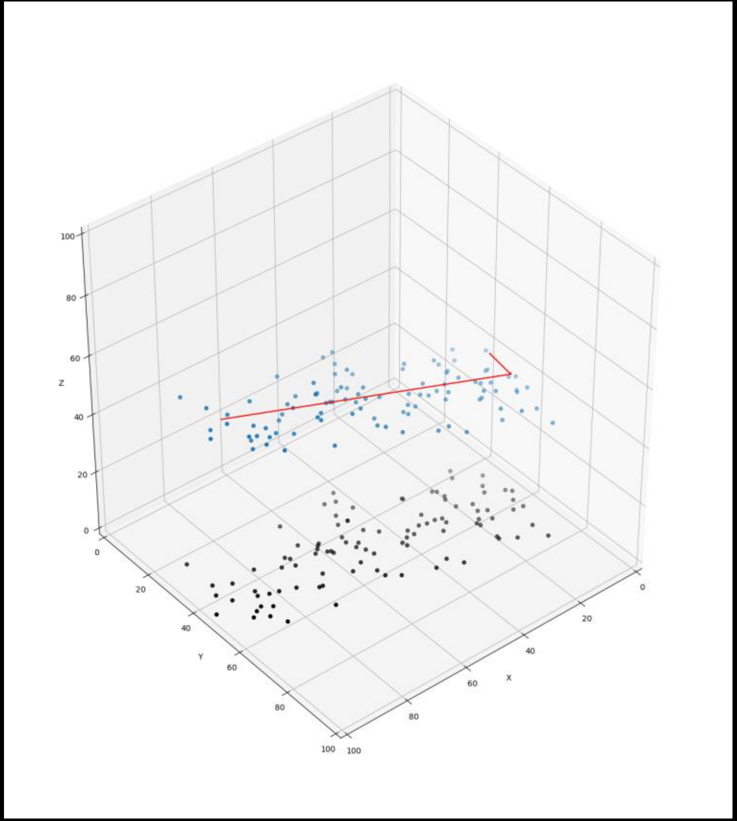


3D Data Set

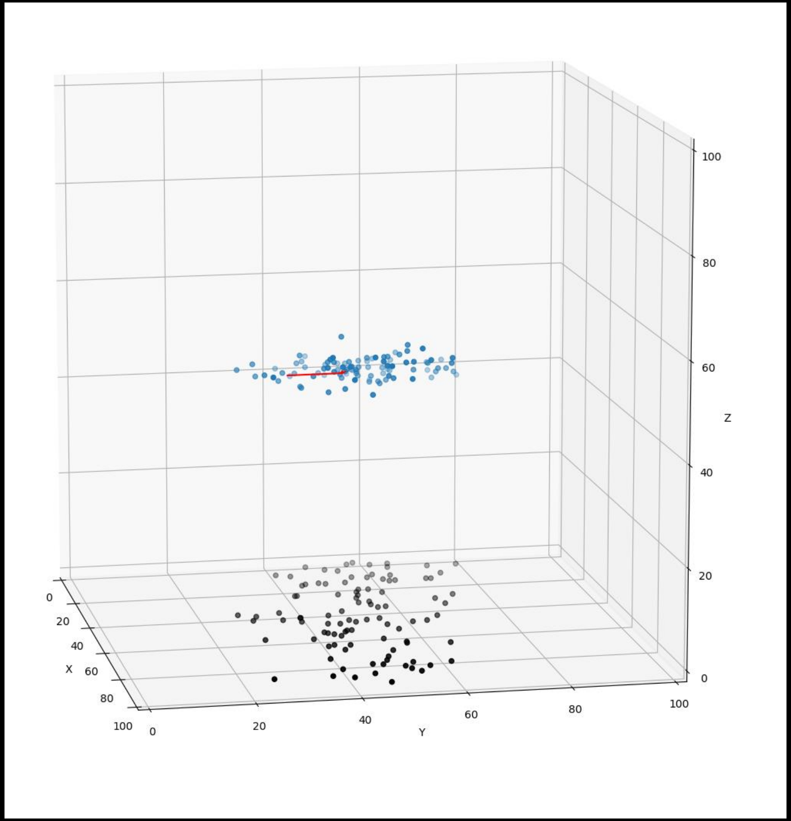


Maybe mostly 1D!

Alternative DR: Principal Component Analysis



Flatter 2D-ish Data Set



View down the 1st Princ. Comp.

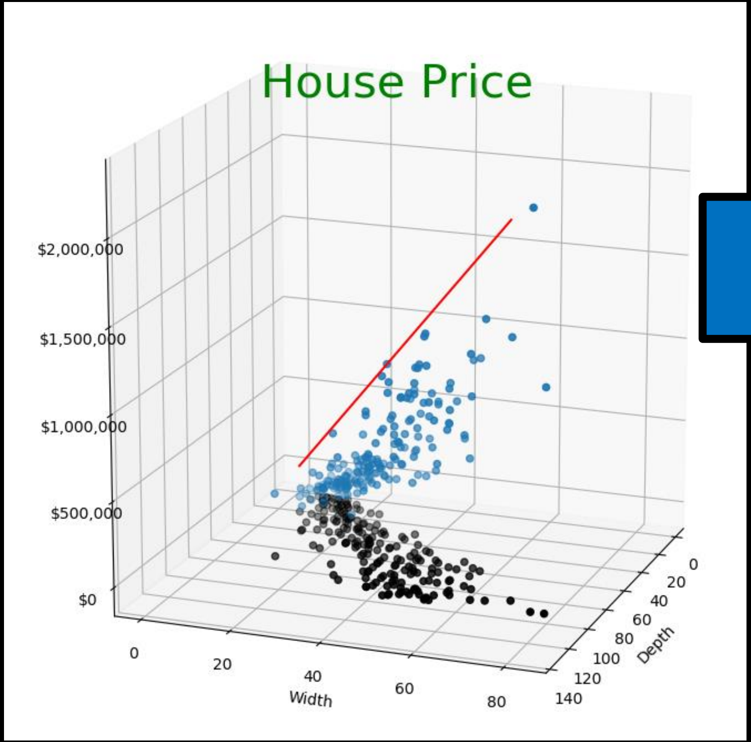
Why So Many Alternatives?

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

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

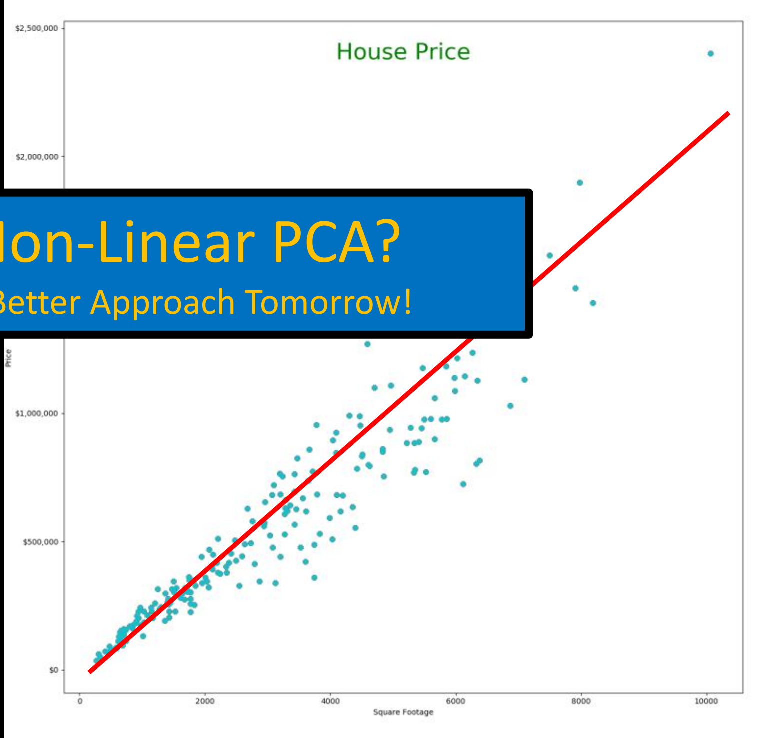
There may be some opportunities to reduce the dimension of the vector here. Perhaps clustering on the geographical coordinates...

