DATA ANALYTICS PROGRAMMING ON SPARK: PYTHON IN THE CLOUD

Chris Jermaine
cmj4@cs.rice.edu

15 Years Ago…

- Say you had a "big" data set, wanted 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 100GB+ in 2002, TB+ now (all the way to dozens of PB)

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

③

But What About the Software?

- 15 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

④

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

⑤

But People Not Happy With SQL Databases

- Also quite expensive: even today, pay $5K to $50K/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 2004, 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?

⑥

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.

⑦

By Early-Mid 2000's...

- The Internet companies (Google, Yahoo!, etc.)...
    - — ...had some of the largest databases in the world

- But they 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

⑧

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

9

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

10

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 no 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

(15)

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

(16)

MapReduce Had a Huge Impact

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

17

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

18

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

19

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

20

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

21

Why Popular? (5)

• Software is free!

22

By 2010, Hadoop Was Widely Used

- Open-source implementation of Google MapReduce
    - — Had Hadoop MapReduce
    - — Plus Hadoop distributed file system

- But the cracks were appearing in the edifice...

Hadoop MR Word Count Java Code

```
import java.util.*;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;

public class WordCount {

  public static int main(String[] args) throws Exception {

    // if we got the wrong number of args, then exit
    if (args.length != 4 || !args[0].equals ("-r")) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // Get the default configuration object
    Configuration conf = new Configuration ();

    // now create the MapReduce job
    Job job = new Job (conf);
    job.setJobName ("WordCount");

    // we'll output text/int pairs (since we have words as keys and counts as values)
    job.setMapOutputKeyClass (Text.class);
    job.setMapOutputValueClass (IntWritable.class);

    // again we'll output text/int pairs (since we have words as keys and counts as values)
    job.setOutputKeyClass (Text.class);
    job.setOutputValueClass (IntWritable.class);

    // tell Hadoop the mapper and the reducer to use
    job.setMapperClass (WordCountMapper.class);
    job.setCombinerClass (WordCountReducer.class);
    job.setReducerClass (WordCountReducer.class);

    // we'll be reading in a text file, so we can use Hadoop's built-in TextInputFormat
    job.setInputFormatClass (TextInputFormat.class);

    // we can use Hadoop's built-in TextOutputFormat for writing out the output text file
    job.setOutputFormatClass (TextOutputFormat.class);

    // set the input and output paths
    TextInputFormat.setInputPaths (job, args[2]);
    TextOutputFormat.setOutputPath (job, new Path (args[3]));

    // set the number of reduce paths
    try {
      job.setNumReduceTasks (Integer.parseInt (args[1]));
    } catch (Exception e) {
      System.out.println("usage: WordCount -r <num reducers> <input> <output>");
      return -1;
    }

    // force the mappers to handle one megabyte of input data each
    TextInputFormat.setMinInputSplitSize (job, 1024 * 1024);
    TextInputFormat.setMaxInputSplitSize (job, 1024 * 1024);

    // this tells Hadoop to ship around the jar file containing "WordCount.class" to all of
    the different
    // nodes so that they can run the job
    job.setJarByClass(WordCount.class);

    // submit the job and wait for it to complete!
    int exitCode = job.waitForCompletion (true) ? 0 : 1;
    return exitCode;
```

Not pretty!
Programmer burden too high...

Many Felt Hadoop MR Too Slow

- Data read from HDFS again for each MR job

- Bad for iterative data processing

    — Many analytics tasks process and reprocess data

25

And API Too Restrictive

- Can only do Map

- Or MapReduce

- Everything else in terms of those operations... annoying!

26

So Hadoop MR Used Less and Less

- Issues led to other dataflow platforms to replace Hadoop MR

  — Notable are Spark and Flink

  — We will study Spark

- Interesting fact:

  — While Hadoop MR is dying...

  — Hadoop DFS is going strong

  — De-facto standard for Big Data management

  — Looks set to stay that way for a very long time

27

Apache Spark

- #1 Hadoop MR killer

- What is Spark?

  — Platform for efficient, distributed data analytics

- Basic abstraction: **Resilient Distributed Data Set** (RDD)

- RDD is a data set buffered in RAM by Spark

  — Distributed across machines in cluster

  — To create and load an RDD (in Python):

```
myRDD = sc.textFile (someFileName) # sc is the Spark context
# or else...
data = [1, 2, 3, 4, 5]
myRDD = sc.parallelize (data) # or
myRDD = sc.parallelize (range (20000)) # or...
```

28

Computations: Series of Xforms Over RDDs

• Example: word count

— Count number of occurs of each distinct word in a corpus

```
def countWords (fileName):
    textFile = sc.textFile (fileName)
    lines = textFile.flatMap (lambda line: line.split(" "))
    counts = lines.map (lambda word: (word, 1))
    aggCounts = counts.reduceByKey (lambda a, b: a + b)
    retrun aggCounts.top (200, key=lamda p: p[1])
```

• What transforms do we see here?

