Chapter 12

Big Data: Achieving Scale

> A change in quantity also entails a change in quality.
>
> – Friedrich Engels

I was once interviewed on a television program, and asked the difference between *data* and *big data*. After some thought, I gave them an answer which I will still hold to today: "size."

Bupkis is a marvelous Yiddish word meaning "too small to matter." Used in a sentence like "He got paid *bupkis* for it," it is a complaint about a paltry sum of money. Perhaps the closest analogy in English vernacular would be the word "peanuts."

Generally speaking, the data volumes we have dealt with thus far in this book all amount to *bupkis*. Human annotated training sets run in the hundreds to thousands of examples, but anything you must pay people to create has a hard time running into the millions. The log of all New York taxi rides for several years discussed in Section 1.6 came to 80 million records. Not bad, but still *bupkis*: you can store this easily on your laptop, and make a scan through the file to tabulate statistics in minutes.

The buzzword *big data* is perhaps reaching its expiration date, but presumes the analysis of truly massive data sets. What *big* means increases with time, but I will currently draw the starting line at around 1 terabyte.

This isn't as impressive as it may sound. After all, as of this writing a terabyte-scale disk will set you back only $100, which *is* bupkis. But acquiring a meaningful data set to fill it will take some initiative, perhaps privileged access inside a major internet company or large volumes of video. There are plenty of organizations wrestling with petabytes and even exabytes of data on a regular basis.

Big data requires a larger scale of infrastructure than the projects we have considered to date. Moving enormous volumes of data between machines requires fast networks and patience. We need to move away from sequential processing, even beyond multiple cores to large numbers of machines floating

S.S. Skiena, *The Data Science Design Manual,*
Texts in Computer Science, https://doi.org/10.1007/978-3-319-55444-0_12

in the clouds. These computations scale to the point where we must consider robustness, because of the near certainty that some hardware component will fail before we get our answer.

Working with data generally gets harder with size. In this section, I will try to sensitize you to the general issues associated with massive data sets. It is important to understand why size matters, so you will be able to contribute to projects that operate at that scale.

12.1 What is Big Data?

How big is big? Any number I give you will be out of date by the time I type it, but here are some 2016 statistics I stumbled across, primarily at `http://www.internetlivestats.com/` :

- Twitter: 600 million tweets per day.
- Facebook: 600 terabytes of incoming data each day, from 1.6 billion active users.
- Google: 3.5 billion search queries per day.
- Instagram: 52 million new photos per day.
- Apple: 130 billion total app downloads.
- Netflix: 125 million hours of TV shows and movies streamed daily.
- Email: 205 billion messages per day.

Size matters: we can do amazing things with this stuff. But other things also matter. This section will look at some of the technical and conceptual complexities of dealing with big data.

Take-Home Lesson: Big data generally consists of massive numbers of rows (records) over a relatively small number of columns (features). Thus big data is often overkill to accurately fit a single model for a given problem. The value generally comes from fitting *many* distinct models, as in training a custom model personalized for each distinct user.

12.1.1 Big Data as Bad Data

Massive data sets are typically the result of opportunity, instead of design. In traditional hypothesis-driven science, we design an experiment to gather exactly the data we need to answer our specific question. But big data is more typically the product of some logging process recording discrete events, or perhaps distributed contributions from millions of people over social media. The data

scientist generally has little or no control of the collection process, just a vague charter to turn all those bits into money.

Consider the task of measuring popular opinion from the posts on a social media platform, or online review site. Big data can be a wonderful resource. But it is particularly prone to biases and limitations that make it difficult to draw accurate conclusions from, including:

- *Unrepresentative participation*: There are sampling biases inherent in any ambient data source. The data from any particular social media site does not reflect the people who don't use it – and you must be careful not to overgeneralize.

  Amazon users buy far more books than shoppers at Walmart. Their political affiliations and economic status differs as well. You get equally-biased but very different views of the world if analyzing data from Instagram (too young), *The New York Times* (too liberal), *Fox News* (too conservative), or *The Wall Street Journal* (too wealthy).

- *Spam and machine-generated content*: Big data sources are worse than unrepresentative. Often they have been engineered to be deliberately misleading.

  Any online platform large enough to generate enormous amounts of data is large enough for there to be economic incentives to pervert it. Armies of paid reviewers work each day writing fake and misleading product reviews. Bots churn out mechanically written tweets and longer texts in volume, and even are the primary consumers of it: a sizable fraction of the hits reported on any website are from mechanical crawlers, instead of people. Fully 90% of all email sent over networks is spam: the effectiveness of spam filters at several stages of the pipeline is the only reason you don't see more of it.

  Spam filtering is an essential part of the data cleaning process on any social media analysis. If you don't remove the spam, it will be lying to you instead of just misleading you.

- *Too much redundancy*: Many human activities follow a power law distribution, meaning that a very small percentage of the items account for a large percentage of the total activity. News and social media concentrates heavily on the latest missteps of the Kardashian's and similar celebrities, covering them with articles by the thousands. Many of these will be almost exact duplicates of other articles. How much more does the full set of them tell you than any one of them would?

  This law of unequal coverage implies that much of the data we see through ambient sources is something we have seen before. Removing this duplication is an essential cleaning step for many applications. Any photo sharing site will contain thousands of images of the Empire State Building, but none of the building I work in. Training a classifier with such images will

produce fabulous features for landmarks, that may or may not be useful for more general tasks.

- *Susceptibility to temporal bias*: Products change in response to competition and changes in consumer demand. Often these improvements change the way people use these products. A time series resulting from ambient data collection might well encode several product/interface transitions, which make it hard to distinguish artifact from signal.

  A notorious example revolves around Google Flu Trends, which for several years successfully forecast flu outbreaks on the basis of search engine queries. But then the model started performing badly. One factor was that Google added an auto-complete mechanism, where it suggests relevant queries during your search process. This changed the distribution of search queries sufficiently to make time series data from before the change incomparable with the data that follows.

  Some of these effects can be mitigated through careful normalization, but often they are baked so tightly into the data to prevent meaningful longitudinal analysis.

Take-Home Lesson: Big data is data we have. Good data is data appropriate to the challenge at hand. Big data is bad data if it cannot really answer the questions we care about.

12.1.2 The Three Vs

Management consulting types have latched onto a notion of the *three Vs of big data* as a means of explaining it: the properties of *volume*, *variety*, and *velocity*. They provide a foundation to talk about what makes big data different. The Vs are:

- *Volume*: It goes without saying that big data is bigger than little data. The distinction is one of class. We leave the world where we can represent our data in a spreadsheet or process it on a single machine. This requires developing a more sophisticated computational infrastructure, and restricting our analysis to linear-time algorithms for efficiency.

- *Variety*: Ambient data collection typically moves beyond the matrix to amass heterogeneous data, which often requires ad hoc integration techniques.

  Consider social media. Posts may well include text, links, photos, and video. Depending upon our task, all of these may be relevant, but text processing requires vastly different techniques than network data and multimedia. Even images and videos are quite different beasts, not to be processed using the same pipeline. Meaningfully integrating these materials into a single data set for analysis requires substantial thought and effort.

- *Velocity*: Collecting data from ambient sources implies that the system is *live*, meaning it is always on, always collecting data. In contrast, the data sets we have studied to date have generally been *dead*, meaning collected once and stuffed into a file for later analysis.

  Live data means that infrastructures must be built for collecting, indexing, accessing, and visualizing the results, typically through a dashboard system. Live data means that consumers want real-time access to the latest results, through graphs, charts, and APIs.

