INTRODUCTION TO DATA SCIENCE

JOHN P DICKERSON

Lecture #25 – 4/27/2017

CMSC320
Tuesdays & Thursdays
3:30pm – 4:45pm

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

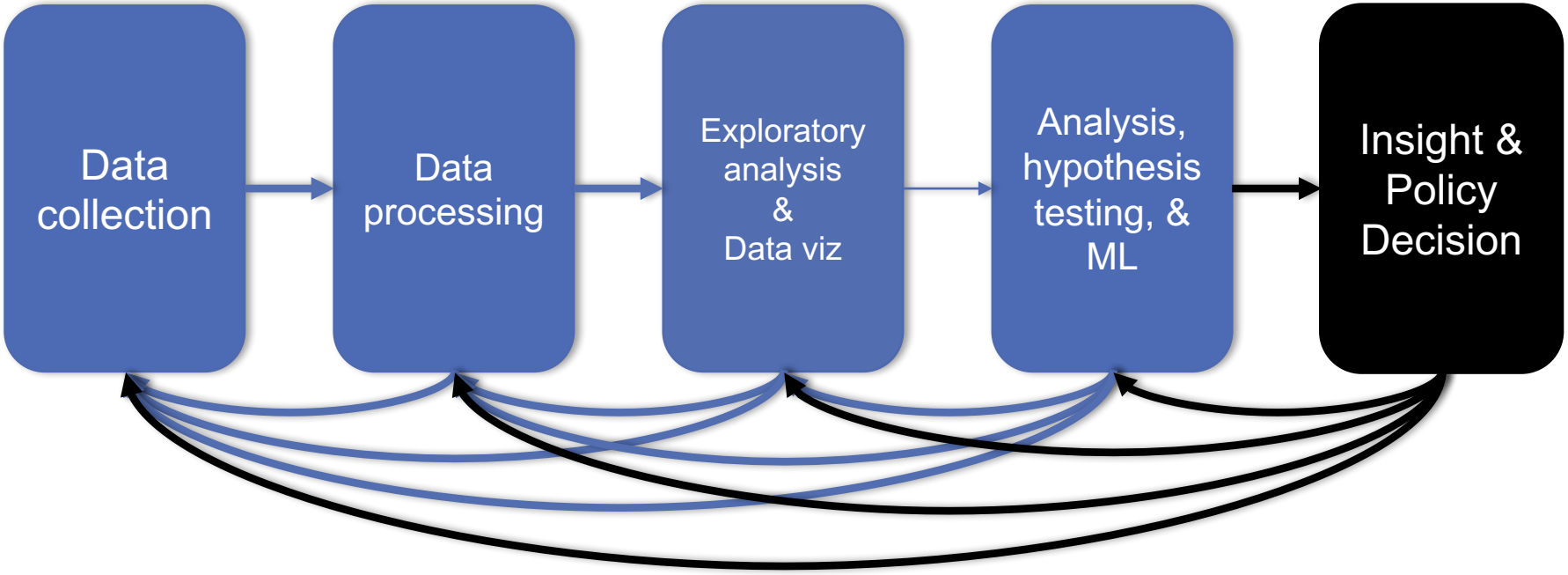
ANNOUNCEMENTS

Fill out the group assignment Google Drive documents:

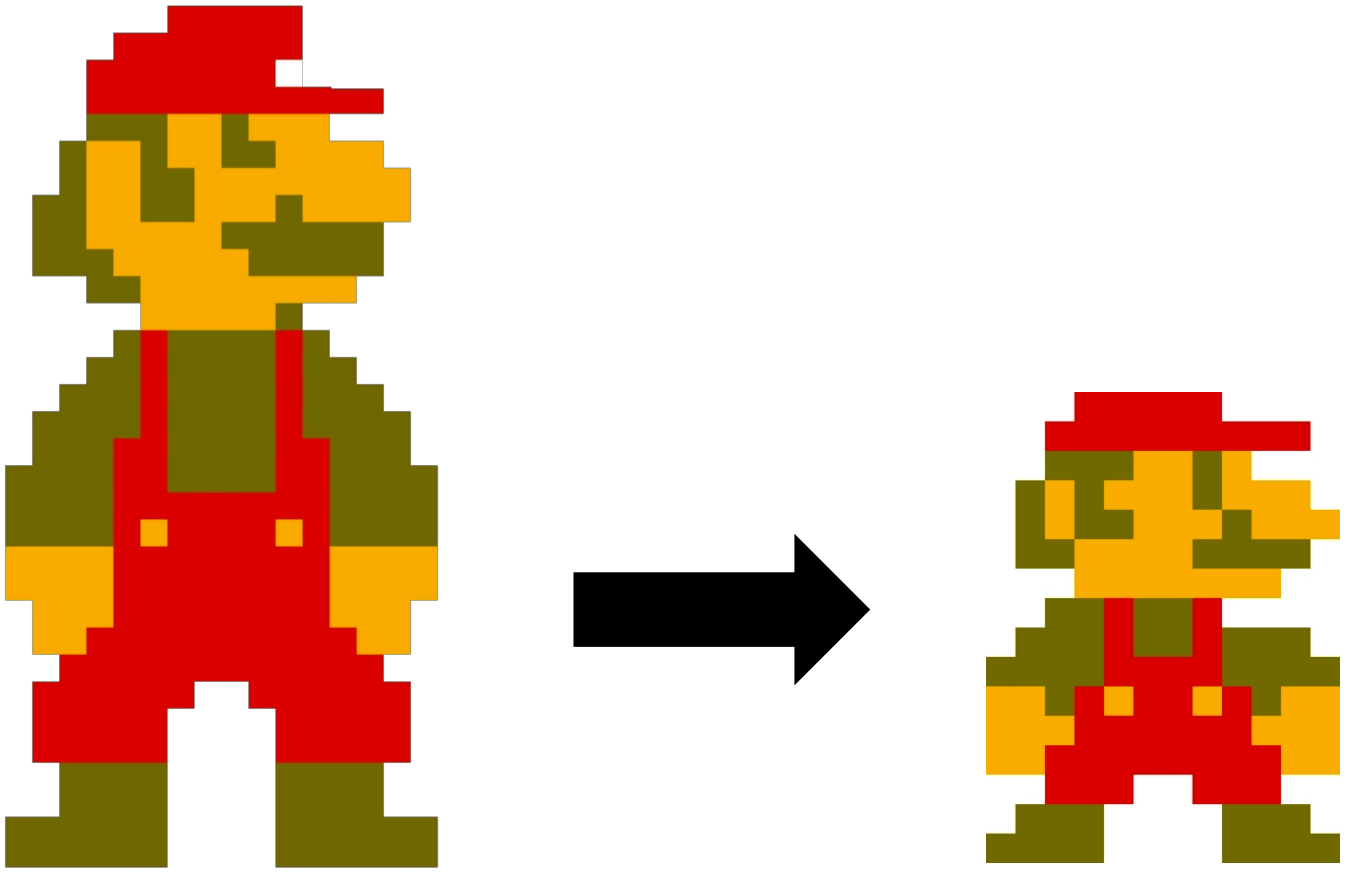
- HW2: https://docs.google.com/spreadsheets/d/1uB_gbTuzFqEzeRkEYHJjEgFNiHwBLsxFv3ptHj8rVBQ/edit#gid=0
- HW3: https://docs.google.com/spreadsheets/d/1dOtND_0QfahrOGpaJn5glb430ccPhA7AFdSoEALPLiM/edit#gid=0

My office hours will be from 10:30-11:30AM tomorrow.

TODAY'S LECTURE



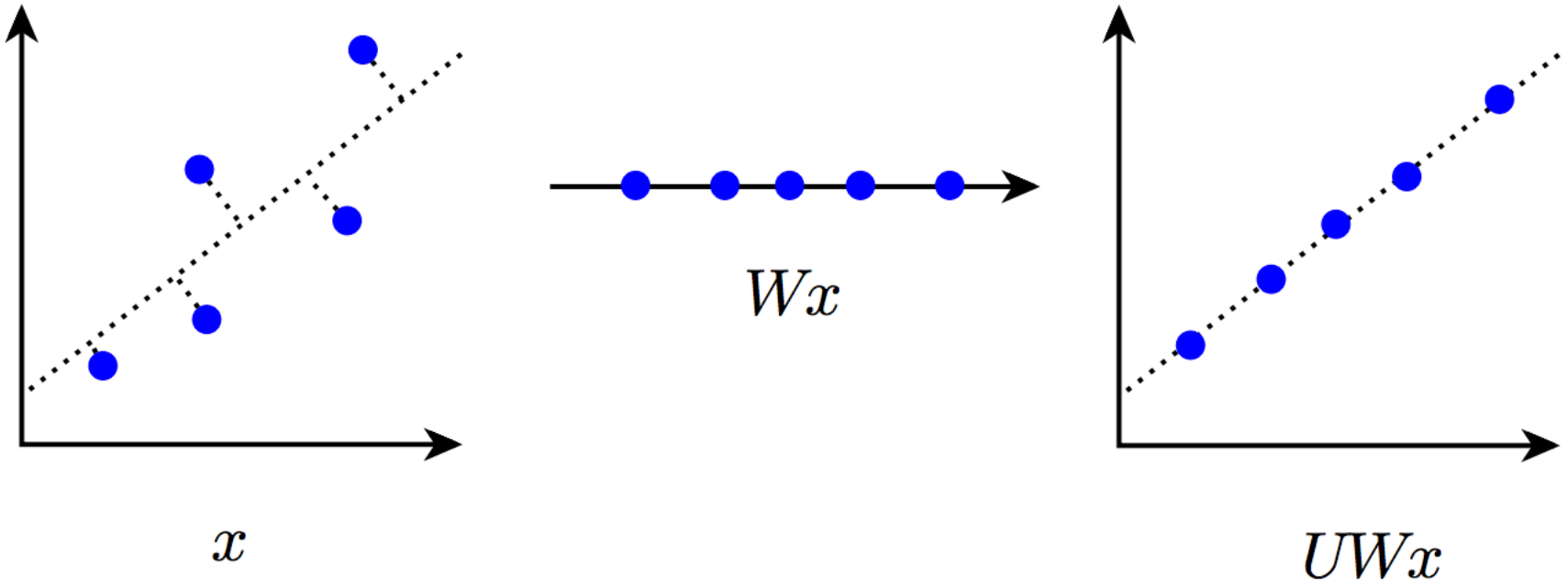
WRAPPING UP LAST LECTURE: DIMENSIONALITY REDUCTION



RECAP: PRINCIPAL COMPONENT ANALYSIS (PCA)

