1

INTRODUCTION TO BIG DATA

Unit Structure

1.1 BIG DATA

Big data is referred to as the collection of a huge data set that includes structured, semi-structure or unstructured data which cannot be stored and analyzed by traditional database management systems. The primary source of big data is various activities done by uses through the internet for various purposes.

The use of the internet is an integral part of our lifestyle and due to that, it is very common to use various digital platforms on the internet for day-to-day work. Lots of people leave their footprint in the form of the data by doing various activities on social media, online shopping websites, online business transactions, online banking systems, online searching, online education system and many others. Subsequently, it is observed that the growth of data is exponential way. So very advanced technology has emerged to manage a huge amount of data.

1.1.1 Introduction to Big data Platform:

The invention of hand-held digital devices has been considering as a prime factor for the growth of internet users. In today's life, the internet is accessed via computers, mobile phones, personal digital assistant devices, gaming stations and digital TV. It is believed that the Internet is the most fast growing technology.

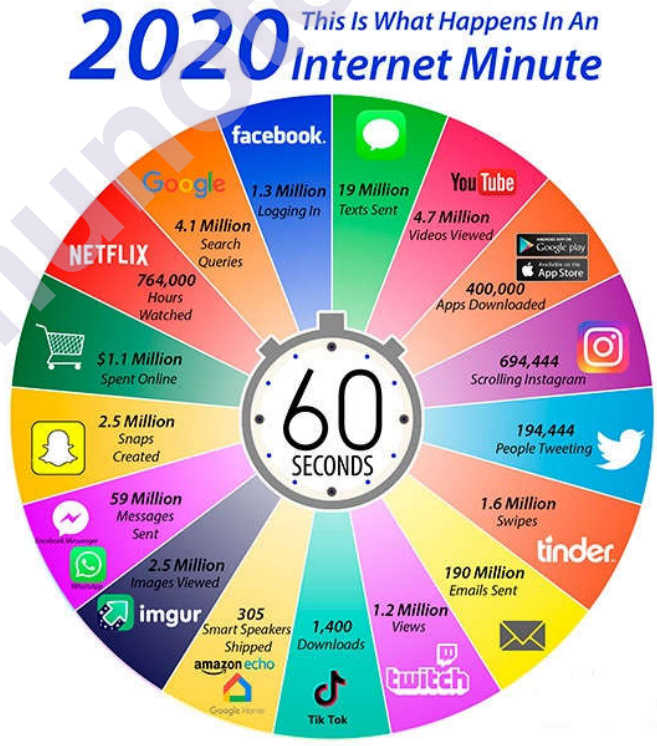


Figure 1 : Internet usage in 2020

Big data cannot be analyzed by conventional technology or it cannot be stored by the traditional database management system. The biggest challenge to work with big data is the exponential growth of data which requires very advanced technology to store it in such a way that can be

utilized for analysis purposes. Various big data platforms enable storing, managing, merging, developing, deploying, operating and analyzing big data. The big data infrastructure generally consists of very advanced data storage systems, high computing servers and big data management technology. A big data platform normally includes very advanced infrastructure which combines the capability of several big data applications. Whereas, the big data analytics software mainly focuses on providing facilities to support analytics for extremely large data sets. In other words, analytics helps to convert a huge amount of data into smart data or high-quality information which provides deeper insights for the decision-making process.

There are many big data tools are available in the market for Big data analytics, few can be listed here. Apache Hadoop, Cassandra, data wrapper, mongo DB, Apache storm, Tableau, R, CDH (Cloudera Distribution for Hadoop), Elastic search, Kaggle, Hive, Spark, OpenText, Oracle Data Mining, BigML, CouchDB, Pentaho, Adverity, Xplenty, Apache SAMOA, Lumify, HPCC, Adverity, Knime, Talend, rapid miner, Microsoft Azure, Amazon Web service, Google bigquery, VMware, Google big data, IBM big data, wavefront, Cloudera enterprise big data, Oracle Big data analytics, DataTorrent, mapR converged data platform, Splunk big data analytics, Big object, Opera solutions signal hub, SAP Big data analytics, Next Pathway, 1010data, GE industrial internet, SGI big data, Teradata big data analytics, Intel big data, HP big data, Dell Big data analytics, Cisco big data, Pentahol big data, Opera solutions big data.

1.1.2 Traits of big data:

Billions of users are connected to the World Wide Web and spending a significant amount of time via mobiles, computers and other devices. Consecutively, there are collections of large-scale unstructured data and it is also increasing with a constant growth rate every day. Hence, it emerges into the necessity of an advanced technology that could support a wide range of data storage, scalable processing and analysis of this data. In this scenario, big data technologies evolved as a revolutionary solution to cope up with all these solutions.

Big data defines with 5V's characteristics. The first 'V' is a symbolization of extra-large scale of the data volume. The second 'V' is a symbolization of a variety of data that emphasis on heterogeneous data (structure, unstructured and semi-structure). The third 'V' is a symbolization of velocity of data that highlights on data-analytics. Figure 2 shows 5 'V' characteristics of big data.

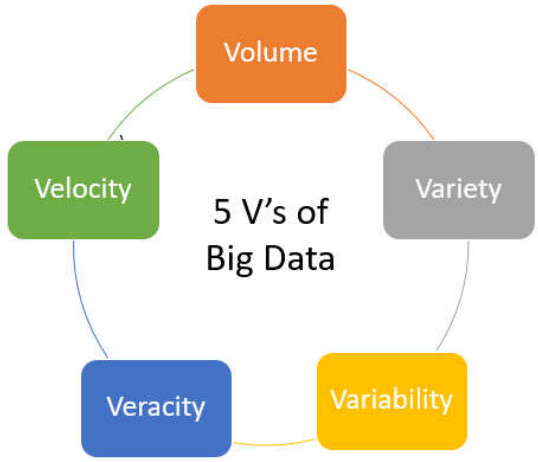


Figure 2 : 5 'V' characteristics of big data

1 Volume

Big data has been defining with five V characteristics. The first V is symbolization of volume. The big data has an extra-large scale data. The volume of data can be measured with zettabytes.

Unit	Abbreviation	Size
byte	B	8 bits
kilobyte	KB	1,024 bytes or 10^3 bytes
megabyte	MB	1,024 KB or 10^6 bytes
gigabyte	GB	1,024 MB or 10^9 bytes
terabyte	TB	1,024 GB or 10^12 bytes
petabyte	PB	1,024 TB or 10^15 bytes
exabyte	EB	1,024 PB or 10^18 bytes
zettabyte	ZB	1,024 EB or 10^21 bytes
yottabyte	YB	1,024 ZB or 10^24 bytes

In real life, millions of users are connected with the World Wide Web and spending a significant amount of time for surfing and online activities with the help of many hand-held devices, such as computers, laptops and tablets.

Due to this, a constant growth rate was found, and mostly this data increasing at petabyte scale. The volume of data was previously measuring into Terabytes, later on, Petabytes and nowadays that is shifted to Zettabytes. Have a look at some statics about today's scenario. Only

Twitter has more than 500 tweets to send every day and hence it generates more than 7 TB of data every day. Whereas, on Facebook, approximately 4 petabytes of the post or likes related data and hence it generates 10 TB data every day. It is also observed that more than 65 billion messages are sent by people via WhatsApp. Some online enterprises are also believed to generate terabytes of data every hour of every day. A new era has begun in the field of transportation and 4 TB of data has been generated by each connected car. On the Internet, 5 billion searches are made from all around the world and the Internet is a huge network of many web servers and web services. This is just for having an idea that how much data we produce and even how much data will be available in the future to dig into it?

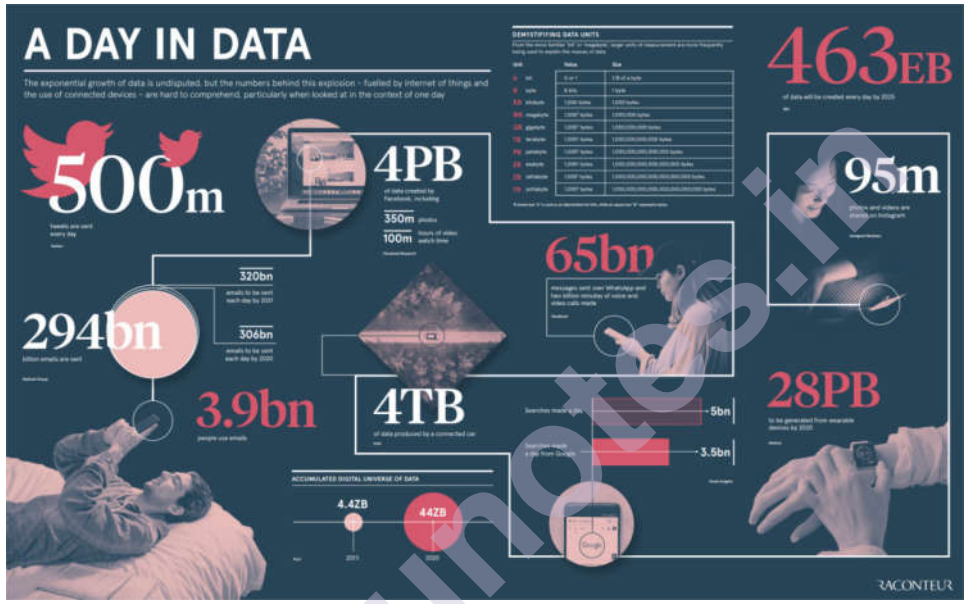


Figure 3 : A day in big data

In other words, we can say that a massive amount of data has generated every day, which has to store. An organization has to manage storage and processing in real-time, which is the biggest challenge related to big data.