Principal Component Analysis Fail



1st Component Off

Data Not Very Linear



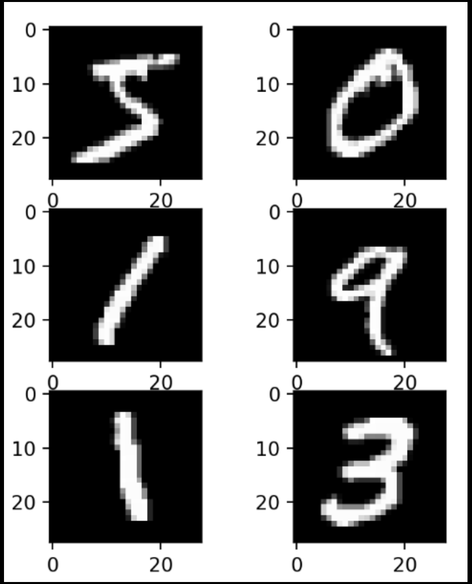
D x W Is Not Linear

But (DxW) Fits Well

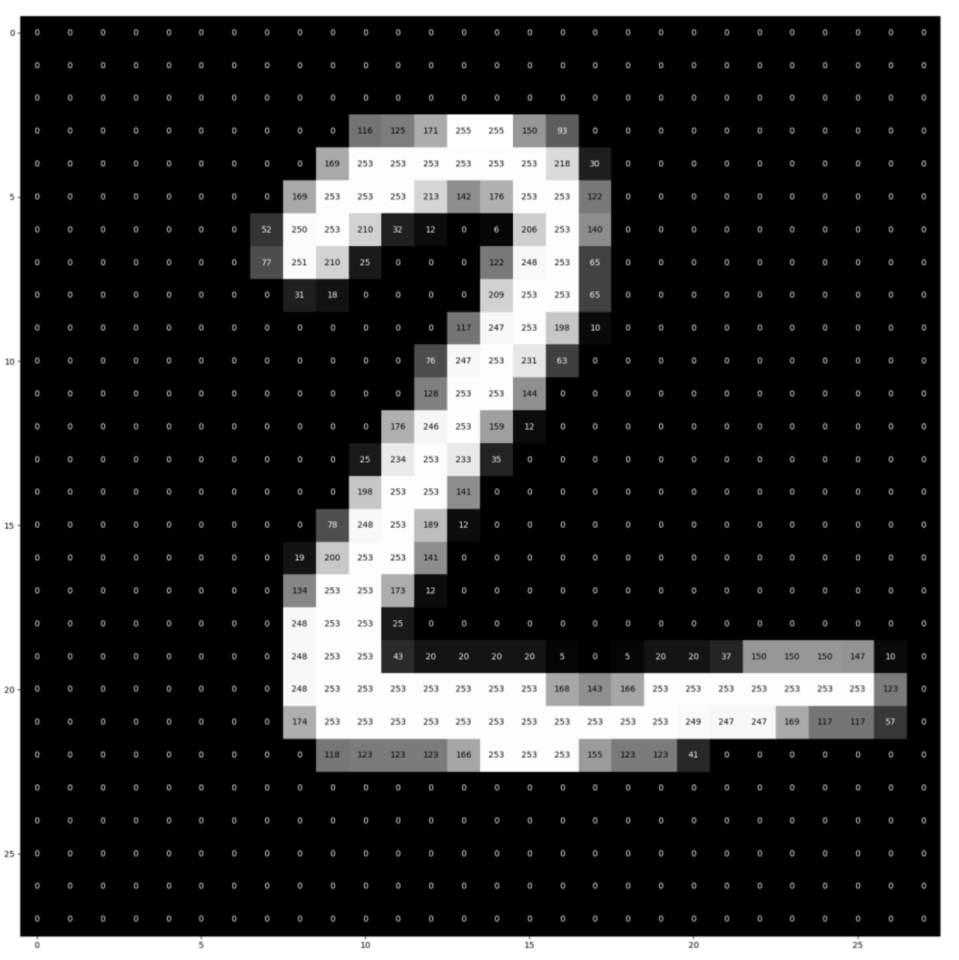
Non-Linear PCA?

A Better Approach Tomorrow!

Why Would An Image Have 784 Dimensions?

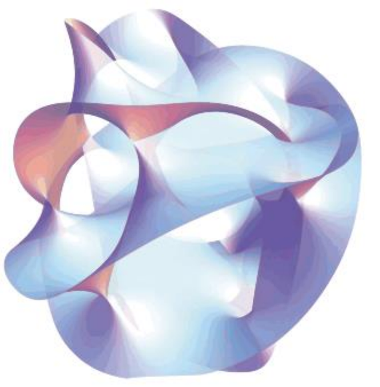


MNIST 28x28
greyscale images

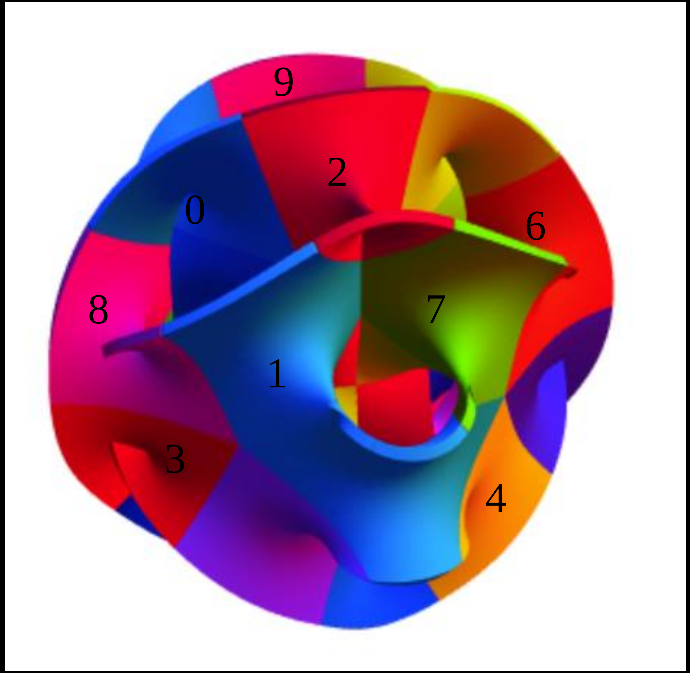


	0					5					10					15					20					25		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	116	125	171	255	255	150	93	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	169	253	253	253	253	253	253	218	30	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	169	253	253	253	213	142	176	253	253	122	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	52	250	253	210	32	12	0	6	206	253	140	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	77	251	210	25	0	0	0	122	248	253	65	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	31	18	0	0	0	0	209	253	253	65	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	117	247	253	198	10	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	76	247	253	231	63	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	128	253	253	144	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	176	246	253	159	12	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	25	234	253	233	35	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	198	253	253	141	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	78	248	253	189	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	19	200	253	253	141	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	134	253	253	173	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	248	253	253	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	248	253	253	43	20	20	20	20	5	0	5	20	20	37	150	150	150	147	10	0
20	0	0	0	0	0	0	0	0	248	253	253	253	253	253	253	253	168	143	166	253	253	253	253	253	253	253	123	0
	0	0	0	0	0	0	0	0	174	253	253	253	253	253	253	253	253	253	253	253	249	247	247	169	117	117	57	0
	0	0	0	0	0	0	0	0	0	118	123	123	123	166	253	253	253	155	123	123	41	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