Dimensionality reduction: main use of PCA for data science applications

If $h_\theta(x) = UWx$, then $Wx \in \mathbb{R}^k$ is a reduced (probably with some loss) representation of input features x



RECAP: HOW TO USE PCA & FRIENDS IN PRACTICE

Unsupervised learning methods are useful for EDA

- Cluster or reduce to a few dimensions and visualize!

Also useful as data prep before supervised learning!

1. Run PCA, get W matrix
2. Transform $\tilde{x}^{(i)} = W x^{(i)}$ – (reduce colinearity, dimension)
3. Train and test your favorite supervised classifier

Or use k-means to set up radial basis functions (RBFs):

1. Get k centers $\mu^{(1)}, \ldots, \mu^{(k)}$
2. Create RBF features $\phi_j^{(i)} = \exp\left(-\frac{\|x^{(i)} - \mu^{(j)}\|_2^2}{2\sigma^2}\right)$

(SOME MORE) RECOMMENDER SYSTEMS

QUICK ASIDE: ASSOCIATION RULES

Last time: CF systems give predictions based on other users' scores of the same item

Complementary idea: Find rules that associate the presence of one set of items with that of another set of items

Customers who bought this item also bought

ThinkGeek Plush Unicorn Slippers, One Size, White
4.5 stars 395
$7.77

Adult New Purple Unicorn Onesie Pajamas Kigurumi Cosplay Costumes Animal Outfit
4.5 stars 168
$23.99 - $28.99

EOS ~ Holiday 2015 Limited Edition Decorative Lip Balm Collection
4.5 stars 156
$5.24 - $22.99

FORMAT OF ASSOCIATION RULES

Typical Rule form:

- **Body → Head**
- **Body and Head can be represented as sets of items (in transaction data) or as conjunction of predicates (in relational data)**
- **Support and Confidence**
  - Usually reported along with the rules
  - Metrics that indicate the strength of the item associations

Examples:

- {diaper, milk} → {beer} [support: 0.5%, confidence: 78%]
- buys(x, "bread") /\ buys(x, “eggs") → buys(x, "milk") [sup: 0.6%, conf: 65%]
- major(x, "CS") /\ takes(x, "DB") → grade(x, "A") [1%, 75%]
- age(x,30-45) /\ income(x, 50K-75K) → owns(x, SUV)
- age=“30-45”, income=“50K-75K” → car=“SUV”

Thanks to: Bamshad Mobasher

ASSOCIATION RULES: BASIC CONCEPTS

Let D be database of transactions

Transaction ID	Items
1000	A, B, C
2000	A, B
3000	A, D
4000	B, E, F

Let I be the set of items that appear in the database:

- **e.g., I = {A, B, C, D, E, F}**

Each transaction t is a subset of I

A rule is an implication among itemsets X and Y, of the form by X → Y, where $X \subset I$, $Y \subset I$, and $X \cap Y = \varnothing$

- **e.g.: {B,C} → {A}**

Thanks to: Bamshad Mobasher

ASSOCIATION RULES: BASIC CONCEPTS

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Itemset

- **A set of one or more items**
  - E.g.: {Milk, Bread, Diaper}
- **k-itemset**
  - An itemset that contains k items

Support count (σ)

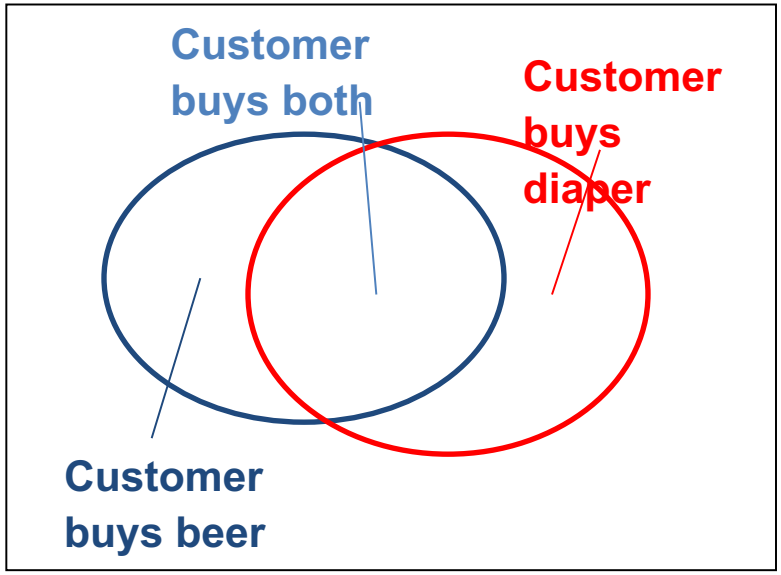
- Frequency of occurrence of an itemset (number of transactions in which it appears)
- E.g. σ({Milk, Bread,Diaper}) = 2

