

MAPREDUCE, DISTRIBUTED FILE SYSTEMS, HADOOP, AND DATA MINING

Chris Jermaine
cmj4@cs.rice.edu
Rice U.

The Plan for Today

1. Start with an intro to MapReduce and Hadoop

- Then have an activity where you will set up your own Hadoop cluster
- Your cluster will be on machines rented from Amazon's EC2 service

2. Then an overview of writing MapReduce programs

- Then have an activity where you will write a simple MapReduce code

3. Overview of some “data mining” algs for “big data”

- Then you will implement a simple algorithm (KMeans clustering) for a single machine

4. Talk about how this alg could be implemented over MapReduce

- Then use your single-machine imp. as a basis for a MapReduce imp.

5. Implement one more algorithm over MapReduce

- “KNN” classification

6. Finish up with a brief discussion of some other tools

10 Years Ago...

- Say you had a “big” data set, wanted general purpose platform to analyze it
 - Analysis: report generation, customer profiling, statistical modelling, etc.
- What do I mean by “big”?
 - Too large to fit in aggregate RAM of a standard distributed/parallel system
- Big is perhaps 100GB+ in 2002, 10TB+ 2012 (all the way up to dozens of PB)
- Key: “too large to fit in aggregate RAM”
 - Generally rules out the classic HPC paradigm
 - HPC is generally unconcerned with secondary storage
 - Plus, what happens when the power goes out?

Hardware: Agreement on “Shared Nothing”

- Store/analyze data on a large number of commodity machines
- Local, non-shared storage attached to each of them
- Only link is via a LAN
- “Shared nothing” refers to no sharing of RAM, storage
- Why preferred?
 - Inexpensive: built out of commodity components (same stuff as in desktop PCs)
 - You are leveraging price/tech wars among Dell, HP, Intel, AMD, etc.
 - Compute resources scales nearly linearly with \$\$
 - Contrast this to a shared RAM machine with uniform memory access

But What About the Software?

- 10 years ago, you'd have two primary options
 - Put your data into an SQL database, or
 - Roll your own software stack to perform your analysis

Clearly, Building Own Software Not Desirable

- Costly, time consuming
 - A \$10M software feature might eat up most of the IT budget for a single firm
 - But Oracle can spread those costs across 100K customers
- Requires expertise not always found in house
- Risky: high potential for failure

But People Not Happy With SQL Databases

- Also quite expensive: even today, pay \$10K and up/year/TB
- Performance often unpredictable, or just flat out poor
 - Only now are there systems that mortals can get to work in TB+ range
- In 2005, not a lot of options for commodity shared nothing
- Software insanely complicated to use correctly
 - Hundreds or even thousands of “knobs” to turn
- Software stack too big/deep, not possible to unbundle
 - If you are doing analysis, ACID not important
 - And yet, you pay for it (\$\$, complexity, performance)
- Difficult to put un- or semi-structured data into an SQL DB
 - How does an archive of 10M emails get put into a set of relations?

And, Many People Just Don't Like SQL

- It is “declarative”
 - In some ways, very nice, since parallelism is implicit
 - But user doesn't really know what's happening under the hood... people don't like
- Also, not easy/natural to specify important computations
 - Such as Google's PageRank, or rule mining, or data clustering, etc.

By Early-Mid 2000's...

- The Internet companies (Google, Yahoo!, etc.)...
 - ...had some of the largest databases in the world
- But they had never used classical SQL databases for webscale data
- How'd they deal with all of the data they had to analyze?
 - Many ways
 - But paradigm with most widespread impact was **MapReduce**
 - First described in a 2004 academic paper, appeared in OSDI
 - Easy read:
<http://research.google.com/archive/mapreduce.html>

What Is MapReduce?

- It is a simple data processing paradigm
- To process a data set, you have two pieces of user-supplied code:
 - A **map** code
 - And a **reduce** code
- These are run (potentially over a large compute cluster) using three data processing phases
 - A **map** phase
 - A **shuffle** phase
 - And a **reduce** phase

The Map Phase

- Assume that the input data are stored in a huge file

This file contains a simple list of pairs of type $(key1, value1)$

- And assume we have a user-supplied function of the form

`map (key1, value1)`

- That outputs a list of pairs of the form $(key2, value2)$

- In the **map** phase of the MapReduce computation

- this `map` function is called for every record in the input data set
- Instances of `map` run in parallel all over the compute cluster

The Shuffle Phase

- The **shuffle** phase accepts all of the `(key2, value2)` pairs from the **map** phase
- And it groups them together
- So that all of the pairs
 - From all over the cluster
 - Having the same `key2` value
 - Are merged into a single `(key2, list <value2>)` pair
- Called a **shuffle** because this is where a potential all-to-all data transfer happens

The Reduce Phase

- Assume we have a user-supplied function of the form

```
reduce (key2, list <value2>)
```

- That outputs a list of `value2` objects
- In the **reduce** phase of the MapReduce computation
 - this `reduce` function is called for every `key2` value output by the shuffle
 - Instances of `reduce` run in parallel all over the compute cluster
 - The output of all of those instances is collected in a (potentially) huge output file

The Distributed File System

- Now MapReduce is a **compute** paradigm
- It is not a **data storage** paradigm
- But any MapReduce system must read/write data from some storage system
- As a result, the MapReduce programming paradigm is tightly integrated with the idea of a **distributed file system (DFS)**
- A DFS is a storage system that allows data to be stored/accessed across machines in a network
- And abstracts away differences between local and remote data
 - Same mechanism to read/write data
 - No matter where data is located in the network

Distributed File Systems for MR

- DFSs have been around for a long time
 - First widely used DFS was Sun's NFS, first introduced in 1985
- How is a DFS for MapReduce going to be different?
- Unlike classical DFSs, it sits on top of each machine's OS
- The OS is not aware of the DFS; you can't "mount" it anywhere
 - So the files in the DFS are not accessible from "My Computer" in Windows
- Why "on top of" rather than "in" the OS?
 - Ease of use, portability, means don't have worries with a heterogeneous cluster
 - Just start up a process on each machine in the cluster
 - No need to tell the OS about anything
 - Means you can have a DFS up and running on a cluster in minutes/hours

Distributed File Systems for MR

- But (in theory) they still give you most of what a classic DFS does
- Replication
 - Put each block at n locations in the cluster
 - That way, if a disk/machine goes down, you are still OK
- Network awareness
 - Smart enough to try to satisfy a data request locally, or from same rack
- Easy to add/remove machines
 - You buy 10 more machines, just tell the DFA about them, and it'll add data to 'em
 - Can take machines off the network; no problem, DFS will realize this and handle
- Load balancing
 - If one machine is getting hit, go to another machine that has the data

Take Home Message From Last 10 Slides

- MapReduce is a distributed programming paradigm
- Needs to run on top of some storage system--a DFS
- DFS should be lightweight, easy to install, OS agnostic
- Thus, you can expect most MR softwares to be tightly integrated with a particular DFS
 - And that DFS will typically run **on top of** the OS of each machine

MapReduce Has Had a Huge Impact

- One of the key technologies in the “NoSQL movement”
- What do people like about it?

Why Popular? (1)

- Schema-less
- You write code that operates over raw data
- No need to spend time/\$\$ loading the data into a database system
- Really nice for unstructured/semi-structured data: text, logs, for example

Why Popular? (2)

- Easier to code than classical HPC system
- Distributed/parallel computing **very** difficult to program correctly
 - pthreads, MPI, semaphores, monitors, condition variables...
 - All are hard to use!
- But MapReduce is a super-simple compute model
- All communication is done during shuffle phase
- All scheduling is taken care of by the MapReduce system
- Radically reduces complexity for the programmer

Why Popular? (3)

- Much more control than an SQL database
- You write the actual code that touches the data
 - In a standard language, such as Java
- You control what happens during the map and reduce phases
- Contrast to an SQL database
 - Where you write SQL that can be compiled to an arbitrary execution plan

Why Popular? (4)

- Fits very nicely into the “cloud computing” paradigm
- Why? So simple, lightweight, hardware agnostic
- Need to run a MapReduce computation?
 - Just rent a bunch of machines from Amazon
 - Give ‘em back when you are done
- Contrast this with a multi-terabyte Oracle database
 - Most SQL databases are NOT lightweight and simple to get going in a few mins

Why Popular? (5)

- Software is free!
- At least, Apache Hadoop is
- Hadoop is a widely-used open source MapReduce implementation
- Slowly being supplanted by other options (Spark, for example)

Hadoop

- Given discussion so far, not surprising that Hadoop has two major components:
 - The Hadoop distributed file system
 - And a MapReduce engine

Hadoop and “The Cloud”

- Because Hadoop (and other tools such as Spark) allow you to use many machines working together...
- ...they are often closely associated with “The Cloud”
- “The Cloud”: computation as a utility
 - Buy as much compute power/storage as you need to solve the task at hand
 - Want to solve problem in 1/10 the time?
 - Buy 10X the compute power, and use Hadoop (or similar tool) to harness it
 - Same cost, but you get the answer much faster!
- Cloud computing is “elastic”: use as much as you need
 - Pay for what you use
 - No fixed costs!
 - Great when your requirements quickly evolve

Is the Cloud A Good Idea?

- Should you/could you ask someone (Microsoft, Amazon, Google) to store your data?
- Several concerns:
 - How to load it up?
 - Security/privacy
 - Persistence
 - Cost

Loading it Up

- Amazon has a very fast Internet connection!
- Undoubtedly, your connection will be the limiting factor
 - At 100Mbps, 1 day to upload a terabyte
 - At 1000Mbps, 2 hours to upload a terabyte
- What if you have even more to move into the cloud?
 - Amazon will accept your storage devices via FedEx

Security/Privacy

- This is a serious concern
- Amazon Web Services is probably more secure than you are
 - Anonymous couldn't hurt them after they booted WikiLeaks
 - But this does not change the fact you are giving your data to someone else
- If something *did* happen
 - Amazon is far less accountable to you than your IT people are to your CEO
 - You would potentially be one of many on the outside looking in

Persistence

- Someone like Amazon is very good at not losing data
- Hadoop is also quite good at not losing data
 - If you replicate data many times, you are quite fault tolerant
 - People do use HDFS as a primary data store
- But is Amazon + Hadoop DFS a good solution?
 - You'd want to put your data on local disk
 - But local storage is lost when an instance spins down (machines are virtual!)
 - And stuff happens: power outages, faults requiring reboot, etc.
 - So should view local disk in EC2 like RAM on your own machines
- That said, are lots of cloud-based options for storage
 - Amazon EBS (Elastic Block Storage), Amazon S3,...
 - But annoying to have to back up a Hadoop cluster!

Cost

- Amazon EBS (tightly coupled with EC2)
 - \$.10 per GB/month SSD (\$100 per terabyte/month)
- Amazon S3
 - High redundancy: handles data loss at two Amazon data centers... from \$30 per terabyte/month down to \$27 per terabyte month if you store > 5PB
 - Medium redundancy: handles data loss at one data center... from \$24 per terabyte down to \$22 per terabyte
- Amazon EC2: \$.50/hour (desktop eq.) \$2/hour small server
- Is this expensive?
 - I have personally had many arguments about this. At first glance, it seems so...
 - But especially in a university, hard to truly account for facilities, power, OH, etc.