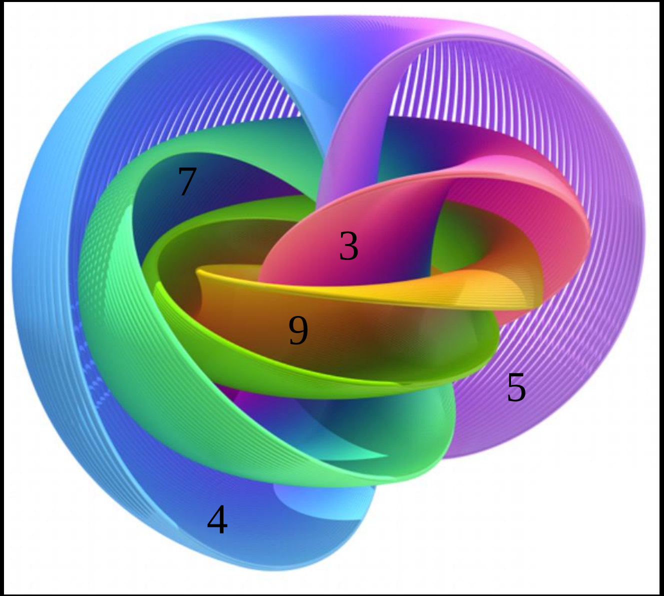
Central Hypothesis of Modern DL



Data Lives On
A Lower Dimensional
Manifold



Maybe Very Contiguous

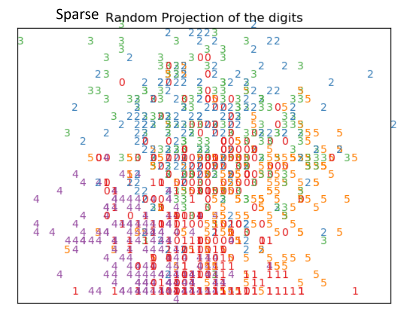


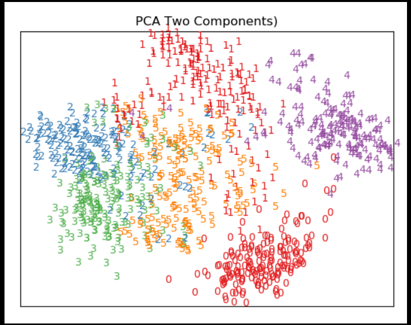
Maybe Less So

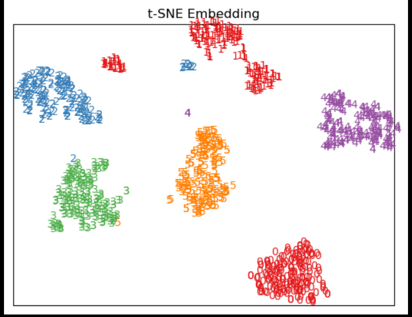
Images from Wikipedia

Testing These Ideas With Scikit-learn

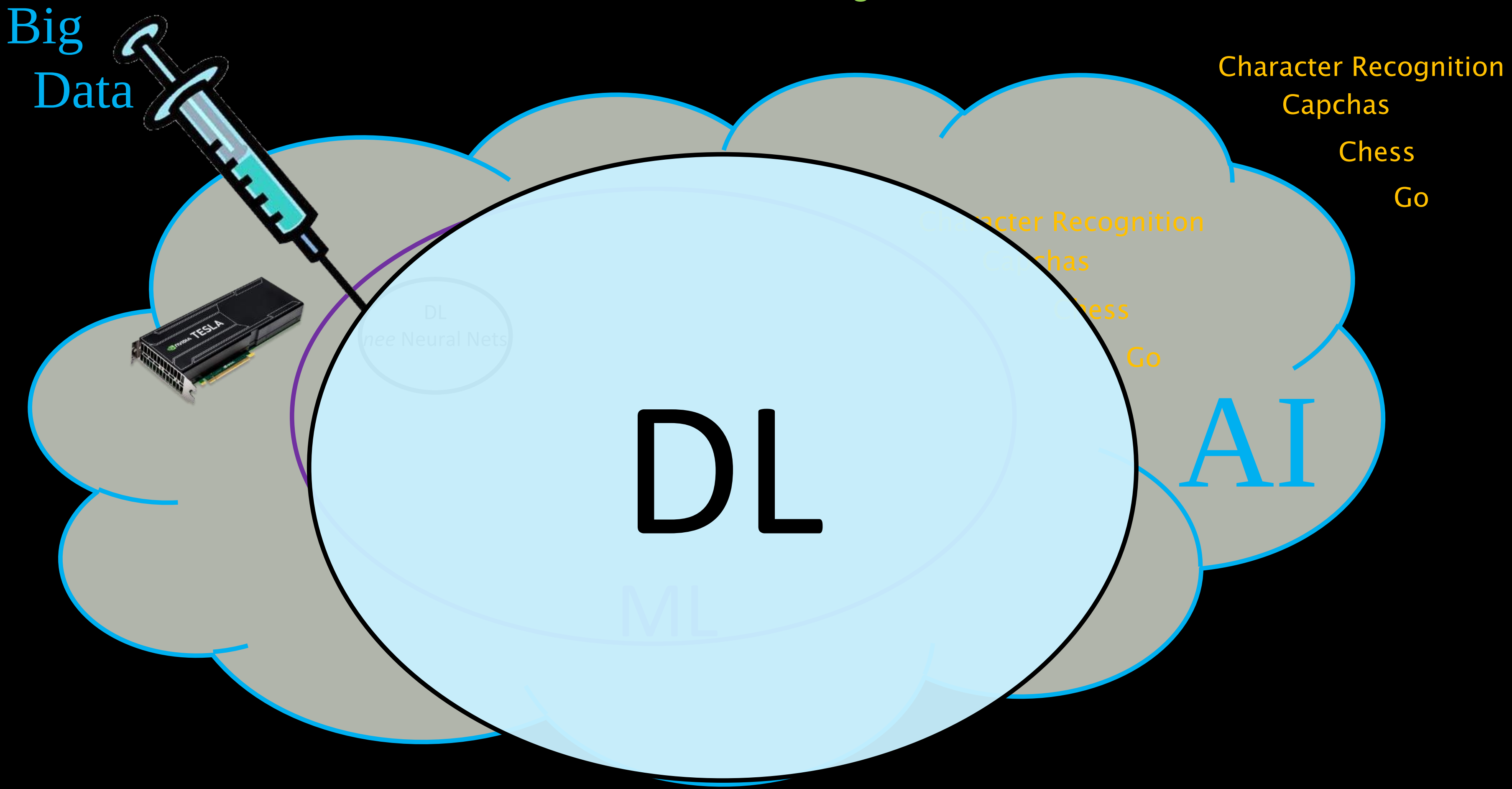
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import (datasets, decomposition, manifold, random_projection)

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()
```







How does all this fit together?



Big Data

Character Recognition

Capchas

Chess

Go

DL

nee Neural Nets

acter Recognition

chas

ess

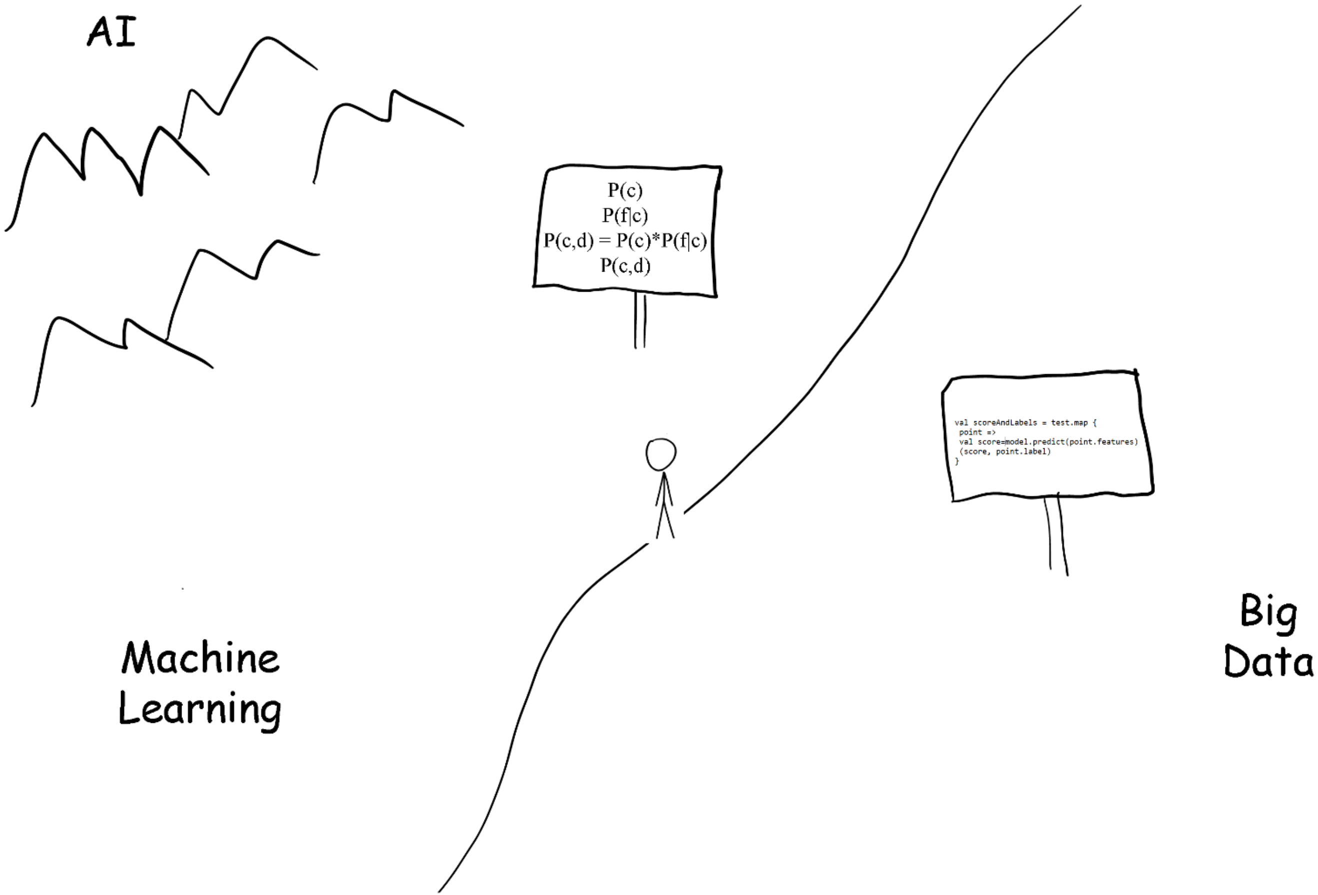
Go

DL

ML

AI

The Journey Ahead



$P(c)$

$P(f|c)$

$P(c,d) = P(c)*P(f|c)$

$P(c,d)$

```
val scoreAndLabels = test.map {
 point =>
 val score=model.predict(point.features)
 (score, point.label)
}
```

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 probablity and statistics.

A Recommender System

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Obvious Applications

We are now advanced enough that we can aspire to a serious application. One of the most significant applications for some very large websites (Netflix, Amazon, etc.) are recommender systems.

“Customers who bought this product also bought these.”

“Here are some movies you might like…”

As well as many types of targeted advertising. However those of you with less commercial ambitions will find the core concepts here widely applicable to many types of data that require dimensionality reduction techniques.

Let’s go all Netflix

Netflix once (2009) had a $1,000,000 contest to with just this very problem[1]. We will start with a similar dataset. It looks like:

Movie Dataset (Movie ID, Title, Genre):

```
31,Dangerous Minds (1995),Drama
32,Twelve Monkeys (a.k.a. 12 Monkeys) (1995),Mystery|Sci-Fi|Thriller
34,Babe (1995),Children|Drama
```

Ratings Dataset (User ID, Movie ID, Rating, Timestamp):