Support

- Fraction of the transactions in which an itemset appears
- E.g. s({Milk, Bread, Diaper}) = 2/5

Frequent Itemset

- An itemset whose support is greater than or equal to a *minsup* threshold



ASSOCIATION RULES: BASIC CONCEPTS

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Association Rule

- X → Y, where X and Y are non-overlapping itemsets
- {Milk, Diaper} → {Beer}

Rule Evaluation Metrics

- **Support (s)**
  - Fraction of transactions that contain both **X** and **Y**
  - **i.e., support of the itemset $X \cup Y$**
- **Confidence (c)**
  - Measures how often items in **Y** appear in transactions that contain **X**

Example:

$$\{\text{Milk}, \text{Diaper}\} \rightarrow \text{Beer}$$

$$s = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{|\text{D}|} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{\sigma(\text{Milk}, \text{Diaper})} = \frac{2}{3} = 0.67$$

ASSOCIATION RULES: INTERESTINGNESS

Another interpretation of support and confidence for *X* → *Y*

- **Support is the probability that a transaction contains {X ∪ Y} or Pr(X /\ Y)**

$$\text{support}(X \rightarrow Y) = \text{support}(X \cup Y) = \sigma(X \cup Y) / |D|$$

- **Confidence is the conditional probability that a transaction will contains Y given that it contains X or Pr(Y | X)**

$$\text{confidence}(X \rightarrow Y) = \sigma(X \cup Y) / \sigma(X)$$
$$= \text{support}(X \cup Y) / \text{support}(X)$$

ASSOCIATION RULES: INTERESTINGNESS

Other considerations of how interesting a rule is:

$$\text{lift}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X) \times \text{supp}(Y)}$$

If lift is equal to 1 ??????????

- Body X and Head Y are independent

If lift is greater than 1 ?????????

- Body X and Head Y are in some sense dependent

Conviction measures frequency of X and Y occurring together, vs. how frequently X occurs but not Y

Many others …

ASSOCIATION RULES IN PRACTICE

Orange3 is a {GUI, Python API, …} that:

- Enumerates frequent itemsets
- Performs association rule mining
- (Wrapper calls to, shared functionality with, Scikit-Learn)

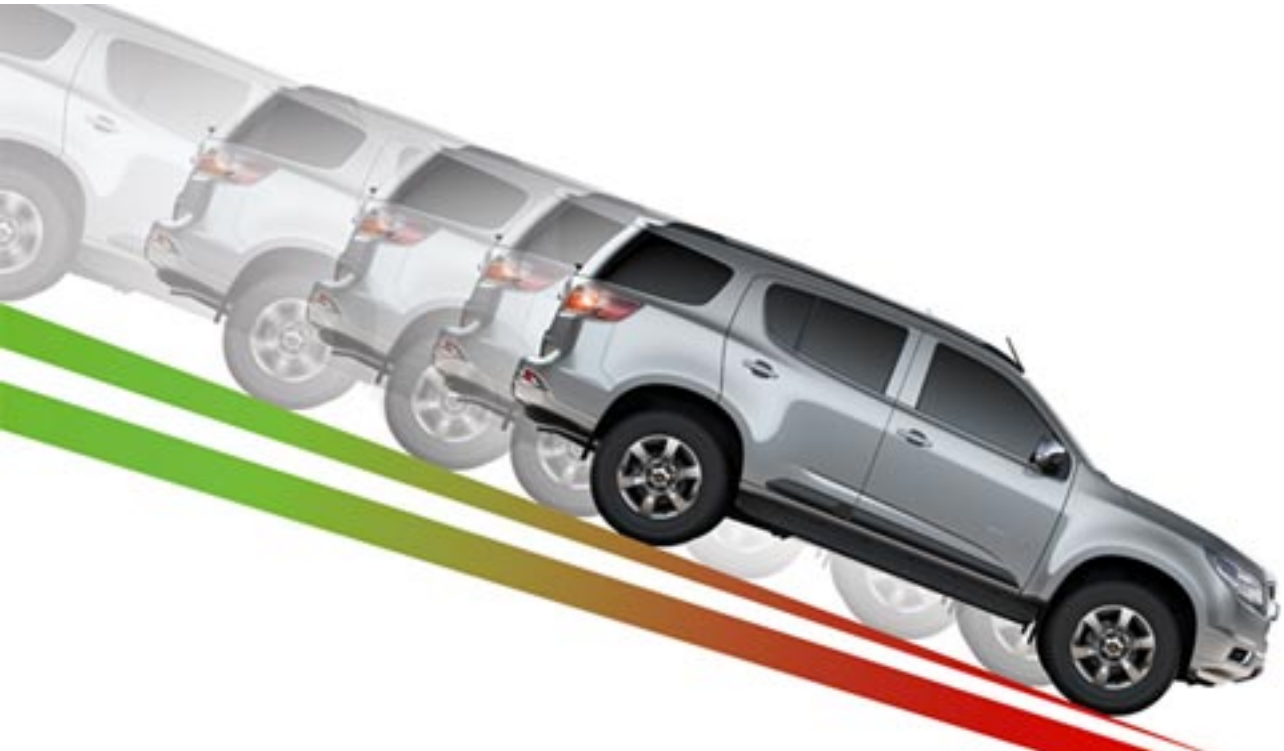
```
conda install –c ales-erjavec orange3
```