  Depending upon the industry, real-time access may involve updating the state of the database within seconds or even milliseconds of actual events. In particular, the financial systems associated with high-frequency trading demand immediate access to the latest information. You are in a race against the other guy, and you profit only if you win.

  Data velocity is perhaps the place where data science differs most substantially from classical statistics. It is what stokes the demand for advanced system architectures, which require engineers who build for scale using the latest technologies.

The management set sometimes defines a fourth V: *veracity*, a measure for how much we trust the underlying data. Here we are faced with the problem of eliminating spam and other artifacts resulting from the collection process, beyond the level of normal cleaning.

12.2 War Story: Infrastructure Matters

I should have recognized the depth of Mikhail's distress the instant he lifted his eyebrow at me.

My Ph.D. student Mikhail Bautin is perhaps the best programmer I have ever seen. Or you have ever seen, probably. Indeed, he finished in *1st place* at the 12th International Olympiad in Informatics with a perfect score, marking him as the best high school-level programmer in the world that year.

At this point, our Lydia news analysis project had a substantial infrastructure, running on a bunch of machines. Text spidered from news sources around the world was normalized and passed through a natural language processing (NLP) pipeline we had written for English, and the extracted entities and their sentiment were identified from the text and stored in a big database. With a series of SQL commands this data could be extracted to a format where you could display it on a webpage, or run it in a spreadsheet.

I wanted us to study the degree to which machine translation preserved detectable sentiment. If so, it provided an easy way to generalize our sentiment analysis to languages beyond English. It would be low-hanging fruit to stick some third-party language translator into our pipeline and see what happened.

I thought this was a timely and important study, and indeed, our subsequent paper [BVS08] has been cited 155 times as of this writing. So I gave the project

to my best Ph.D. student, and even offered him the services of a very able masters student to help with some of the technical issues. Quietly and obediently, he accepted the task. But he did raise his eyebrow at me.

Three weeks later he stepped into my office. The infrastructure my lab had developed for maintaining the news analysis in our database was old-fashioned and crufty. It would not scale. It offended his sense of being. Unless I let him rewrite the entire thing from scratch using modern technology, he was leaving graduate school immediately. He had used his spare time during these three weeks to secure a very lucrative job offer from a world-class hedge fund, and was stopping by to say farewell.

I can be a very reasonable man, once things are stated plainly enough. Yes, his dissertation could be on such an infrastructure. He turned around and immediately got down to work.

The first thing that had to go was the central MYSQL database where all our news and sentiment references were stored. It was a bottleneck. It could not be distributed across a cluster of machines. He was going to store everything in a distributed file system (HDFS) so that there was no single bottleneck: reads and writes could happen all over our cluster.

The second thing that had to go was our jury-rigged approach to coordinating the machines in our cluster on their various tasks. It was unreliable. There was no error-recovery mechanism. He was going to rewrite all our backend processing as MapReduce jobs using Hadoop.

The third thing that had to go was the ad hoc file format we used to represent news articles and their annotations. It was buggy. There were exceptions everywhere. Our parsers often broke on them for stupid reasons. This is why G-d had invented XML, to provide a way to rigorously express structured data, and efficient off-the-shelf tools to parse it. Any text that passed through his code was going to pass an XML validator first. He refused to touch the disease-ridden Perl scripts that did our NLP analysis, but isolated this code completely enough that the infection could be contained.

With so many moving parts, even Mikhail took some time to get his infrastructure right. Replacing our infrastructure meant that we couldn't advance on any other project until it was complete. Whenever I fretted that we couldn't get any experimental analysis done until he was ready, he quietly reminded me about the standing offer he had from the hedge fund, and continued right on with what he was doing.

And of course, Mikhail was right. The scale of what we could do in the lab increased ten-fold with the new infrastructure. There was much less downtime, and scrambling to restore the database after a power-glitch became a thing of the past. The APIs he developed to regulate access to the data powered all our application analysis in a convenient and logical way. His infrastructure cleanly survived a porting to the Amazon Cloud environment, running every night to keep up with the world's news.

The take-home lesson here is that infrastructure matters. Most of this book talks about higher-level concepts: statistics, machine learning, visualization – and it is easy to get hoity-toity about what is science and what is plumbing.

But civilization does not run right without effective plumbing. Infrastructures that are clean, efficient, scalable, and maintainable, built using modern software technologies, are essential to effective data science. Operations that reduce technical debt like refactoring, and upgrading libraries/tools to currently supported versions are not no-ops or procrastinating, but the key to making it easier to do the stuff you really want to do.

12.3 Algorithmics for Big Data

Big data requires efficient algorithms to work on it. In this section, we will delve briefly into the basic algorithmic issues associated with big data: asymptotic complexity, hashing, and streaming models to optimize I/O performance in large data files.

I do not have the time or space here to provide a comprehensive introduction to the design and analysis of combinatorial algorithms. However, I can confidently recommend *The Algorithm Design Manual* [Ski08] as an excellent book on these matters, if you happen to be looking for one.

12.3.1 Big Oh Analysis

Traditional algorithm analysis is based on an abstract computer called the *Random Access Machine* or *RAM*. On such a model:

- Each simple operation takes exactly one step.
- Each memory operation takes exactly one step.

Hence counting up the operations performed over the course of the algorithm gives its running time.

Generally speaking, the number of operations performed by any algorithm is a function of the size of the input n: a matrix with n rows, a text with n words, a point set with n points. Algorithm analysis is the process of estimating or bounding the number of steps the algorithm takes as a function of n.

For algorithms defined by **for**-loops, such analysis is fairly straightforward. The depth of the nesting of these loops defines the complexity of the algorithm. A single loop from 1 to n defines a *linear-time* or $O(n)$ algorithm, while two nested loops defines a *quadratic-time* or $O(n^2)$ algorithm. Two sequential for-loops that do not nest are still linear, because $n + n = 2n$ steps are used instead of $n \times n = n^2$ such operations.

Examples of basic loop-structure algorithms include:

- *Find the nearest neighbor of point p*: We need to compare p against all n points in a given array a. The distance computation between p and point $a[i]$ requires subtracting and squaring d terms, where d is the dimensionality of p. Looping through all n points and keeping track of the closest point takes $O(d \cdot n)$ time. Since d is typically small enough to be thought of as a constant, this is considered a linear-time algorithm.

- *The closest pair of points in a set*: We need to compare every point $a[i]$ against every other point $a[j]$, where $1 \leq i \neq j \leq n$. By the reasoning above, this takes $O(d \cdot n^2)$ time, and would be considered a quadratic-time algorithm.

- *Matrix multiplication*: Multiplying an $x \times y$ matrix times a $y \times z$ matrix results in an $x \times z$ matrix, where each of the $x \cdot z$ terms is the dot product of two y-length vectors:

```
C = numpy.zeros((x, z))

for i in range(0,x-1):
    for j in range(0, z-1):
        for k in range(0, y-1):
            C[i][j] += A[i][k] * B[k][j]
```

  This algorithm takes $x \cdot y \cdot z$ steps. If $n = \max(x, y, z)$, then this takes at most $O(n^3)$ steps, and would be considered a cubic-time algorithm.

For algorithms which are defined by conditional **while** loops or recursion, the analysis often requires more sophistication. Examples, with very concise explanations, include:

- *Adding two numbers*: Very simple operations might have no conditionals, like adding two numbers together. There is no real value of n here, only two, so this takes constant time or $O(1)$.