```
2,144,3.0,835356016
2,150,5.0,835355395
2,153,4.0,835355441
2,161,3.0,835355493
```

We won’t use the genres or timestamp fields for our analysis.

1) https://en.wikipedia.org/wiki/Netflix_Prize

Starting Point

What we are given is a large (100,480,507 ratings) and sparse (that is a little better than 1% of 8,532,958,530 matrix elements) list of ratings for users:

← 480,189 Users →

	5		**3**											**1**
1														
			3			5								
								3			3			3
	2			4							5		5	

← 17,770 Movies →

Objective

For any given user we would like to use their ratings, in combination with all the existing user ratings, to determine which movies they might prefer. For example, a user might really like *Annie Hall* and *The Purple Rose of Cairo* (both Woody Allen movies, although our database doesn't have that information). Can we infer from other users that they might like *Zelig*? That would be finding a latent variable. These might also include affinities for an actor, or director, or genre, etc.

Users

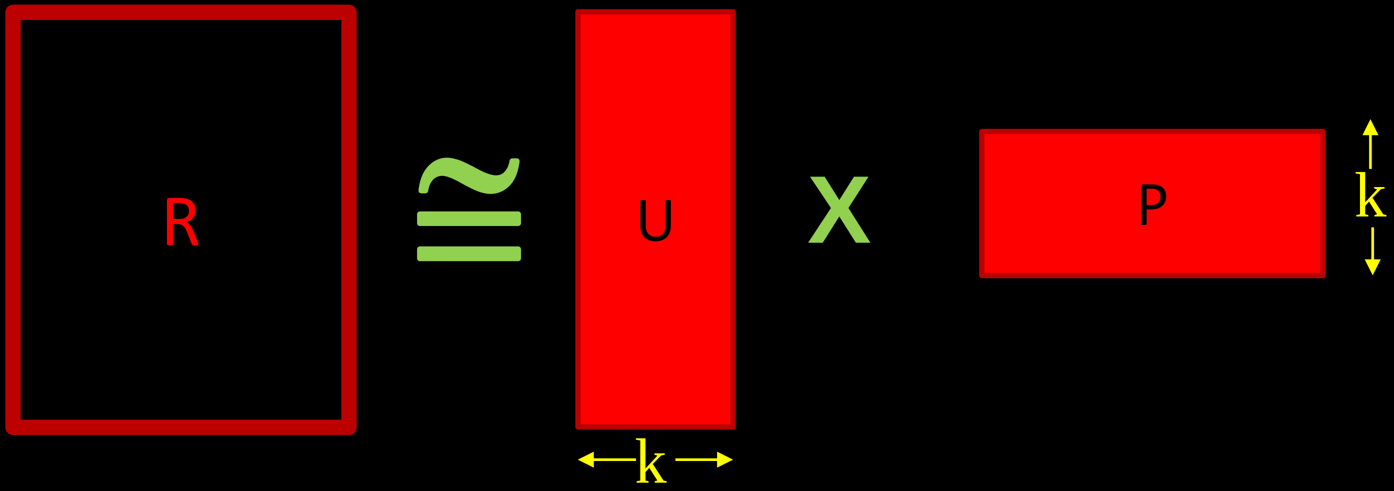
3	**5**	**2**	**3**	**5**	**2**	**4**	**3**	**1**	**4**	**3**	**1**	**3**	**2**	**1**
1	4	1	3	2	1	3	2	5	5	3	4	3	4	1
1	3	5	3	3	5	5	4	2	2	2	5	4	3	3
4	1	2	4	2	3	5	2	3	1	1	3	1	2	3
2	2	4	1	4	4	3	5	1	2	4	5	2	5	4

Movies

Matrix Factorization

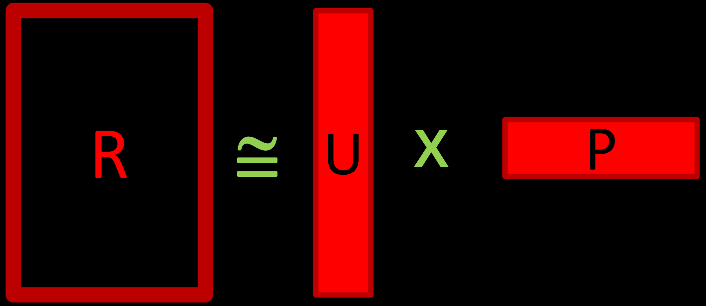
This resulting large, dense, matrix would be too big to actually keep around. We need to find a compressed representation where we can reproduce any given element we request. This will have to be lossy.

There are different ways to decompose a matrix. We will approximate our matrix as the product of two smaller matrices. The rank, k, of the new matrices will determine how accurate this approximation will be.



Lossy Compression Becomes Approximate Solution

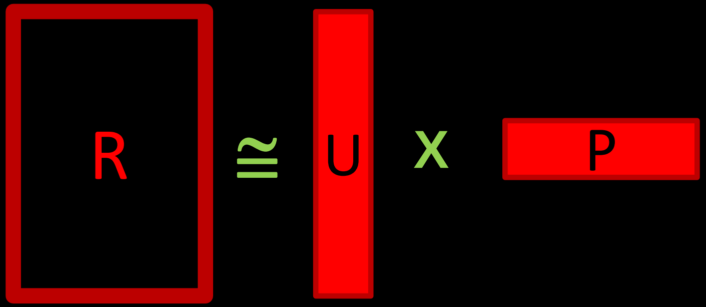
The process of lossy compressing the sparse R matrix is also going to provide us a means to construct its missing members (the dense matrix).



We will call our smaller matrices a *user feature matrix* and a *product feature matrix*. *This approximation is also going to smooth out the zeros and in the process give us our projected ratings.*

Why are we getting this two-for-one?

This provides an excellent introduction to a profound perspective on Machine Learning.



One way of thinking about learning is that we are compressing everything we know about the world into a smaller representation. Sometimes, but not usually, this can be seen explicitly, as here.

You can do this too.

Let's say you worked in a 1990's video store, but had never heard of Steven Spielberg. If you paid careful attention to the rental records you might notice that many people that rented *E.T.* also rented *Raiders of the Lost Ark* and *Jaws* and *Close Encounters* and *Jurassic Park*. So if a customer told you they really enjoyed an Indiana Jones movie, you might suggest they try *Jurassic Park*. All without knowing who the director was. You have inferred a hidden connection (latent effect).

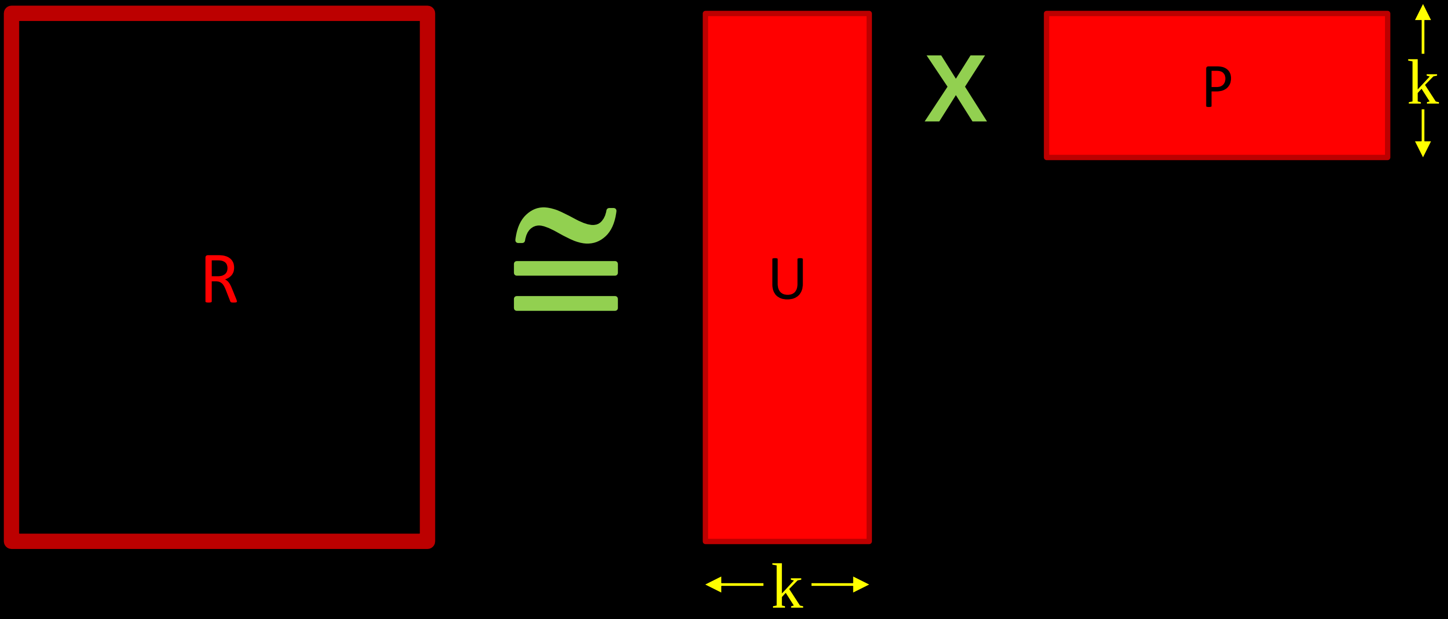
One can imaging many such hidden categories in our movie data: actors, genres, release dates, etc.

You can also imagine that the renters themselves possess these preferences hidden in their own data. Without it being explicitly noted, we might easily see that Mary likes documentaries and Joe loves movies with Cher.

We are thinking of reduced ways to represent these people ("likes documentaries") vs. the raw data!

Matrix Factorization

The rank k can now also be thought of as the number of *latent effects* we are incorporating. But it will not be as intuitively explicit as a simple category, and we will have to investigate an optimal size for this parameter.



Defining our error

In ML, defining the *error* (or *loss*, or *cost*) is often the core of defining the objective solution. Once we define the error, we can usually plug it into a canned solver which can minimize it. Defining the error can be obvious, or very subtle, or have multiple acceptable methods.

Clustering: For k-means we simply used the geometrical distance. It was actually the sum of the squared distances, but you get the idea.

Image Recognition: If our algorithm tags a picture of a cat as a dog, is that a larger error than if it tags it as a horse? Or a car? How would you quantify these?

Recommender: We will take the Mean Square Error distance between our given matrix and our approximation as a starting point.

Mean Square Error plus Regularization

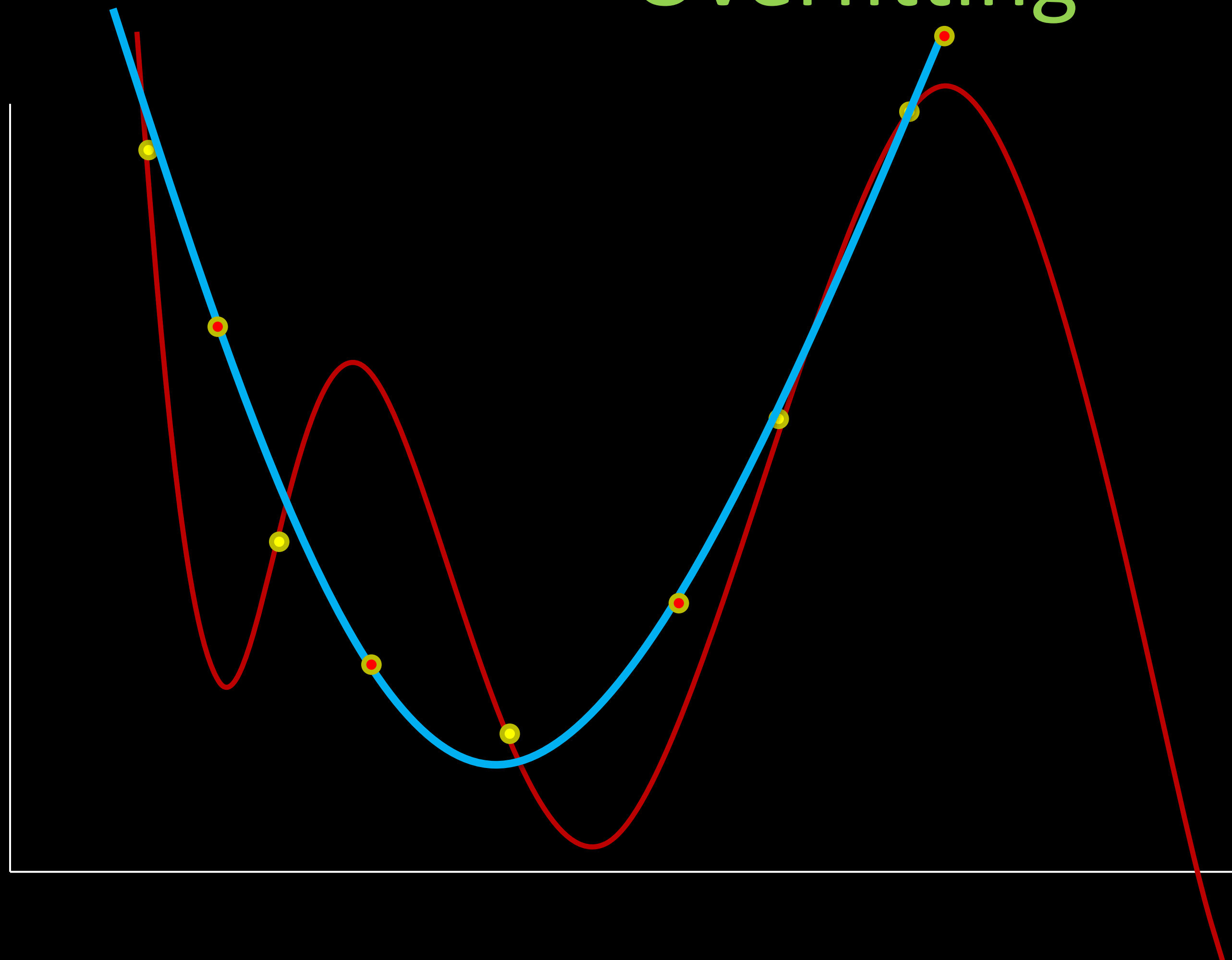
We will also add a term to discourage *overfitting* by damping large elements of U or P. This is called *regularization* and versions appear frequently in error functions.