2 Variety

The second 'V' is a symbol of a variety of data that means the big data can be found as structured data, unstructured data or semi-structured data. In an online environment, the source of data could be different and hence the data can have a different format subsequently the data may have a variety of format. Due to the presence of text, media, links and application programs as a part of today's websites, a variety of data is found as a part of the big data. In the case of convention data technology, data could be processed only if it is structured and represented in the two-dimensional table. On the other hand, the major portion of today's websites and social media data consist of text, images and videos, which are very complex and difficult to process. Herein, text, links, maps, network hierarchies and streaming data are unstructured and cannot be stored in that 2-D format.

Some of the data are semi-structure, which is more structure in nature, compare to unstructured data. It cannot process with the help of a relational database. Normally, a tree-like structure such as XML is used to store semi-structured data. It is also known as the key-value pair structure. XML and JSON are an example of these kinds of data storage formats.

	Structured Data	Unstructured Data
Type of data	It is represented as numbers, dates, strings and alphanumeric values etc.	It may consist of text, images, audio and videos etc.
Storage structure	It can be easily stored in 2-dimensal structure of row and column. So, it can be stored with Excel or RDBMS.	It can be stored with (NOSQL) Non relational structure, Big Table, graph data and many other advanced data structures.
Source of data	It is part of major business data stored with ERP systems and other MIS system.	It is normally present at a part of online systems and web data.
Growth rate	It is increasing at the growth rate of 20-30%	It is increasing at the growth rate of 80-90%
Analysis Process	It is very easy to analysis it with RDMS and with use of simple algorithms.	It is very complicated to preprocess, process and analysis of it. It requires very complex and advanced technology for analysis purpose such as text analysis algorithms, Artificial Intelligent and Neural Network.

Due to all these challenges, many innovations have provided solutions to process data in various formats such as Big Table, graph data and many others. Even due to these data challenges, NoSQL technology emerged as a solution and it has been adapted by many.

3 Velocity

The third ‘V’ is a symbol of the velocity of the data. A Velocity is related to the speed at which data are arriving and it has to store. Similarly, velocity is related to 'How much the data received in a specific period?' and that could accommodate into the database. Sometimes, velocity is also referred as the measurement of the speed at which the data it is moving towards the data repository. For the conventional system, it is impossible to manage the constant flow of data that comes from various data streams connected with RFID sensors. More than that, for the real-time system, it is essential to analyze this data in real life as the life of the data is short.

For real-life applications, batch processing is not a good option specifically for data streams. The real-time computing system, which accepts data from many data streams and computing systems has to execute the query and identifies current trends based on the recent and up-to-date data in real-time. The Google map traffic analysis system is this kind of real-time system which processes a massive amount of current traffic-related data and provides valuable information in real-time.

4 Varacity

The next 'V' stands for 'Varacity' or 'Validity' of the data. The veracity refers to the trustworthiness and quality of the data used for analysis. Nowadays, the data is available in huge amounts but the quality of data is a big question. Only high-quality data yields meaningful information, which seems a difficult task in an online environment. The source of data and its authenticity must be considered at the time of data preprocessing. The handling of noise, inaccurate data and missing data must be done to increase the quality of the data. The process of validating data is a big challenge due to the consideration of context analysis for text data.

5 Variability

The next 'V' stands for 'Variability' or uncertainty of the data. The variability of the data suggests too many changes in the data. Due to changing nature of the data, the data processing methods and the models has to also change according to the data. The constant changes and innovation in the technology lead to the addition of new things into the Internet, and hence new kinds of data formats and processing methods involve automatically. The general methodology for various kinds of objects cannot be applicable. Subsequently, new algorithms and processing approaches have to introduce to manage constantly variable data. The conventional technology only focused on the analysis of historical data collected over a period of time from the same enterprise system. This system design is for specific kinds of processing requirements concerning that data only. Internet and advanced IoT systems are capable to connect many different systems with different components. This emerges as needs flexible algorithms that can work well with a wide range of data variety.

1.1.3 Challenges of conventional systems:

The conventional systems are mainly made to manage enterprise level data but, it is normally do not focus on gathering data from out of the organization. Due to this, the data of conventional system has predefined and fixed structure, where as big data system has mix of various kinds of data. More than this, volume of data for conventional system is limited to gigabytes to terabytes only whereas big data system has to store and manage zetabytes of data with cloud and other advanced data storage system. The conventional system can analysed data with algorithms which are suitable to process structured data only. The analysis of structured data may be done by various functions such as statistical functions and date functions. In the market now-a-days the most commonly used statistics

softwares for that is SPSS software. Statistical methods are most suitable for quantitative data. In statistic, a wide range of aggregation functions are available which can be applicable o groups. In contrast to that, statistical methods cannot be applicable to heterogeneous data. Hence, a wide variety of algorithms are needed to process structured, semi-structured and unstructured data. The analysis of unstructured data or text data is very complicated in nature, compare to structured data. For example, search engines has to perform text analysis on web data, it may required key word extraction, semantic analysis and similarity matching etc.

Another limitation of the conventional data management system is related to the storage capacity of data. In the case of a conventional data management system data is generated at the rate of per hour or per day. The business data can be stored at the centralized level and shared with all remote devices. The data has a fixed schema and it is not possible to change the structure at run time. The data manipulation functions are predefined and various data operations are performed on regular basis. Subsequently, the analysis process is also implied according to the data. In contrast to that, big data has flexible schema and heterogeneous data. More than that, big data is generated at the speed of exponential rate. Due to that, data has to store with a flat-file structure or in such a way that can be shared over a wide network. The latest technological revolution has made it possible with cloud storage and clustering storage systems. Subsequently, the processing method has to adapt the relevant technology for future analysis. In short, big data analysis systems should be flexible, scalable and more tolerant to failure to manage the need of the time. That should also allow distributed and allowed parallel processing to speed up the analysis task.

1.1.4 Web data:

The traditional system focus on a data management system that processes mostly transaction data such as enterprise resource planning system (ERP) and customer relationship management (CRP) system. The major source of this kind of system is transaction data produce due to various business transactions which have to be processed via predefine business methods.

On the other side, the Web of data is today's reality and exist due to the relationship among the data on the internet. The web data consist of Website data, Domain name data, News data, Web activity data, Web search data, IP address data, Click Stream data, Sentiment web data, Web traffic data and Semantic web data. The entire collection of interrelated data set on the web is also sometimes referred to as linked data or Semantic Web. An example of a Linked Dataset is DBPedia, which includes Wikipedia data. A significant feature of DBPedia is it makes it possible to get the content of Wikipedia in RDF format.

Web analytics is a process of measuring web traffic, web search and web uses. Many web search engines perform web analysis and help internet users to search from a huge collection of web pages present on the Internet. The analysis of web data is possible with the use of HTML, XML, RDF, OWL, SPARQL, etc.

In addition to that, the volume of Web data is constantly increasing. Along with that, a variety of data sources continuously generating various kinds of data and makes web data more complicated and unstructured. The data on the Internet arises due to social media, social networking links, social media posts, image data, video data, click stream data and many other activities. Another source of data is various surveys, online surveys, experiments and observations of the people. Sometimes, market survey data, industry reports, consumer analysis reports, various kinds of business reports and comparative analysis reported also loads tons of different types of data on the internet. In this era, due to the presence of GPS and GIS, lots of location related data is also generated by mobile devices and other geospatial systems. Many security systems, produces images and videos in massive amount with the use of surveillance and other security devices. With the help of many remote sensors, RFID devices, IoT systems and many other real-time tracking systems load a massive amount of data. Satellite images and weather-related data are also an integral part of Internet data.

1.1.5 Analytic processes and tools:

Data analysis is a process that transforms raw data into very useful information. Data analysis is very useful for generating various statistics related to data, meaningful insights and valuable explanations to manage data-driven business decisions. There are many software and applications which perform various data analysis tasks. It is crucial to choose an appropriate tool to execute, from a wide range of data analytics tools. The selection process for data analytics tools may consider many parameters such as price, robustness, supported data models, learning curve, scalability, expandability, visualization facility and many others.

Data analysis generally follows well-defined steps. It is very important to understand the importance of process along with know-how of data to yield meaningful insights and valuable patterns. Normally, to carry out the analytics process following steps are required to conduct : (1) data collection (2) data cleaning and preprocessing (3) data analysis (4) visualising the output (5) understanding the results.

Data collection: The first step of the data analysis process is to understand the source of the data, the format of the data and the collection procedure. Based on all this, the data collection procedure has to be defined. Nowadays various data collection tools are also used to capture data in real-time, such as barcode readers, cameras, voice detecting machines, sensors and automatic weighing machines.