Activity One

- Today, we'll be using Amazon EC2
- Set up a Hadoop cluster on EC2 and run a MapReduce job

See <http://cmj4.web.rice.edu/GettingStarted.html> for the description

Writing a Hadoop MapReduce Program

- A Hadoop program needs:
 - A Java class with a `main` that configures and submits the job
 - A class that extends the `Hadoop Mapper` class (The “Mapper”)
 - A class that extends the `Hadoop Reducer` class (The “Reducer”)
 - Optionally: A class that extends the `Hadoop Reducer` class (The “Combiner”)

The Main Class

- You'll be studying an example in a minute! What does it do?
- In the Main class you first create a Configuration object:

```
Configuration conf = new Configuration ();
```
- This is basically a map from String objects to String objects
- It is used to configure the MapReduce job
 - Useful for communicating configuration to mappers and reducers

The Main Class (cont'd)

- Then you build a `Job` object out of the `Configuration` object

```
Job job = new Job (conf);
```

- A `Job` is a runnable configuration
- Wraps up the `Configuration` object that was used to create it
- But has nice interface to add a bunch of stuff to it:

— What mapper/reducer to use

```
job.setMapperClass (WordCountMapper.class);
```

```
job.setReducerClass (WordCountReducer.class);
```

The Main Class (cont'd)

- Then you build a `Job` object out of the `Configuration` object

```
Job job = new Job (conf);
```

- A `Job` is a runnable configuration
- Wraps up the `Configuration` object that was used to create it
- But has nice interface to add a bunch of stuff to it:
 - What mapper/reducer to use
 - What the `InputFormatClass` is (tells Hadoop how to process input data)
`job.SetInputFormatClass (TextInputFormat.class);`

The Main Class (cont'd)

- Then you build a `Job` object out of the `Configuration` object

```
Job job = new Job (conf);
```

- A `Job` is a runnable configuration
 - Wraps up the `Configuration` object that was used to create it
 - But has nice interface to add a bunch of stuff to it:
 - What mapper/reducer to use
 - What the `InputFormatClass` is (tells Hadoop how to process input data)
 - What the `OuputFormatClass` is (tells Hadoop how to write out output data)
- ```
job.SetInputFormatClass (TextInputFormat.class);
```

## The Main Class (cont'd)

- Then you build a `Job` object out of the `Configuration` object

```
Job job = new Job (conf);
```

- A `Job` is a runnable configuration
  - Wraps up the `Configuration` object that was used to create it
  - But has nice interface to add a bunch of stuff to it:
    - What mapper/reducer to use
    - What the `InputFormatClass` is (tells Hadoop how to process input data)
    - What the `OuputFormatClass` is (tells Hadoop how to write out output data)
    - How many reducers to use
- ```
job.setNumReduceTasks (num);
```

The Main Class (cont'd)

- Then you build a `Job` object out of the `Configuration` object

```
Job job = new Job (conf);
```

- A `Job` is a runnable configuration
- Wraps up the `Configuration` object that was used to create it
- But has nice interface to add a bunch of stuff to it:
 - What mapper/reducer to use
 - What the `InputFormatClass` is (tells Hadoop how to process input data)
 - What the `OuputFormatClass` is (tells Hadoop how to write out output data)
 - How many reducers to use
 - What `.jar` file to send around the cluster

```
job.setJarByClass (WordCount.class);
```

Many others! But these only configured if default is not OK

The Main Class (cont'd)

- Then it runs the job

```
job.waitForCompletion (true); // true means you print out progress info to the
```

- That's it!

The Main Class (cont'd)

- Then it runs the job

```
job.waitForCompletion (true); // true means you print out progress info to the
```

- That's it!
- Well, sort of....
- As you'll see when you work on the next activity, generally need to configure the `InputFormatClass` and the `OutputFormatClass`

```
TextInputFormat.setInputPaths (job, path);  
TextInputFormat.setMinInputSplitSize (job, value1);  
TextInputFormat.setMaxInputSplitSize (job, value2);
```

- This code asks the `TextInputFormat` class to write to the `Configuration` object inside of the `Job` object

The Mapper Class

- Your mapper must extend the base `Mapper` class. Ex:

```
public class MyMapper extends Mapper <LongWritable, Text, Text, IntWritable> {}
```

- First two type params (`LongWritable` and `Text`)...
 - ...specify the (key, value) pairs that the map tasks will process
 - Must match the output of the `FileInputFormat` that you are using
 - Ex: `TextInputFormat` spews out (`LongWritable`, `Text`) pairs
 - The first half of the pair is the position in the input text file
 - The second half is a line from the text file
 - `LongWritable` and `Text` are writable Hadoop versions of `Long`, `String`

The Mapper Class (cont'd)

- Must extend the `Mapper` class. Ex:

```
public class MyMapper extends Mapper <LongWritable, Text, Text, IntWritable> {}
```

- The second two type params (`Text` and `IntWritable`)...
 - ...specify the (key, value) pairs that will be sent to the reducer

The Mapper Class (cont'd)

- The code for the base Mapper class is as follows:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    // Called once at the beginning of the task.  
    protected void setup(Context context) throws IOException, InterruptedException { /*nothing*/ }  
    // Called once for each key/value pair in the input split. Most applications should override this...  
    protected void map(KEYIN key, VALUEIN value, Context context) throws IOException, InterruptedException {  
        context.write((KEYOUT) key, (VALUEOUT) value);  
    }  
    // Called once at the end of the task.  
    protected void cleanup(Context context) throws IOException, InterruptedException { /*nothing*/ }  
    // Expert users can override this method for more complete control over the execution of the Mapper.  
    public void run(Context context) throws IOException, InterruptedException {  
        setup(context);  
        while (context.nextKeyValue()) {  
            map(context.getCurrentKey(), context.getCurrentValue(), context);  
        }  
        cleanup(context);  
    }  
}
```

The Reducer Class

- Your mapper must extend the base `Reducer` class. Ex:

```
public class MyReducer extends Reducer <Text, IntWritable, Text, IntWritable> {}
```

- First two type params (`Text` and `IntWritable`)...

- Must match the output of the map task

- Second two type params (`Text` and `IntWritable`)...

- Are what is written to the output of the MapReduce program

- Must match the output of the `FileOutputFormat` that you are using

- Ex: `TextOutputFormat` is a generic that can output anything as a line of text

- So it can write out (`Text`, `IntWritable`) pairs

The Reducer Class (cont'd)

- The code for the base `Reducer` class is as follows:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {  
    protected void setup(Context context) throws IOException, InterruptedException { /*nothing*/ }  
    // Called once for each key. Most applications should override this...  
    protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context) throws... {  
        for (VALUEIN value : values) {  
            context.write((KEYOUT) key, (VALUEOUT) value);  
        }  
    }  
  
    protected void cleanup(Context context) throws IOException, InterruptedException { /*nothing*/ }  
    // Expert users can override this method for more complete control over the execution of the Reducer.  
    public void run(Context context) throws IOException, InterruptedException {  
        setup(context);  
        while (context.nextKeyValue()) {  
            reduce(context.getCurrentKey(), context.getCurrentValues(), context);  
        }  
        cleanup(context);  
    }  
}
```

Now That We've Covered the Basics

- Let's talk in more detail about how a MapReduce job is run...

Step (1): Fire up the Mappers

- Hadoop asks the `FileInputFormat` that you specified...
 - ...to break the input files into splits
- For each split...
 - ...a `TaskTracker` somewhere in the cluster spawns a JVM to run a map task
- In each map task...
 - ...the `FileInputFormat` you specified creates a `RecordReader` for the split
 - This `RecordReader` is used to feed records to the `Context` in the mapper

Step (2): Collect the Mapper Output

- As output key-value pairs are written to the output...
 - ...they are partitioned, with one partition per reducer
 - You can control how this is done if you would like to
- As the pairs are partitioned...
 - ...they are serialized to binary format
 - You can control how this is done if you would like to (often a very good idea!)
 - And then sorted
 - You can also control the sort order, and allow comparison w/o de-serialization
 - If too many records for RAM are accumulated, a run is sorted and spilled
- All of this is happening within the JVM attached to each map task
 - So there is one sorted list of records per (mapper, reducer) pair

Step (3): The Shuffle Phase

- At some point, TaskTrackers begin spawning reduce tasks
- As reduce tasks are spawned...
 - ...they start asking the mappers for the partitions belonging to them, via HTTP

Step (4): The Merge

- Thus, reduce tasks get one or more sorted runs from each mapper
 - Once a map task obtains all of its runs, it merges them
 - In this way, it obtains a list of all of the records, sorted based upon keys
- As everything is merged...
 - The records are broken into groups
 - And each group is sent to the reducer
 - By default, all keys with “equal” values are in same group

key: 1	key: 1	key: 2	key: 3	key: 4	key: 4	key: 5	key: 7	key: 7
val: 5	val: 2	val: 11	val: 6	val: 22	val: 9	val: 13	val: 6	val: 7
group 1		group 2	group 3	group 4		group 5	group 6	

Step (4): The Merge

- Thus, reduce tasks get one or more sorted runs from each mapper
 - Once a map task obtains all of its runs, it merges them
 - In this way, it obtains a list of all of the records, sorted based upon keys
- As everything is merged...
 - The records are broken into groups
 - And each group is sent to the reducer
 - But it is possible to do something like this:

key: 1	key: 1	key: 2	key: 3	key: 4	key: 4	key: 5	key: 7	key: 7
val: 5	val: 2	val: 11	val: 6	val: 22	val: 9	val: 13	val: 6	val: 7
group 1			group 2			group 3	group 4	

- Set your `SortComparator` to order using the key
- Set your `GroupingComparator` to order using the key div 2

Step (5): The Reduce

- Each group of records with “equal” key vals is sent to the reducer
 - Which processes the group with a call to `Mapper.map`
- The reduce asks the `FileOutputFormat` for a `RecordWriter`
 - This `RecordWriter` will create an output file for the reduce task
- As records are written to the output `Context`...
 - They are fed to the returned `RecordWriter`
- That’s it! (mostly...)

Step (2.5): The Combiner

- Can optionally specify a “combiner” that extends `Reducer`
- This performs a pre-aggregation at the map task
- If a map task decides it is producing a lot of data
 - It can start calling the specified combiner as it spills runs to disk
- Classic example: `WordCount`
 - The map task gets lines of text, breaks into (word, 1) pairs
 - The reducer gets a list of (word, num) pairs and totals count for each word
 - Add a combiner that does the same thing as the reducer!
- Can give a big win
 - If typical word appears 10 times in split
 - Potential 10X reduction in data transferred via shuffle

Step (2.5): The Combiner

- Can optionally specify a “combiner” that extends `Reducer`
- This performs a pre-aggregation at the map task
- If a map task decides it is producing a lot of data
 - It can start calling the specified combiner as it spills runs to disk
- Classic example: `WordCount`
 - The map task gets lines of text, breaks into (word, 1) pairs
 - The reducer gets a list of (word, num) pairs and totals count for each word
 - Add a combiner that does the same thing as the reducer!
- But not always a win
 - Hadoop chooses whether or not to invoke the combiner
 - If you **really** need pre-aggregation, then write it into your mapper

Activity Two: Writing WordCount

- Will write the classic first MapReduce program
 - Has the same functionality as the WordCount you ran after setting up Hadoop
- A tiny tutorial... regular expressions in Java
- To break a line of text into tokens...
 - First create a `Pattern` object that recognizes things that might be words
`Pattern wordPattern = Pattern.compile ("[a-zA-Z][a-zA-Z0-9]+");`
 - Then when you have a `String` object “str”
`Matcher myMatcher = wordPattern.matcher (str);`
 - A call to “`myMatcher.find ()`” then returns true if found another in `str`
 - A call `myMatcher.group ()` then returns the next word
- Instructions available at

<http://cmj4.web.rice.edu/WordCount>

An Intro to Data Mining

- One of the most common things one does with “big data”...
 - ...is to “mine” it
- Are several common data mining tasks
 - (1) Build a “classifier” (aka “unsupervised learning”)...
 - Build a model that can be used to label data points (data points could be text docs, employees, customers, etc.)
 - Ex: label might be “+1: will lose this customer in six months”, or “-1: won’t lose this customer in six months”
 - Typically, you are given a set of labeled data to “train” the model

An Intro to Data Mining

- One of the most common things one does with “big data”...
 - ...is to “mine” it
- Are several common data mining tasks
 - (2) “Cluster” the data (“unsupervised learning”)
 - Given a large data set, assign labels to data points without any pre-labeled data
 - Typically the algorithm will look at how similar data points are
 - Similar points get same label (in the same “cluster”)
 - Useful for summarizing the data... cluster 1M customers into 10 groups, give the VP a summary of each group

An Intro to Data Mining

- One of the most common things one does with “big data”...
 - ...is to “mine” it
- Are several common data mining tasks
 - (3) Find “outliers” in the data
 - Given a large data set, find points that are unlike any other
 - Allows you to find strange events, bring to the attention of an analyst
 - Ex: find network events that are strange, don't fit typical pattern... might be an attack

An Intro to Data Mining

- One of the most common things one does with “big data”...
 - ...is to “mine” it
- Are several common data mining tasks
 - (4) Find patterns in the data
 - Imagine you have a large database describing billions of events
 - Each event is made up of a list of individual components...
 - Ex: a retail event might be a purchase of {beer, diapers, Frosted Flakes}
 - Find rules of the form <“purchases beer” implies “purchases diapers”>

Our Plan

- To keep scope manageable
 - Will focus on first two mining tasks: clustering and classification
 - Will further focus on text documents
 - Though the methods we'll consider are widely applicable
 - And focus on implementation over “big data” using Hadoop
- OK, so say we have a very large database of text documents (aka a “corpus”) that we want to mine...

The Classic Workflow

- First, build a *dictionary* for the corpus.
 - Given d distinct words...
 - A *dictionary* is a map from each word to an integer from 1 to d
- Then, process each doc to obtain a “bag of words”
 - Start with an array/vector x of length d , initialized to all zeros
 - Then, for each word in the doc:
 - (1) look it up in the dictionary to get its corresponding int i
 - (2) Increment $x[i]$

The Classic Workflow

- Example:

- Doc is “This was followed by radiotherapy.”