- *Binary search*: We seek to locate a given search key k in a sorted array A, containing n items. Think about searching for a name in the telephone book. We compare k against the middle element $A[n/2]$, and decide whether what we are looking for lies in the top half or the bottom half. The number of halvings until we get down to 1 is $\log_2(n)$, as we discussed in Section 2.4. Thus binary search runs in $O(\log n)$ time.

- *Mergesort*: Two sorted lists with a total of n items can be merged into a single sorted list in linear time: take out the smaller of the two head elements as first in sorted order, and repeat. Mergesort splits the n elements into two halves, sorts each, and then merges them. The number of halvings until we get down to 1 is again $\log_2(n)$ (do see Section 2.4), and merging all elements at all levels yields an $O(n \log n)$ sorting algorithm.

This was a very fast algorithmic review, perhaps too quick for comprehension, but it did manage to provide representatives of six different algorithm complexity classes. These complexity functions define a spectrum from fastest to slowest, defined by the following ordering:

$$O(1) \ll O(\log n) \ll O(n) \ll O(n \log n) \ll O(n^2) \ll O(n^3).$$

Take-Home Lesson: Algorithms running on big data sets must be linear or near-linear, perhaps $O(n \log n)$. Quadratic algorithms become impossible to contemplate for $n > 10,000$.

12.3.2 Hashing

Hashing is a technique which can often turn quadratic algorithms into linear-time algorithms, making them tractable for dealing with the scale of data we hope to work with.

We first discussed hash functions in the context of locality-sensitive hashing (LSH) in Section 10.2.4. A *hash function* h takes an object x and maps it to a specific integer $h(x)$. The key idea is that whenever $x = y$, then $h(x) = h(y)$. Thus we can use $h(x)$ as an integer to index an array, and collect all similar objects in the same place. Different items are *usually* mapped to different places, assuming a well-designed hash function, but there are no guarantees.

Objects that we seek to hash are often sequences of simpler elements. For example, files or text strings are just sequences of elementary characters. These elementary components usually have a natural mapping to numbers: character codes like Unicode by definition map symbols to numbers, for example. The first step to hash x is to represent it as a sequence of such numbers, with no loss of information. Let us assume each of the $n = |S|$ character numbers of x are integers between 0 and $\alpha - 1$.

Turning the vector of numbers into a single representative number is the job of the hash function $h(x)$. A good way to do this is to think of the vector as a base-α number, so

$$h(x) = \sum_{i=0}^{n-1} \alpha^{n-(i+1)} x_i \pmod{m}.$$

The mod function ($x \bmod m$) returns the remainder of x divided by m, and so yields a number between 0 and $m - 1$. This n-digit, base-α number is doomed to be huge, so taking the remainder gives us a way to get a representative code of modest size. The principle here is the same as a roulette wheel for gambling: the ball's long path around the wheel ultimately ends in one of $m = 38$ slots, as determined by the remainder of the path length divided by the circumference of the wheel.

Such hash functions are amazingly useful things. Major applications include:

- *Dictionary maintenance*: A hash table is an array-based data structure using $h(x)$ to define the position of object x, coupled with an appropriate collision-resolution method. Properly implemented, such hash tables yield constant time (or $O(1)$) search times in practice.

  This is much better than binary search, and hence hash tables are widely used in practice. Indeed, Python uses hashing below the hood to link variable names to the values they store. Hashing is also the fundamental

idea behind distributed computing systems like MapReduce, which will be discussed in Section 12.6.

- *Frequency counting*: A common task in analyzing logs is tabulating the frequencies of given events, such as word counts or page hits. The fastest/easiest approach is to set up a hash table with event types as the key, and increment the associated counter for each new event. Properly implemented, this algorithm is linear in the total number of events being analyzed.

- *Duplicate removal*: An important data cleaning chore is identifying duplicate records in a data stream and removing them. Perhaps these are all the email addresses we have of our customers, and want to make sure we only spam each of them once. Alternately, we may seek to construct the complete vocabulary of a given language from large volumes of text.

  The basic algorithm is simple. For each item in the stream, check whether it is already in the hash table. If not insert it, if so ignore it. Properly implemented, this algorithm takes time linear in the total number of records being analyzed.

- *Canonization*: Often the same object can be referred to by multiple different names. Vocabulary words are generally case-insensitive, meaning that "The" is equivalent to "the." Determining the vocabulary of a language requires unifying alternate forms, mapping them to a single key.

  This process of constructing a *canonical representation* can be interpreted as hashing. Generally speaking, this requires a domain-specific simplification function doing such things as reduction to lower case, white space removal, stop word elimination, and abbreviation expansion. These canonical keys can then be hashed, using conventional hash functions.

- *Cryptographic hashing*: By constructing concise and *uninvertible* representations, hashing can be used to monitor and constrain human behavior. How can you prove that an input file remains unchanged since you last analyzed it? Construct a hash code or *checksum* for the file when you worked on it, and save this code for comparison with the file hash at any point in the future. They will be the same if the file is unchanged, and almost surely differ if any alterations have occurred.

  Suppose you want to commit to a bid on a specific item, but not reveal the actual price you will pay until all bids are in. Hash your bid using a given cryptographic hash function, and submit the resulting hash code. After the deadline, send your bid in again, this time without encryption. Any suspicious mind can hash your now open bid, and confirm the value matches your previously submitted hash code. The key is that it be difficult to produce collisions with the given hash function, meaning you cannot readily construct another message which will hash to the same code. Otherwise you could submit the second message instead of the first, changing your bid after the deadline.

12.3.3 Exploiting the Storage Hierarchy

Big data algorithms are often *storage-bound* or *bandwidth-bound* rather than *compute-bound*. This means that the cost of waiting around for data to arrive where it is needed exceeds that of algorithmically manipulating it to get the desired results. It still takes half an hour to just to read 1 terabyte of data from a modern disk. Achieving good performance can rest more on smart data management than sophisticated algorithmics.

To be available for analysis, data must be stored somewhere in a computing system. There are several possible types of devices to put it on, which differ greatly in speed, capacity, and latency. The performance differences between different levels of the *storage hierarchy* is so enormous that we cannot ignore it in our abstraction of the RAM machine. Indeed, the ratio of the access speed from disk to cache memory is roughly the same (10^6) as the speed of a tortoise to the exit velocity of the earth!

The major levels of the storage hierarchy are:

- *Cache memory*: Modern computer architectures feature a complex system of registers and caches to store working copies of the data actively being used. Some of this is used for prefetching: grabbing larger blocks of data around memory locations which have been recently accessed, in anticipation of them being needed later. Cache sizes are typically measured in megabytes, with access times between five and one hundred times faster than main memory. This performance makes it very advantageous for computations to exploit *locality*, to use particular data items intensively in concentrated bursts, rather than intermittently over a long computation.

- *Main memory*: This is what holds the general state of the computation, and where large data structures are hosted and maintained. Main memory is generally measured in gigabytes, and runs hundreds to thousands of times faster than disk storage. To the greatest extent possible, we need data structures that fit into main memory and avoid the paging behavior of virtual memory.

- *Main memory on another machine*: Latency times on a local area network run into the low-order milliseconds, making it generally faster than secondary storage devices like disks. This means that distributed data structures like hash tables *can* be meaningfully maintained across networks of machines, but with access times that can be hundreds of times slower than main memory.

- *Disk storage*: Secondary storage devices can be measured in terabytes, providing the capacity that enables big data to get big. Physical devices like spinning disks take considerable time to move the read head to the position where the data is. Once there, it is relatively quick to read a large block of data. This motivates pre-fetching, copying large chunks of files into memory under the assumption that they will be needed later.