Data cleaning and preprocessing: It is very essential to conduct a data cleaning process to convert raw data into high-quality data. The data cleaning process may include the process for removal of duplicate data, removal of outliers and removal of errors. Sometimes, it is also essential to identify and fill the gap between data that are collected from different sources to integrate them into a single database. The data cleaning process may carry out manually or by using automated data cleaning tools. Along

with the data cleaning process, it is also very essential to conduct an exploratory analysis of the data. This step helps to understand the characteristic of data and the relationship among them. Sometimes the existing co-relationship of the data is very essential to find out to establish a hypothesis.

Data analysis: A data analytic process mainly depends on the goal of the process and the availability of the data. There are lots of statistical techniques used for analysis, a few are listed here univariate or bivariate analysis, regression analysis, time series analysis, descriptive analysis and predictive analysis.

The descriptive analysis identifies the underlying relationship among the data. This kind of analysis may help to find, a summary of the data, to describe the data, and to determine the next processing step to be carried out. The predictive analysis helps to find future values for future, based on the historical data. This kind of analysis may help to predict market sales based on the previous year's sales data.

Visualiting the output: Data visualization is equally important as the data analytics process. The output of the analysis process must be clearly well presented and understandable. Sometimes data visualization tools are used to increase the readability of the data. More specifically, these tools are used when the volume of data is very large. Google charts, Infogram and Tableau are well known examples of data visualization tools.

Understanding the results: Understanding of final output is a very crucial step. For instance, the output may be misleading or erroneous due to several reasons. In this situation, it is very essential to identify the reason behind it, and to determine correct approach.

1.1.6 Analysis Vs Reporting:

In this digital era, the wealth of information brings into existence due to modern analytics technology. Analysis and reporting both are valuable for the same. The goal of the analysis process is to inspect the data and transform it into useful information. The goal of the reporting process is transforming the output of the analytic process in a presentable format. The main purpose of conducting the analysis process is examining, interpreting, comparing and predicting the data. Whereas reporting process is mainly focusing on highlighting organizing, summarizing and formatting processes. Sometimes, visualization of output may enhance with the use of chats, maps, graphs and linking of data.

1.1.7 Modern data analytic tools:

Big data analytics uses the large quantities of data that generates and gathers from various sources and converts into meaningful information. There are many big data tools, and having the most in-demand by data scientist. Some vital tools of big data are the following:

No.	Tools	Benefits
1	R	R programming language is the most common choice of many data scientists today. R is free and available under an open-source license. R available for different types of hardware and software e.g. Windows, Unix systems and the Mac. The most attractive feature of 'R' is the extendibility and integration of a rich library of packages.
2	Python	Python is a very powerful yet, open source language and an easy-to-learn language. It offers statistical and mathematical functions. Few famous libraries are NumPy, SciPy, etc. It is a high-level language with high readability and object-oriented programming functionality.
3	PIG and HIVE	Hadoop is distributed File System that allows the storage of data in a distributed manner. The ecosystem is consists of many tools. Hadoop MapReduce facilitates the processing of large volumes of data in a parallel and distributed manner. HIVE and PIG are also an integral part of the Hadoop ecosystem. They facilitate processing and analysis. More specifically, HIVE is a data warehouse with HiveQL, which is the query language for large datasets stored in HDFS. PIG runs on Hadoop cluster and processes and analyzes large datasets using a scripting language.
4	Tableau	Tableau is a very easy-to-learn data visualization tool that converts numeric and textual data into beautiful visuals. It is user friendly, mobile friendly, simple yet, fast. Anyone without knowledge of coding can also use Tableau.
5	Jupyter Notebook	Jupyter Notebook is a free, open-source and online data analytics tool. It supports 40+ programming languages so, it is known as a multi-language computing environment. It allows the use of python's wide variety of packages and visualization tools.
6	Google Data Studio	Google data studio is a free data analytics tool that can automatically integrate with other Google applications such as Google Analytics, Google Ads, Google Sheets and Google BigQuery.

1.2. STATISTICAL CONCEPTS:

1.2.1 Sampling distributions

We consider sample as an analytic subset of a larger population in statistics. Samples allow researchers to conduct their studies with more manageable data and in a timely manner. Random samples do not have much bias if they are large enough, but achieving such a sample may be expensive and time consuming. In simple random sampling, every entity in the population is identical.

What is a Sampling Distribution?

A sampling distribution is a probability distribution of a statistic obtained from a larger number of samples. It is the distribution of frequencies of a range of different outcomes that could possibly occur for a statistic of a population.

A population may refer to an entire group of people, objects, events, hospital visits, or measurements. A population can thus be said to be an aggregate observation of subjects grouped together by a common feature.

- A sampling distribution is a statistic that is arrived out through repeated sampling from a larger population.
- It describes a range of possible outcomes that of a statistic, such as the mean or mode of some variable, as it truly exists a population.
- The majority of data analyzed by researchers are actually drawn from samples and not populations.

Understanding Sampling Distribution

Huge amount of data drawn and used by academicians, statisticians, researchers, marketers, analysts, etc. are actually samples, not populations. Consider this example, a medical researcher that wanted to compare the average weight of all babies born in North America from 1995 to 2005 to those born in South America within the same time period cannot within a reasonable amount of time draw the data for the entire population of over a million childbirths that occurred over the ten-year time frame. He will instead only use the weight of, say, 100 babies, in each continent to make a conclusion. The weight of 200 babies used is the sample and the average weight calculated is the sample mean.

Few Definitions

A ***sample*** is a subset of the population.

A ***population*** is a collection of all the elements of interest.

The ***sampled population*** is the population from which the sample is drawn.

An ***element*** is the entity on which data are collected.

A ***frame*** is a list of the elements that the sample will be selected from.

1.2.2 Re-sampling

Once we have a data sample, it can be used to estimate the population parameter. The problem is that we only have a single estimate of the population parameter. One way to address this is by estimating the population parameter multiple times from our data sample. This is called re-sampling.

Statistical re-sampling methods are procedures that describe how to economically use available data to estimate a population parameter. The result can be both a more accurate estimate of the parameter (such as taking the mean of the estimates) and a quantification of the uncertainty of the estimate (such as adding a confidence interval).

Two commonly used re-sampling methods that you may encounter are k-fold cross - validation and the bootstrap.

- **Bootstrap**. Samples are drawn from the dataset with replacement (allowing the same sample to appear more than once in the sample), where those instances not drawn into the data sample may be used for the test set.

- **k-fold Cross Validation**. A dataset is partitioned into k groups, where each group is given the opportunity of being used as a held out test set leaving the remaining groups as the training set.

The k-fold cross-validation method specifically lends itself to use in the evaluation of predictive models that are repeatedly trained on one subset of the data and evaluated on a second held-out subset of the data.

Generally, re-sampling techniques for estimating model performance operate similarly. Re-sampling methods are very easy to use, requiring little mathematical knowledge. They are methods that are easy to understand and implement compared to specialized statistical methods that may require deep technical skill in order to select and interpret.

The re-sampling methods are easy to learn and easy to apply. They require no mathematics beyond introductory high-school algebra, etc are applicable in an exceptionally broad range of subject areas.

A downside of the methods is that they can be computationally very expensive, requiring tens, hundreds, or even thousands of re-samples in order to develop a robust estimate of the population parameter.

The key idea is to resample form the original data either directly or via a fitted model to create replicate datasets, from which the variability of the interest can be assessed without long-winded and error-prone analytical calculation. Because this approach involves repeating the original data analysis procedure with many replicate sets of data, these are sometimes called computer intensive methods.

Each new subsample from the original data sample is used to estimate the population parameter. The sample of estimated population parameters can then be considered with statistical tools in order to quantify the expected value and variance, providing measures of the uncertainty of the estimate. Statistical sampling methods can be used in the selection of a subsample from the original sample.

A key difference is that process must be repeated multiple times. The problem with this is that there will be some relationship between the samples as observations that will be shared across multiple subsamples. This means that the subsamples and the estimated population parameters are not strictly identical and independently distributed. This has implications for statistical tests performed on the sample of estimated population parameters downstream, i.e. paired statistical tests may be required.

Subset of samples can be used to fit a model and the remaining samples are used to estimate the efficacy of the model. This process is repeated multiple times and the results are aggregated and summarized. The difference in techniques usually depends on the method in which subsamples are chosen.

1.2.3 Statistical Inference

Statistical inference makes propositions about a population, using data drawn from the population with some form of sampling. Given a hypothesis about a population, for which we wish to draw inferences, statistical inference consists of selecting a statistical model of the process that generates the data and deducing propositions from the model.

"The majority of the problems in statistical inference can be considered to be problems related to statistical modelling". Sir David Cox has said, "How [the] translation from subject-matter problem to statistical model is done is often the most critical part of an analysis".

The conclusion of a statistical inference is a statistical proposition. Some common forms of statistical proposition are the following:

- a point estimate, i.e. a particular value that best approximates some parameter of interest;
- an interval estimate, e.g. a confidence interval (or set estimate), i.e. an interval constructed using a dataset drawn from a population so that, under repeated sampling of such datasets, such intervals would contain the true parameter value with the probability at the stated confidence level;
- a credible interval, i.e. a set of values containing, for example, 95% of posterior belief;
- rejection of a hypothesis;
- Clustering or classification of data points into groups.