— `flatMap`, `map`, `reduceByKey`, `top`

• Let's go through them...

But First… What's a Lambda?

• Basically, a function that I can pass like a variable

• Key ability: can "capture" its surroundings at creation

```
def addTwelveToResult (myLambda):
    return myLambda (3) + 12

a = 23
aCoolLambda = lambda x : x + a
addTwelveToResult (aCoolLambda) # prints 38

a = 45
addTwelveToResult (aCoolLambda) # prints ???
```

Lambdas and Comprehensions

• Lambdas can return many items

```
def sumThem (myLambda):
     tot = 0
     for a in myLambda ():
          tot = tot + a
     return tot

x = np.array([1, 2, 3, 4, 5])
iter = lambda : (j for j in x)
sumThem (iter) # prints 15
```

Lambdas and Comprehensions

• Lambdas can return many items

```
def sumThem (myLambda):
     tot = 0
     for a in myLambda ():
          tot = tot + a
     return tot

x = np.array([1, 2, 3, 4, 5])
iter = lambda : (j for j in x)
sumThem (iter) # prints 15
```

• Exercise: change `iter` so accepts param, adds to each item, then:

```
def sumThem (myLambda, addMeIn):
     tot = 0
     for a in myLambda (addMeIn):
          tot = tot + a
     return tot                # try sumThem (iter, 5)
```

flatMap ()

```
def countWords (fileName):
    textFile = sc.textFile (fileName)
    lines = textFile.flatMap (lambda line: line.split(" "))
```

- Process every data item in the RDD

- Apply lambda to it

- Lambda argument to return zero or more results

- Each result added to resulting RDD

(33)

map ()

```
def countWords (fileName):
    textFile = sc.textFile (fileName)
    lines = textFile.flatMap (lambda line: line.split(" "))
    counts = lines.map (lambda word: (word, 1))
```

- Process every data item in the RDD

- Apply lambda to it

- But the lambda must return **exactly** one result

(34)

reduceByKey ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(" "))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
```

- Data must be (*Key, Value*) pairs

- Shuffle so that all (*K, V*) pairs with same *K* on same machine

- Organize into $(K, (V_1, V_2, ..., V_n))$ pairs

- Use the lambda to "reduce" the list to a single value

top ()

```
def countWords (fileName):
     textFile = sc.textFile (fileName)
     lines = textFile.flatMap (lambda line: line.split(" "))
     counts = lines.map (lambda word: (word, 1))
     aggCounts = counts.reduceByKey (lambda a, b: a + b)
     retrun aggCounts.top (200, key=lamda p: p[1])
```

- Data must be (*Key, Value*) pairs

- Takes two params... first is number to return

- Second (optional): lambda to use to obtain key for comparison

- Note: `top` **collects** an RDD, moving from cloud to local

- So result is **not** an RDD

filter ()

- Not used in example code

- But needed for first activity

- Accepts a boolean-valued lambda

- That lambda applied to each item in RDD

- Item removed from result if and only if lambda returns `false`

An Important Note

- Lazy evaluation... if I run this code:

```
textFile = sc.textFile (fileName)
lines = textFile.flatMap (lambda line: line.split(" "))
counts = lines.map (lambda word: (word, 1))
aggCounts = counts.reduceByKey (lambda a, b: a + b)
```

- Nothing happens! (Other than Spark remembers the ops)

  — Spark does not execute until an attempt made to collect an RDD

  — When we hit `top()`, **then** all of these are executed

- Why do this?

  — By waiting until last possible second, opportunities for "pipelining" exploited

  — Only ops that require a shuffle can't be pipelined

Another Important Note

- Lambda capture **by value** at a difficult-to-determine instant

    — All referenced variables serialized and broadcast

    — Sometime between definition of RDD transform...

    — And lazy evaluation

    — Very useful! Standard way of broadcasting local state to distributed computations

    — So be careful! Never rely on side-effects...

Activity One

- Word count on Amazon EC2

    — Word count is everyone's favorite first-big-data-computation!

- See `cmj4.web.rice.edu/DSDay2/GettingStarted.html`

Next, We'll Look at Text Analytics on Spark

- 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]

45

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]

— After "by" and "radiotherapy", have x = [0, 0, 1, 1, 0, 1, 1, 0, 1, 0]

46

The Classic Workflow

- $x = [0, 0, 1, 1, 0, 1, 1, 0, 1, 0]$ is then treated as a *feature vector*

- Now want to figure out how to classify (label) text documents...

    — For example: "+1: this patient had breast cancer"

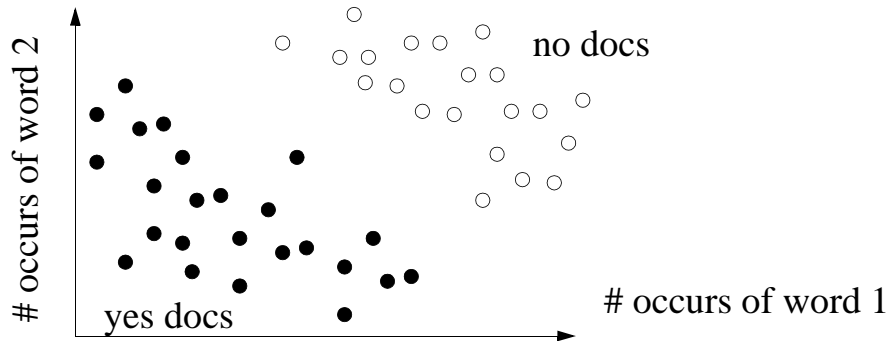   — "-1: this patient didn't have breast cancer"

- How?

47

The Classic 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
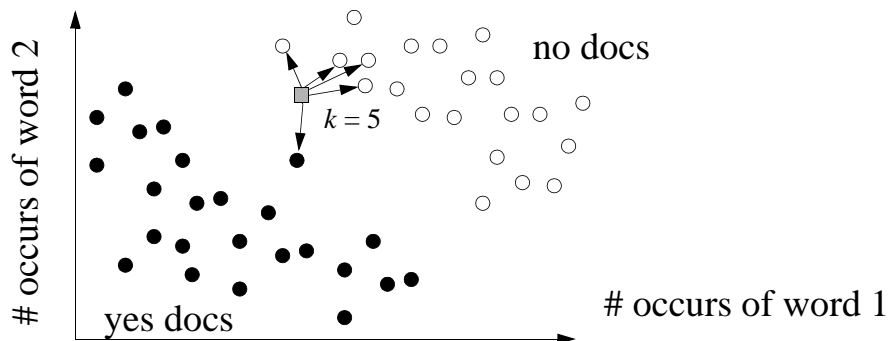
48

A Common Classifier is "KNN"

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

A Common Classifier is "KNN"

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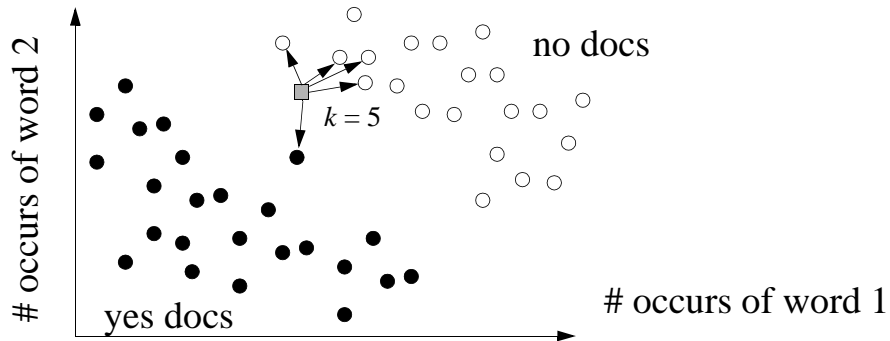- Idea: place docs in multi-dim space

- 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 first one that you'll be implementing on Spark
- 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

---

A Common Classifier is "KNN"

- This is the first one that you'll be implementing on Spark
- 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
- How to define distance?
    — Often use *ED* (note: small values mean "close"):

$$ED(x_1, x_2) \; = \; \sum_{w=1}^{d} (x_1[w] - x_2[w])^2$$

— We will use **cosine similarity** (note: big values mean "close"):

$$Cos(x_1, x_2) \; = \; \sum_{i = 1}^{d} (x_1[w] \times x_2[w])$$

Just About Ready for Activity Two

• But first some new RDD transformations we'll be using

— `join ()`, `groupByKey ()`, `aggregateByKey ()`

join ()

- Given two data sets `rddOne`, `rddTwo` of (*Key, Value*) pairs
- `rddOne.join (rddTwo)` returns (*K*, (*V1, V2*)) pairs
- constructed from all (*K1, V1*) from `rddOne`, (*K2, V2*) from `rddTwo`, where *K1 == K2*
- Can blow up RDD size if join is many-to-many
- Requires expensive shuffle!

groupByKey ()

- Data must be (*Key, Value*) pairs
- Shuffle so that all (*K, V*) pairs with same *K* on same machine
- Organize into $(K, (V_1, V_2, ..., V_n))$ pairs
- Store each list as a `ResultIterable` for future processing
- Like `reduceByKey ()` but without the reduce

aggregateByKey ()

- Like `reduceByKey ()`

- Data must be (*Key, Value*) pairs

- Organize into $(K, (V_1, V_2, ..., V_n))$ pairs

- Then aggregate the list, like `reduceByKey ()`

- With `reduceByKey ()` aggregate directly, can be restrictive...

  — what if values are cities, and we want the list of unique cities?