$$\text{Error} = |\,R - U\text{x}P\,|^2 + \lambda(\text{Penalty for large elements})$$

The | | notation means “sum the squares of all the elements and then take the square root”.

You may wonder how we can have “too little” error – the pursuit of which leads to overfitting. Think back to our clustering problem. We could drive the error as low as we wanted by adding more clusters (up to 5000!). But we weren’t really finding new clusters. Variations of this phenomena occur throughout machine learning.

Overfitting



One solution is to keep using higher order terms, but to penalize them. These *regularization hyperparameters* that enable our solution to have good *generalization* will show up again in our workshop, and throughout your machine learning endeavors.

Think of this as Occam's Razor for machine learning.

Mean Square Error plus Regularization

Here is exactly our error term with regularization. MLLIB scales this factor for us based on the number of ratings (this tweak is called ALS-WR).

$$\text{Error} = | R - UxP |^2 + \lambda(|U|^2 + |P|^2)$$

The | | notation means "sum the squares of all the elements and then take the square root".

Additionally, we need to account for our missing (unrated) values. We just zero out those terms. Here it is term-by-term:

$$\text{Error} = \Sigma_{l,j}\, w_{l,j}\, (R_{l,j} - (UxP)_{ij})^2 + \lambda(|U|^2 + |P|^2) \qquad w_{l,j} = 0 \text{ if } R_{l,j} \text{ is unknown}$$

Note that we now have two hyperparameters, k and λ, that we need to select intelligently.

Alternating Least Squares

To actually find the U and P that minimize this error we need a solving algorithm.

SGD, a go-to for many ML problems and one we will use later, is not practical for billions of parameters, which we can easily reach with these types of problems. We are dealing with *Users X Items* elements here.

Instead we use Alternating Least Squares (ALS), also built into MLLIB.

- Alternating least squares cheats by holding one of the arrays constant and then doing a classic least squares fit on the other array parameters. Then it does this for the other array.

- This is easily parallelized.

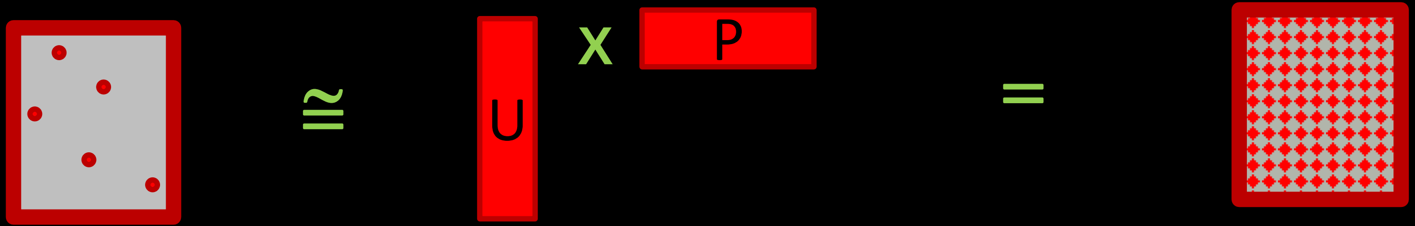
- It works well with sparse inputs. The algorithm scales linearly with observed entries.

Here Is Our Plan

Given Ratings

Use ALS

Predicted Ratings



Sparse

Dense

Users

	5		3											1
1														
			3			5								
								3			3			3
	2			4							5		5	

Movies

Users

3	5	2	3	5	2	4	3	1	4	3	1	3	2	1
1	4	1	3	2	1	3	2	5	5	3	4	3	4	1
1	3	5	3	3	5	5	4	2	2	2	5	4	3	3
4	1	2	4	2	3	5	2	3	1	1	3	1	2	3
2	2	4	1	4	4	3	5	1	2	4	5	2	5	4

Movies

Training, Validation and Test Data

We use the training data to create our solution, the UxP matrix here.

The validation data is used to verify we are not overfitting: to stop training after enough iterations, to adjust λ or k here, or to optimize the many other *hyperparameters* you will encounter in ML.

The test data must be saved to judge our final solution.

Reusing, or subtly mixing, the training, validation and t

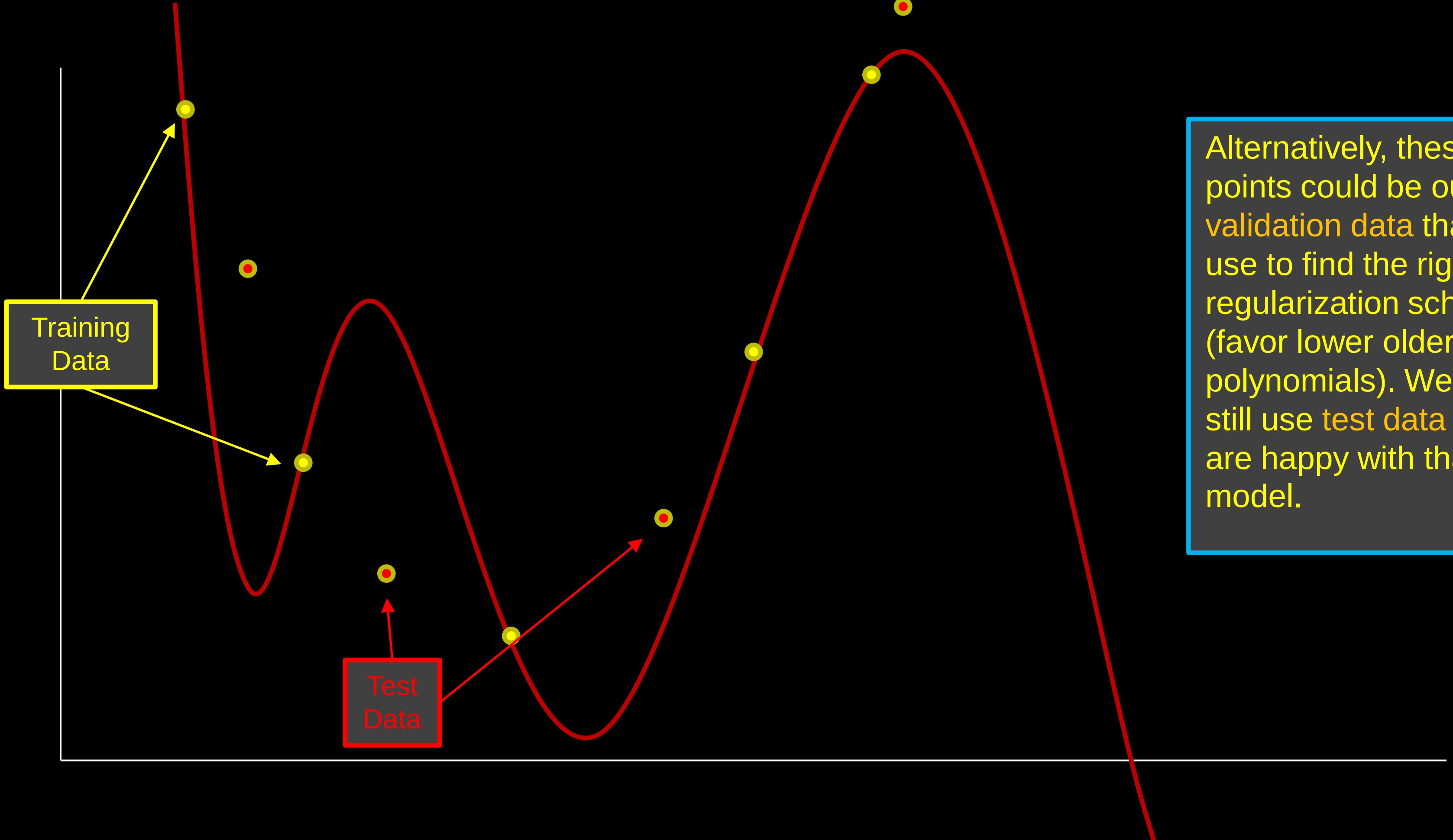
confusion.

What proportions of your data to use for each of these

might want to start by copying from similar work or exa

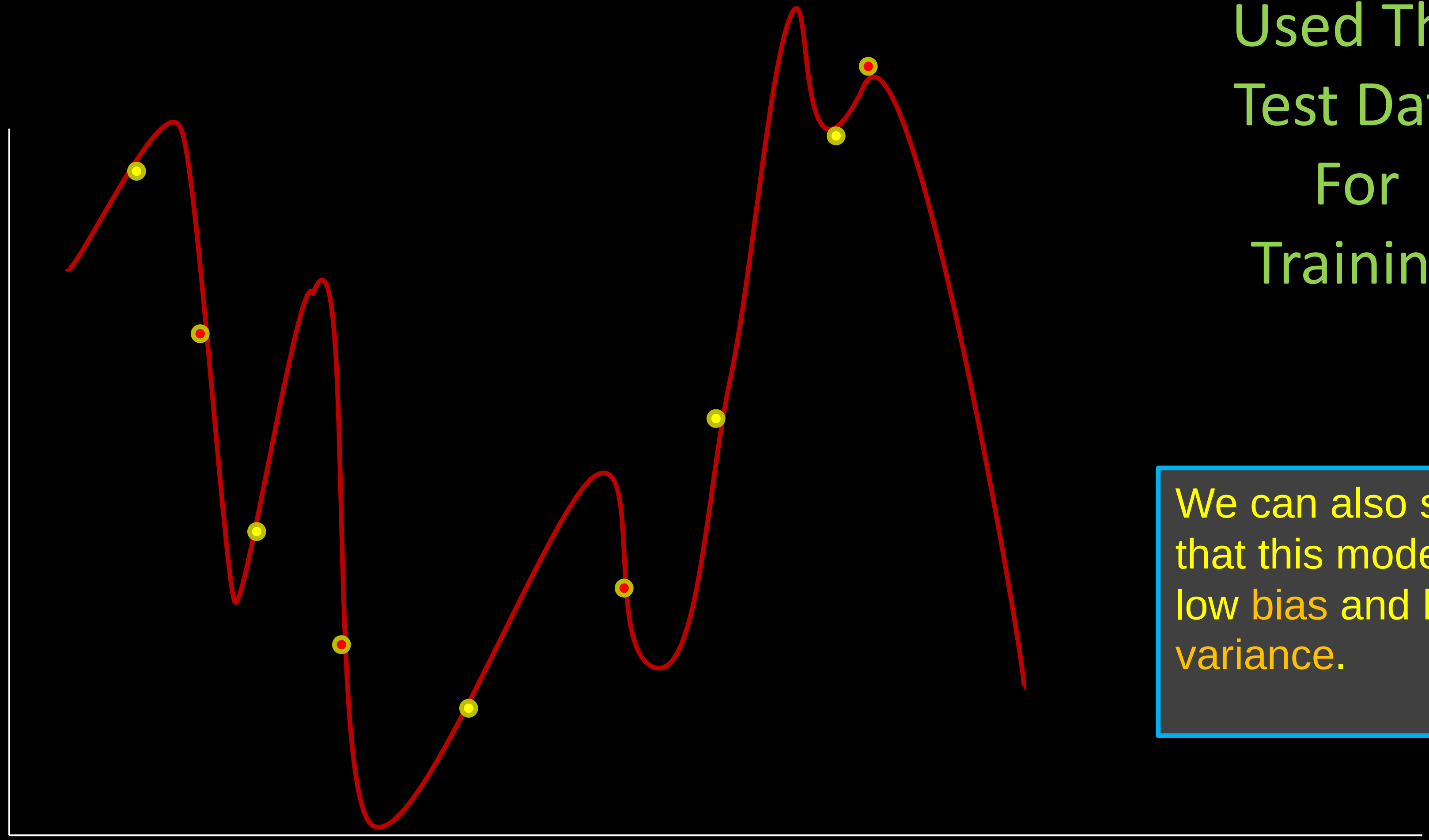
There are techniques to slice-and-cycle share the training and validation data, called *cross-validation*. Don't try this with the test data!

Reality Check By Test Data



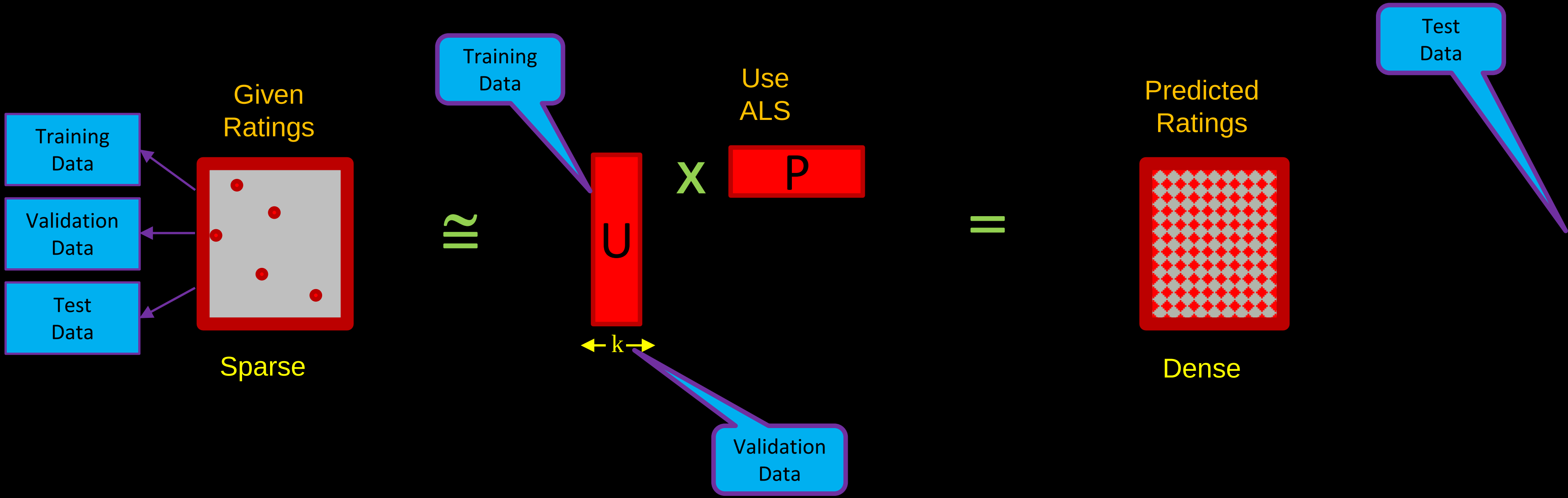
Alternatively, these new points could be our validation data that we use to find the right regularization scheme (favor lower older polynomials). We would still use test data after we are happy with that model.

Used The Test Data For Training



We can also say that this model has low bias and high variance.

Where does our data come into play?



Let’s Build A Recommender

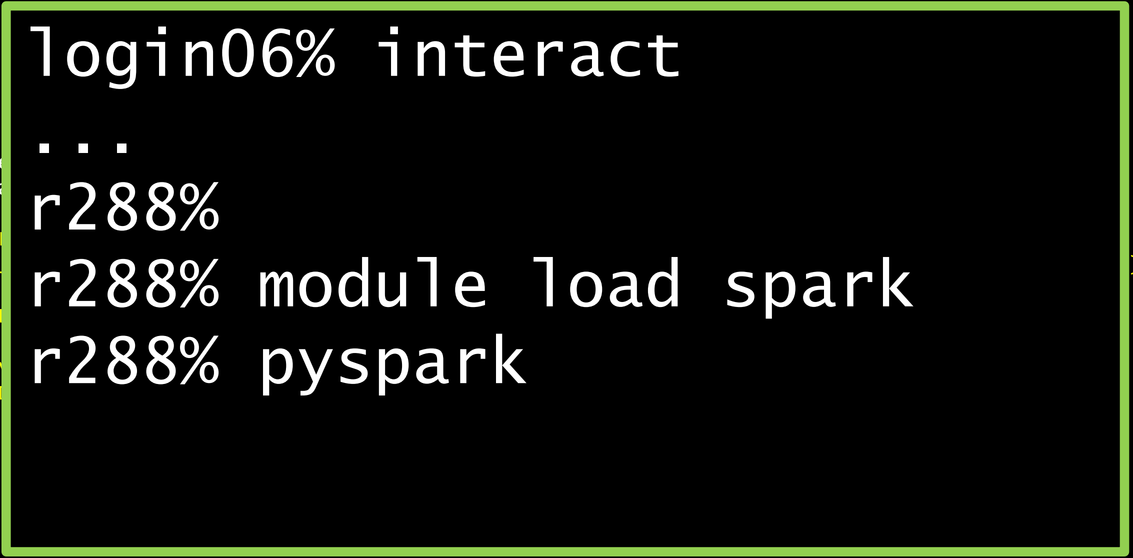
We have all the tools we need, so let’s fire up PySpark and create a scalable recommender. Our plan is:

1. Load and parse data files
2. Create ALS model
3. Train it with varying ranks (k) to find reasonable hyperparameters

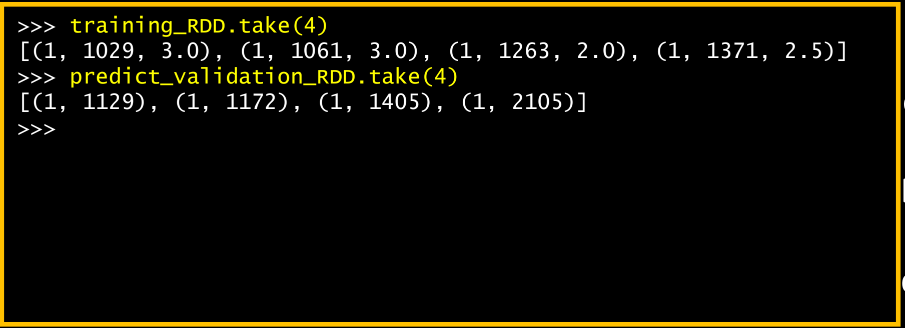
4. Add a new user
5. Get top recommendations for new user

Building a Recommender

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

Using Python version 3.7.4 (d
SparkSession available as 'sp
>>>
>>> ratings_raw_RDD = sc.text
>>> ratings_RDD = ratings_raw_                                                               ),float(tokens[2])))
>>>
>>> training_RDD, validation_
>>>
>>> predict_validation_RDD = 
>>> predict_test_RDD = test_R
```



```
login06% interact
...
r288%
r288% module load spark
r288% pyspark
```