Models and assumptions

Any statistical inference requires some assumptions. A **statistical model** is a set of assumptions concerning the generation of the observed data and similar data. Descriptions of statistical models usually emphasize the role of population quantities of interest, about which we wish to draw inference. Descriptive statistics are typically used as a preliminary step before more formal inferences are drawn.

Paradigms for inference

Different schools of statistical inference have become established. These schools or "paradigms" are not mutually exclusive, and methods that work well under one paradigm often have attractive interpretations under other paradigms.

There are four paradigms:

(i) Classical statistics or error statistics,

(ii) Bayesian statistics,

(iii) Likelihood based statistics and

(iv) Akaikean Information Criterion based statistics.

- The practice of statistics falls broadly into two categories:

(1) Descriptive or

(2) Inferential.

When we are just describing or exploring the observed sample data, we are doing descriptive statistics. However, we are often also interested in understanding something that is unobserved in the wider population, this could be the average blood pressure in a population of pregnant women for example, or the true effect of a drug on pregnancy rate, or whether a new treatment perform better or worse than the standard treatment. In these situations we have to recognise that almost always we observe only one sample or do one experiment. If we took another sample or did another experiment, then the result would almost certainly vary. This means that there is uncertainty in our result, if we took another sample or did another experiment and based our conclusion solely on the observed sample data, **we may even end up drawing a different conclusion!**

- The purpose of statistical inference is to estimate this sample to sample variation or uncertainty. Understanding how much our results may differ if we did the study again, or how uncertain our findings are, allows us to take this uncertainty into account when drawing conclusions. It allows us to provide a plausible range of values for the true value of something in the population, such as the mean, or size of an effect and it allows us to make statements about whether our study provides evidence to reject a hypothesis.

Estimating uncertainty:

- Almost of all of the statistical methods you will come across are based on sampling distribution. This is a completely abstract concept. It is the theoretical distribution of a sample statistic such as the sample mean over infinite independent random samples. We typically only do one experiment or one study and certainly don't replicate a study so many times that we could empirically observe the sampling distribution. It is thus a theoretical concept. However we can estimate what the sampling distribution looks like for our sample statistic or point estimate of interest based on only one sample or one experiment or one study. The spread of the sampling distribution is captured by its standard deviation, just like the spread of a sample distribution is captured by the standard deviation.

Do not get confused between the sample distribution and sampling distribution, one is the distribution of the individual observations that we observe or measure, and the other is the theoretical distribution of the sample statistic that we don't observe.

We should not get confused between the standard deviation of the sample distribution and the standard deviation of the sampling distribution, we call the standard deviation of the sampling distribution the standard error. This is useful because the standard deviation of the sampling distribution captures the error due to sampling, it is thus a measure of the precision of the point estimates or put another way, a measure of the uncertainty of our estimate. Since we often want to draw conclusions about something in a population based on only one study, understanding how our sample statistics may vary from sample to sample, as captured by the standard error, is also really useful. The standard error allows us to try to answer questions such as: what is a plausible range of values for the mean in this population given the mean that I have observed in this particular sample? The standard error is thus integral to all statistical inference, it is used for all of the hypothesis tests and confidence intervals that you are likely to ever come across.

1.2.4 Prediction error

A prediction error is the failure of some expected event to occur. When predictions fail, humans can use meta-cognitive functions, examining prior predictions and failures and deciding. For example, whether there are correlations and trends such as consistently being unable to foresee outcomes accurately in particular situations. Applying that type of knowledge can inform decisions and improve the quality of future predictions.

Predictive analytics software processes new and historical data to forecast activity, behavior and trends. The programs apply statistical analysis techniques, analytical queries and machine learning algorithms to data sets to create predictive models that quantify the likelihood of a particular event happening.

Errors are an inescapable element of predictive analytics that should also be quantified and presented along with any model, often in the form of a confidence interval that indicates how accurate its predictions are expected to be. Analysis of prediction errors from similar or previous models can help determine confidence intervals.

In artificial intelligence (AI), the analysis of prediction errors can help guide machine learning (ML), similarly to the way it does for human learning. In reinforcement learning, for example, an agent might use the goal of minimizing error feedback as a way to improve. Prediction errors, in that case, might be assigned a negative value and predicted outcomes a positive value, in which case the AI would be programmed to attempt to maximize its score. That approach to ML, sometimes known as error-driven learning, seeks to stimulate learning by approximating the human drive for mastery.

1.3. DATA ANALYSIS:

Regression analysis is a set of statistical processes for estimating the relationships between a dependent variable and one or more independent variables. The most common form of regression analysis is linear regression, in which one finds the line that most closely fits the data according to a specific mathematical criterion.

For example, the method of ordinary least squares computes the unique line that minimizes the sum of squared differences between the true data and that line. For specific mathematical reasons, this allows the researcher to estimate the conditional expectation of the dependent variable when the independent variables take on a given set of values. Less common forms of regression use slightly different procedures to estimate alternative location parameters or estimate the conditional expectation across a broader collection of non-linear models.

Regression analysis is primarily used for two conceptually distinct purposes.

First, regression analysis is widely used for *prediction* and *forecasting*, where its use has substantial overlap with the field of machine learning.

Second, in some situations regression analysis can be used to infer causal relationships between the independent and dependent variables.

Importantly, regressions by themselves only reveal relationships between a dependent variable and a collection of independent variables in a fixed dataset. To use regressions for prediction or to infer causal relationships, respectively, a researcher must carefully justify why existing relationships have predictive power for a new context or why a relationship between two variables has a causal interpretation.

1.3.1 Regression modeling

Regression is a form of machine learning where we try to predict a continuous value based on some variables. It is a form of supervised learning where a model is taught using some features from existing data.

From the existing data the regression model then builds its knowledge base. Based on this knowledge base the model can later make predictions for outcomes on new data.

Continuous values are numerical or quantitative values that have to be predicted and are not from an existing set of labels or categories. There are lots of examples of regression where it is heavily used on a daily basis and in many cases it has a direct business impact.

Types of Regressions Models:

- Linear Regression
- Logistic Regression
- Polynomial Regression
- Stepwise Regression
- Ridge Regression
- Lasso Regression

1.4. ANALYSIS OF TIME SERIES:

1.4.1 Linear systems analysis

A CEO of car manufacturing company is interested in knowing what will be approximate sale of cars for next 2 years. Airline Company is eager to know how many passengers are likely to travel through their flights in next 2 months. Manufacturer of perishable sweet items would want to know how much demand will be there for next 2 weeks. Head of Supply Chain Company wants to know how much will be petrol and diesel prices for next 2 days. A CFO of an IT company is interested in knowing stock prices for next 2 hours.

Everybody sitting at higher positions are taking decisions is of utmost importance. Only resource they have with them is historical data. **Time series analysis** is a technique with which one can forecast for the future, based on historical data. In all such scenarios, one can use historical data and apply time series analysis on the data to create a model which can aid in getting some idea about future. It is important to note that the historical data has to be time-dependent (collected with respect to time function).

Univariate time series is one where data is collected with respect to only one variable, with respect to periodic time instance, over s period where as **multivariate time series** in one in which data for multiple variables is collected for a certain time period. Recording temperature values every hour for a week is an example of univariate time series. Whereas, recording temperature, pressure and humidity every hour for a week is an example of multivariate time series.

Data collected for the time series can be linear or non-linear. Linear data, when plotted in the form of a graph, will be sequential in nature. Any data point with be connected to only two other datapoints, previous and the

next. On the other hand, non-linear data when plotted in the form of graph will not result into a straight line.

A. Components of time series data

Any time series may have some inherent properties / components – Trend, Seasonality, Cyclicity and Irregularity.

(1) **Trend** is an important component of any time series which is a result of overall long term effect of environmental factors. Trend may show inclining or declining effect over a period.

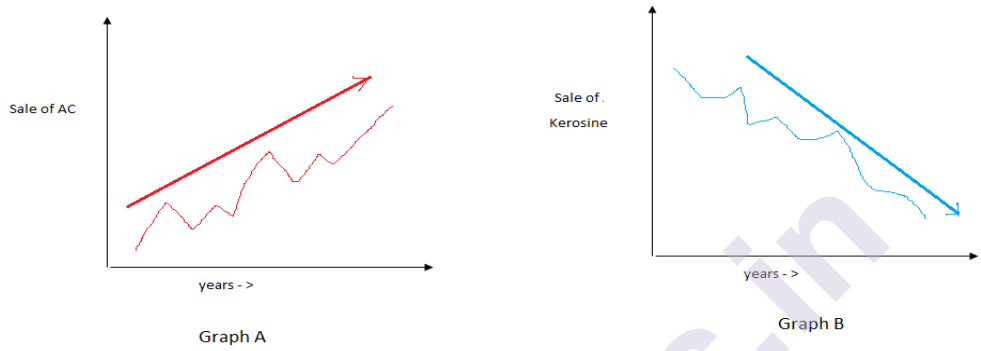


Figure 4.1: Trend component of time series

As seen in graph A, in figure 4.1, there has been overall increase in the sale of air conditioners and overall decrease in sale of kerosene for cooking purpose.

(2) **Seasonality** is the short term movement in data due to seasonal factors. E.g. there can be notable increase of sale of warm clothes during winter season or even sudden increase in the sale of washing machines during rainy season can be attributed to seasonal fluctuation.