- `aggregateByKey ()` takes three args

  — The "zero" to init the aggregation

  — Lambda that takes `x1`, `x2` and aggs them, where `x1` already aggregated, `x2` not

  — Lambda that takes `x1`, `x2` and aggs them, where both already aggregated

57

Activity Two

- KNN over bag-of-words vectors

- See `cmj4.web.rice.edu/DSDay2/kNNBasic.html`

58

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

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

  — $IDF(w) = \log\left(\dfrac{\text{size of corpus}}{\displaystyle\sum_{x \text{ in corpus}} 1 \text{ if } (x[w] \geq 1),\ 0 \text{ otherwise}} \right)$, the "inverse doc freq"

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

- Often helps cause it weights rare words more highly

- In our next activity, we use TF-IDF in our KNN classifier

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-gram*s, not just individual words
  - Given "This was followed by radiotherapy", 2-grams are: "This was", "was fol-lowed", "followed by", "by radiotherapy"
  - Helps to address bag-of-words' total lack of contextual knowledge

Activity Three

- KNN over TF-IDF bag-of-words vectors

- See `cmj4.web.rice.edu/DSDay2/kNNTFIDF.html`

Linear Regression

- KNN often works well, but it's expensive

- Requires complete scan of data

- Alternative: LR

Linear Regression

- KNN often works well, but it's expensive

- Requires complete scan of data

- Alternative: LR

- In LR, compute a vector of regression weights *r*

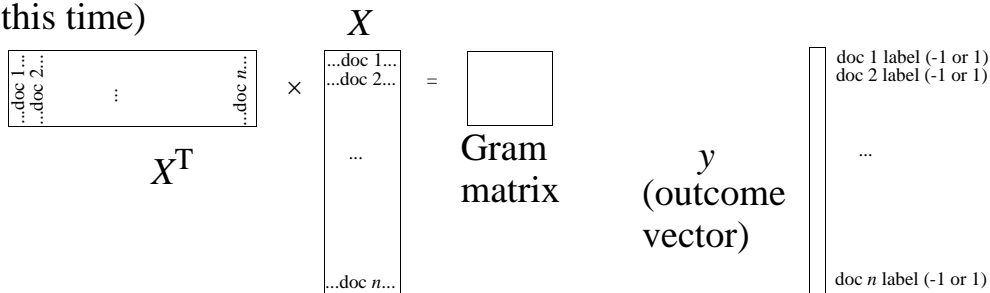- Then to classify TF-IDF vector *x*, compute:

$$\sum_{w=1}^{d} x[w] \times r[w]$$

- If greater than 0, say "yes", otherwise "no"

- Drawback: simple regression only allows two classes

  — KNN trivially extends to many classes

63

How To Compute?

- Let *X* be the data matrix, *y* the outcome vector

  — Each row in *X* is a doc, column TF-IDF value for a word

- The Gram matrix is $X^T X$... we did this yesterday! (but no trimming this time)

- *r* is computed as (Gram matrix)$^{-1} X^T y$

64

How To Compute on Spark?

- Gram matrix

  — We know matrix multiply fastest, but not feasible here

  — Why? *X* is distributed in an RDD

  — So use `map ()` that computes outer product

  — And then use `aggregate ()` to sum them, collect locally, invert locally

  — `aggregate ()` just like `aggregateByKey ()`, but no grouping by key

- Then, use a `map()` to multiply (Gram mat)$^{-1}$ by every data record by to get (Gram mat)$^{-1}X^{\mathrm{T}}$ stored as one vec per doc

- Finally, multiply every result rec by corresponding outcome/class label and sum using `aggregate ()` to get (Gram matrix)$^{-1}X^{\mathrm{T}}y$

- That's it! Well, almost...

Gram Matrix Too Expensive

- Have to sum 20,000 different 20,000 by 20,000 matrices

- Not gonna happen fast enough

- So map each 20,000-dimensional vector down to 1,000 dims

  — Now we sum 20,000 different 1,000 by 1,000 matrices

  — Much more reasonable!

Gram Matrix Too Expensive

- How to map?

  — Could use something like PCA, which is classic

  — But too expensive!

- We just use a random mapping...

  — In practice, as good as PCA unless you map to very low dimensions

- Fill a matrix M of 1,000 columns and 20,000 rows with samples from a Normal (0, 1) distribution

- Then, before we compute (Gram matrix)$^{-1}X^{T}y$...

  — Just multiply each data vector with M

- 20,000-dimensional problem becomes 1,000 dimensional!

Activity Four

- LR over TF-IDF bag-of-words vectors
- Tries to classify religion vs. not
- See `cmj4.web.rice.edu/DSDay2/LinReg.html`

That's It!

- Questions?