Latency issues generally act like a volume discount: we pay a lot for the first item we access, but then get a bunch more very cheaply. We need to organize our computations to take advantage of this, using techniques like:

- *Process files and data structures in streams*: It is important to access files and data structures sequentially whenever possible, to exploit prefetching. This means arrays are better than linked structures, because logically-neighboring elements sit near each other on the storage device. It means making entire passes over data files that read each item once, and then perform all necessary computations with it before moving on. Much of the advantage of sorting data is that we can jump to the appropriate location in question. Realize that such random access is expensive: think sweeping instead of searching.

- *Think big files instead of directories*: One can organize a corpus of documents such that each is in its own file. This is logical for humans but slow for machines, when there are millions of tiny files. Much better is to organize them in one large file to efficiently sweep through all examples, instead of requiring a separate disk access for each one.

- *Packing data concisely*: The cost of decompressing data being held in main memory is generally much smaller than the extra transfer costs for larger files. This is an argument that it pays to represent large data files concisely whenever you can. This might mean explicit file compression schemes, with small enough file sizes so that they can be expanded in memory.

  It does mean designing file formats and data structures to be concisely encoded. Consider representing DNA sequences, which are long strings on a four-letter alphabet. Each letter/base can be represented in 2 bits, meaning that four bases can be represented in a single 8-bit byte and thirty-two bases in a 64-bit word. Such data-size reductions can greatly reduce transfer times, and are worth the computational effort to pack and unpack.

  We have previously touted the importance of readability in file formats in Section 3.1.2, and hold to that opinion here. Minor size reductions are likely not worth the loss of readability or ease of parsing. But cutting a file size in half is equivalent to doubling your transfer rate, which may matter in a big data environment.

12.3.4 Streaming and Single-Pass Algorithms

Data is not necessarily stored forever. Or even at all. In applications with a very high volume of updates and activity, it may pay to compute statistics on the fly as the data emerges so we can then throw the original away.

In a *streaming* or *single-pass* algorithm, we get only one chance to view each element of the input. We can assume some memory, but not enough to store the

bulk of the individual records. We need to decide what to do with each element when we see it, and then it is gone.

For example, suppose we seek to compute the mean of a stream of numbers as it passes by. This is not a hard problem: we can keep two variables: s representing the running sum to date, and n the number of items we have seen so far. For each new observation a_i, we add it to s and increment n. Whenever someone needs to know the current mean $\mu = \bar{A}$ of the stream A, we report the value of s/n.

What about computing the variance or standard deviation of the stream? This seems harder. Recall that

$$V(A) = \sigma^2 = \frac{\sum_{i=1}^{n}(a_i - \bar{A})^2}{n-1}$$

The problem is that the sequence mean $\bar{A}$ cannot be known until we hit the end of the stream, at which point we have lost the original elements to subtract against the mean.

But all is not lost. Recall that there is an alternate formula for the variance, the mean of the square minus the square of the mean:

$$V(a) = (\frac{1}{n}\sum_{i=1}^{n}(a_i)^2) - (\bar{A})^2$$

Thus by keeping track of a running sum of squares of the elements in addition to n and s, we have all the material we need to compute the variance on demand.

Many quantities cannot be computed exactly under the streaming model. An example would be finding the *median* element of a long sequence. Suppose we don't have enough memory to store half the elements of the full stream. The first element that we chose to delete, whatever it is, could be made to be the median by a carefully-designed stream of elements yet unseen. We need to have all the data simultaneously available to us to solve certain problems.

But even if we cannot compute something exactly, we can often come up with an estimate that is good enough for government work. Important problems of this type include identifying the most frequent items in a stream, the number of distinct elements, or even estimating element frequency when we do not have enough memory to keep an exact counter.

Sketching involves using what storage we do have to keep track of a partial representation of the sequence. Perhaps this is a frequency histogram of items binned by value, or a small hash table of values we have seen to date. The quality of our estimate increases with the amount of memory we have to store our sketch. *Random sampling* is an immensely useful tool for constructing sketches, and is the focus of Section 12.4.

12.4 Filtering and Sampling

One important benefit of big data is that with sufficient volume you can afford to throw most of your data away. And this can be quite worthwhile, to make

your analysis cleaner and easier.

I distinguish between two distinct ways to throw data away, filtering and sampling. *Filtering* means selecting a relevant subset of data based on a specific criteria. For example, suppose we wanted to build a language model for an application in the United States, and we wanted to train it on data from Twitter. English accounts for only about one third of all tweets on Twitter, so filtering out all other languages leaves enough for meaningful analysis.

We can think of filtering as a special form of cleaning, where we remove data not because it is erroneous but because it is distracting to the matter at hand. Filtering away irrelevant or hard-to-interpret data requires application-specific knowledge. English is indeed the primary language in use in the United States, making the decision to filter the data in this way perfectly reasonable.

But filtering introduces biases. Over 10% of the U.S. population speaks Spanish. Shouldn't they be represented in the language model, *amigo*? It is important to select the right filtering criteria to achieve the outcome we seek. Perhaps we might better filter tweets based on location of origin, instead of language.

In contrast, *sampling* means selecting an appropriate size subset in an arbitrary manner, without domain-specific criteria. There are several reasons why we may want to subsample good, relevant data:

- *Right-sizing training data*: Simple, robust models generally have few parameters, making big data unnecessary to fit them. Subsampling your data in an unbiased way leads to efficient model fitting, but is still representative of the entire data set.
- *Data partitioning*: Model-building hygiene requires cleanly separating training, testing, and evaluation data, typically in a 60%, 20%, and 20% mix. Constructing these partitions in an unbiased manner is necessary for the veracity of this process.
- *Exploratory data analysis and visualization*: Spreadsheet-sized data sets are fast and easy to explore. An unbiased sample is representative of the whole while remaining comprehensible.

Sampling n records in an efficient and unbiased manner is a more subtle task than it may appear at first. There are two general approaches, deterministic and randomized, which detailed in the following sections.

12.4.1 Deterministic Sampling Algorithms

Our straw man sampling algorithm will be *sampling by truncation*, which simply takes the first n records in the file as the desired sample. This is simple, and has the property that it is readily *reproducible*, meaning someone else with the full data file could easily reconstruct the sample.

However, the order of records in a file often encodes semantic information, meaning that truncated samples often contain subtle effects from factors such as:

- *Temporal biases*: Log files are typically constructed by appending new records to the end of the file. Thus the first n records would be the oldest available, and will not reflect recent regime changes.

- *Lexicographic biases*: Many files are sorted according to the primary key, which means that the first n records are biased to a particular population. Imagine a personnel roster sorted by name. The first n records might consist only of the As, which means that we will probably over-sample Arabic names from the general population, and under-sample Chinese ones.

- *Numerical biases*: Often files are sorted by identity numbers, which may appear to be arbitrarily defined. But ID numbers can encode meaning. Consider sorting the personnel records by their U.S. social security numbers. In fact, the first five digits of social security numbers are generally a function of the year and place of birth. Thus truncation leads to a geographically and age-biased sample.

  Often data files are constructed by concatenating smaller files together, some of which may be far more enriched in positive examples than others. In particularly pathological cases, the record number might completely encode the class variable, meaning that an accurate but totally useless classifier may follow from using the class ID as a feature.

So truncation is generally a bad idea. A somewhat better approach is *uniform sampling*. Suppose we seek to sample n/m records out of n from a given file. A straightforward approach is to start from the ith record, where i is some value between 1 and m, and then sample every mth record starting from i. Another way of saying this is that we output the jth record if $j (\mathrm{mod}\ m) = i$. Such uniform sampling provides a way to balance many concerns:

- We obtain exactly the desired number of records for our sample.