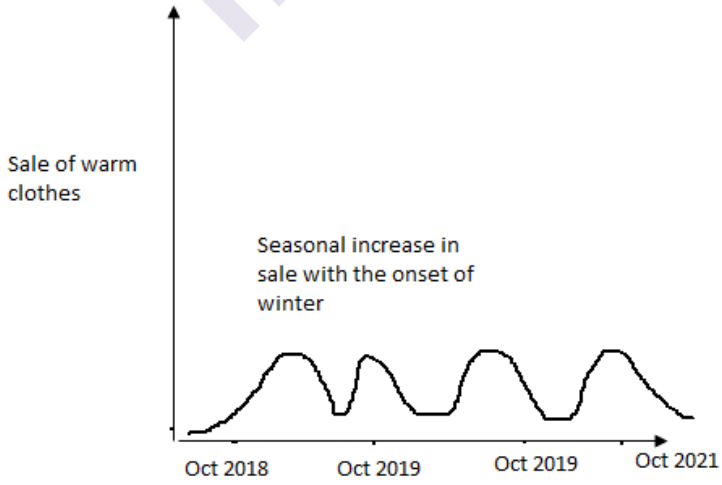


Figure 4.2: Seasonality component of time series

(3) **Cyclicity** is a pattern observed when the data is collected for a very long duration, say 40-50 years. This pattern repeats over a period, but the gap between two time instances may not be fixed. E.g. recession occurring time and again, but it is difficult to predict the next occurrence.

Figure 4.3: Cyclicity component of time series

(4) **Irregularities / random component** are the sudden changes in data which are unlikely to be repeated. Such a sudden change in data cannot be predicted by other components like trend / seasonality or cyclicity. These variations are mostly accidental in nature and may result in to change in trends / seasonality and cyclicity in the forthcoming period. Natural calamities can be an example which may cause irregularities in data. E.g. Covid pandemic has resulted steep increase in the sale of electronic gadgets such as tablets, laptops and cell phones on account of online lectures from schools and colleges.

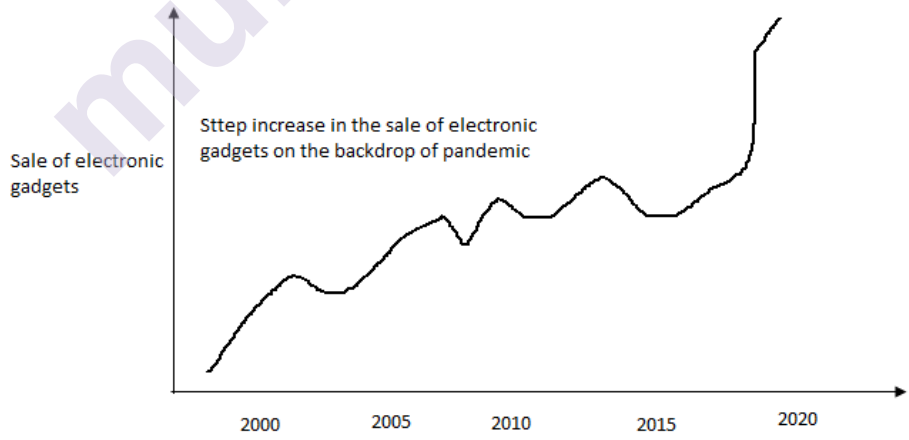


Figure 4.4: Irregularity component of time series

There are certain situation when data is not changing with respect to time, then time series analysis is not applicable to such situations. E.g. If average rainfall over the years in 3-4 decades is approximately same, then it implies that time factor has not affected the rainfall or one can conclude that rainfall is independent of time. There is no point in applying time series analysis to such situations.

B. Types of analysis on time series data

Time series analysis can be categorized into Descriptive, Diagnostic, Predictive and Prescriptive analysis.

Descriptive analysis gives idea about what happened in the past. It helps in interpretation of the patterns followed by the data. It can be represented in the form of data visualizations like graphs, charts, dashboards etc. Variations in the data can be tracked with the help of descriptive analysis.

Diagnostic analysis is like an extension of descriptive analysis, which helps in answering the reasoning behind variations in the data. This is often referred to as root-cause analysis. Techniques like data discovery, data mining and drilling down data come handy for this purpose

Predictive analysis tries to generate a model based on the historical data. The model understands the basic pattern and trends of the data. The same model is then applied to predict for the future. E.g. based on the sale of apartments in a city for last 50 years, a model can help predict the same for next 5 years.

Prescriptive analysis takes predictive data, a step higher and helps to decide what action should be taken. E.g. If certain number of demand is predicted for next year for electric vehicles, then accordingly production planning can be prepared by a company.

It is a prerequisite for any time series forecasting that the data is stationary. If components like trends, seasonality, cyclicity and irregularity are present in the data, it is considered as non-stationary. It is necessary to smoothen the data before it is used for further forecasting. Mean, variance and covariance values help deciding whether the data is stationary or non-stationary. Stationary data may have seasonality component but not the trend component and mean, variance and covariance should not change as per time. To illustrate on non-stationary data further, consider plotting blood pressure against time. It may have minor seasonal variation but definitely no trend. It will never continuously increase or decrease with time. Plotted in the form of a graph, blood pressure values will look as a flat line with no slope. In some medical conditions, there can be irregularities as well, there can be sudden spikes in blood pressed and medical practitioners are definitely interested to find out the root cause behind such spikes and remove them.

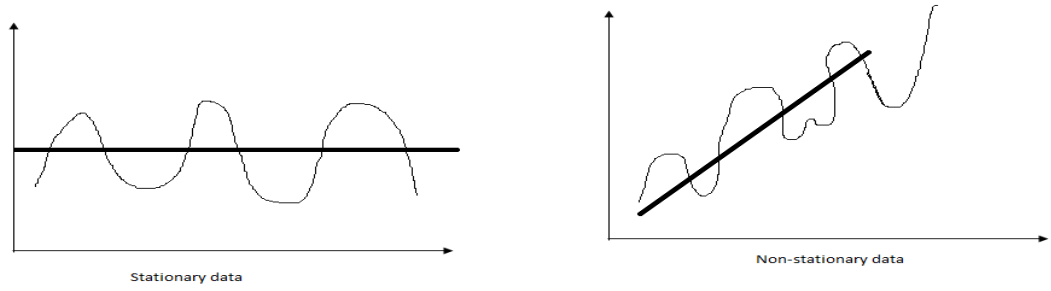


Figure 4.5: Stationary Vs Non-stationary data

Smoothing of data (i.e. converting non-stationary data to stationary) can be achieved by applying **moving average** to the data. Moving average technique removes the randomness in the data. Consider the figure 4.6, the graph represents monthly sales figures for 3 consecutive years. Though there is overall increase in sales, there are variations in between.

Year / Month	Sales
1-01	266
1-02	145.9
1-03	183.1
1-04	119.3
1-05	180.3
1-06	168.5
1-07	231.8
1-08	224.5
1-09	192.8
1-10	122.9
1-11	336.5
1-12	185.9
2-01	194.3
2-02	149.5
2-03	210.1
2-04	273.3
2-05	191.4
2-06	287

Year / Month	Sales
2-07	226
2-08	303.6
2-09	289.9
2-10	421.6
2-11	264.5
2-12	342.3
3-01	339.7
3-02	440.4
3-03	315.9
3-04	439.3
3-05	401.3
3-06	437.4
3-07	575.5
3-08	407.6
3-09	682
3-10	475.3
3-11	581.3
3-12	646.9

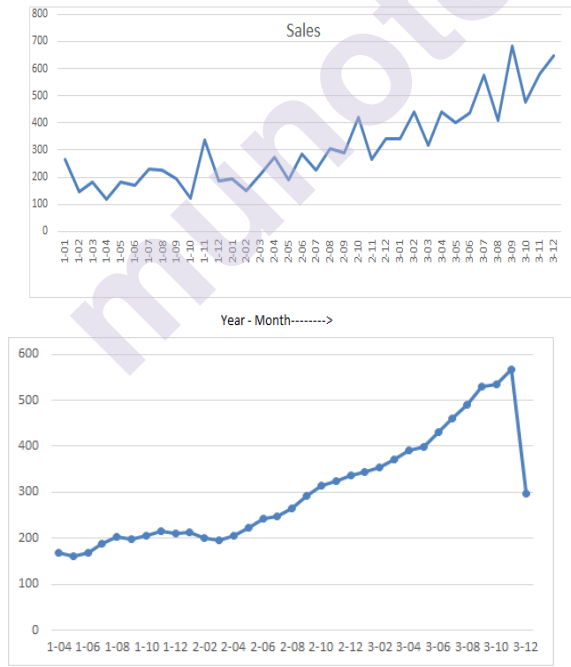


Figure 4.6: Monthly sales figures (Stationary and non-stationary)

After applying moving average – MV4 (Take of first 4 data values and calculate average, then take 2nd, 3rd, 4th and 5th data values to calculate average, then 3rd to 6th and so on.). Next, calculate centered moving average of every 2 data values to further smoothen the rough edges. Plot the line graph of Centered Moving average instead of actual data values), against the time frequencies for data collection i.e. every month of 3 years.

