COMP6208 – Advanced Machine Learning

1

Big Data Processing

Summary

Big data is currently a very hot topic in machine learning, data analytics and computer science in general. Distributed computing techniques can help solve these problems, however, there can be problems with more traditional distributed computing methods as these require moving data to the processing units; with even moderately big data this is a massive bottleneck due to the limited I/O speeds of networks and disks. To be truly effective, distributed processing needs to be coupled with distributed data storage, and computational tasks need to be "pushed to the data" in order to minimise the amount of data that needs to be transferred. MapReduce is a popular computational framework that aims to achieve these goals by posing the computational processing task as *map* and *reduce* functions that are readily distributed to the data.

Key points

- What is big data?
  - No one really knows – it's just a trendy buzzword!
    - "**Big data** is an evolving term that describes any voluminous amount of structured, semi-structured and unstructured **data** that has the potential to be mined for information."
    - "**Big data** is a buzzword, or catch phrase, used to describe a massive volume of both structured and unstructured **data** that is so **large** that it's difficult to process using traditional database and software techniques."
  - For this lecture, we'll consider big data to be data that is too **large to be effectively stored on a single machine or processed in a time-efficient manner by a single machine.**
- In order to process big data we need to use **distributed computation.**
  - Use **clusters** of machines to perform processing, taking advantage of many CPUs working in **parallel.**
    - Each machine in a cluster is called a **node**.
  - **Data transfer isn't free**, so it doesn't make sense to store the data centrally and pull it to the processors.
    - Reading data from a disk is slow
      - Mechanical disks typically 140MB/s peak
      - Consumer SSDs go up to 600MB/s or more, but have limited storage.
      - Enterprise SSDs go up to GB/s speed, but have limited storage and are *extremely* expensive!
      - In all cases actual speeds are much slower, especially if you're both reading and writing.
      - Actual speeds can be improved by:
        - Bulk reads and writes (avoiding small files; reading and writing large blocks).
        - Attaching multiple drives to a single machine on different buses.
    - Networking is also slow
      - Gigabit Ethernet can only do about 125MB/s at the theoretical maximum
      - Infiniband can do >600MB/s but is *really* expensive
        - The fastest networking would still be limited by the slowest disk
    - ***Take away message: moving data between machines is really bad for processing performance.***
  - If we **distribute the data** across the cluster, then can we **push computation to the data**!
    - Older models of distributing computing **pull** the data from a centralized storage to the nodes, but this is not efficient for "big data" processing due to bandwidth limitations.
    - Data can be distributed in different ways:
      - **Horizontally Partition** the dataset across multiple nodes – this is commonly called **sharding** (each partition is known as a **shard**) & is a popular way of distributing database systems.
      - Store data on a **distributed file system (DFS)**.
        - Typically, the DFS will work with data at the block level.

Written by Jonathon Hare. Send any comments/bugs/typos to jsh2@ecs.soton.ac.uk.

        - File system blocks belonging to a single logical file will be **spread across** the disks in each machine in the cluster.
        - DFS blocks are usually much bigger than those found on a normal file system (normal desktop fs: 512B to 4KB; typical DFS: 64MB)
            - Assumption is that the individual data files are large (orders of magnitude larger than the block size) thus meaning that the task of keeping track of all the blocks is more manageable as there will be fewer.
            - This is also more optimal for efficiently transferring blocks between nodes.
            - It does however mean that you **don't want to store lots of small files on the DFS**!
                - Related small files representing records of the same dataset can be combined into a larger container file.
        - **Block replication** (across different nodes) can be used to ensure redundancy in case a drive or node in the cluster fails.
            - **Rack-awareness** can ensure that block replicas are in different physical racks making for even more redundancy.
- **MapReduce** is a framework for *efficient distributed computing over distributed data*
    - The MapReduce conceptual programming model is founded in ideas from functional programming, although they are used in a much more flexible way in the MapReduce model.
        - Fundamentally, the MapReduce programming model specifies a program that **transforms a list of *<key, value>* pairs of data into a list of values.**
        - The model has three parts: a Map and Reduce function that are written by the user, and a distributed *shuffle* phase that is provided by the framework.
            - The **Map** function takes in a key-value pair from the data being processed and outputs zero or more key-value pairs of data (which can be of a different type to the input): ***Map(k1, v1) → list(k2, v2)***
            - The **shuffle** phase collates all the Map function outputs by their key, producing a list of *<key, [values]>*, sorted by key: ***list(k2, v2) → list(k2, list(v2))***
            - The **Reduce** function is handed each *<key, [values]>* pair and produces zero or more values as a result: ***Reduce(k2, list(v2)) → list(v3)***
        - The MapReduce programming model is useful because it is easily parallelizable.
            - Provided that each map operation is independent of the others, all maps can be performed in parallel (in practice this is limited by the number of data sources and processing units).
            - The reduce phase can be performed in parallel as the data for each key is processed independently.
        - It should be noted that in practice, MapReduce programs often contain **multiple Map and Reduce operations**, and sometimes even **omit the Reduce operation**.
    - MapReduce is designed to work over distributed data.
        - Assumption is that Map operations run "near" to the data they are operating on in order to minimize the amount of data that is transferred.
            - Generally the data is stored on a distributed file system or is sharded.
            - Requires intelligent scheduling to best make use of cluster resources.
                - Data replication can give the scheduler more options.
        - Restrictions on the data:
            - Input and output is assumed to be in the form of key-value pairs.
            - How does this work for (say) text files that are Mapped line-by-line?
                - Create a key from the line number? random key? Null key?
            - Actual implementations are quite flexible about input and output formats and allow for custom schemes to select (or generate) keys & values.
    - **Hadoop** is an open-source cluster-computing framework with MapReduce and DFS implementations.
        - Initiated by Doug Cutting and Mike Cafarella at Yahoo! as part of the Nutch web search engine, and based on ideas from the Google MapReduce and GFS (Google File System – Google's internal DFS implementation).
        - Now maintained/developed by the Apache Foundation.