- It is quick and reproducible by anyone given the file and the values of i and m.

- It is easy to construct multiple disjoint samples. If we repeat the process with a different offset i, we get an independent sample.

Twitter uses this method to govern API services that provide access to tweets. The free level of access (the spritzer hose) rations out 1% of the stream by giving every 100th tweet. Professional levels of access dispense every tenth tweet or even more, depending upon what you are willing to pay for.

This is generally better than truncation, but there still exist potential periodic temporal biases. If you sample every mth record in the log, perhaps every item you see will be associated with an event from a Tuesday, or at 11PM each night. On files sorted by numbers, you are in danger of ending up with items with the same lower order digits. Telephone numbers ending in "000" or repeat

digits like "8888" are often reserved for business use instead of residential, thus biasing the sample. You can minimize the chances of such phenomenon by forcing m to be a large-enough prime number, but the only certain way to avoid sampling biases is to use randomization.

12.4.2 Randomized and Stream Sampling

Randomly sampling records with a probability p results in a selection of an expected $p \cdot n$ items, without any explicit biases. Typical random number generators return a value between 0 and 1, drawn from a uniform distribution. We can use the sampling probability p as a threshold. As we scan each new record, generate a new random number r. When $r \leq p$, we accept this record into our sample, but when $r > p$ we ignore it.

Random sampling is a generally sound methodology, but it comes with certain technical quirks. Statistical discrepancies ensure that certain regions or demographics will be over-sampled relative to population, however in an unbiased manner and to a predictable extent. Multiple random samples will not be disjoint, and random sampling is not reproducible without the seed and random generator.

Because the ultimate number of sampled records depends upon randomness, we may end up with slightly too many or too few items. If we need *exactly* k items, we can construct a random permutation of the items and truncate it after the first k. Algorithms for constructing random permutations were discussed in Section 5.5.1. These are simple, but require large amounts of irregular data movement, making them potentially bad news for large files. A simpler approach is to append a new random number field to each record, and sort with this as the key. Taking the first k records from this sorted file is equivalent to randomly sampling exactly k records.

Obtaining a fixed size random sample from a stream is a trickier problem, because we cannot store all the items until the end. Indeed, we don't even know how big n will ultimately be.

To solve this problem, we will maintain a uniformly selected sample in an array of size k, updated as each new element arrives from the stream. The probability that the nth stream element belongs in the sample is k/n, and so we will insert it into our array if random number $r \leq k/n$. Doing so must kick a current resident out of the table, and selecting which current array element is the victim can be done with another call to the random number generator.

12.5 Parallelism

Two heads are better than one, and a hundred heads better than two. Computing technology has matured in ways that make it increasingly feasible to commandeer multiple processing elements on demand for your application. Microprocessors routinely have 4 cores and beyond, making it worth thinking about parallelism even on individual machines. The advent of data centers and cloud

computing has made it easy to rent large numbers of machines on demand, enabling even small-time operators to take advantage of big distributed infrastructures.

There are two distinct approaches to simultaneously computing with multiple machines, namely parallel and distributed computing. The distinction here is how tightly coupled the machines are, and whether the tasks are CPU-bound or memory/IO-bound. Roughly:

- *Parallel processing* happens on one machine, involving multiple cores and/or processors that communicate through threads and operating system resources. Such tightly-coupled computation is often CPU-bound, limited more by the number of cycles than the movement of data through the machine. The emphasis is solving a particular computing problem faster than one could sequentially.

- *Distributed processing* happens on many machines, using network communication. The potential scale here is enormous, but most appropriate to loosely-coupled jobs which do not communicate much. Often the goal of distributed processing involves sharing resources like memory and secondary storage across multiple machines, more so than exploiting multiple CPUs. Whenever the speed of reading data from a disk is the bottleneck, we are better off having many machines reading as many different disks as possible, simultaneously.

In this section, we introduce the basic principles of parallel computing, and two relatively simple ways to exploit it: data parallelism and grid search. MapReduce is the primary paradigm for distributed computing on big data, and will be the topic of Section 12.6.

12.5.1 One, Two, Many

Primitive cultures were not very numerically savvy, and supposedly only counted using the words *one*, *two*, and *many*. This is actually a very good way to think about parallel and distributed computing, because the complexity increases very rapidly with the number of machines:

- *One*: Try to keep all the cores of your box busy, but you are working on one computer. This isn't distributed computing.

- *Two*: Perhaps you will try to manually divide the work between a few machines on your local network. This is barely distributed computing, and is generally managed through ad hoc techniques.

- *Many*: To take advantage of dozens or even hundreds of machines, perhaps in the cloud, we have no choice but to employ a system like MapReduce that can efficiently manage these resources.

Complexity increases hand-in-hand with the number of agents being coordinated towards a task. Consider what changes as social gatherings scale in size. There is a continual trend of making do with looser coordination as size increases, and a greater chance of unexpected and catastrophic events occurring, until they become so likely that the unexpected must be expected:

- 1 *person*: A date is easy to arrange using personal communication.
- > 2 *persons*: A dinner among friends requires active coordination.
- > 10 *persons*: A group meeting requires that there be a leader in charge.
- > 100 *persons*: A wedding dinner requires a fixed menu, because the kitchen cannot manage the diversity of possible orders.
- > 1000 *persons*: At any community festival or parade, no one knows the majority of attendees.
- $> 10,000$ *persons*: After any major political demonstration, somebody is going to spend the night in the hospital, even if the march is peaceful.
- $> 100,000$ *persons*: At any large sporting event, one of the spectators will presumably die that day, either through a heart attack [BSC+11] or an accident on the drive home.

If some of these sound unrealistic to you, recall that the length of a typical human life is $80\ years \times 365\ days/year = 29,200\ days$. But perhaps this sheds light on some of the challenges of parallelization and distributed computing:

- *Coordination*: How do we assign work units to processors, particularly when we have more work units than workers? How do we aggregate or combine each worker's efforts into a single result?
- *Communication*: To what extent can workers share partial results? How can we know when all the workers have finished their tasks?
- *Fault tolerance*: How do we reassign tasks if workers quit or die? Must we protect against malicious and systematic attacks, or just random failures?

Take-Home Lesson: Parallel computing works when we can minimize communication and coordination complexity, and complete the task with low probability of failure.

12.5.2 Data Parallelism

Data parallelism involves partitioning and replicating the data among multiple processors and disks, running the same algorithm on each piece, and then collecting the results together to produce the final results. We assume a *master* machine divvying out tasks to a bunch of slaves, and collecting the results.

A representative task is aggregating statistics from a large collection of files, say, counting how often words appear in a massive text corpus. The counts for each file can be computed independently as partial results towards the whole, and the task of merging these resulting count files easily computed by a single machine at the end. The primary advantage of this is simplicity, because all counting processes are running the same program. The inter-processor communication is straightforward: moving the files to the appropriate machine, starting the job, and then reporting the results back to the master machine.

The most straightforward approach to multicore computing involves data parallelism. Data naturally forms partitions established by time, clustering algorithms, or natural categories. For most aggregation problems, records can be partitioned arbitrarily, provided all subproblems will be merged together at the end, as shown in Figure 12.1.

For more complicated problems, it takes additional work to combine the results of these runs together later. Recall the k-means clustering algorithm (Section 10.5.1), which has two steps:

1. For each point, identifies which current cluster center is closest to it.
2. Computes the new centroid of the points now associated with it.

Assuming the points have been spread across multiple machines, the first step requires the master to communicate all current centers to each machine, while the second step requires each slave to report back to the master the new centroids of the points in its partition. The master then appropriately computes the averages of these centroids to end the iteration.