- Dictionary is: {(patient, 1), (status, 2), (followed, 3), (radiotherapy, 4), (negative, 5), (was, 6), (this, 7), (treated, 8), (by, 9), (with, 10)}

- x is [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

- First process “this”, giving $x = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$

The Classic Workflow

- Example:
 - Doc is “This was followed by radiotherapy.”
 - Dictionary is: {(patient, 1), (status, 2), (followed, 3), (radiotherapy, 4), (negative, 5), (was, 6), (this, 7), (treated, 8), (by, 9), (with, 10)}
- x is [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 - First process “this”, giving $x = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$
 - Then process “was”, giving $x = [0, 0, 0, 0, 0, 1, 1, 0, 0, 0]$

The Classic Workflow

- Example:
 - Doc is “This was followed by radiotherapy.”
 - Dictionary is: {(patient, 1), (status, 2), (followed, 3), (radiotherapy, 4), (negative, 5), (was, 6), (this, 7), (treated, 8), (by, 9), (with, 10)}
- x is [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 - First process “this”, giving $x = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$
 - Then process “was”, giving $x = [0, 0, 0, 0, 0, 1, 1, 0, 0, 0]$
 - Then process “followed”, giving $x = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0]$

The Classic Workflow

- Example:
 - Doc is “This was followed by radiotherapy.”
 - Dictionary is: {(patient, 1), (status, 2), (followed, 3), (radiotherapy, 4), (negative, 5), (was, 6), (this, 7), (treated, 8), (by, 9), (with, 10)}
- x is [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 - First process “this”, giving $x = [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]$
 - Then process “was”, giving $x = [0, 0, 0, 0, 0, 1, 1, 0, 0, 0]$
 - Then process “followed”, giving $x = [0, 0, 1, 0, 0, 1, 1, 0, 0, 0]$
 - After “by” and “radiotherapy”, have $x = [0, 0, 1, 1, 0, 1, 1, 0, 1, 0]$
- $x = [0, 0, 1, 1, 0, 1, 1, 0, 1, 0]$ is then treated as a *feature vector*

Some Other Tricks Are Used Mining Text

- Rather than using x ...
- Use vector y , where $y[w]$ is the “TF-IDF” of the word in the doc

— $TF(x, w) = \frac{x[w]}{\sum_{w'} x[w']}$; this is the “term frequency”

— $IDF(w) = \log \left(\frac{\sum_{x \text{ in corpus, } w' \text{ in dictionary}} x[w']}{\sum_{x \text{ in corpus}} x[w]} \right)$, the “inverse document freq”

— Then $y[w] = TF(x, w) \times IDF(w)$

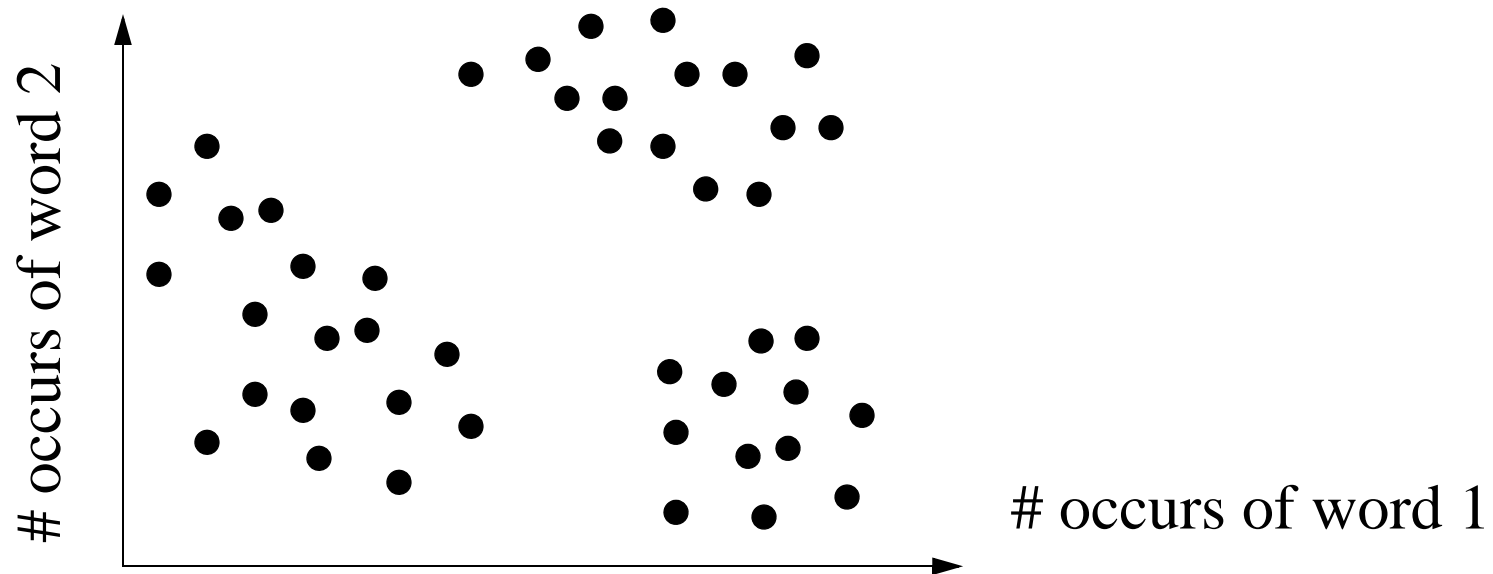
- Often helps cause it weights rare words more highly
- In our next activity, won't use TF-IDF, but will normalize the word-count vectors so we ignore the document length

Some Other Tricks Are Used Mining Text

- Remove *stop words* since they convey little meaning
 - “the” “which” “at” “which” “on”...
- Perform *stemming* of words
 - “running”, “run”, “runner”, “runs” are all reduced to “run”
- Use only the top k most frequent words
 - Removes obscure terms, mis-spellings, etc.
- Use *n-grams*, not just individual words
 - Given “This was followed by radiotherapy”, 2-grams are: “This was”, “was followed”, “followed by”, “by radiotherapy”
 - Helps to address bag-of-words’ total lack of contextual knowledge

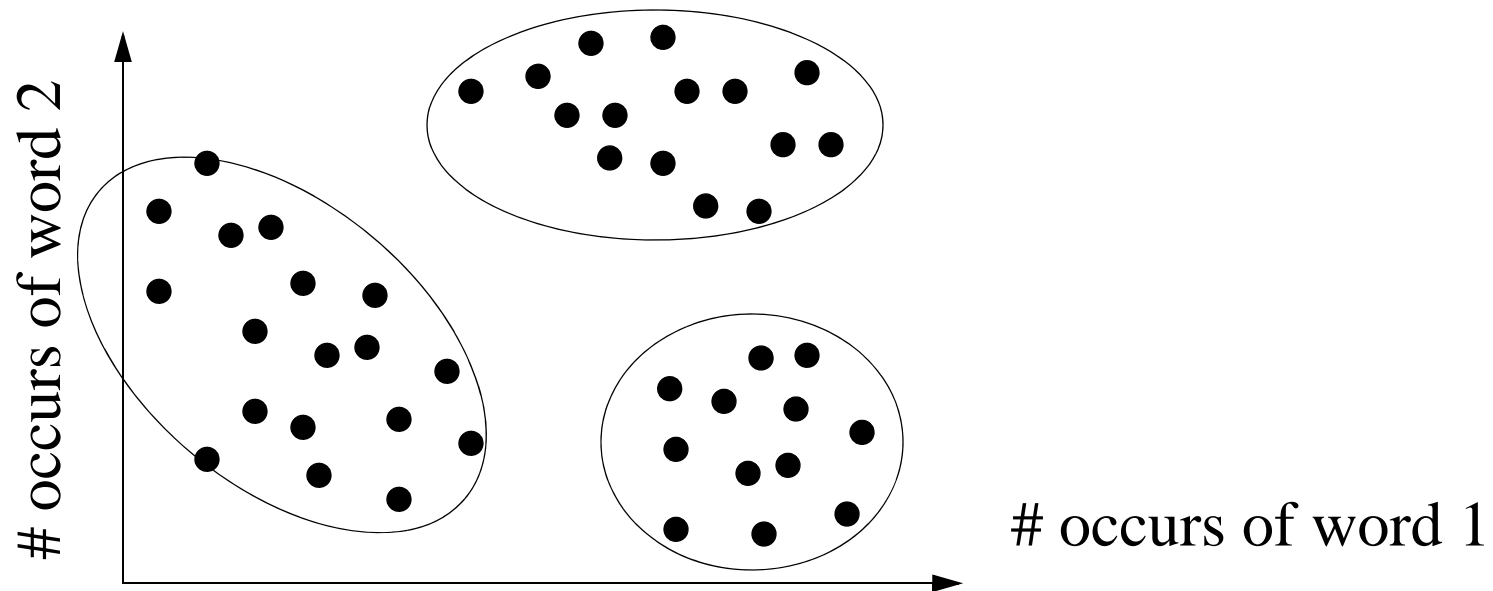
Clustering

- Typically you view the data again as falling in a multi-dim space



Clustering

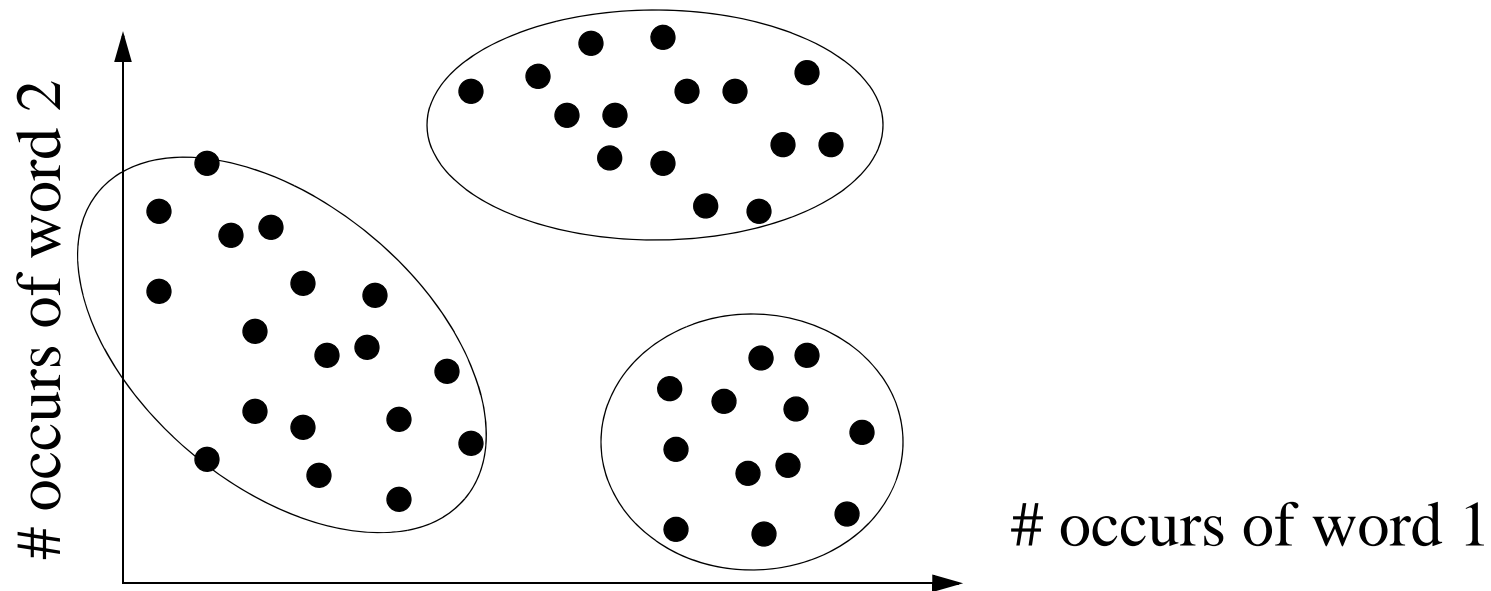
- Typically you view the data again as falling in a multi-dim space



- And the goal is to group them so similar points in same “cluster”

