Behavioral Data Mining

Lecture 7
Hadoop and MapReduce
(slides from Matei Zaharia)
Wrap-up from last time

- MCMC + Gibbs Sampling
- What it all means
Back to Inference

• **Estimation**: Given a joint distribution $Pr(X, Y)$ on observed data $Y$ and unobserved data $X$, we want to estimate $X$ given $Y$. We may want:
  - MAP estimates: the mode of the posterior $Pr(X | Y)$
  - Condition means: $E(X | Y)$

In practice it may only be possible to get a local max or mean.

• **Model Inference**: Since we cant know the true $Pr(X, Y)$, we choose a family of models $M$ with tractable $Pr(X, Y | M)$ and then find a “best” model (e.g. minimum loss).
  - Most model inference formulations are not closed form, so an iteration is needed to find the best model.
MCMC

Basics:

• The expected value of a sum is the sum of expected values (no independence needed), so a random mean can be approximated as a mean of random values with the same mean.

• A Markov chain is a sequence of random values $X_0, X_1, X_2, \ldots$ such that the distribution of $X_{i-1}$ depends only on the value of $X_i$.

• So if you can generate a Markov chain whose stationary distribution is the posterior probability, any posterior statistics can be estimated.
**MCMC**

**Metropolis Hastings:** $X_t$ is the current state.

1. Sample a point $Y$ from a proposal distribution $q(.|X_t)$
2. With probability $\alpha(X,Y)$ accept the new point and set $X_{t+1} = Y$.

where

$$\alpha(X,Y) = \min \left( 1, \frac{\pi(Y)q(X|Y)}{\pi(Y)q(Y|X)} \right)$$

And $\pi(.)$ is the distribution of interest, usually the posterior probability.

The stationary distribution of this sampler is $\pi(.)$ and we can estimate the statistics of any variable derived from $X$. 
MCMC

1. Sample a point $Y$ from a proposal distribution $q(.|X_t)$

2. With probability $\alpha(X, Y)$ accept the new point and set $X_{t+1} = Y$.

\[
\alpha(X, Y) = \min \left( 1, \frac{\pi(Y)q(X|Y)}{\pi(X)q(Y|X)} \right)
\]

Very simple, but it's not magic.

Note that the sampler will spend most time in high-probability states. If the proposed $Y$ is too “far” from $X_t$ it will have low probability and the chain will never move.

A very successful strategy for this is the Gibbs Sampler, which changes one variable at a time.
Gibbs Sampler

Proposal distribution is:

\[ q_i(Y_i \mid X) = \pi_i(Y_i \mid X_{-i}) \]

where \( X_{-i} = X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n \)

For this \( q_i \), the acceptance probability is 1.

Its exceptionally simple if the \( X_i \) are binary or categorical variables. Then \( q_i(Y_i \mid X) \) is simply a vector of probabilities from which we can directly sample.
Gibbs Sampler

There are lots of principled and semi-principled improvements:

1. Update independent blocks of variables in parallel.
2. Delay (mini-batch) updates to model parameters, rather than updating with every block – Smola et al. paper
3. Use collapsed inference for continuous variables where possible.
4. Draw multiple samples for each variable (skip-ahead) in one time step:
   – Bernoulli samples $\rightarrow$ Binomial samples
   – Categorical samples $\rightarrow$ Multinomial samples
   – Or approximate both with Poisson distributions
Open Problem

Gibbs sampler approaches so far have been much slower than algebraic (e.g. variational LDA) ones.

• With collapsed sampling, the number of “operations” is essentially the same.

• But implementation choices lead to huge constant factors
  – Java random numbers are quite slow
  – Updating random memory locations is very slow

• Instead, dense parametric GPU-based random number generators can do the same calculations orders of magnitude faster.
Outline

• Design patterns for Behavioral Modeling
• Stochastic Gradient Descent
• Second-Order SGD
• MCMC + Gibbs Sampling
• What it all means
Convex vs non-Convex Optimization

• Many simple optimization problems are convex (e.g. regression), and the choice of optimization strategy affects only the rate of convergence.

• But most non-trivial models are non-convex (e.g. factor and cluster models, latent variable models), and have multiple local risk minima. The optimization strategy can have a significant affect on final accuracy.
Convex vs non-Convex Optimization

• Classical gradient descent is a “deterministic” algorithm. It will usually find a nearby local minimum of risk.

• SGD adds “randomness” to the gradient estimates. It moves both with and against the gradient, and can move away from local minima.

• Gibbs samplers in principle explore the entire posterior space, but in practice often wander only near a local minimum.
Convex vs non-Convex Optimization

- For this reason, it’s common to start Gibbs samplers (or other MCMC estimators) from many random initial points.
- Since some trajectories can wander far from the minima, they can be periodically pruned.
- The acceptance probability can be adjusted (down) – to reduce the “temperature” of the sampler.
- This process (called annealing) eventually causes the sampler to settle in a true local minimum.
- If the loss is a negative log probability, then the local risk minimum is a local mode of probability.
Summary

• Design patterns for Behavioral Modeling
• Stochastic Gradient Descent
• Second-Order SGD
• MCMC + Gibbs Sampling
• What it all means
What is MapReduce?