12.5.3 Grid Search

A second approach to exploit parallelism involves multiple independent runs on the same data. We have seen that many machine learning methods involve parameters which impact the quality of the ultimate result, such as selecting the right number of clusters k for k-means clustering. Picking the best one means trying them all, and each of these runs can be conducted simultaneously on different machines.

Grid search is the quest for the right meta-parameters in training. It is difficult to predict exactly how varying the learning rate or batch size in stochastic-gradient descent affects the quality of the final model. Multiple independent fits can be run in parallel, and in the end we take the best one according to our evaluation.

Effectively searching the space over k different parameters is difficult because of interactions: identifying the best value single of each parameter separately

does not necessarily produce the best parameter set when combined. Typically, reasonable minimum and maximum values for each parameter p_i are established by the user, as well as the number of values t_i for this parameter to be tested at. Each interval is partitioned into equally-spaced values governed by this t_i. We then try all parameter sets which can be formed by picking one value per interval, establishing the grid in grid search.

How much should we believe that the best model in a grid search is *really* better than the others? Often there is simple variance that explains the small differences in performance on a given test set, turning grid search into cherry-picking for the number which makes our performance sound best. If you have the computational resources available to conduct a grid search for your model, feel free to go ahead, but recognize the limits of what trial-and-error can do.

12.5.4 Cloud Computing Services

Platforms such as Amazon AWS, Google Cloud, and Microsoft Azure make it easy to rent large (or small) numbers of machines for short-term (or long-term) jobs. They provide you with the ability to get access to exactly the right computing resources when you need them, provided that you can pay for them, of course.

The cost models for these service providers are somewhat complicated, however. They will typically be hourly charges for each virtual machine, as a function of the processor type, number of cores, and main memory involved. Reasonable machines will rent for between 10 and 50 cents/hour. You will pay for the amount of long-term storage as a function of gigabyte/months, with different cost tiers depending upon access patterns. Further, you pay bandwidth charges covering the volume of data transfer between machines and over the web.

Spot pricing and reserved instances can lead to lower hourly costs for special usage patterns, but with extra caveats. Under *spot pricing*, machines go to the highest bidder, so your job is at risk of being interrupted if someone else needs it more than you do. With *reserved instances*, you pay a certain amount up front in order to get a lower hourly price. This makes sense if you will be needing one computer 24/7 for a year, but not if you need a hundred computers each for one particular day.

Fortunately, it can be free to experiment. All the major cloud providers provide some free time to new users, so you can play with the setup and decide on their dime whether it is appropriate for you.

12.6 MapReduce

Google's MapReduce paradigm for distributed computing has spread widely through open-source implementations like Hadoop and Spark. It offers a simple programming model with several benefits, including straightforward scaling to

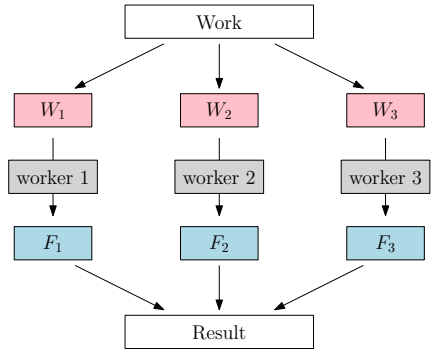

Figure 12.1: Divide and conquer is the algorithmic paradigm of distributed computing.

hundreds or even thousands of machines, and fault tolerance through redundancy.

The level of abstraction of programming models steadily increases over time, as reflected by more powerful tools and systems that hide implementation details from the user. If you are doing data science on a multiple-computer scale, MapReduce computing is probably going on under the hood, even if you are not explicitly programming it.

An important class of large-scale data science tasks have the following basic structure:

- Iterate over a large number of items, be they data records, text strings, or directories of files.
- Extract something of interest from each item, be it the value of a particular field, frequency counts of each word, or the presence/absence of particular patterns in each file.
- Aggregate these intermediate results over all items, and generate an appropriate combined result.

Representatives of these class of problems include word frequency counting, k-means clustering, and PageRank computations. All are solvable through straightforward iterative algorithms, whose running times scale linearly in the size of the input. But this can be inadequate for inputs of massive size, where the files don't naturally fit in the memory of a single machine. Think about web-scale problems, like word-frequency counting over billions of tweets, k-means clustering on hundred of millions of Facebook profiles, and PageRank over all websites on the Internet.

The typical solution here is *divide and conquer*. Partition the input files among m different machines, perform the computations in parallel on each of them, and then combine the results on the appropriate machine. Such a solution

works, in principle, for word counting, because even enormous text corpora will ultimately reduce to relatively small files of distinct vocabulary words with associated frequency counts, which can then be readily added together to produce the total counts.

But consider a PageRank computation, where for each node v we need to sum up the PageRank from all nodes x where x points to v. There is no way we can cut the graph into separate pieces such that all these x vertices will sit on the same machine as v. Getting things in the right place to work with them is at the heart of what MapReduce is all about.

12.6.1 Map-Reduce Programming

The key to distributing such computations is setting up a distributed hash table of buckets, where all the items with the same key get mapped to the same bucket:

- *Word count*: For counting the total frequency of a particular word w across a set of files, we need to collect the frequency counts for all the files in a single bucket associated with w. There they can be added together to produce the final total.

- *k-means clustering*: The critical step in k-means clustering is updating the new centroid c' of the points closest to the current centroid c. After hashing all the points p closest to c to a single bucket associated with c, we can compute c' in a single sweep through this bucket.

- *PageRank*: The new PageRank of vertex v is the sum of old PageRank for all neighboring vertices x, where (x, v) is a directed edge in the graph. Hashing the PageRank of x to the bucket for all adjacent vertices v collects all relevant information in the right place, so we can update the PageRank in one sweep through them.

These algorithms can be specified through two programmer-written functions, *map* and *reduce*:

- *Map*: Make a sweep through each input file, hashing or *emitting* key-value pairs as appropriate. Consider the following pseudocode for the word count mapper:

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);
```

- *Reduce*: Make a sweep through the set of values v associated with a specific key k, aggregating and processing accordingly. Pseudocode for the word count reducer is:

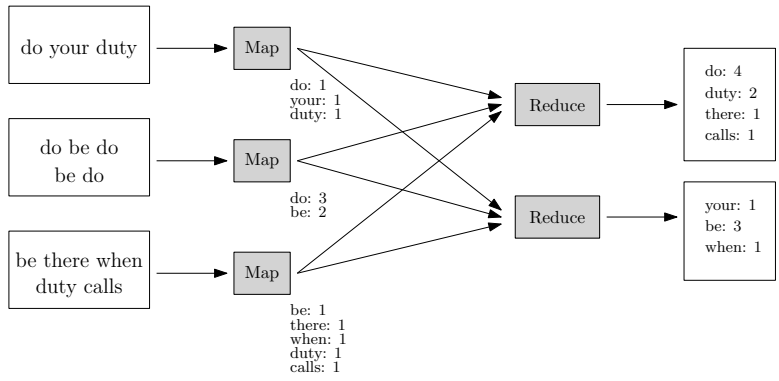

Figure 12.2: Word count in action. Count combination has been performed before mapping to reduce the size of the map files.