Clustering

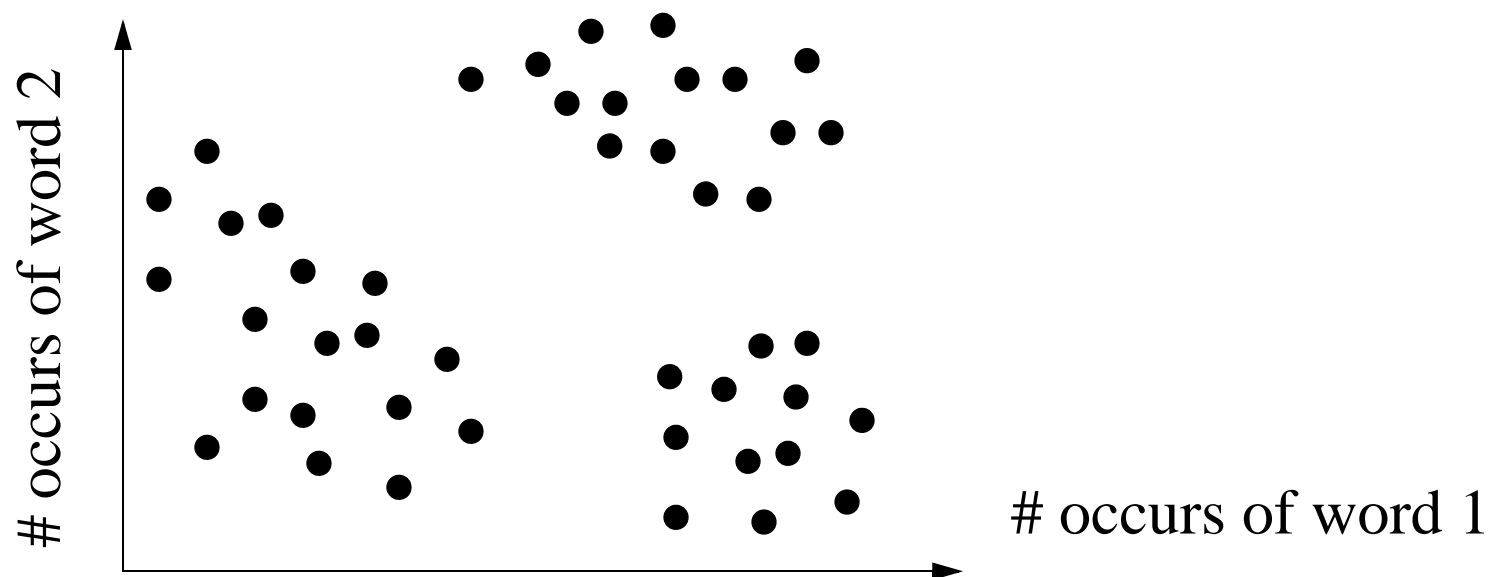
- Typically you view the data again as falling in a multi-dim space



- And the goal is to group them so similar points in same “cluster”
 - Data reduction: report back summaries of the clusters rather than actual data
 - Also serves as a way to segment the data when no obvious labels are available

KMeans Clustering

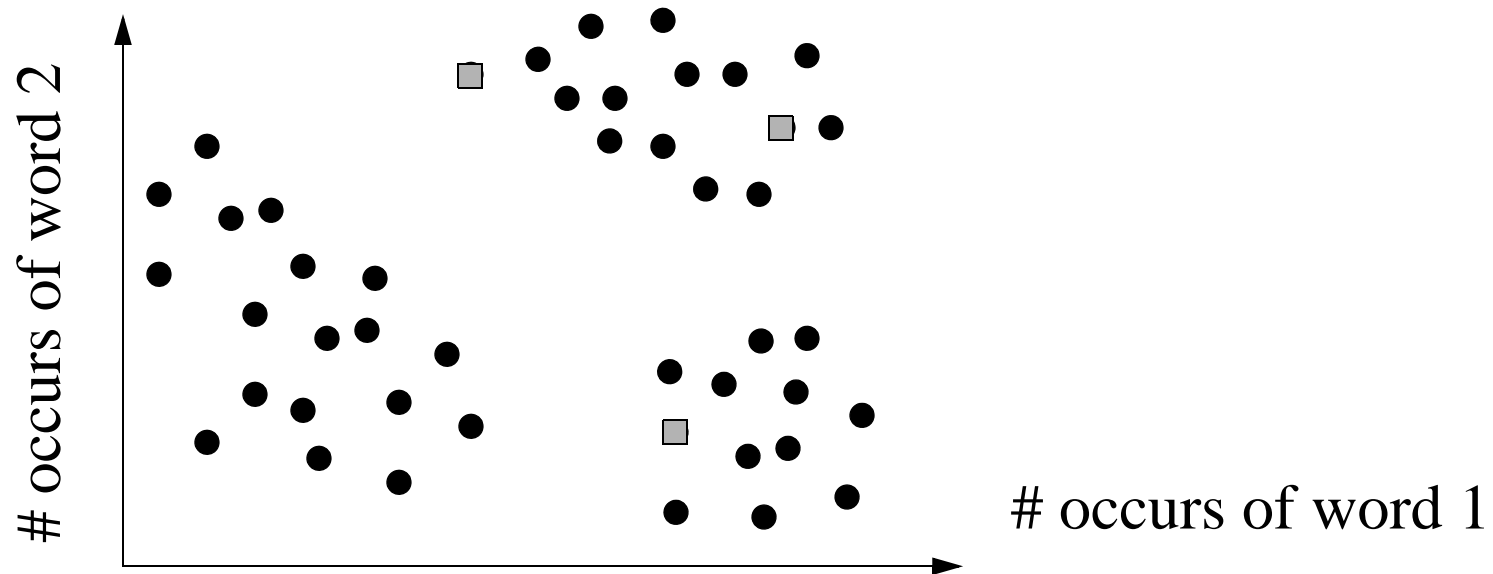
- Classical algorithm for clustering is called “KMeans”



- Tries to find k points (“centroids”) in data space...
- Such that average distance of each point to its closest centroid is minimized
- Unfortunately, very difficult problem to solve exactly!
- So solved approximately using an iterative algorithm...

KMeans Clustering

- Classical algorithm for clustering is called “KMeans”

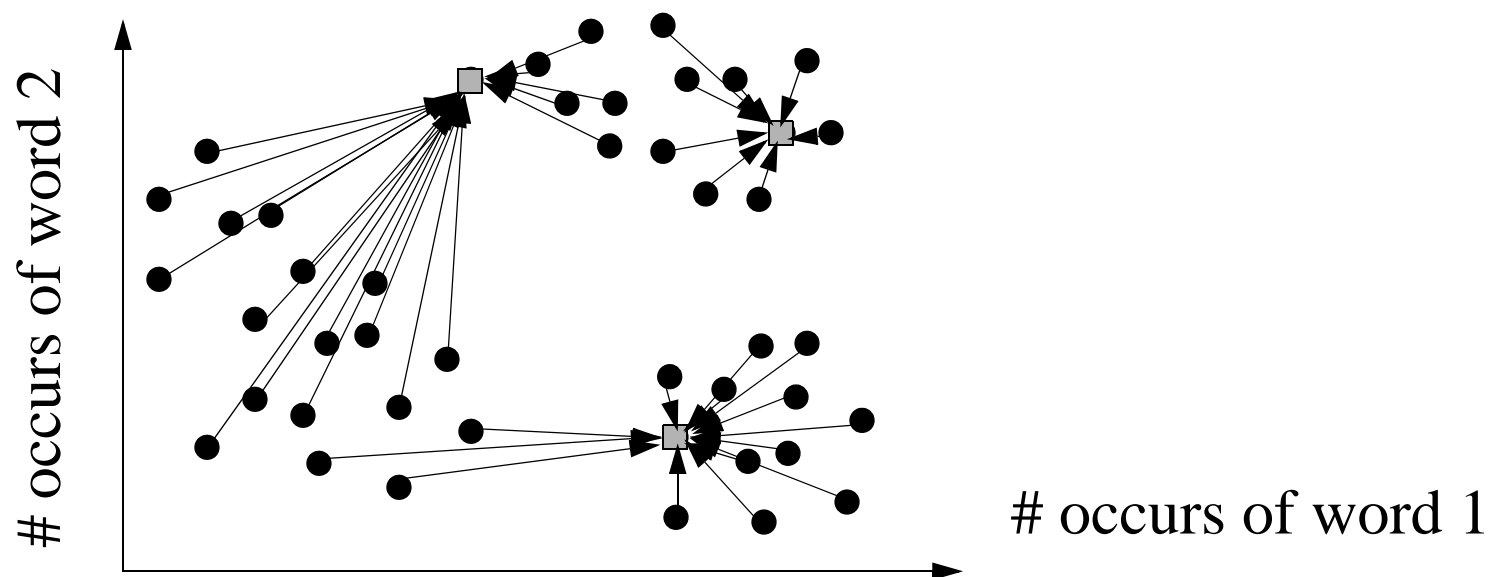


At all times, we have a set of k candidate centroids...

Typically initialized by choosing k database points at random

KMeans Clustering

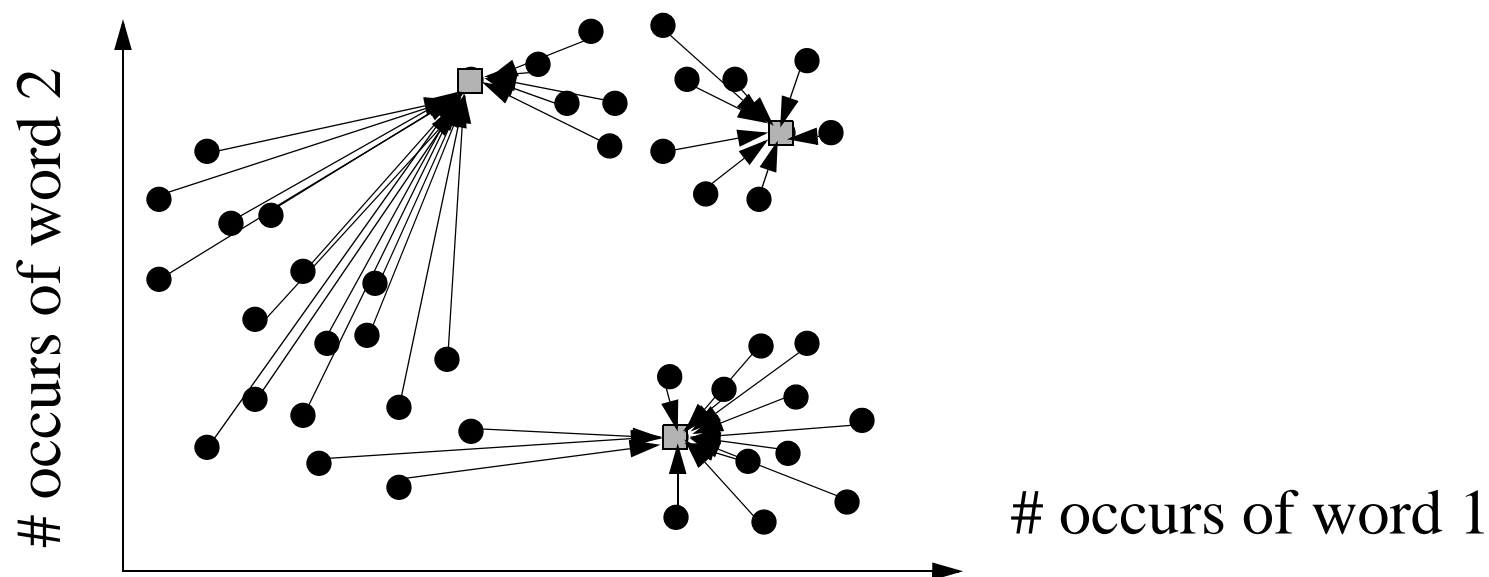
- Classical algorithm for clustering is called “KMeans”



At every iteration you start with an “E Step” that assigns each database point to its closest centroid...