- **Data-parallel** programming model for *clusters of commodity machines*

- Pioneered by Google
  - Processes 20 PB of data per day

- Popularized by open-source **Hadoop** project
  - Used by Yahoo!, Facebook, Amazon, …
What is MapReduce used for?

• At Google:
  – Index building for Google Search
  – Article clustering for Google News
  – Statistical machine translation

• At Yahoo!:
  – Index building for Yahoo! Search
  – Spam detection for Yahoo! Mail

• At Facebook:
  – Data mining
  – Ad optimization
  – Spam detection
Outline

- MapReduce architecture
- Fault tolerance in MapReduce
- Sample applications
- Getting started with Hadoop
MapReduce Design Goals

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 50 MB/s = 23 days
   - Scan on 1000-node cluster = 33 minutes

2. **Cost-efficiency:**
   - Commodity nodes (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)
Typical Hadoop Cluster

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo terasort):
  8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)
Typical Hadoop Cluster
Challenges

• **Cheap nodes fail, especially if you have many**
  – Mean time between failures for 1 node = 3 years
  – MTBF for 1000 nodes = 1 day
  – **Solution:** Build fault-tolerance into system

• **Commodity network = low bandwidth**
  – **Solution:** Push computation to the data

• **Programming distributed systems is hard**
  – **Solution:** Data-parallel programming model: users write “map” and “reduce” functions, system handles work distribution and fault tolerance
Hadoop Components

• Distributed file system (HDFS)
  – Single namespace for entire cluster
  – Replicates data 3x for fault-tolerance

• MapReduce implementation
  – Executes user jobs specified as “map” and “reduce” functions
  – Manages work distribution & fault-tolerance
Hadoop Distributed File System

- Files split into 128MB blocks
- Blocks replicated across several datanodes (usually 3)
- Single namenode stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only
MapReduce Programming Model

- Data type: key-value records

- Map function:
  \[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]

- Reduce function:
  \[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]
MapReduce Programming Model

- To map tabular data to MapReduce, use keys which are row/column tuples:
  - Row-based:
    \[ K = (K_{row}, K_{column}) \]
  - Column-based:
    \[ K = (K_{column}, K_{row}) \]
  - or lumped row-based:
    \[ K = K_{row}, V = list(v_{col1}, v_{col2}, v_{col3}, \ldots) \]
Example: Word Count

def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
Word Count Execution

Input

the quick brown fox

the fox ate the mouse

how now brown cow

Map

the, 1
brown, 1
fox, 1

Map

how, 1
now, 1
brown, 1
mouse, 1

Map

cow, 1

Reduce

the, 1
fox, 1
the, 1

Reduce

quick, 1

Reduce

ate, 1
cow, 1
mouse, 1
quick, 1

Output

brown, 2
fox, 2
how, 1
now, 1
the, 3
SQL analogue

Think:

```
SELECT red_fcn(cols) FROM
    (SELECT map_fcn(cols) FROM data_source)
GROUP BY key
```
MapReduce Execution Details

- Single *master* controls job execution on multiple *slaves* as well as user scheduling
- Mappers preferentially placed on same node or same rack as their *input block*
  - Push computation to data, minimize network use
- Mappers save outputs to local disk rather than pushing directly to reducers
  - Allows having more reducers than nodes
  - Allows recovery if a reducer crashes
An Optimization: The Combiner

• A combiner is a local aggregation function for repeated keys produced by same map
• For associative ops. like sum, count, max
• Decreases size of intermediate data
• Very often the same function as the reducer

• Example: local counting for Word Count:

  ```python
  def combiner(key, values):
      output(key, sum(values))
  ```
Word Count with Combiner

The quick brown fox
the fox ate
the mouse
how now brown cow

Input
Map & Combine
Shuffle & Sort
Reduce
Output

Map
the, 2
brown, 1
fox, 1

Map
the, 1
brown, 1
fox, 1

Map
how, 1
now, 1
brown, 1
mouse, 1

Reduce
brown, 2
fox, 2
how, 1
now, 1
the, 3

Reduce
ate, 1
cow, 1
mouse, 1
quick, 1
Outline

- MapReduce architecture
- Fault tolerance in MapReduce
- Sample applications
- Getting started with Hadoop
Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     • Okay for a map because it had no dependencies
     • Okay for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block (user-controlled)

➤ Note: For this and the other fault tolerance features to work, your map and reduce tasks must be side-effect-free
Fault Tolerance in MapReduce

2. If a node crashes:
   – Relaunch its current tasks on other nodes
   – Relaunch any maps the node previously ran
     • Necessary because their output files were lost along with the crashed node
Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   – Launch second copy of task on another node
   – Take the output of whichever copy finishes first, and kill the other one