```
Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
```

The efficiency of a MapReduce program depends upon many things, but one important objective is keeping the number of emits to a minimum. Emitting a count for each word triggers a message across machines, and this communication and associated writes to the bucket prove costly in large quantities. The more stuff that is mapped, the more that must eventually be reduced.

The ideal is to *combine* counts from particular input streams locally before, and then emit only the total for each distinct word per file. This could be done by adding extra logic/data structures to the map function. An alternate idea is to run mini-reducers in memory after the map phase, but before inter-processor communication, as an optimization to reduce network traffic. We note that optimization for in-memory computation is one of the major performance advantages of Spark over Hadoop for MapReduce-style programming.

Figure 12.2 illustrates the flow of a MapReduce job for word counting, using three mappers and two reducers. Combination has been done locally, so the counts for each word used more than once in an input file (here *doc* and *be*) have been tabulated prior to emitting them to the reducers.

One problem illustrated by Figure 12.2 is that of mapping *skew*, the natural imbalance in the amount of work assigned to each reduce task. In this toy example, the top reducer has been assigned map files with 33% more words and 60% larger counts than its partner. For a task with a serial running time of T, perfect parallelization with n processors would yield a running time of T/n. But the running time of a MapReduce job is determined by the largest, slowest piece. Mapper skew dooms us to a largest piece that is often substantially higher than the average size.

One source of mapper skew is the luck of the draw, that it is rare to flip n coins and end up with exactly as many heads as tails. But a more serious problem is that key frequency is often power law distributed, so the most frequent key will come to dominate the counts. Consider the word count problem, and assume that word frequency observes Zipf's law from Section 5.1.5. Then the frequency of the most popular word (*the*) should be greater than the sum of the thousand words ranked from 1000 to 2000. Whichever bucket *the* ends up in is likely to prove the hardest one to digest.[1]

12.6.2 MapReduce under the Hood

All this is fine. But how does a MapReduce implementation like Hadoop ensure that all mapped items go to the right place? And how does it assign work to processors and synchronize the MapReduce operations, all in a fault-tolerant way?

There are two major components: the distributed hash table (or file system), and the run-time system handling coordination and managing resources. Both of these are detailed below.

Distributed File Systems

Large collections of computers can contribute their memory space (RAM) and local disk storage to attack a job, not just their CPUs. A distributed file system such as the Hadoop Distributed File System (HDFS) can be implemented as a distributed hash table. After a collection of machines register their available memory with the coordinating runtime system, each can be assigned a certain hash table range that it will be responsible for. Each process doing mapping can then ensure that the emitted items are forwarded to the appropriate bucket on the appropriate machine.

Because large numbers of items might be mapped to a single bucket, we may choose to represent buckets as disk files, with the new items appended to the end. Disk access is slow, but disk throughput is reasonable, so linear scans through files are generally manageable.

One problem with such a distributed hash table is fault tolerance: a single machine crash could lose enough values to invalidate the entire computation. The solution is to replicate everything for reliability on commodity hardware. In particular, the run-time system will replicate each item on three different machines, to minimize the chances of losing data in a hardware failure. Once the runtime system senses that a machine or disk is down, it gets to work replicating the lost data from these copies to restore the health of the file system.

[1] This is one of the reasons the most frequent words in a language are declared *stop* words, and often omitted as features in text analysis problems.

MapReduce Runtime System

The other major component of MapReduce environments for Hadoop or Spark is their *runtime system*, the layer of software which regulates such tasks as:

- *Processor scheduling*: Which cores get assigned to running which map and reduce tasks, and on which input files? The programmer can help by suggesting how many mappers and reducers should be active at any one time, but the assignment of jobs to cores is up to the runtime system.
- *Data distribution*: This might involve moving data to an available processor that can deal with it, but recall that typical map and reduce operations require simple linear sweeps through potentially large files. Thus moving a file might be more expensive than just doing the computation we desire locally.

  Thus it is better to *move processes to data*. The runtime system should have configuration of which resources are available on which machine, and the general layout of the network. It can make an appropriate decision of which processes should run where.
- *Synchronization*: Reducers can't run until something has been mapped to them, and can't complete until after the mapping is done. Spark permits more complicated work flows, beyond synchronized rounds of map and reduce. It is the runtime system that handles this synchronization.
- *Error and fault tolerance*: The reliability of MapReduce requires recovering gracefully from hardware and communications failures. When the runtime system detects a worker failure, it attempts to restart the computation. When this fails, it transfers the uncompleted tasks to other workers. That this all happens seamlessly, without the involvement of the programmer, enables us to scale computations to large networks of machines, on the scale where hiccups become likely instead of rare events.

Layers upon Layers

Systems like HDFS and Hadoop are merely layers of software that other systems can build on. Although Spark can be thought of as a competitor for Hadoop, in fact it can leverage the Hadoop distributed file system and is often most efficient when doing so. These days, my students seem to spend less time writing low-level MapReduce jobs, because they instead use software layers working at higher levels of abstraction.

The full big data ecosystem consists of many different species. An important class are *NoSQL* databases, which permit the distribution of structured data over a distributed network of machines, enabling you to combine the RAM and disk from multiple machines. Further, these systems are typically designed so you can add additional machines and resources as you need them. The cost of this flexibility is that they usually support simpler query languages than full SQL, but still rich enough for many applications.

The big data software ecosystem evolves much more rapidly than the foundational matters discussed in this book. Google searches and a scan of the O'Reilly book catalog should reveal the latest technologies when you are ready to get down to business.

12.7 Societal and Ethical Implications

Our ability to get into serious trouble increases with size. A car can cause a more serious accident than a bicycle, and an airplane more serious carnage than an automobile.

Big data can do great things for the world, but it also holds the power to hurt individuals and society at large. Behaviors that are harmless on a small scale, like scraping, become intellectual property theft in the large. Describing the accuracy of your model in an excessively favorable light is common for PowerPoint presentations, but has real implications when your model then governs credit authorization or access to medical treatment. Losing access to your email account is a bonehead move, but not properly securing personal data for 100 million customers becomes potentially criminal.

I end this book with a brief survey of common ethical concerns in the world of big data, to help sensitize you to the types of things the public worries about or should worry about:

- *Integrity in communications and modeling*: The data scientist serves as the conduit between their analysis and their employer or the general public. There is a great temptation to make our results seem stronger than they really are, by using a variety of time-tested techniques:
  - We can report a correlation or precision level, without comparing it to a baseline or reporting a p-value.
  - We can cherry pick among multiple experiments, and present only the best results we get, instead of presenting a more accurate picture.
  - We can use visualization techniques to obscure information, instead of reveal it.

  Embedded within every model are assumptions and weaknesses. A good modeler knows what the limitations of their model are: what they trust it to be able to do and where they start to feel less certain. An honest modeler communicates the full picture of their work: what they know and what they are not so sure of.

  Conflicts of interest are a genuine concern in data science. Often, one knows what the "right answer" is before the study, particularly the result that the boss wants most to hear. Perhaps your results will be used to influence public opinion, or appear in testimony before legal or governmental authorities. Accurate reporting and dissemination of results are essential behavior for ethical data scientists.

- *Transparency and ownership*: Typically companies and research organizations publish data use and retention policies to demonstrate that they can be trusted with their customer's data. Such transparency is important, but has proven to be subject to change just as soon as the commercial value of the data becomes apparent. It is often easier to get forgiveness than to get permission.