The graph will then look as shown in figure 4.6. Except the last part, the graph is much smoother. Decomposition procedure helps in understanding trend and seasonality factors in time series. De-trending and removing seasonal effect followed by step to identify irregularity causing factors in the original data can prepare data for applying models for forecasting.

Next important task is to forecast based on historical non-stationary data. Certain tools with programming languages like R, Python can also be used for forecasting purpose. Or mathematical models also can be used for this purpose, 2 such models are widely used and they are

a) Additive model:

X_t = Trend + Seasonal + Irregular

In a party, a cook assumes that on an average, people will eat 2 rotis and accordingly will prepare the food. But, if some people are hungry, may be they will eat one extra roti. So, one who east 1 roti normally, will eat 2. One who eats 2 in normal situation, will eat 3. This is 1 extra to normal situation, irrespective of what original number is. In such a case, additive model is used for forecasting.

b) Multiplicative model:

X_t = Trend * Seasonal * Irregular

When there is increase in product prices, it is in percentage terms. E.g. Price of laptop increases by 5% than previous year, cost of certain model laptop which costed 50,000 Rs. previous year will now cost 52,500 Rs. The one which costed 70,000 Rs. previous year will now cost 73,500 Rs. So, the increase in cost is not fixed but in terms of percentage, and such scenarios, multiplicative model is best suited.

So, we can summarize that additive model is useful when the seasonal variation is relatively constant over time.

The multiplicative model is useful when the seasonal variation increases over time.

In the additive model, we take the addition of trend, seasonal and irregular factor and divide it by centered moving average.

Exponential smoothing is a feature available in Excel worksheet, which takes care of this entire process. After applying exponential smoothing, the graph will show actual as well as predicted values of sales, which can be further extrapolated for forecasting. As we can observe, actual and forecasted values are pretty matching with each other. Hence, we can say that this model is accurate and can be used for forecasting future sales values. Other data smoothing techniques like random walk, simple exponential smoothing are also available. Once smoothing is done, we need to right click on the line chart and add equation, R^2 value on the graph. Also after adding trend line on the graph, one can forecast for the future. R^2 helps in indicating how good the model is for prediction.

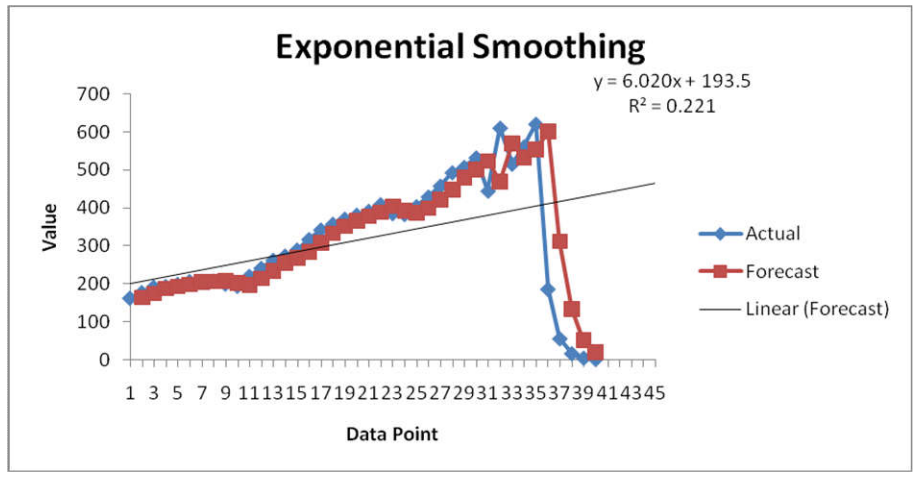


Figure 4.7: Sales values (actual vs. predicted) along with forecast line.

1.4.2 Nonlinear dynamics:

In case of non-linear data, data points are connected to each other in multiple ways. As shown in figure 4.7, the number of data point and degree of connections with each other may vary. Further, elements can also be heterogeneous in nature. This also called as **topology** of the system. Consider a pendulum moving with certain initial state and velocity it will follow certain pattern of movements.

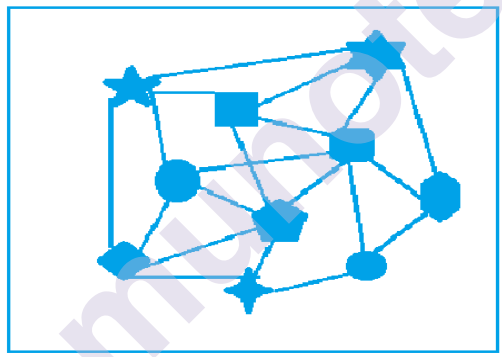


Figure 4.8: Non-linear data

But, if the pendulum is bent in between, it will be controlled by 2 equilibriums.The resultant motion will become non-linear. Hypothetically, imagine the earth is also controlled by another plant, which will exert its own gravitational force effect on the earth, the entire structure of earth's orbit will change and may look like 2 connected ovals. This will again result into non-linear motion. If data for non-linear motion is plotted as graph, instead of sequential nature of line, it will look curved, more like a quadratic equation. Such a change in movement is called as **chaos**. Chaos theory studies behavior of dynamical systems, sensitive to initial conditions (referred to as butterfly effect). Motion of pendulum with 2 pendulums, recorded in isolation, is predictable. But when combined, reveal non-linear behavior.

Two sound waves, perfectly out of synch with each other, rather than adding with each other, will cancel the effect of both. Many human being

working in tandem, may synergize overall output, much higher than addition of individual outputs. Non-linear systems may shift to whole new regime, even if there is small change in input condition. Such a change is called as **phase-transition.**

For a quick comparison between linear and non-linear time series data, a linear data will reveal a straight line when plotted in graph, whereas non-linear data will generated a curved shaped graph. A linear data, when presented in an equation, will be first degree equation whereas non-linear data will be a quadratic equation. It is crucial to find out whether data is linear or non-linear before deciding the techniques to use for forecasting purpose. When represented graphically, non-linear time-series data will generally one of following shapes:

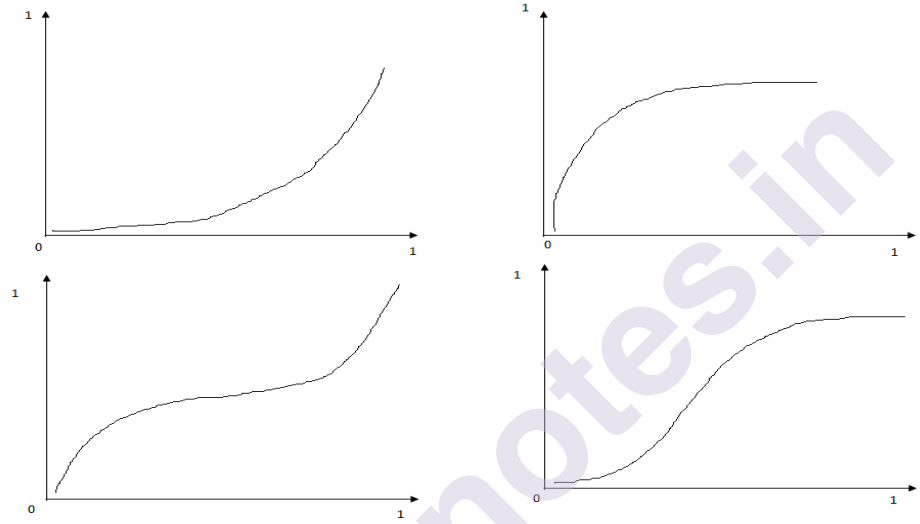


Figure 4.9: Graphical representation of non-linear time series data

Structural breaks (outside forces that may cause sudden and permanent change in the pattern of the data) play vital role while studying non-linear time series data. Identifying the presence of structural breaks, estimating their timings and studying behavior of data before, after and during the breaks needs to be studied while dealing with non-linear data.

Brock- Dechert-Scheinkman test (denoted as the BDS test) is the most widely used test for detecting non-linearity of the data. The BDS test gets its name from its original authors William Brock, Davis Dechert and Jose Scheinkman, who develop it in 1987. It is generally used indirectly to test alternative hypothesis for non-linearity. The BDS test uses the correlation function (also called the correlation integral) as the statistic test. In case of non-linear data, which is time dependent, BDS test checks dependence of data points in the space where point are plotted. Naturally, unlike linear data, there is more than 2 dependence of datapoints in case on non-linear data. This is denoted as checking spatial dependence check.

ARIMA (Auto Regressive Integrated Moving Average) Model is used for forecasting in non-linear time series data. ARIMA model is denoted as ARIMA (p, d, q) where p is The number of lag observations included in

the model, also called the lag order. D is the number of times that the raw observations are differenced, also called the degree of differencing and q is the size of the moving average window, also called the order of moving average. Steps in ARIMA are stated as

1. Model identification. Use plots and summary statistics to identify trends, seasonality, and auto regression elements to get an idea of the amount of differencing and the size of the lag that will be required.

2. Parameter Estimation. Use a fitting procedure to find the coefficients of the regression model.

3. Model Checking. Use plots and statistical tests of the residual errors to determine the amount and type of temporal structure not captured by the model. The process is repeated until either a desirable level of fit is achieved on the in-sample or out-of-sample observations (e.g. training or test datasets).