More information:
https://blog.biolab.si/2016/04/25/association-rules-in-orange/

In general:

- **Can be useful for interpretable, fast data mining**
- **Typically doesn't consider order, scalability issues …**

SCALING IT UP: STOCHASTIC GRADIENT DESCENT (SGD)



RECAP: GRADIENT DESCENT

Algorithm for any* hypothesis function $h_\theta: \mathbb{R}^n \to \mathcal{Y}$, loss function $\ell: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, step size α :

Initialize the parameter vector:

- $\theta \leftarrow 0$

Repeat until satisfied (e.g., exact or approximate convergence):

- **Compute gradient:** $g \leftarrow \sum_{i=1}^{m} \nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$
- **Update parameters:** $\theta \leftarrow \theta - \alpha \cdot g$

What if m is big?
What if n is big?

*must be reasonably well behaved

STOCHASTIC GRADIENT DESCENT

Algorithm for any* hypothesis function $h_\theta: \mathbb{R}^n \to \mathcal{Y}$, loss function $\ell: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$, step size α :

Initialize the parameter vector:

- $\theta \leftarrow 0$

Repeat until satisfied (e.g., exact or approximate convergence):

- **Randomly shuffle the input set x**
- **For i in {1, 2, … m}, shuffled:**
  - Compute gradient: $g \leftarrow \nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$
  - Update parameters: $\theta \leftarrow \theta - \alpha \cdot g$

*must be reasonably well behaved

SGD CONTINUED

Can also make use of "mini-batch" stochastic gradient descent:

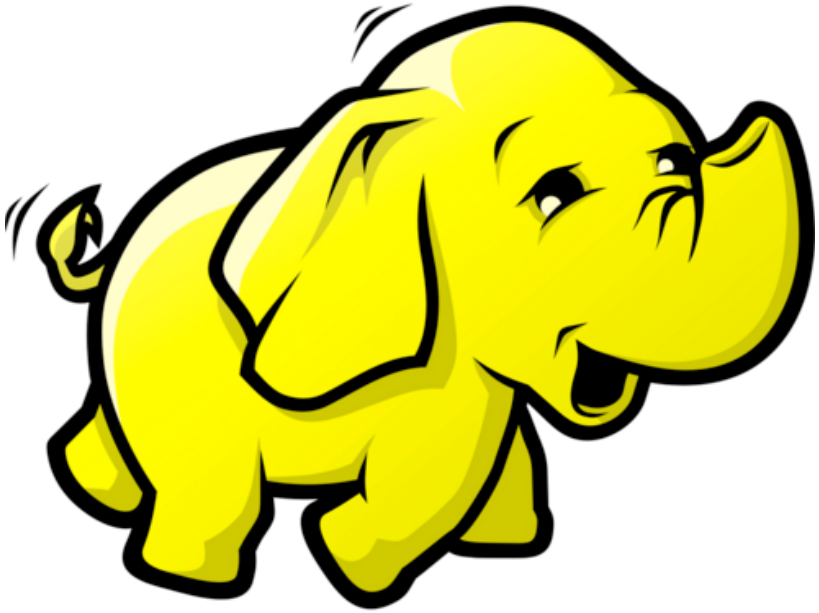
- High-level idea: at every outer iteration, shuffle the input data, then partition into k mini-batches of size m/k
- Perform inner loop of SGD on these mini-batches

Batches reduce variance in update, make use of vectorization

Issues with SGD ??????????????

- Convergence: "almost surely" converges to the global optimum, assuming convexity and reasonable learning rate
- Batch size: hyperparameter? Not really – figure out how many examples fit in RAM/GPU memory, then choose the nearest power of 2 and go with that
- Sensitive to feature scaling (if batching over features)

SCALING IT UP: BIG DATA & MAPREDUCE



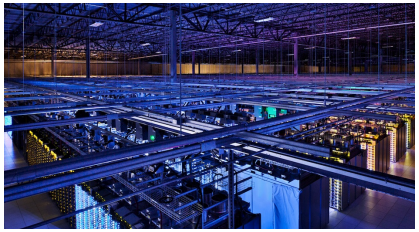
Thanks to: Jeff Dean, Sanjay Ghemawa, Zico Kolter

“Big data”

My laptop
8GB RAM
500GB Disk

Big data?
No

?



Google Data Center
??? RAM/Disk
(>> PBs)

Big data?
Yes

Some notable inflection points

1. Your data fits in RAM on a single machine

2. Your data fits on disk on a single machine

3. Your data fits in RAM/disk on a “small” cluster of machines (you don’t need to worry about machines dying)

4. Your data fits in RAM/disk on a “large” cluster of machine (you need to worry about machines dying)

It’s probably reasonable to refer to 3+ as “big data”, but many would only consider 4

Do you have big data?