        - (Mostly) written in Java.
        - Hadoop MapReduce provides the distributed MapReduce framework
            - Most MapReduce programs are written in JVM languages (Java, Scala, Groovy,etc), but it is possible to use any language through "Hadoop Streaming".
        - A number of filesystems can be used, but Hadoop is distributed with HDFS (Hadoop DFS).
            - Supports: rack-awareness, block replication, high-availability (no single point of failure), etc.
            - HDFS allows the MapReduce scheduler to be data aware, and thus minimize network traffic.
        - In Hadoop terminology a **Mapper** is responsible to applying the Map function to a subset of the data and a **Reducer** applies the Reduce function to the sorted output from all the mappers (or a subset of that data if there are multiple Reducers).
    - **Disadvantages of MapReduce**
        - Standard MapReduce programs follow an acyclic dataflow paradigm – i.e. a stateless Map followed by a stateless Reduce.
            - This doesn't mesh well with machine learning algorithms, which are often iterative.
                - You can't for example rely on the result of the previous iteration, if the iteration is defined by a single Map invocation.
            - It's designed for batch data, rather than streaming or real-time data.
            - Practical frameworks like Hadoop have some features that can help (to an extent):
                - Ability to save *extra* data from the Mapper, that doesn't automatically get sent to the reducer (although can be loaded there).
                - Shared memory across groups of Mappers.
                - Ability to write flexible chains (map-reduce-map-reduce…; map-map-map-reduce…), and even skip the reduce all-together.
            - But there is still an **overhead** if you are passing over the same data again and again.
                - Other solutions like *Apache Spark* (which is built on Apache Hadoop, but doesn't use the MapReduce component) work around this by having nodes load the data (locally) once, and then **cache it in memory** for the iterations.
- Spark is built around the concept of **distributed datasets** (i.e. data stored on HDFS across a cluster).
    - Known as **RDD**s (Resilient Distributed Datasets).
        - RDDs can be transparently cached in-memory.
            - Useful if you're re-using the data (perhaps iterating over it multiple times).
    - You write programs that perform operations on RDDs.
        - Those operations are **automatically run close to the data.**
        - Two main classes of operation:
            - **Transformations**: these take an RDD and return a new RDD.
                - e.g.: map, filter, groupByKey, reduceByKey, aggregateByKey...
            - **Actions**: these take an RDD and return a result value.
                - e.g.: reduce, collect, count, first, take, countByKey, foreach…
    - A spark program follows the **directed acyclic graph (DAG)** pattern, and can be **arbitrarily complex**, unlike MapReduce, which is highly prescribed.

Further reading

- Dean, Jeffrey & Ghemawat, Sanjay (2004). "MapReduce: Simplified Data Processing on Large Clusters". Retrieved November 23, 2011.
- Hare, Jonathon S., Samangooei, Sina & Lewis, Paul H. (2012) Practical scalable image analysis and indexing using Hadoop. *Multimedia Tools and Applications*, 1-34. (doi:10.1007/s11042-012-1256-0).

Appendix 1: Word Count with MapReduce

Synopsis: read corpus of text files and counts how often words occur.

Input: collection of text files, read line-by line.

Output: count of number of times each word occurs across corpus.

Map function:

Input: <*Key:* line number, *Value:* line of text>

Outputs: <*Key:* word, *Value:* 1>

Takes a line as input and breaks it into words by splitting on whitespace (or similar). Then, for each word, emit a <key,value> pair with the word and 1.

Reduce function:

Input: <*Key:* Word, *Value:* [list of 1's corresponding to each occurrence]>

Outputs: <*Key:* word, *Value:* count>

Each reducer sums the counts for each word and emits a single <key,value> with the word and sum.