KMeans Clustering

- Classical algorithm for clustering is called “KMeans”

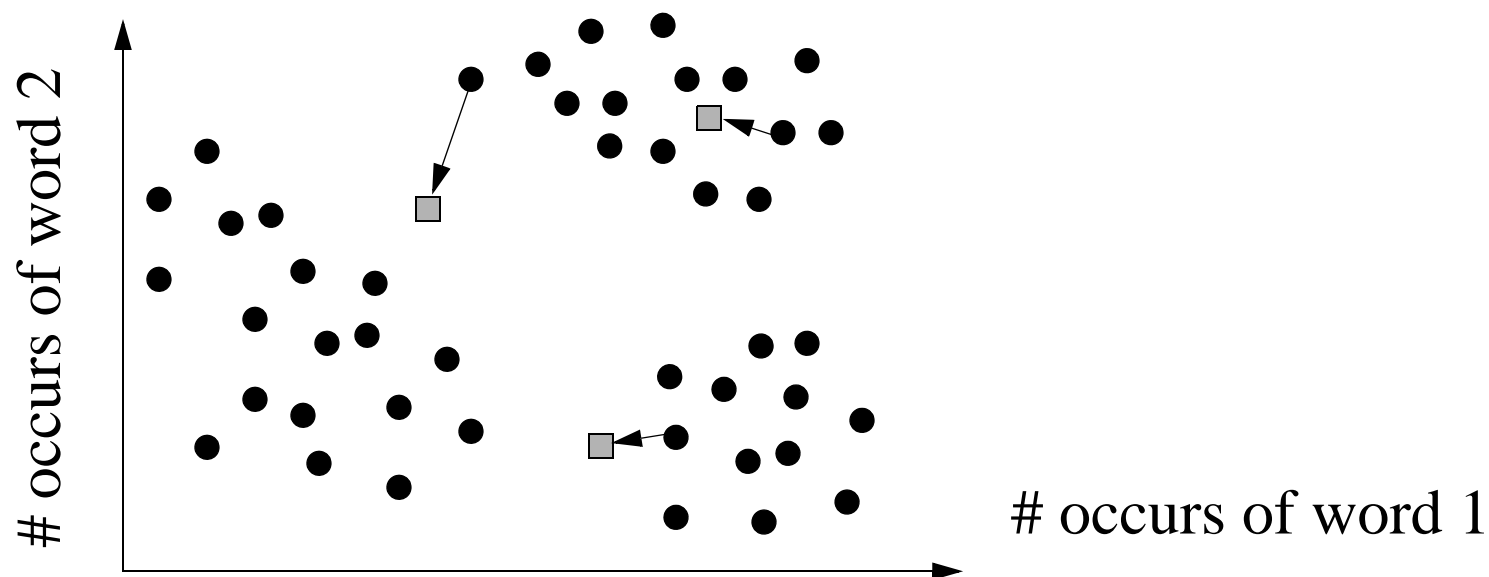


Then you have an “M Stem” where the centroids are updated to be the mean of all of the points that have been assigned to them

Since this are vectors, compute the mean using vector arithmetic: mean of $\langle 1, 2, 4 \rangle$, $\langle 1, 3, 5 \rangle$, and $\langle 1, 4, 3 \rangle$ is $\langle 3, 9, 12 \rangle / 3$ or $\langle 1, 3, 4 \rangle$

KMeans Clustering

- Classical algorithm for clustering is called “KMeans”

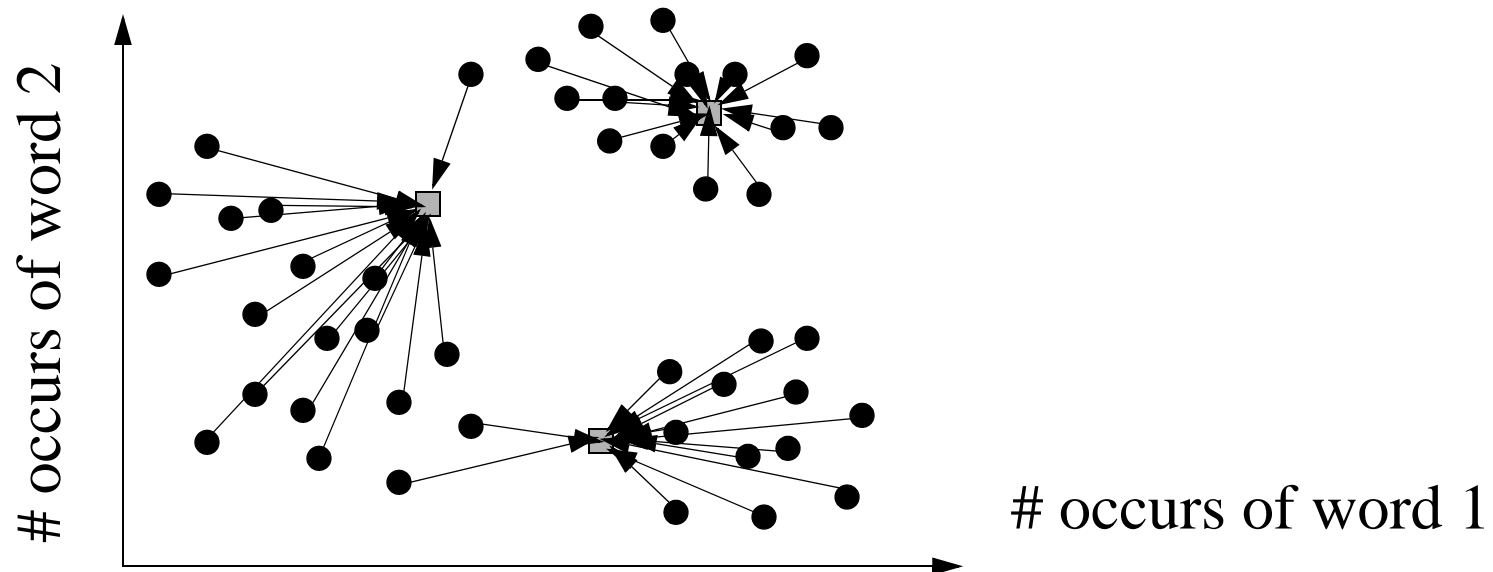


Then you have an “M Stem” where the centroids are updated to be the mean of all of the points that have been assigned to them

Since this are vectors, compute the mean using vector arithmetic: mean of $\langle 1, 2, 4 \rangle$, $\langle 1, 3, 5 \rangle$, and $\langle 1, 4, 3 \rangle$ is $\langle 3, 9, 12 \rangle / 3$ or $\langle 1, 3, 4 \rangle$

KMeans Clustering

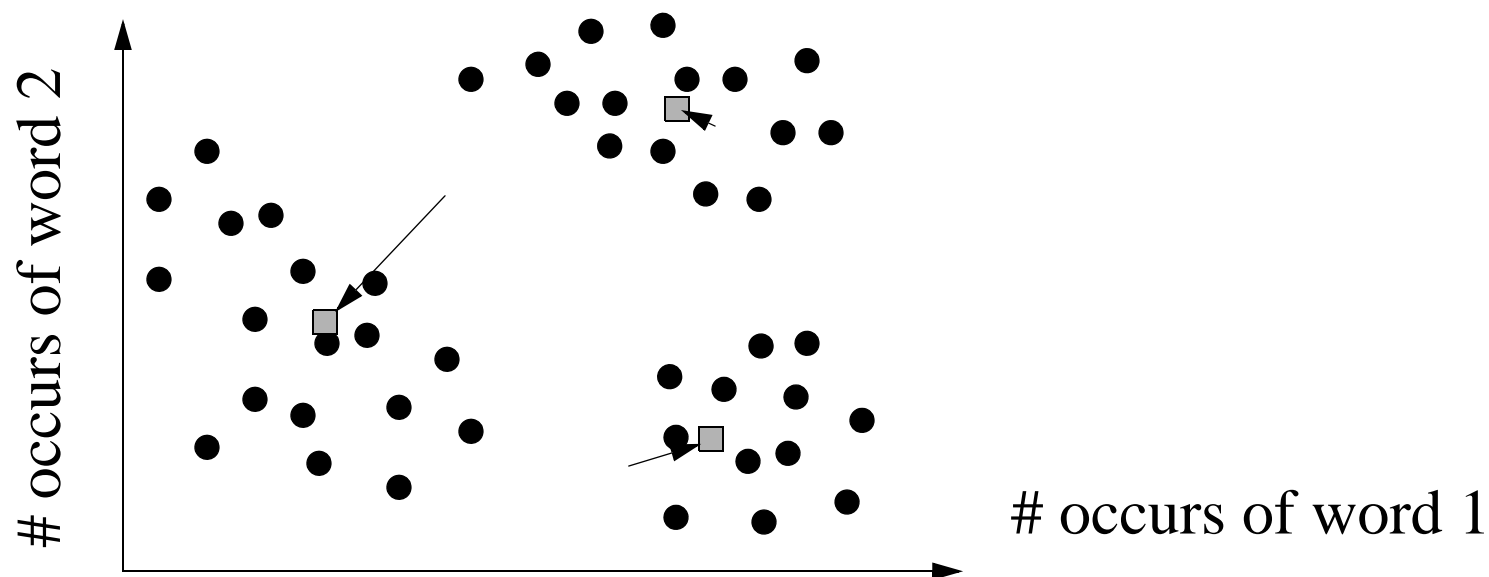
- Classical algorithm for clustering is called “KMeans”



Then you repeat the “E Step” again....

KMeans Clustering

- Classical algorithm for clustering is called “KMeans”



And the “M Step” again....

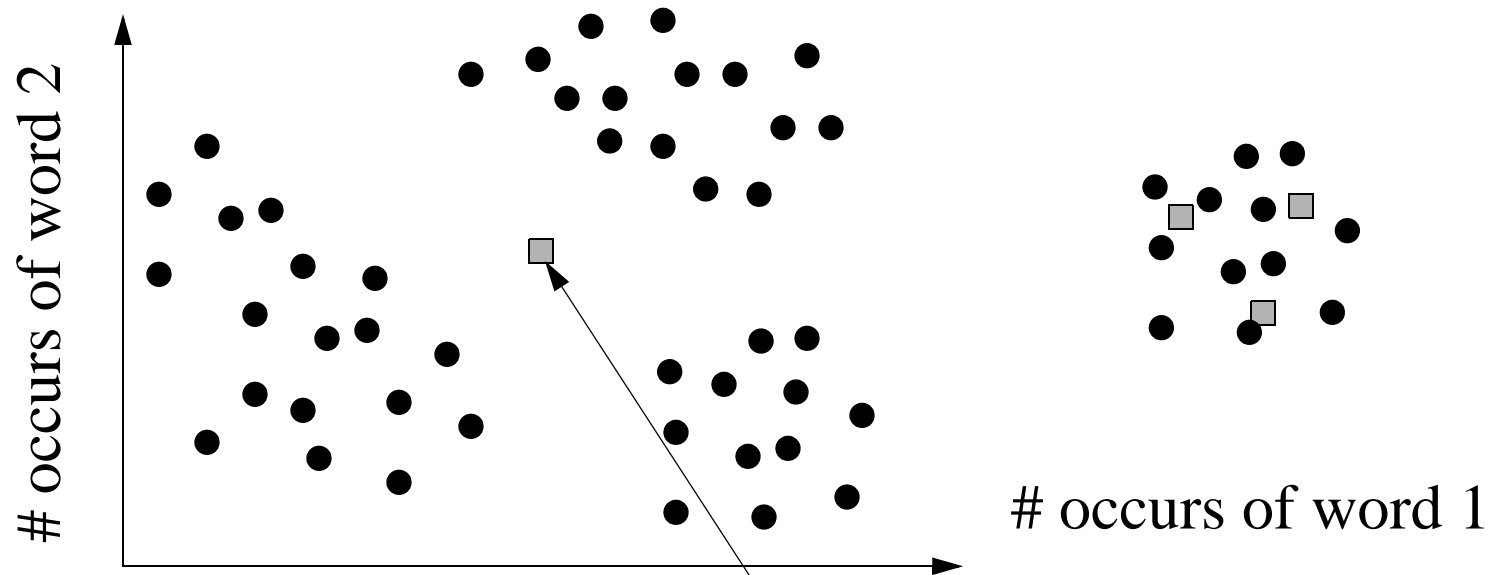
And already our clustering looks pretty good!

You repeat this until “convergence”, where no data points change cluster membership...

Mathematically this is guaranteed to converge

KMeans: Getting Stuck

- One problem is that while KMeans always converges...
 - It is vulnerable to getting “stuck” at a so-called locally optimal solution

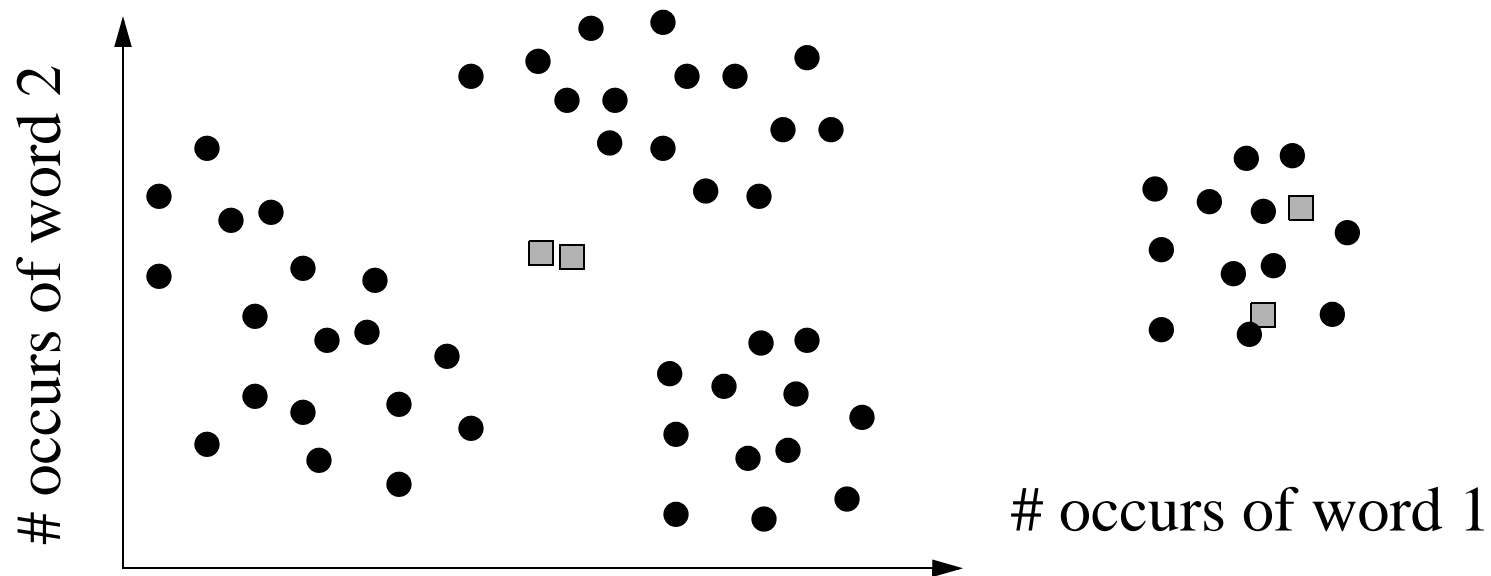


In this example, we have a very poor grouping, but it can never change...