If your data fits on a single machine (even on disk), then it's almost always better to think about how you can design an efficient single-machine solution, unless you have extremely good reasons for doing otherwise

scalable system	cores	twitter	uk-2007-05
GraphChi [10]	2	3160s	6972s
Stratosphere [6]	16	2250s	-
X-Stream [17]	16	1488s	-
Spark [8]	128	857s	1759s
Giraph [8]	128	596s	1235s
GraphLab [8]	128	249s	833s
GraphX [8]	128	419s	462s
Single thread (SSD)	1	300s	651s
Single thread (RAM)	1	275s	-

Table 2: Reported elapsed times for 20 PageRank iterations, compared with measured times for single-threaded implementations from SSD and from RAM. GraphChi and X-Stream report times for 5 PageRank iterations, which we multiplied by four.

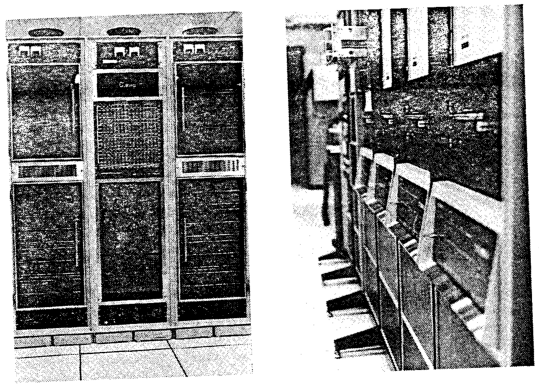
scalable system	cores	twitter	uk-2007-05
Stratosphere [6]	16	950s	-
X-Stream [17]	16	1159s	-
Spark [8]	128	1784s	$\geq$ 8000s
Giraph [8]	128	200s	$\geq$ 8000s
GraphLab [8]	128	242s	714s
GraphX [8]	128	251s	800s
Single thread (SSD)	1	153s	417s

Table 3: Reported elapsed times for label propagation, compared with measured times for single-threaded label propagation from SSD.

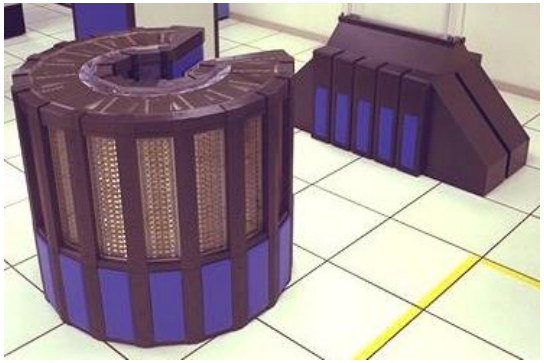
Tables from [McSherry et al., 2015 "Scalability! But at what COST"]

Distributed computing

Distributed computing rose to prominence in the 70s/80s, often built around “supercomputing,” for scientific computing applications



1971 – CMU C.mmp
(16 PDP-11 processors)



1984 – Cray-2
(4 vector processors)

Message passing interface

In mid-90s, researchers built a common interface for distributed computing called the message passing interface (MPI)

MPI provided a set of tools to run multiple processes (on a single machine or across many machines), that could communicate, send data between each other (all of “scattering”, “gathering”, “broadcasting”), and synchronize execution

Still common in scientific computing applications and HPC (high performance computing

Downsides to MPI

MPI is extremely powerful but has some notable limitations

1. MPI is complicated: programs need to explicitly manage data, synchronize threads, etc

2. MPI is brittle: if machines die suddenly, can be difficult to recover (unless explicitly handled by the program, making them more complicated)

A new paradigm for data processing

When Google was building their first data centers, they used clusters of off-the-shelf commodity hardware; machines had different speeds and failures were common given cluster sizes

Data itself was distributed (redundantly) over many machines, as much as possible wanted to do the computation on the machine where the data is stored

Led to the development of the MapReduce framework at Google [Ghemawat, 2004], later made extremely popular through the Apache Hadoop open source implementation

AN EXAMPLE PROGRAM

Present the concepts of MapReduce using the "typical example" of MR, Word Count

- Input: a volume of raw text, of unspecified size (could be KB, MB, TB, it doesn't matter!)
- Output: a list of words, and their occurrence count.

(Assume that words are split correctly; ignore capitalization and punctuation.)

Example:

- **The doctor went to the store. =>**
  - The, 2
  - Doctor, 1
  - Went, 1
  - To, 1
  - Store, 1

MAP? REDUCE?

Mappers read in data from the filesystem, and output (typically) modified data

Reducers collect all of the mappers output on the keys, and output (typically) reduced data

The outputted data is written to disk

All data is in terms of key-value pairs (“The” → 2)

MAPREDUCE VS HADOOP

The paper is written by two researchers at Google, and describes their programming paradigm

Unless you work at Google, or use Google App Engine, you won't use it! (And even then, you might not.)