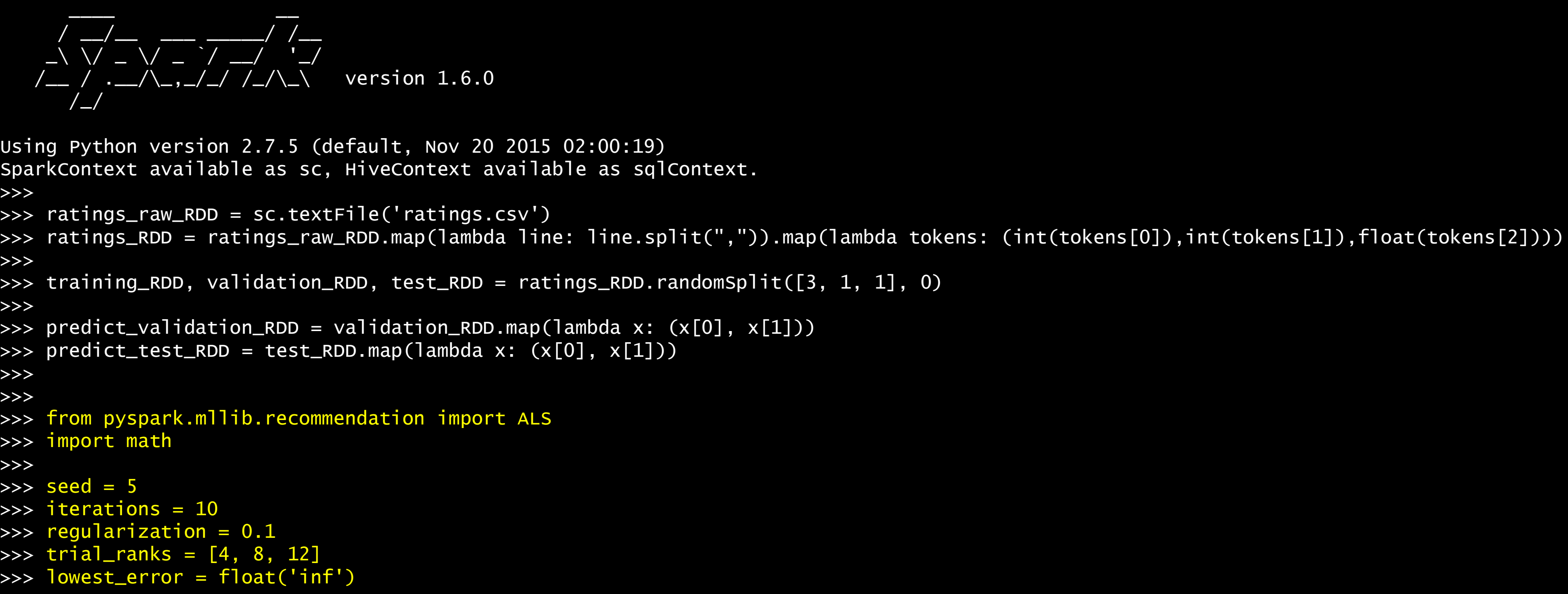


```
>>> training_RDD.take(4)
[(1, 1029, 3.0), (1, 1061, 3.0), (1, 1263, 2.0), (1, 1371, 2.5)]
>>> predict_validation_RDD.take(4)
[(1, 1129), (1, 1172), (1, 1405), (1, 2105)]
>>>
```