• On behavioral data (usually power law) you will often get structural stragglers – nodes whose random share of the data is very large. These require special treatment – truncation, splitting,…
Takeaways

• By providing a data-parallel programming model, MapReduce can control job execution in useful ways:
  – Automatic division of job into tasks
  – Automatic placement of computation near data
  – Automatic load balancing
  – Recovery from failures & stragglers

• User focuses on application, not on complexities of distributed computing
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• Getting started with Hadoop
I. Search

• **Input:** (lineNumber, line) records
• **Output:** lines matching a given pattern

• **Map:**
  
  ```
  if(line matches pattern):
    output(line)
  ```

• **Reduce:** identify function
  – Alternative: no reducer (map-only job)
2. Sort

- **Input:** (key, value) records
- **Output:** same records, sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick partitioning function $h$ such that $k_1 < k_2 \Rightarrow h(k_1) < h(k_2)$
3. Inverted Index

• **Input:** (filename, text) records
• **Output:** list of files containing each word

• **Map:**
  ```python
  foreach word in text.split():
    output(word, filename)
  ```

• **Combine:** uniquify filenames for each word

• **Reduce:**
  ```python
  def reduce(word, filenames):
    output(word, sort(filenames))
  ```
Inverted Index Example

hamlet.txt
  to be or not to be

12th.txt
  be not afraid of greatness

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt
4. Most Popular Words

- **Input:** (filename, text) records
- **Output:** the 100 words occurring in most files

- Two-stage solution:
  - **Job 1:**
    - Create inverted index, giving (word, list(file)) records
  - **Job 2:**
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job

- Optimizations:
  - Map to (word, 1) instead of (word, file) in Job 1
  - Estimate count distribution in advance by sampling
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• Getting started with Hadoop
Getting Started with Hadoop

• Run jobs on icluster1.eecs.berkeley.edu
OR
• Download from hadoop.apache.org
• To install locally, unzip and set JAVA_HOME
• Details: hadoop.apache.org/core/docs/current/quickstart.html

• Three ½ ways to write jobs:
  – Java API
  – Hadoop Streaming (for Python, Perl, etc)
  – Pipes API (C++)
  – Write in Scala and run a jar with the runtime
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable ONE = new IntWritable(1);

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {

        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            output.collect(new text(itr.nextToken()), ONE);
        }
    }
}
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output,
        Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    FileInputFormat.setInputPaths(conf, args[0]);
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setOutputKeyClass(Text.class); // out keys are words (strings)
    conf.setOutputValueClass(IntWritable.class); // values are counts

    JobClient.runJob(conf);
}
Word Count in Python with Hadoop Streaming

Mapper.py:
```python
import sys
for line in sys.stdin:
    for word in line.split():
        print(word.lower() + "\t" + 1)
```

Reducer.py:
```python
import sys
counts = {}
for line in sys.stdin:
    word, count = line.split("\t")
    dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
    print(word.lower() + "\t" + 1)
```
The Good, the Bad, the Ugly

- MapReduce is extremely popular, especially since the release of Hadoop.
- Availability in cloud services (like EC2) makes it usable by just about anyone.
- While you can’t do everything in Hadoop, its task coverage seems to be very high.
- Hash partitioning, the basic sort/group model, and its error recovery make it relatively simple to use as a programming model.
- Leverages Java serialization, reflection,…
- Fits well with higher-level interfaces like Hive, Pig etc.
The Good, the Bad, the Ugly

- By itself, Hadoop/MR is too low-level for large-scale programming.
- It is procedural, and doesn’t support query optimization.
- Job turnaround time is high because of rigidity in the design.
- On a cluster, you don’t have to do everything “well”, and its easy to compensate for a weak implementation with extra cycles.
The Good, the Bad, the Ugly

• Mapper/reducer code has poor structure compared to columnar design (e.g. SQL functions).

• Hard memory partitioning in Hadoop make it a poor match for machine learning algorithms with large models.

• “stateless” mappers and reducers are also inefficient for ML algorithms that work incrementally.
Outline

• MapReduce architecture
• Fault tolerance in MapReduce
• Sample applications
• Getting started with Hadoop
• Hadoop and machine learning
Conclusions

• MapReduce’s data-parallel programming model hides complexity of distribution and fault tolerance

• Principal philosophies:
  – *Make it scale*, so you can throw hardware at problems
  – *Make it cheap*, saving hardware, programmer and administration costs (but requiring fault tolerance)

• MapReduce is not suitable for all problems, but when it works, it may save you a lot of time

• Even with many cores, data representation dominates performance. Use efficient designs with one or many CPUs.
Resources

• Hadoop: http://hadoop.apache.org/core/
• Hadoop docs: http://hadoop.apache.org/core/docs/current/
• Pig: http://hadoop.apache.org/pig
• Hive: http://hadoop.apache.org/hive
• Hadoop video tutorials from Cloudera: http://www.cloudera.com/hadoop-training