Since this every point on the LHS is closest to this guy

KMeans: Getting Unstuck

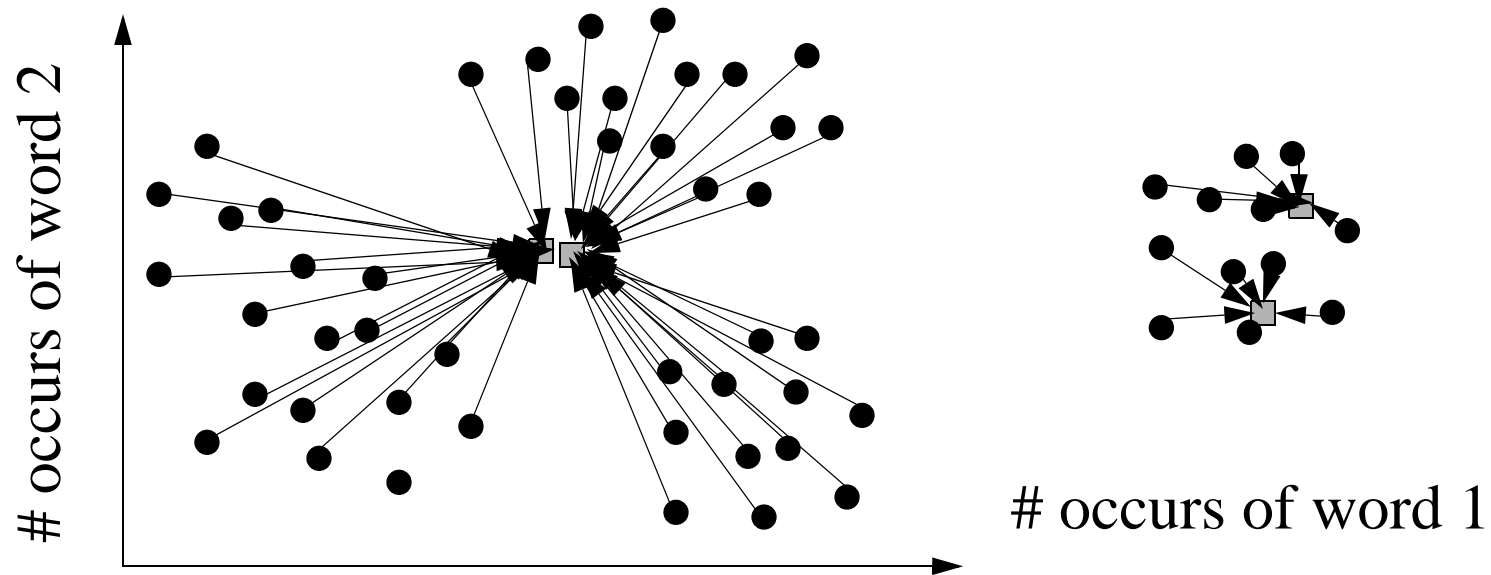
- One problem is that while KMeans always converges...
 - It is vulnerable to getting “stuck” at a so-called locally optimal solution



Simple solution: (1) kill the centroids with the fewest points assigned, then (2)
Create a new centroid right next to the centroid with the most points assigned

KMeans: Getting Unstuck

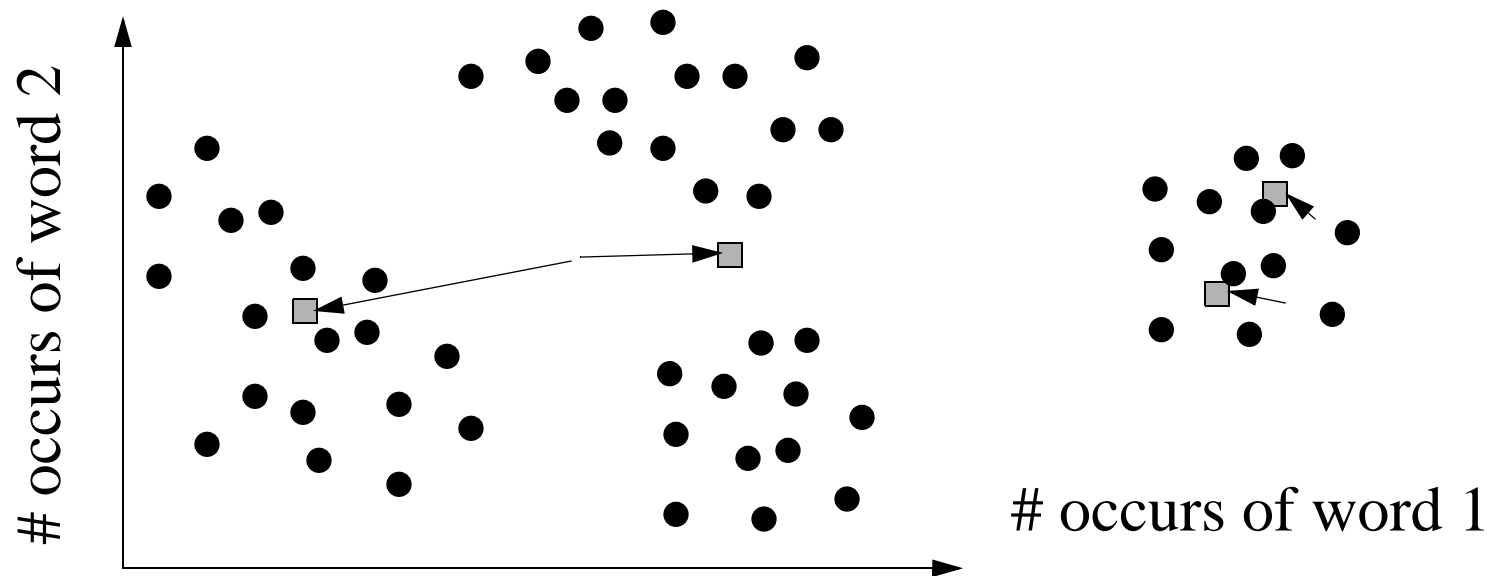
- One problem is that while KMeans always converges...
 - It is vulnerable to getting “stuck” at a so-called locally optimal solution



Then you will tend to get unstuck...

KMeans: Getting Unstuck

- One problem is that while KMeans always converges...
 - It is vulnerable to getting “stuck” at a so-called locally optimal solution



Then you will tend to get unstuck...

Though you might have to do this several times...

Heuristic: kill the smallest cluster whenever it is less than 5% of the size of the largest

Activity Three: K-Means Clustering

- You'll implement the K-Means algorithm for a single machine
 - As a prelude to “Hadoopifying” it for the next exercise
- See <http://cmj4.web.rice.edu/Kmeans.html>
- Your task is to fill in around 10 lines of code in `KMeans.java`
- To do this, three important files you'll download & examine:
 - `IDoubleVector.java`: interface that provides vector functionality
 - `VectorizedObject.java`: provides a key/value pair with a vector attached
 - `KMeans.java`: where you will add your code

K-Means over MapReduce

- First, we need to identify what the mappers and reducers do
- Initial idea...
- Mappers: scan data, compute what cluster each point belongs to
 - Output is a set of (`clusterID`, `data`) pairs
 - Where `data` is closest to the centroid with ID `clusterID`
- Reducers: accept data, sorted on `clusterID`
 - Have many reducers
 - Each accepts (`Text cID`, `iterable<VectorizedObject> vals`) pairs
 - Averages all of the data in `vals`, puts results in object `avg`
 - Writes (`cID`, `avg`) to the context

What's the problem with this?

- It sends the entire data set as output from the Map phase!
- Will be very expensive...
- Solution?

What's the problem with this?

- It sends the entire data set as output from the Map phase!
- Will be very expensive...
- Solution?
 - Might we use a combiner?
 - accepts `(Text cID, iterable<VectorizedObject> vals) pairs`
 - output `(cID, sum) pairs`
 - where `sum` is a `VectorizedObject` that has the sum of everything in `vals`...
 - plus the count of the number of points that went into that sum

What's the problem with this?

- It sends the entire data set as output from the Map phase!
- Will be very expensive...
- Solution?
 - Might we use a combiner?
 - accepts `(Text cID, iterable<VectorizedObject> vals)` pairs
 - output `(cID, sum)` pairs
 - where `sum` is a `VectorizedObject` that has the sum of everything in `vals`...
 - plus the count of the number of points that went into that sum
- Then the reducer:
 - When it gets a `(Text cID, iterable<VectorizedObject> vals)` pair
 - Adds up everything in `vals`, divides by the sum of the counts stored in there
 - And outputs the result as the new centroid

This is a better solution...

- Except that you are not guaranteed that the combiner will run
- And also, the reduce phase is distributed all over the cluster
 - Problematic since we can't kill the smallest cluster centrally
- Might we do even better?

Our solution...

- Sum up the points associated with each cluster in each mapper
 - Do **not** send out any data as you process the points
- Then, in the mapper's `cleanup` method...
 - `(cID, sum)` pairs
 - where `sum` is a `VectorizedObject` that has the sum of everything in `vals...`
 - ...plus the count of the number of points that went into that sum

Our solution...

- Only use a **single** reducer
 - OK since amount of data is bounded by k times the number of mappers
- The reducer:
 - When it gets a `(Text cID, iterable<VectorizedObject> vals)` pair
 - Adds up everything in `vals`, divides by the sum of the counts stored in there
 - And stores the result in a data structure as the new centroid
- Then, in the reducer's `cleanup` method...
 - Kill and replace the smallest cluster if necessary
 - Then output all of the new centroids
- MapReduce purists won't like this solution
 - Why? It is stateful!

Activity Three: MR K-Means Clustering

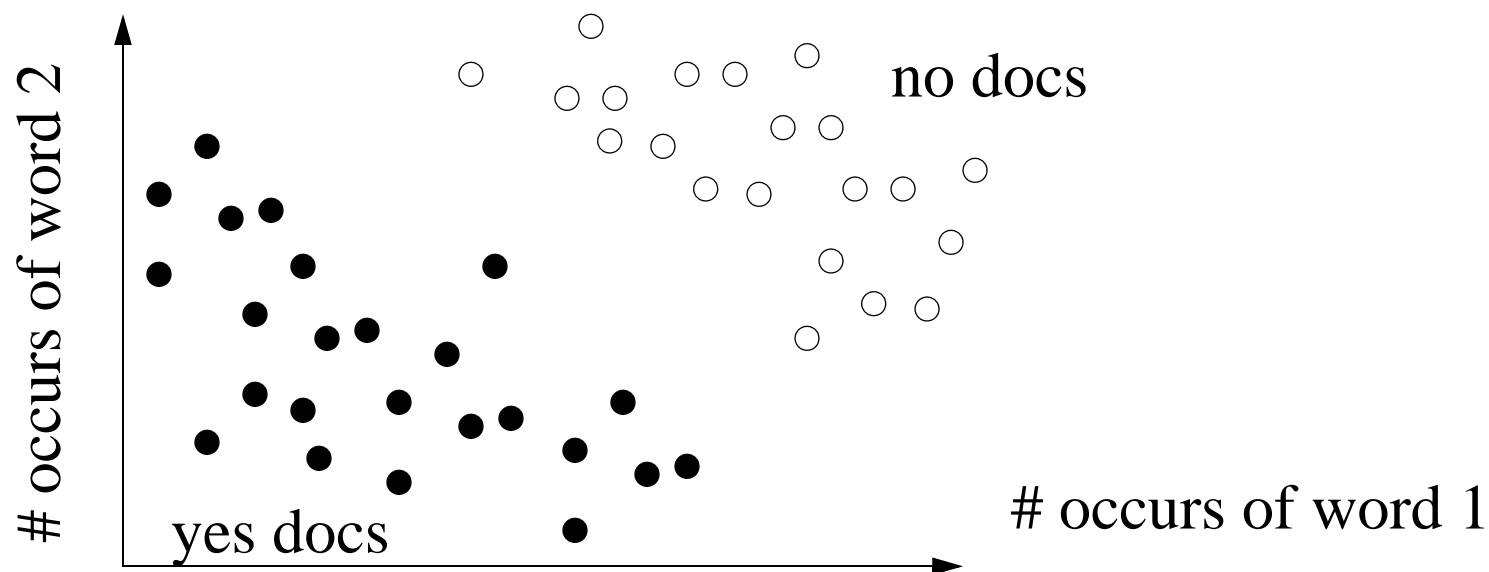
- You'll implement the K-Means algorithm over Hadoop
 - As a prelude to “Hadoopifying” it for the next exercise
- `http://cmj4.web.rice.edu/MapRedKMeans.html`