ovie,rating) data.

Ds.

data for our prediction RDDs.



```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   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.
>>>
>>> ratings_raw_RDD = sc.textFile('ratings.csv')
>>> ratings_RDD = ratings_raw_RDD.map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),int(tokens[1]),float(tokens[2])))
>>>
>>> training_RDD, validation_RDD, test_RDD = ratings_RDD.randomSplit([3, 1, 1], 0)
>>>
>>> predict_validation_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
>>> predict_test_RDD = test_RDD.map(lambda x: (x[0], x[1]))
>>>
>>>
>>> from pyspark.mllib.recommendation import ALS
>>> import math
>>>
>>> seed = 5
>>> iterations = 10
>>> regularization = 0.1
>>> trial_ranks = [4, 8, 12]
>>> lowest_error = float('inf')
```

Import mllib and set some variables we are about to use.

```
>>> ratings_raw_RDD = sc.textFile('ratings.csv')
>>> ratings_RDD = ratings_raw_RDD.map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),int(tokens[1]),float(tokens[2])))
>>>
>>> training_RDD, validation_RDD, test_RDD = ratings_RDD.randomSplit([3, 1, 1], 0)
>>>
>>> predict_validation_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
>>> predict_test_RDD = test_RDD.map(lambda x: (x[0], x[1]))
>>>
>>>
>>> from pyspark.mllib.recommendation import ALS
>>> import math
>>>
>>> seed = 5
>>> iterations = 10
>>> regularization = 0.1
>>> trial_ranks = [4, 8, 12]
>>> lowest_error = float('inf')
>>>
>>> for k in trial_ranks:
>>>    model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>    #Coercing ((u,p),r) tuple format to accomodate join
>>>    predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>    ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>    error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>    print ('For k=',k,'the RMSE is', error)
>>>    if error < lowest_error:
>>>        best_k = k
>>>        lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
>>>
>>> print('The best rank is size', best_k)
The best rank is size 4
```