Open Source implementation is Hadoop MapReduce

- **Not developed by Google**
- **Started by Yahoo!; now part of Apache**

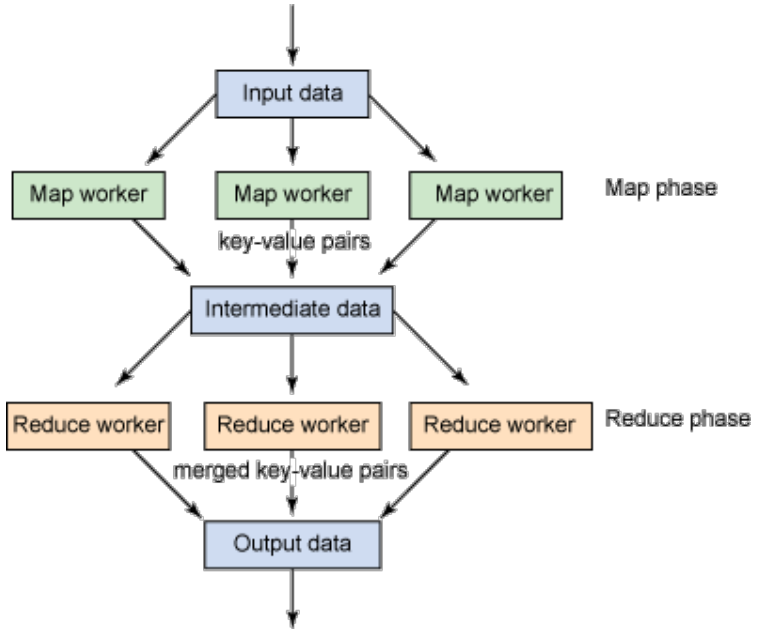
Google's implementation (at least the one described) is written in C++

Hadoop is written in Java

MAJOR COMPONENTS

User Components:

- Mapper
- Reducer
- Combiner (Optional)
- Partitioner (Optional) (Shuffle)
- Writable(s) (Optional)



System Components:

- Master
- Input Splitter*
- Output Committer*
- * You can use your own if you really want!

Image source: http://www.ibm.com/developerworks/java/library/l-hadoop-3/index.html

KEY NOTES

Mappers and Reducers are typically single threaded and deterministic

- Determinism allows for restarting of failed jobs, or speculative execution

Need to handle more data? Just add more Mappers/Reducers!

- No need to handle multithreaded code
- Since they're all independent of each other, you can run (almost) arbitrary number of nodes

Mappers/Reducers run on arbitrary machines. A machine typically multiple map and reduce slots available to it, typically one per processor core

Mappers/Reducers run entirely independent of each other

- In Hadoop, they run in separate JVMs

BASIC CONCEPTS

All data is represented in key-value pairs of an arbitrary type

Data is read in from a file or list of files, from distributed FS

Data is chunked based on an input split

- A typical chunk is 64MB (more or less can be configured depending on your use case)

Mappers read in a chunk of data

Mappers emit (write out) a set of data, typically derived from its input

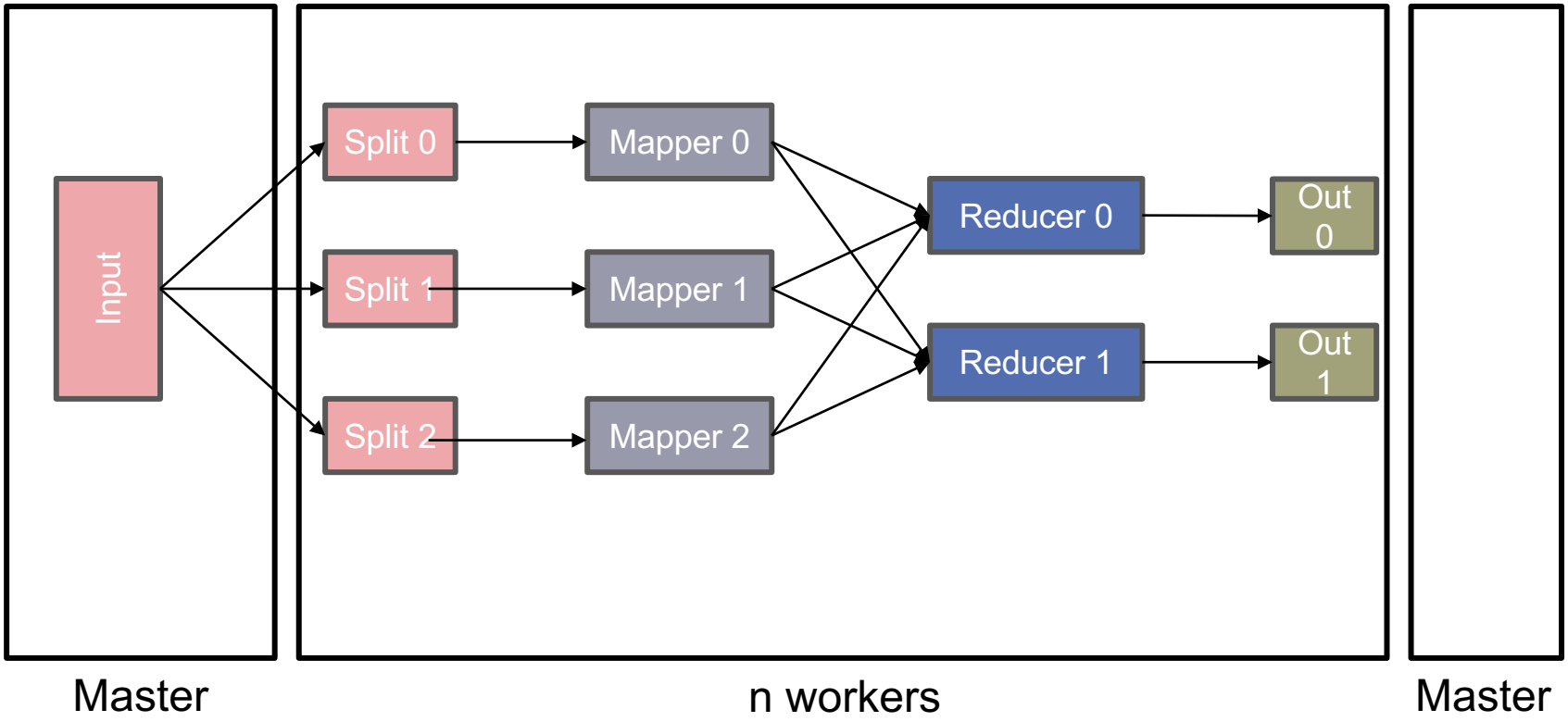
Intermediate data (the output of the mappers) is split to a number of reducers