The Classic Classification Workflow

- Assume we have a set of labeled data
 - For example, check to see if the patient was billed for BC in next 6 months
- This gives a set of (x, label) pairs
- Feed these as *training data* into your classifier-of-choice
- Then, when have a new record to classify
 - Convert it into a bag-of-words
 - And feed it into the classifier for labeling

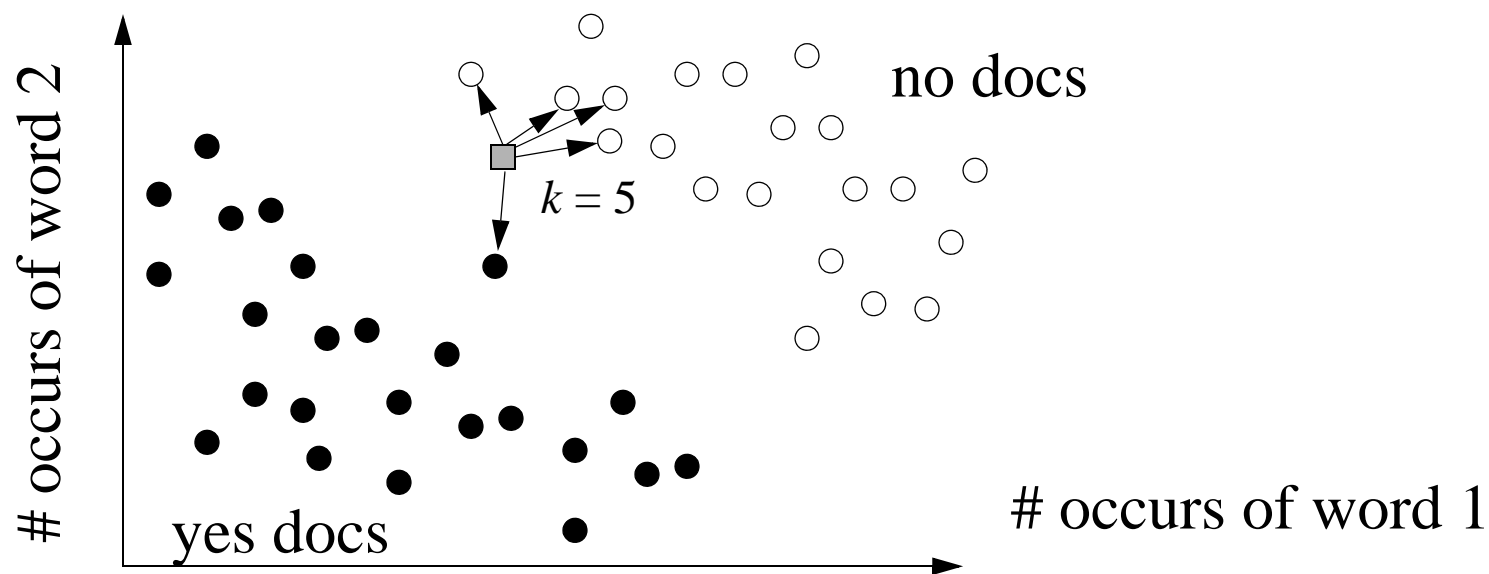
A Common Classifier is "KNN"

- This is the one that you'll be implementing on Hadoop
- Idea: place docs in multi-dim space



A Common Classifier is “KNN”

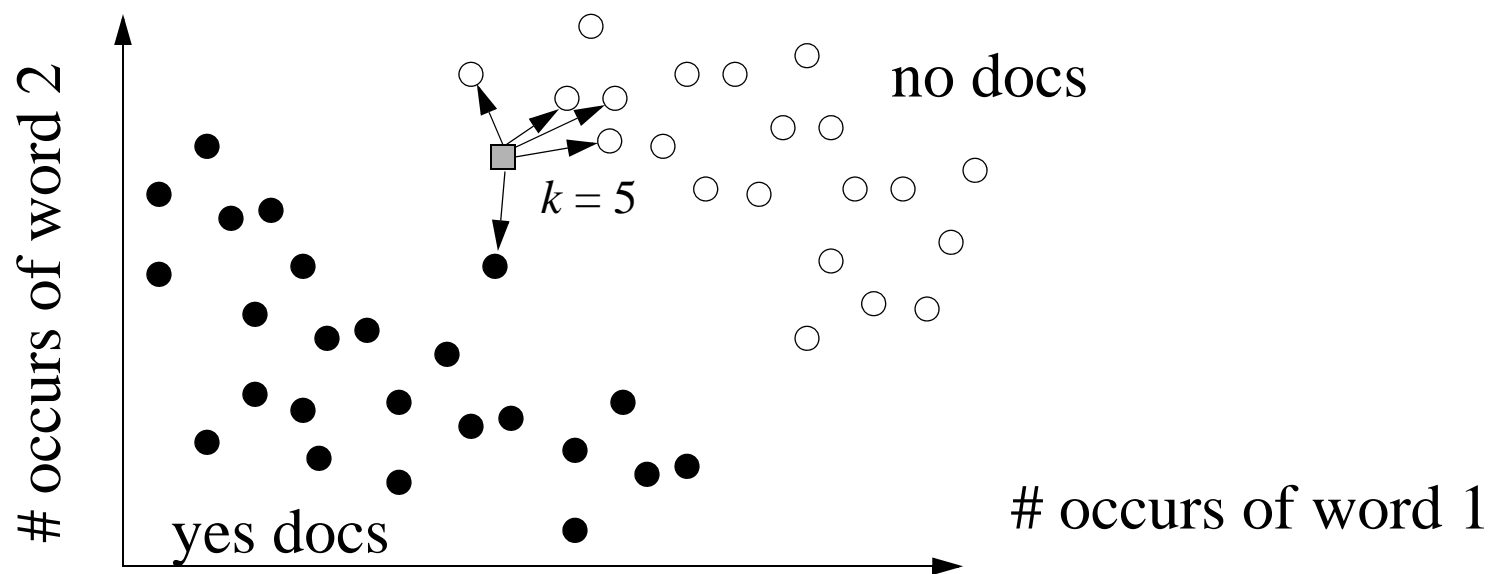
- This is the one that you’ll be implementing on Hadoop
- Idea: place docs in multi-dim space



- To classify a new doc...
 - You place it in the space, and find its k nearest neighbors (hence “KNN”)

A Common Classifier is “KNN”

- This is the one that you’ll be implementing on Hadoop
- Idea: place docs in multi-dim space



- To classify a new doc...
 - You place it in the space, and find its k nearest neighbors (hence “KNN”)
 - And you give it the most common label of its k nearest neighbors

KNN Over MapReduce

- Say you want to perform KNN classification
 - Want to classify m points
 - Have a database of n training point
 - Data are high-dimensional
- Really no way to avoid doing an m by n computation
 - For each point p_1 in the training database
 - For each point p_2 in the points to classify
 - Compute the distance from p_1 to p_2
 - If this distance is in the bottom k distances to p_2 , record $(p_2, \text{dist}, p_1.\text{label})$
 - For each point p_2 to classify
 - compute the most common label in the bottom k distances

KNN Over MapReduce

- Really no way to avoid doing an m by n computation

For each point p_1 in the training database

For each point p_2 in the points to classify

Compute the distance from p_1 to p_2

If this distance is in the bottom k distances to p_2 , record $(p_2, \text{dist}, p_1.\text{label})$

For each point p_2 to classify

compute the most common label in the bottom k distances

- You could try (for example) to index the list of points to classify
 - So you can avoid looking at the p_2 's that are far from the current p_1
 - But this is known to be ineffective in high dimensions (many attributes)
- So brute force (using MapReduce) is a reasonable choice

KNN Over MapReduce

- Say we are gonna use MapReduce
 - What should the general strategy be?
 - Many options!

KNN Over MapReduce

- Here's one...
- Say we have a `RecordKey` with two pieces of data:
 - Key: a `String`
 - Distance: a `Double`
- And at each mapper, we run a version of the aforementioned loop:
 - For each point `p1` in the training database
 - For each point `p2` in the points to classify
 - Compute the distance from `p1` to `p2`
 - But what do we do here?

KNN Over MapReduce

- Here's one...
- Say we have a `RecordKey` with two pieces of data:
 - Key: a `String`
 - Distance: a `Double`
- And at each mapper, we run a version of the aforementioned loop:
 - For each point `p1` in the training database
 - For each point `p2` in the points to classify
 - Compute the distance from `p1` to `p2`
 - Emit a `((p2.identifier, distance), p1.class)` pair

KNN Over MapReduce

- Here's one...
- Say we have a `RecordKey` with two pieces of data:
 - Key: a `String`
 - Distance: a `Double`
- And at each mapper, we run a version of the aforementioned loop:

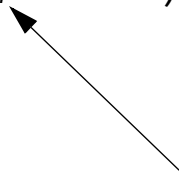
For each point `p1` in the training database

For each point `p2` in the points to classify

Compute the distance from `p1` to `p2`

Emit a `((p2.identifier, distance), p1.class)` pair

First entry is a `RecordKey`



KNN Over MapReduce

- And at each mapper, we run a version of the aforementioned loop:

- For each point p_1 in the training database

- For each point p_2 in the points to classify

- Compute the distance from p_1 to p_2

- Emit a $((p_2.identifier, distance), p_1.class)$ pair

- Why is this useful?

- In the shuffle phase...

- ...we can sort by `RecordKeys`

- Sort order is first by key, then by distance

KNN Over MapReduce

- Why is this useful?

- In the shuffle phase...

- ...we can sort by `RecordKeys`

- Sort order is first by key, then by distance

- So as the output of the map phase, you get a big mess

((doc 1, 4), N)	((doc 2, 9), N)	((doc 3, 8), N)	((doc 4, 2), N)
((doc 4, 5), N)	((doc 4, 0), Y)	((doc 4, 2), Y)	((doc 3, 4), Y)
((doc 2, 7), N)	((doc 1, 2), N)	((doc 2, 1), N)	((doc 3, 3), Y)
((doc 1, 8), N)	((doc 1, 6), Y)	((doc 3, 6), N)	((doc 2, 7), N)

KNN Over MapReduce

- Sort order is first by key, then by distance
- So as the output of the map phase, you get a big mess

((doc 1, 4), N)	((doc 2, 9), N)	((doc 3, 8), N)	((doc 4, 2), N)
((doc 4, 5), N)	((doc 4, 0), Y)	((doc 4, 2), Y)	((doc 3, 4), Y)
((doc 2, 7), N)	((doc 1, 2), N)	((doc 2, 1), N)	((doc 3, 3), Y)
((doc 1, 8), N)	((doc 1, 6), Y)	((doc 3, 6), N)	((doc 2, 7), N)

- But after sorting you get

((doc 1, 2), N)	((doc 1, 4), N)	((doc 1, 6), Y)	((doc 1, 8), N)
((doc 2, 1), N)	((doc 2, 7), N)	((doc 2, 7), N)	((doc 2, 9), N)
((doc 3, 3), Y)	((doc 3, 4), Y)	((doc 3, 6), N)	((doc 3, 8), N)
((doc 4, 0), Y)	((doc 4, 2), Y)	((doc 4, 2), N)	((doc 4, 5), N)

KNN Over MapReduce

- But after sorting you get

((doc 1, 2), N)	((doc 1, 4), N)	((doc 1, 6), Y)	((doc 1, 8), N)
((doc 2, 1), N)	((doc 2, 7), N)	((doc 2, 7), N)	((doc 2, 9), N)
((doc 3, 3), Y)	((doc 3, 4), Y)	((doc 3, 6), N)	((doc 3, 8), N)
((doc 4, 0), Y)	((doc 4, 2), Y)	((doc 4, 2), N)	((doc 4, 5), N)

- OK, this is looking more useful, but how to reduce?

KNN Over MapReduce

- But after sorting you get

((doc 1, 2), N)	((doc 1, 4), N)	((doc 1, 6), Y)	((doc 1, 8), N)
((doc 2, 1), N)	((doc 2, 7), N)	((doc 2, 7), N)	((doc 2, 9), N)
((doc 3, 3), Y)	((doc 3, 4), Y)	((doc 3, 6), N)	((doc 3, 8), N)
((doc 4, 0), Y)	((doc 4, 2), Y)	((doc 4, 2), N)	((doc 4, 5), N)

- OK, this is looking more useful, but how to reduce?