ARIMA includes both auto regression and moving average features. It needs at least 50 and on an average 100 records to build a proper model. The ARIMA model tend to be unstable, both with respect to changes in observations and changes in model specification. Because of the large data requirements, the lack of convenient updating procedure, ARIMA becomes high cost model.

1.4.3 Rule induction:

Rule induction is a process of deriving if-then rules as a part of data mining. Rules are most popular symbolic representation of knowledge. Rules are not only very easy but also natural and in human understandable form. Such decision rules help in discovering inherent relationships amongst the data sets as well as use them for business. Consider an example – If it is 8 pm on Saturday, then there will be lot of rush in the restaurants. Predictions based on such rules are based on everyday observation for long duration. Rules are easier to understand than decision trees. Consider a scenario which has more than 30-35 decision situations. A decision tree built based on such decision points will not only be a very large diagram but will be difficult to understand as well. Hence, decision rules are more preferred over decision trees or any other technique for classification.

Such rules can be extracted from a decision tree. Rules consist of attribute – value pairs which can be traced from a root of a decision tree to a particular node. These rules are mutually exclusive (without conflict / overlap) and exhaustive (covering all possible scenarios of decision making).

For deciding income tax to be paid by a person, following rules can be followed (The given example is totally hypothetical and for academic purpose only).

If a person is a senior citizen and earning in slab 1	Then No income tax
If a person is a senior citizen and earning in slab 1	Then 5% income tax
If a person is salaried, not a senior citizen and earning in slab 1 and gender -Male	Then 5% income tax
If a person is salaried, not a senior citizen and earning in slab 2 and gender -Male	Then 10% income tax
If a person is salaried , not a senior citizen and earning in slab 1 and gender -Female	Then No income tax
If a person is salaried , not a senior citizen and earning in slab 2 and gender -Female	Then 5% income tax
If a person is business person , not a senior citizen and earning in slab 1 and gender -Male	Then 15% income tax
If a person is business person , not a senior citizen and earning in slab 1 and gender -Female	Then 10% income tax

a. Rule Induction algorithms:

Apart from inducing rules from decision trees, certain algorithms can also be used for rule induction process. Training data can be used for deriving rules. Generally one rule is learnt by using the process of machine learning. For more number of rules, iterations are carried out on the dataset for every new rule.

i. Learn one rule:

This rule follows greedy search technique where it searches for a rule which has high accuracy but less coverage classifying all positive examples for a given instance. Strength of this algorithm lies in its ability to create relations amongst the given attributes under test and cover maximum number of dataset for these attributes. Consider a situation where in a decision of playing cricket match is based on certain parameters such as weather, rains, cloudiness, light intensity, temperature, nature of grass on the playground and soil quality. Based on possible alternatives to all these parameters, final ruleset is designed.

E.g. Rule number 1 can be - If quality of soil is good, and no grass on the ground, and light intensity is good, and no cloudiness and no rains, match will be played.

Another rule can be - If heavy rains, even if no grass, soil quality is good, match will be played.

ii. Sequential covering:

This is widely used algorithm for rule based classification for learning disjunction rules. In this algorithm, based on learn one rule, one rule is discovered. After that all the data covered by this rule is removed. Then the same process is repeated in a sequential manner for all other rules.

iii. FOIL:

First Order Inductive Learning is a rule based algorithm which is a natural extension of Sequential Covering algorithm and Learn One rule algorithms. FOIL used the concept of inductive logic which involves analyzing and understanding evidences and then use them for prediction. Look at the example, wherein the evidence say 80% of youth go for movies on weekends, and the fact that A is a youth, one can predict that this person will go and watch movie on a weekend. The algorithm works in iteration forming new rules, and for every new rule, all previous positive and negative examples are eliminated.

iv. AQ:

Algorithm Quasi Optimal is a powerful machine learning methodology aimed at learning symbolic decision rules from a set of examples and counterexamples (negative examples). AQ starts with assigning class (labels) to input data. So it can be treated as supervised algorithm. AQ involve 4 major steps – data preparation, rule learning, postprocessing and optional testing.AQ is used in two ways, for theory formation (TF) and Pattern Discovery (PD). AQ segregates all ambiguous data/ event into 4 categories – Positive, where all ambiguous data is gathered into a class. Negative, where ambiguous data is eliminated. Eliminate, where ambiguous data is not used further. Majority, where ambiguous data is labeled to a class where it mostly appears. Further, the algorithm selects only most relevant attributes. This avoid unnecessary rule formation in a highly noisy situation. In the beginning, a general rule is formed by comparing with positive and negative examples, and keep repeating this process by refining previous rules.

v. CN2:

CN2 algorithm works best in a noisy environment. It is a classification technique for inducing simple if-then rules to predict a class to which data related to an event belongs to. There is inbuilt process for removing empty columns, removing instances with unknown target values and imputing missing values with mean values. Two algorithms, search algorithm (decides which are the best rules) and control algorithm (exerts criteria for deciding best rules) which are part of CN2, work in tandem to induce rules, in an ordered and unordered set.

vi. RIPPER:

It stands for **R**epeated **I**ncremental **P**runing to **P**roduce **E**rror **R**eduction. The Ripper Algorithm is a Rule-based **classification algorithm**. It derives

a set of rules from the training set. It is a widely used rule induction algorithm. RIPPER algorithm is used when the dataset is imbalanced one (Unequal number of data elements in different classes). Amongst imbalanced datasets, this algorithm selects the majority class as a default class. The algorithm starts with the assumption that records belonging to default class are positive example and all other classes with reducing frequenting of data elements are considered as negative examples. **Sequential Covering Algorithm** is used to generate the rules that discriminate between +ve and -ve examples. Then RIPPER considers next class for deriving the rules. It starts with empty rules and then keeps adding best conjunct (conditions connected by AND) to the antecedents (If part). All such conjuncts are evaluated by a metric. When the rule starts covering negative examples, the algorithm stop execution.

Once a rule is derived, all positive and negative examples are covered by a rule are eliminated and the rule is added to **rule set.**

Accuracy of such rule induction system can be calculated based on number of correct data elements covered by a rule and number of total number of data elements covered by a rule. It is possible that there are more than one rules are applied for uncovering such hidden relationships in the dataset. In such a case, prioritization of rules depending on the requirement is carried out. Such prioritization of rules will avoid conflict while triggering the rules.

b. Conflict resolution techniques:

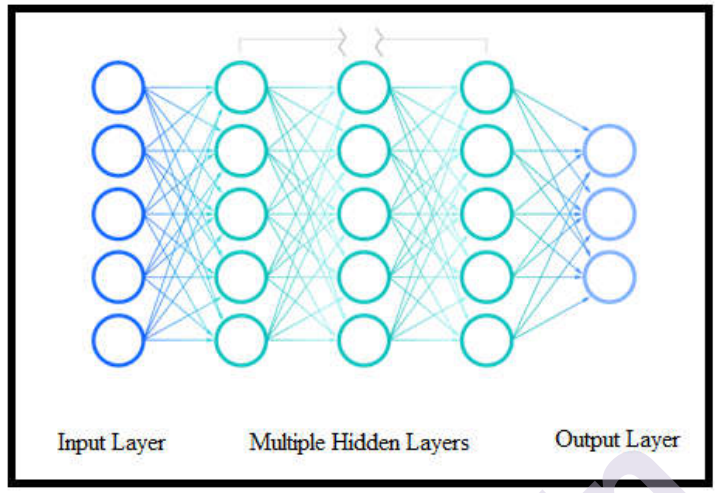
To avoid multiple rules being triggered at the same time, or conflict between rules and class which it belongs to, following conflict resolution techniques are used.

i. Size ordering – In this technique, the rules with maximum number of attributes is given the highest priority.
ii. Class based ordering – Rules with maximum frequency class is considered at the priority.
iii. Rule based ordering – Rules are arranged into a long list of priority based on some measure of rule qualities such as accuracy, coverage and experts' opinion.

1.5. NEURAL NETWORKS:

A neural network is a computational data model which captures and represents complex input & output mechanism. The main motives come for the development of neural network technology is from the thought to develop an artificial system which can perform "intelligent" tasks similar to human brain. NN (Neural networks) reflect the behavior of the human brain. It allows computer programs to recognize patterns and solve common Artificial intelligence problems. NN is also known as Artificial Neural Networks (ANNs). NNs are having many layers, which mainly divide in three categories like an input layer, one or more hidden layers, and an output layer. Node is also known as artificial neuron. Each node connects to another node and each node has an associated weightage and

specific threshold. If the output of any specific node is above the threshold value, then that specific node is activated and it sends data to the next layer of the network.



NNs rely on training data to learn and continuously improve their accuracy over time. For getting the better accuracy there is need to tune up the learning algorithms. Tasks in speech recognition and image recognition can be completed within minutes when it takes several hours in the manual human expert's identification. Google's search algorithm is one of the most well known examples of NNs. Face recognition or character recognition is not the only the problems that NNs can solve. NNs have been successfully applied to wide spectrum of data-intensive applications like:

Fraud Detection - Detect fake transactions of credit card and automatically refuse such charges.

Process Modeling and Controlling - Creating a NN model for a physical plant for best automation.