Run our ALS model on various ranks to see which is best.

```
>>> for k in trial_ranks:
>>>     model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>     #Coercing ((u,p),r) tuple format to accomodate join
>>>     predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>     ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>     error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>     print ('For k=',k,'the RMSE is', error)
>>>     if error < lowest_error:
>>>         best_k = k
>>>         lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.943861262524O242
For k= 12 the RMSE is 0.9390638322819614
```

The ALS.train() routines gives us:

```
>>> model.predictAll(predict_validation_RDD).take(2)
[Rating(user=463, product=4844, rating=2.7640960482284322), Rating(user=380, product=4844, rating=2.399938320644199)]
```

To do the "RMS error" math, we want elements with a *(Given,Predicted)* value for each *(User,Movie)* key:

```
>>> ratings_and_preds_RDD.take(2)
[((119, 145), (4.0, 2.903215714486778)), ((407, 5995), (4.5, 4.604779028840272))]
```

So the next two lines get us from *here to there*.

```
>>> for k in trial_ranks:
>>>     model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>     #Coercing ((u,p),r) tuple format to accomodate join
>>>     predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>     ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>     error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>     print ('For k=',k,'the RMSE is', error)
>>>     if error < lowest_error:
>>>         best_k = k
>>>         lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
```

```
>>> model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2])).take(2)
[((463, 4844), 2.7640960482284322), ((380, 4844), 2.399938320644199)]
```

That map gets us to a pair RDD with *[(User,Movie), rating]* format.

Now do this with the validation RDD:

```
>>> validation_RDD.take(2)
[(1, 1129, 2.0), (1, 1172, 4.0)]
>>>
>>> validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).take(2)
[((1, 1129), 2.0), ((1, 1172), 4.0)]
```

```
>>> for k in trial_ranks:
>>>     model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>     #Coercing ((u,p),r) tuple format to accomodate join
>>>     predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>     ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>     error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>     print ('For k=',k,'the RMSE is', error)
>>>     if error < lowest_error:
>>>         best_k = k
>>>         lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.943861262524O242
For k= 12 the RMSE is 0.9390638322819614
```

To collect rating values for common (User,Movie) keys calls for a join()

Data before join:

```
>>> predictions_RDD.take(2)
[((463, 4844), 2.7640960482284322), ((380, 4844), 2.399938320644199)]
>>>
>>> validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).take(2)
[((1, 1129), 2.0), ((1, 1172), 4.0)]
```

Results of join:

```
>>> ratings_and_preds_RDD.take(2)
[((119, 145), (4.0, 2.903215714486778)), ((407, 5995), (4.5, 4.604779028840272))]
```

```
>>> for k in trial_ranks:
>>>     model = ALS.train(training_RDD, k, seed=seed, iterations=iterations, lambda_=regularization)
>>>     #Coercing ((u,p),r) tuple format to accomodate join
>>>     predictions_RDD = model.predictAll(predict_validation_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>>     ratings_and_preds_RDD = validation_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>>     error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>>     print ('For k=',k,'the RMSE is', error)
>>>     if error < lowest_error:
>>>         best_k = k
>>>         lowest_error = error
>>>
For k= 4 the RMSE is 0.9357038861004305
For k= 8 the RMSE is 0.9438612625240242
For k= 12 the RMSE is 0.9390638322819614
>>>
>>> print('The best rank is size', best_k)
The best rank is size 4
>>>
>>> model = ALS.train(training_RDD, best_k, seed=seed, iterations=iterations, lambda_=regularization)
>>> predictions_RDD = model.predictAll(predict_test_RDD).map(lambda r: ((r[0], r[1]), r[2]))
>>> ratings_and_preds_RDD = test_RDD.map(lambda r: ((r[0], r[1]), r[2])).join(predictions_RDD)
>>> error = math.sqrt(ratings_and_preds_RDD.map(lambda r: (r[1][0] - r[1][1])**2).mean())
>>> print ('For testing data the RMSE is %s' % (error))
For testing data the RMSE is 0.9406803213698973
```

This is our fully tested model (smallest dataset).
These results were reported against the test_RDD.

Adding a User

```
.
.
.
>>>
>>> new_user_ID = 0
>>> new_user = [
         (0,100,4), # City Hall (1996)
         (0,237,1), # Forget Paris (1995)
         (0,44,4),  # Mortal Kombat (1995)
         (0,25,5),  # etc....
         (0,456,3),
         (0,849,3),
         (0,778,2),
         (0,909,3),
         (0,478,5),
         (0,248,4)
        ]
>>>
>>> new_user_RDD = sc.parallelize(new_user)
>>>
>>> updated_ratings_RDD = ratings_RDD.union(new_user_RDD)
>>>
>>> updated_model = ALS.train(updated_ratings_RDD, best_rank, seed=seed, iterations=iterations,
lambda_=regularization)
>>>
```

I checked that ID 0 is unused with a quick `ratings_RDD.filter(lambda x: x[0]=='0').count()"`

Note that we are joining, and then training, with ALL data now - the ratings RDD. We are confident we know what we are doing and are done testing.

Let’s get some predictions…

```
.
.
.
>>>
>>> movies_raw_RDD = sc.textFile('movies.csv')
>>> movies_RDD = movies_raw_RDD.map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),tokens[1]))
>>>
>>> new_user_rated_movie_ids = map(lambda x: x[1], new_user)
>>> new_user_unrated_movies_RDD = movies_RDD.filter(lambda x: x[0] not in new_user_rated_movie_ids).map(lambda x: (new_user_ID, x[0]))
>>> new_user_recommendations_RDD = updated_model.predictAll(new_user_unrated_movies_RDD)
```

```
>>> new_user_unrated_movies_RDD.take(3)
[(0, 1), (0, 2), (0, 3)]
>>> new_user_recommendations_RDD.take(2)
[Rating(user=0, product=4704, rating=3.606560950463134), Rating(user=0, product=4844, rating=2.1368358868224036)]
```

Let see some titles

```
.
.
.
>>>
>>> product_rating_RDD = new_user_recommendations_RDD.map(lambda x: (x.product, x.rating))
>>> new_user_recommendations_titled_RDD = product_rating_RDD.join(movies_RDD)
>>> new_user_recommendations_formatted_RDD = new_user_recommendations_titled_RDD.map(lambda x: (x[1][1],x[1][0]))
>>>
>>> top_recomends = new_user_recommendations_formatted_RDD.takeOrdered(10, key=lambda x: -x[1])
>>> for line in top_recomends: print (line)
...
('Maelstr\xf6m (2000)', 6.2119957527973355)
('King Is Alive', 6.2119957527973355)
('Innocence (2000)', 6.2119957527973355)
('Dangerous Beauty (1998)', 6.189751978239315)
('Bad and the Beautiful', 6.005879185976944)
('Taste of Cherry (Ta'm e guilass) (1997)', 5.96074819887891)
('The Lair of the White Worm (1988)', 5.958594728894122)
('Mifune's Last Song (Mifunes sidste sang) (1999)', 5.934820295566816)
('Business of Strangers', 5.899232655788708)
>>>
>>> one_movie_RDD = sc.parallelize([(0, 800)]) # Lone Star (1996)
>>> rating_RDD = updated_model.predictAll(one_movie_RDD)
>>> rating_RDD.take(1)
[Rating(user=0, product=800, rating=4.100848893773136)]
```

Looks like we can sort by value after all!

Behind the scenes takeOrdered() just does the key/value swap and SortByKey that we previously did ourselves.

```
>>> new_user_recommendations_titled_RDD.take(2)
[(111360, (1.0666741148393921, 'Lucy (2014)')), (49530, (1.8020006042285814, 'Blood Diamond (2006)'))]
>>> new_user_recommendations_formatted_RDD.take(2)
[('Lucy (2014)', 1.0666741148393921), ('Blood Diamond (2006)', 1.8020006042285814)]
```

Exercises

1) We noticed that out top ranked movies have ratings higher than 5. This makes perfect sense as there is no ceiling implied in our algorithm and one can imagine that certain combinations of factors would combine to create “better than anything you’ve seen yet” ratings.

Maybe you have a friend that really likes Anime. Many of her ratings for Anime are 5. And she really likes Scarlett Johansson and gives her movies lots of 5s. Wouldn’t it be fair to consider her rating for *Ghost in the Shell* to be a 7/5?

Nevertheless, we may have to constrain our ratings to a 1-5 range. Can you normalize the output from our recommender such that our new users only sees ratings in that range?

2) We haven’t really investigated our convergence rate. We specify 10 iterations, but is that reasonable? Graph your error against iterations and see if that is a good number.

3) I mentioned that our larger dataset does benefit from a rank of 12 instead of 4 (as one might expect). The larger datasets (ratings-large.csv and movies-large.csv) are available to you in ~training/LargeMovies. Prove that the error is less with a larger rank. How does this dataset benefit from more iterations? Is it more effective to spend the computation cycles on more iterations or larger ranks?

4) We could have used the very similar *pyspark.ml.recommendation* API, which uses dataframes. It requires a little more type checking, so we used the classic RDD API *pyspark.mllib.recommendation* instead - for conciseness. Try porting this example to that API. Is this a better way to work?