  To what extent do users own the data that they have generated? Ownership means that they should have the right to see what information has been collected from them, and the ability to prevent the future use of this material. These issues can get difficult, both technically and ethically. Should a criminal be able to demand all references to their crime be struck from a search engine like Google? Should my daughter be able to request removal of images of her posted by others without her permission?

  Data errors can propagate and harm individuals, without allowing a mechanism to people to access and understand what information has been collected about them. Incorrect or incomplete financial information can ruin somebody's credit rating, but credit agencies are forced by law to make each person's record available to them and provide a mechanism to correct errors. However, data provenance is generally lost in the course of merging files, so these updates do not necessarily get back to all derivative products which were built from defective data. Without it, how can your customers discover and fix the incorrect information that you have about them?

- *Uncorrectable decisions and feedback loops*: Employing models as hard screening criteria can be dangerous, particularly in domains where the model is just a proxy for what you really want to measure. Correlation is not causation. But consider a model suggesting that it is risky to hire a particular job candidate because people like him who live in lower-class neighborhoods are more likely to be arrested. If all employers use such models, these people simply won't get hired, and are driven deeper into poverty through no fault of their own.

  These problems are particularly insidious because they are generally uncorrectable. The victim of the model typically has no means of appeal. And the owner of the model has no way to know what they are missing, i.e. how many good candidates were screened away without further consideration.

- *Model-driven bias and filters*: Big data permits the customization of products to best fit each individual user. Google, Facebook, and others analyze your data so as to show you the results their algorithms think you most want to see.

  But these algorithms may contain inadvertent biases picked up from machine learning algorithms on dubious training sets. Perhaps the search engine will show good job opportunities to men much more often than to women, or discriminate on other criteria.

Showing you exactly what you say you want to see may prevent you from seeing information that you really need to see. Such filters may have some responsibility for political polarization in our society: do you see opposing viewpoints, or just an echo chamber for your own thoughts?

- *Maintaining the security of large data sets*: Big data presents a bigger target for hackers than a spreadsheet on your hard drive. We have declared files with 100 million records to be *bupkis*, but that might represent personal data on 30% of the population of the United States. Data breaches of this magnitude occur with distressing frequency.

  Making 100 million people change their password costs 190 man-years of wasted effort, even if each correction takes only one minute. But most information cannot be changed so readily: addresses, ID numbers, and account information persist for years if not a lifetime, making the damage from batch releases of data impossible to ever fully mitigate.

  Data scientists have obligations to fully adhere to the security practices of their organizations and identify potential weaknesses. They also have a responsibility to minimize the dangers of security breaches through encryption and anonymization. But perhaps most important is to avoid requesting fields and records you don't need, and (this is absolutely the most difficult thing to do) deleting data once your project's need for it has expired.

- *Maintaining privacy in aggregated data*: It is not enough to delete names, addresses, and identity numbers to maintain privacy in a data set. Even anonymized data can be effectively de-anonymized in clever ways, by using orthogonal data sources. Consider the taxi data set we introduced in Section 1.6. It never contained any passenger identifier information in the first place. Yet it does provide pickup GPS coordinates to a resolution which might pinpoint a particular house as the source, and a particular strip joint as the destination. Now we have a pretty good idea who made that trip, and an equally good idea who might be interested in this information if the bloke were married.

  A related experiment identified particular taxi trips taken by celebrities, so as to figure out their destination and how well they tipped [Gay14]. By using Google to find paparazzi photographs of celebrities getting into taxis and extracting the time and place they were taken, it was easy to identify the record corresponding to that exact pickup as containing the desired target.

Ethical issues in data science are serious enough that professional organizations have weighed in on best practices, including the *Data Science Code of Professional Conduct* (`http://www.datascienceassn.org/code-of-conduct.html`) of the Data Science Association and the *Ethical Guidelines for Statistical Practices* (`http://www.amstat.org/about/ethicalguidelines.cfm`) of the American Statistical Association.

I encourage you to read these documents to help you develop your sense of ethical issues and standards of professional behavior. Recall that people turn to data scientists for wisdom and consul, more than just code. Do what you can to prove worthy of this trust.

12.8 Chapter Notes

There exist no shortage of books on the topic of big data analysis. Leskovec, Rajarman and Ullman [LRU14] is perhaps the most comprehensive of these, and a good place to turn for a somewhat deeper treatment of the topics we discuss here. This book and some companion videos are available at `http://www.mmds.org`.

My favorite hands-on resources on software technologies are generally books from O'Reilly Media. In the context of this chapter, I recommend their books on data analytics with Hadoop [BK16] and Spark [RLOW15].

O'Neil [O'N16] provides a thought-provoking look at the social dangers of big data analysis, emphasizing the misuse of opaque models relying on proxy data sources that create feedback loops which exacerbate the problems they are trying to solve.

The analogy of disk/cache speeds to tortoise/escape velocity is due to Michael Bender.

12.9 Exercises

Parallel and Distributed Processing

12-1. *[3]* What is the difference between parallel processing and distributed processing?

12-2. *[3]* What are the benefits of MapReduce?

12-3. *[5]* Design MapReduce algorithms to take large files of integers and compute:

- The largest integer.
- The average of all the integers.
- The number of distinct integers in the input.
- The mode of the integers.
- The median of the integers.

12-4. *[3]* Would we expect map skew to be a bigger problem when there are ten reducers or a hundred reducers?

12-5. *[3]* Would we expect the problem of map skew to increase or decrease when we combine counts from each file before emitting them?

12-6. *[5]* For each of the following *The Quant Shop* prediction challenges dream up the most massive possible data source that might reasonably exist, who might have it, and what biases might lurk in its view of the world.

(a) *Miss Universe.*

(b) *Movie gross.*

(c) *Baby weight.*

(d) *Art auction price.*

(e) *White Christmas.*

(f) *Football champions.*

(g) *Ghoul pool.*

(h) *Gold/oil prices.*

Ethics

12-7. *[3]* What are five practical ways one can go about protecting privacy in big data?

12-8. *[3]* What do you consider to be acceptable boundaries for Facebook to use the data it has about you? Give examples of uses which would be unacceptable to you. Are these forbidden by their data usage agreement?

12-9. *[3]* Give examples of decision making where you would trust an algorithm to make as good or better decisions as a person. For what tasks would you trust human judgment more than an algorithm? Why?

Implementation Projects

12-10. *[5]* Do the stream sampling methods we discussed really produce uniform random samples from the desired distribution? Implement them, draw samples, and run them through the appropriate statistical test.

12-11. *[5]* Set up a Hadoop or Spark cluster that spans two or more machines. Run a basic task like word counting. Does it really run faster than a simple job on one machine? How many machines/cores do you need in order to win?

12-12. *[5]* Find a big enough data source which you have access to, that you can justify processing with more than a single machine. Do something interesting with it.

Interview Questions

12-13. *[3]* What is your definition of big data?

12-14. *[5]* What is the largest data set that you have processed? What did you do, and what were the results?

12-15. *[8]* Give five predictions about what will happen in the world over the next twenty years?

12-16. *[5]* Give some examples of best practices in data science.

12-17. *[5]* How might you detect bogus reviews, or bogus Facebook accounts used for bad purposes?

12-18. *[5]* What do the map function and the reduce function do, under the Map-Reduce paradigm? What do the combiner and partitioner do?

12-19. *[5]* Do you think that the typed login/password will eventually disappear? How might they be replaced?

12-20. *[5]* When a data scientist cannot draw any conclusion from a data set, what should they say to their boss/customer?

12-21. *[3]* What are hash table collisions? How can they be avoided? How frequently do they occur?

Kaggle Challenges

12-22. Which customers will become repeat buyers?
`https://www.kaggle.com/c/acquire-valued-shoppers-challenge`

12-23. Which customers are worth sending junk mail to?
`https://www.kaggle.com/c/springleaf-marketing-response`

12-24. Which hotel should you recommend to a given traveler?
`https://www.kaggle.com/c/expedia-hotel-recommendations`