- Create one group for everyone associated with the same doc we want to classify
- Group 1: ((doc 1, 2), N) ((doc 1, 4), N) ((doc 1, 6), Y) ((doc 1, 8), N)
- Group 2: ((doc 2, 1), N) ((doc 2, 7), N) ((doc 2, 7), N) ((doc 2, 9), N)
- Group 3: ((doc 3, 3), Y) ((doc 3, 4), Y) ((doc 3, 6), N) ((doc 3, 8), N)
- Group 4: ((doc 4, 0), Y) ((doc 4, 2), Y) ((doc 4, 2), N) ((doc 4, 5), N)

KNN Over MapReduce

- OK, this is looking more useful, but how to reduce?
 - Create one group for everyone associated with the same doc we want to classify
 - Group 1: ((doc 1, 2), N) ((doc 1, 4), N) ((doc 1, 6), Y) ((doc 1, 8), N)
 - Group 2: ((doc 2, 1), N) ((doc 2, 7), N) ((doc 2, 7), N) ((doc 2, 9), N)
 - Group 3: ((doc 3, 3), Y) ((doc 3, 4), Y) ((doc 3, 6), N) ((doc 3, 8), N)
 - Group 4: ((doc 4, 0), Y) ((doc 4, 2), Y) ((doc 4, 2), N) ((doc 4, 5), N)
- In other words, the primary sort order on key
 - Is by `RecordKey.key` first
 - Then by `RecordKey.distance`
- But the secondary sort order (the grouping)
 - Is by `RecordKey.key` only

KNN Over MapReduce

- In other words, the primary sort order on key
 - Is by `RecordKey.key` first
 - Then by `RecordKey.distance`
- But the secondary sort order (the grouping)
 - Is by `RecordKey.key` only
- In Hadoop...
 - We handle by defining two `RawComparator` classes over `RecordKey`
 - One is called `RecordKeySortComparator`
 - One is called `RecordKeyGroupingComparator`
 - These define the two desired sort orders

KNN Over MapReduce

- In Hadoop...
 - We handle by defining two `RawComparator` classes over `RecordKey`
 - One is called `RecordKeySortComparator`
 - One is called `RecordKeyGroupingComparator`
 - These define the two desired sort orders
- Then in the main program, we register them with

```
job.setGroupingComparatorClass (RecordKeyGroupingComparator);  
job.setSortComparatorClass (RecordKeySortComparator);
```
- A quick note about the Hadoop `RawComparator` class
 - Needs to have two comparison routines
 - One over `RecordKey` objects
 - One over serialized `RecordKey` objects (passed in as byte arrays)

KNN Over MapReduce

- So at this point, we have

- Group 1: ((doc 1, 2), N) ((doc 1, 4), N) ((doc 1, 6), Y) ((doc 1, 8), N)
- Group 2: ((doc 2, 1), N) ((doc 2, 7), N) ((doc 2, 7), N) ((doc 2, 9), N)
- Group 3: ((doc 3, 3), Y) ((doc 3, 4), Y) ((doc 3, 6), N) ((doc 3, 8), N)
- Group 4: ((doc 4, 0), Y) ((doc 4, 2), Y) ((doc 4, 2), N) ((doc 4, 5), N)

- How to reduce?

- Just scan over everything in the group (will be from low to high distance)
- As soon as you see k items, find the most frequent label

- Ex: the reducer for the first group will get

- ((doc 1, ???), <N, N, Y, N>)
- So if k is 2, you find the most common label in the set {N, N}

KNN Over MapReduce

- So at this point, we have

- Group 1: ((doc 1, 2), N) ((doc 1, 4), N) ((doc 1, 6), Y) ((doc 1, 8), N)
- Group 2: ((doc 2, 1), N) ((doc 2, 7), N) ((doc 2, 7), N) ((doc 2, 9), N)
- Group 3: ((doc 3, 3), Y) ((doc 3, 4), Y) ((doc 3, 6), N) ((doc 3, 8), N)
- Group 4: ((doc 4, 0), Y) ((doc 4, 2), Y) ((doc 4, 2), N) ((doc 4, 5), N)

- Ex: the reducer for the first group will get

- ((doc 1, ???), <N, N, Y, N>)

- So if k is 2, you find the most common label in the set {N, N}

- Ex: the reducer for the second group will get

- ((doc 2, ???), <N, N, N, N>)

- So if k is 2, you again find the most common label in the set {N, N}

One Little Note

- During implementation/debugging, I found this to be annoying:
 - $((\text{doc } 1, ???), \langle N, N, Y, N \rangle)$
 - Why? You lose the distances associated with the labels
 - Not needed to get correct answer, but makes debugging difficult
- So in my implementation...
 - I didn't use $(\text{RecordKey}, \text{Text})$ pairs as output from the mapper
 - Instead I used $(\text{RecordKey}, \text{RecordKey})$ pairs
 - Allowed me to have the label **and** the distance in the reducer
 - Ex, given: $((\text{doc } 1, 2), (N, 2)) ((\text{doc } 1, 4), (N, 4)) ((\text{doc } 1, 6), (Y, 6)) ((\text{doc } 1, 8), (N, 8))$
 - The input to the reducer will be $((\text{doc } 1, ???), \langle (N, 2), (N, 4), (Y, 6), (N, 8) \rangle)$

One Little Note

- So in my implementation...
 - I didn't use (RecordKey, Text) pairs as output from the mapper
 - Instead I used (RecordKey, RecordKey) pairs
 - Allowed me to have the label **and** the distance in the reducer
 - Ex, given: ((doc 1, 2), (N, 2)) ((doc 1, 4), (N, 4)) ((doc 1, 6), (Y, 6)) ((doc 1, 8), (N, 8))
 - The input to the reducer will be ((doc1, ???), <(N, 2), (N, 4), (Y, 6), (N, 8)>)
- In the end...
 - Mapper is <LongWritable, Text, RecordKey, RecordKey>
 - Reducer is <RecordKey, RecordKey, Text, Text>
 - Final pairs output are (document key, predicted label)

A Note on Serialization

- If you define your own class for mapper output...
 - ...as we have done here with `RecordKey`
 - You really need to tell Hadoop how to serialize/deserialize it
- Why?
 - Your own serialization/deserialization will be faster than using the default Java
 - Plus, can then write a good `RawComparator`
- How to do this?
 - Check out the `RecordKeySerialization` class
 - Has three methods...
 - First one (`accept`) just returns true when we see a class we can ser/deser
 - Second one (`getSerializer`) returns a class to serialize `RecordKey` objects
 - Third (`getDeserializer`) returns a class to deserialize `RecordKey` objects

Activity Five

- Design, write, and run a KNN classifier
 - See <http://cmj4.web.rice.edu/MapRedKNN.html>

Outlier Detection (Likely Won't Get Here!)

- We'll go through one more mining algorithm in detail
- Implement it on Hadoop? If you dare!
- Outlier detection
 - Given n data points, find those that are really weird
- Common definition uses KNN
 - A point is an “outlier” if the distance to its k th NN is in top t

Outlier Detection

- Naive algorithm: quadratic (won't work for big n)

For all of the points $p1$ in the data set

Scan all of the points $p2$ in the data set, finding the KNN

Sort points on distance to KNN, return top t

- A better algorithm

Maintain a list of the top k outliers

Let *small* be the distance to the kNN for the "worst" outlier in top list

For all of the points $p1$ in the data set

Scan all of the points $p2$ in the data set in random order

If kNN so far ever exceeds *small*, then stop; $p1$ can't be an outlier

If you made it all the way through, add $p1$ to top list

Outlier Detection

Maintain a list of the top k outliers

Let $small$ be the distance to the k NN for the "worst" outlier in top list

For all of the points $p1$ in the data set

Scan all of the points $p2$ in the data set in random order

If k NN so far ever exceeds $small$, then stop; $p1$ can't be an outlier

If you made it all the way through, add $p1$ to top list

- Guaranteed to work. Why?
- As you process more points...
 - ..." k NN so far" can only increase
 - So " k NN so far" is a lower bound on the k NN distance
 - Often can discard $p1$ very early
 - For even high-D data sets: less than 1000 points on avg are processed

Outlier Detection on MapReduce

- How to run on MapReduce?
 - No precise translation
 - Why? There is a sequential dependency on $p1$
 - Here's one suggestion for a similar algorithm...

Outlier Detection on MapReduce

- Assume input data stored in "Data"
- Create a list of (ID, (point, kNN so far list)) in HDFS
 - Call this "Candidates"
 - "kNN so far list"s are initially set to be empty
 - The ID is just a counter from 1 to size of "Candidates"

Outlier Detection on MapReduce

- Assume input data stored in "Data"
- Create a list of (ID, (point, kNN so far list)) in HDFS
 - Call this "Candidates"
 - "kNN so far list"s are initially set to be empty
 - The ID is just a counter from 1 to size of "Candidates"
- In map phase, process both "Candidates" and "Data"
- When you get a point p from "Data":
 - Send out $(1, p), (2, p), (3, p), \dots, (1000, p)$ as output
 - The "1" in $(1, p)$ is the ID of a point
 - Also, send out (i, p) for 100 randomly selected values of i in $(1001 \dots n)$
 - This means that the first 1000 points in candidates will get ALL of the data
 - And the rest of the points will get only a subset of the data

Outlier Detection on MapReduce

- In map phase, process both "Candidates" and "Data"
- When you get a point p from "Data":
 - Send out $(1, p), (2, p), (3, p), \dots, (1000, p)$ as output
 - The "1" in $(1, p)$ is the ID of a point
 - Also, send out (i, p) for 100 randomly selected values of i in $(1001\dots n)$
 - This means that the first 1000 points in candidates will get ALL of the data
 - And the rest of the points will get only a subset of the data
- When you get a point p from "Candidates"
 - Send it out directly, use ID as the key

Outlier Detection on MapReduce

- So at the reducers:
 - For each point...
 - ...you'll get each point from "Candidates"...
 - ...plus some portion of "Data"
 - A candidate with ID in $1 \dots 1000$ will get all of the data
 - A candidate with ID in $1001 \dots n$ will get a random subset of the data
- For a given point:
 - Scan thru data assigned it from "Candidates"...
 - ...and use t update its "kNN so far list"

Outlier Detection on MapReduce

- A candidate with a ID value in 1...1000...
 - ...will have its exact kNN list computed
 - It got all of the data!
 - Either add to the "top list" or discard
- All others
 - Will have a list of lower bounds in "kNN so far list"
 - discard if lower bound exceeds the current value of *small*

Repeat Iteratively

- Repeat this MapReduce iteratively until everyone...
 - Is either in the "top list"
 - Or has been discarded!
- The hope is...
 - That a significant fraction of the points are killed every iteration
 - So has $n \log n$ time complexity

End With Some Popular Hadoop Tools

- **Mahout** is an open-source Apache project that contains many machine learning and data mining algorithms
- Most (though not all) run on top of Hadoop
 - Major move towards Spark
 - Though all are supposed to be scalable
- A “Mahout” is an elephant handler
 - Hadoop’s mascot is an elephant
 - Hence the name “Mahout”

End With Some Popular Hadoop Tools

- **Hive** is an open-source Apache project
- Gives you a scripting language that looks a lot like SQL
- Remember, people like MapReduce because they dislike SQL!
- But Hive still has some advantages over an SQL database
 - Ex: no need to load data into a database
 - Can make a file in HDFS (storing emails, for example) look like a DB table
- FaceBook is a major developer of Hive

End With Some Popular Hadoop Tools

- **Pig** is an open-source Apache project
- Gives you a scripting language that looks a bit like SQL

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS (line:chararray);
-- Extract words from each line and put them into a pig bag
-- datatype, then flatten the bag to get one word on each row
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
-- filter out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\\w+';
-- create a group for each word
word_groups = GROUP filtered_words BY word;
-- count the entries in each group
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;
-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

* shamelessly stolen from Wikipedia

And Mention Are Some Non-Hadoop Tools

- Are many “key-value” stores for Big Data
- Used for writing programs...
 - ...that do work a lot closer to traditional transactional databases
 - ...but where you want to use a Hadoop-like system architecture
 - ...and you don't care so much about ACID (lesser guarantees)
- Part of Hadoop universe, not part of Hadoop
- Example: MongoDB, Apache Cassandra

Hadoop Itself Is Evolving

- Moving away from Hadoop MapReduce
- To using Apache Spark for data processing
- Runs on top of HDFS
- Has MapReduce plus many other operations
- Faster, aggressive in-RAM caching, cleaner API
- Why didn't we cover Spark? Big reason: strongly linked to Scala.

That's It!

Questions?