Reducers receive each key of data, along with ALL of the values associated with it (this means each key must always be sent to the same reducer)

- Essentially, <key, set<value>>

Reducers emit a set of data, typically reduced from its input which is written to disk

DATA FLOW


Input
Split 0
Split 1
Split 2
Mapper 0
Mapper 1
Mapper 2
Reducer 0
Reducer 1
Out 0
Out 1
Master
n workers
Master

INPUT SPLITTER

Is responsible for splitting your input into multiple chunks

These chunks are then used as input for your mappers

Splits on logical boundaries. The default is 64MB per chunk

- Depending on what you're doing, 64MB might be a LOT of data! You can change it

Typically, you can just use one of the built in splitters, unless you are reading in a specially formatted file

MAPPER

Reads in input pair <K,V> (a section as split by the input splitter)

Outputs a pair <K', V'>

Ex. For our Word Count example, with the following input: "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am."

The output would be:

- <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

REDUCER

Accepts the Mapper output, and collects values on the key

- All inputs with the same key must go to the same reducer!

Input is typically sorted, output is output exactly as is

For our example, the reducer input would be:

- <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

The output would be:

- <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

COMBINER

Essentially an intermediate reducer

- Is optional

Reduces output from each mapper, reducing bandwidth and sorting

Cannot change the type of its input

- Input types must be the same as output types

OUTPUT COMMITTER

Is responsible for taking the reduce output, and committing it to a file

Typically, this committer needs a corresponding input splitter (so that another job can read the input)

Again, usually built in splitters are good enough, unless you need to output a special kind of file

PARTITIONER (SHUFFLER)

Decides which pairs are sent to which reducer

Default is simply:

- Key.hashCode() % numOfReducers

User can override to:

- Provide (more) uniform distribution of load between reducers
- Some values might need to be sent to the same reducer
  - Ex. To compute the relative frequency of a pair of words <W1, W2> you would need to make sure all of word W1 are sent to the same reducer
- Binning of results

MASTER

Responsible for scheduling & managing jobs

Scheduled computation should be close to the data if possible

- Bandwidth is expensive! (and slow)
- This relies on a Distributed File System (e.g. GFS)!

If a task fails to report progress (such as reading input, writing output, etc), crashes, the machine goes down, etc, it is assumed to be stuck, and is killed, and the step is re-launched (with the same input)

The Master is handled by the framework, no user code is necessary

MAPREDUCE IN PYTHON

```
def mapreduce_execute(data, mapper, reducer):
    values = map(mapper, data)

    groups = {}
    for items in values:
        for k,v in items:
            if k not in groups:
                groups[k] = [v]
            else:
                groups[k].append(v)

    output = [reducer(k,v) for k,v in groups.items()]
    return output
```

MAPREDUCE IN PYTHON

Don't do the last slide …

Python's `mrjob` library:

- write mappers and reducers in Python
- Deploy on Hadoop systems, Amazon Elastic MR, Google Cloud

```
from mrjob.job import MRJob

class WordOccurrenceCount(MRJob):
    def mapper(self, _, line):
        for word in line.split(" "):
            yield word, 1

    def reducer(self, key, values):
        yield key, sum(values)
```

MAPREDUCE?

Good:

- All you need to do is write a mapper and a reducer
- Can get away with not exposing any of the internals (data splitting, locality issues, redundancy, etc) if you're using a ready-made engine

Bad:

- Lots of reading/writing from disk (in part because this helps with redundancy)
- Sometimes communication between processes is necessary
- Talk about later: parameter servers, GraphLab aka Dato, etc

NEXT CLASS:

SCALING IT UP: (MORE) BIG DATA & MAPREDUCE & DISTRIBUTED COMPUTATION