Machine Diagnostics - Detect the failure of machine and automatically shut down the machine systems when this problem occurs.

Targeted Marketing and survey – For getting highest response rate for a particular marketing campaign.

Quality Control and Maintenance – Identifying the product defects based on the recorded data.

Portfolio Management - Allocate the assets in a portfolio in for maximum return with minimum risk.

Medical Diagnosis Application - Help doctors with their diagnosis by analyzing the image data such as MRIs & X-rays.

Financial Forecasting & Credit Rating – Do the financial forecasting with the available data also calculate the credit rating based on current financial conditions.

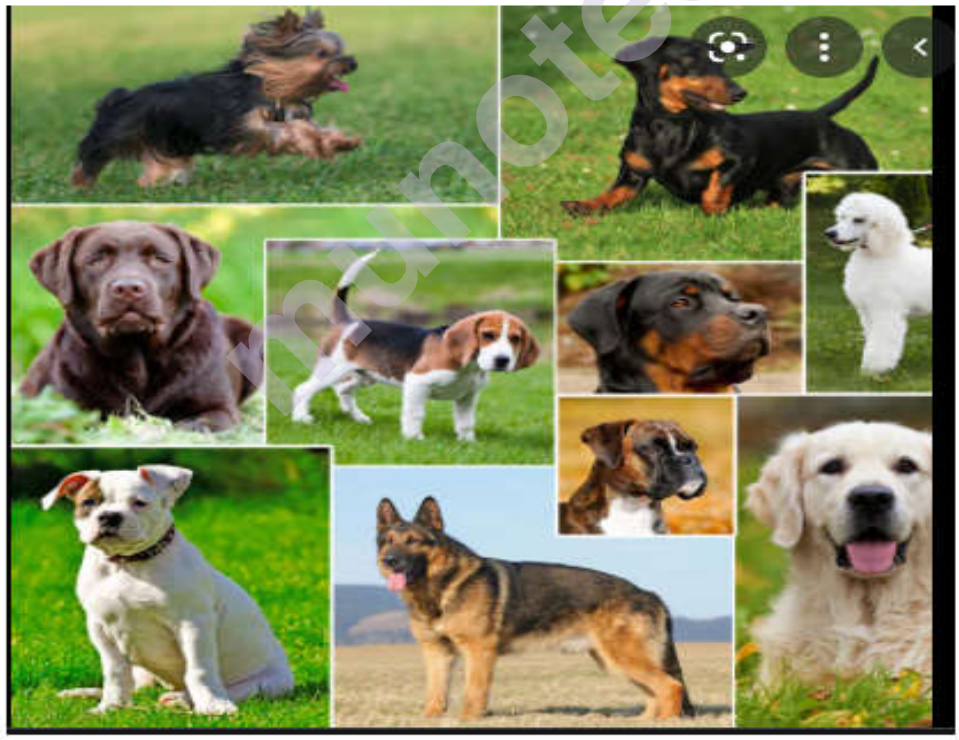
Military Application -Target Recognition - Determine target if any enemy present in given data.

1.5.1 Learning and Generalization:

First step in NNs training is **generalization. Generalization** specifies how good our model is for learning from the provided data and applying the learnt information. When we train a NN, some data we will use for train the model and some we will reserve for checking the performance of model. Here we are explaining generalization of NN with an example.

We are training a NN which should give the decision about given image is of dog or not. We have some pictures of dogs, each dog belonging to a certain breed and having different features like color, strips, height and many more. We have a total 12 pictures of dog. We will use 10 pictures for training and remaining 2 for checking the accuracy of model.

Now we will show this to a person and train them with 10 breeds of dogs and after training ask person to detect other dogs from testing data. Hopefully person will give answer about asked question. Here 10 breeds should be enough to understand and identify the unique features of a dog. This concept of learning is called **generalization in which Learning** from some data and **correctly** applying the gained knowledge on other data



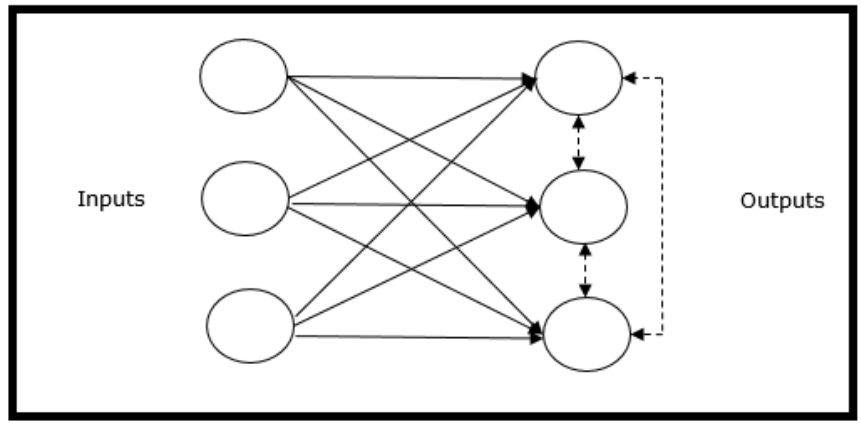
Training Set

Testing Set

1.5.2 Competitive Learning:

Competitive learning is a specific form of unsupervised learning in NNs. This type of learning is done without any supervision of a teacher. This is independent learning process. At the time of training of NN under unsupervised learning, the similar input vectors combined and form a cluster. In this system when a new input pattern is applied, then the NN gives a response indicating the class to which input pattern belongs. There is a no any feedback from the environment as to what should be the desired output and whether the generated result is correct or incorrect. In this type of learning the network and discover the patterns. This is based on the concept of Competitive Learning Network.

Competitive Network is like a single layer feed-forward network having feedback connection between the outputs. The connections between the outputs are inhibitory type, which is shown by dotted lines, which means the competitors never support themselves. Here the competition done between the output nodes specifically during the training. Output node unit which has the highest activation to a given input pattern will be declared the winner node. During training, the output unit that provides the highest activation to a given input pattern is declared the specific weights of the winner and is moved closer to the input pattern; whereas the rest of the neurons are remain unchanged. In this strategy winner-take-all and only the winning neuron is updated other remain as it is.



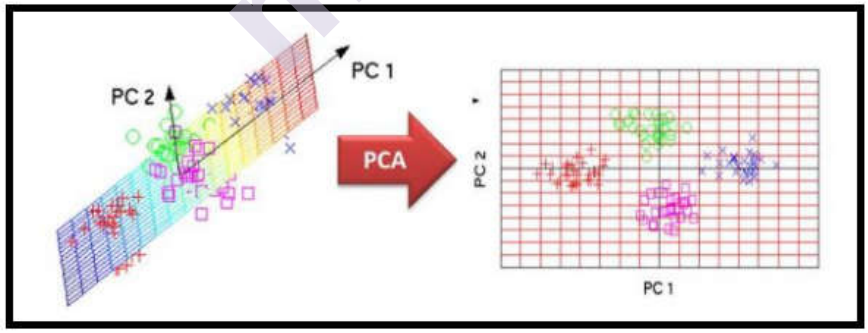
1.5.3 Principal Component Analysis and Neural Networks:

Principal Component Analysis (PCA) is an unsupervised learning methodology which is generally used to reduce the dimensionality of large datasets or generally use to simplify the complexity of dataset by transforming a large set of variables into a smaller one while trying to retain most of the information of the original dataset.PCA reduces data by geometrically projecting it onto lower dimensions which in turn are called as Principal Components (PC).

The purpose of this method is to find the best summary of our data by using the least amount of principal components. By choosing principal components we minimizing our distance between the original data and its projected values on the principal components, as a result of minimizing the distance we maximize the variance of the projected points, same we can repeat for all other principal components.

The basic idea of PCA is to preserve maximal variance for a data set with a minimal set of linear descriptors. High dimensional datasets are projected into a smaller number of dimensions maximizing the variance on the new axes. PCA is a very important Statistical analysis tool and therefore many researchers are working to improve the algorithm for better performance and better data interpretation.

Let's take an example, if we have a training set consisting of 250 images of "person wearing glasses" and "person not wearing glasses" having 4096 features per image, when we directly apply NN to our dataset it would take a huge amount of time for the training purpose, but if we pre-process our data using PCA it will reduce the dimensions of our dataset to (250,250) from the original (250,4096) hence when we apply NN to our resulting dataset the time required to train the dataset will reduce drastically without a huge loss in accuracy.



Principal Components Analysis

1.6. FUZZY LOGIC:

The term "Fuzzy Logic" refers to the data which is imprecise or vague. This concept was first introduced in 1965 by Lotfi A Zadeh, A Barkley Professor in Electronics and computer Science, who was basically a Mathematician. He is also called as "Father of Fuzzy Logic". He realized

that the legacy application were not capable of handling imprecise data and mainly focused on handling precise (Boolean) data such as True / False in the form of 1 and 0. In real world, more often one has to process unclear data than clear data.

To further elaborate, consider a question "Is the car is moving?. "We have only two answers – YES or NO. It's pretty simple to handle such situations and transform them in to software systems. But imagine a situation where we think of developing an autopilot application for a car. Software is expected to handle a decision making situation wherein a decision needs to be made about applying brakes or pushing accelerator based on the existing speed of the car, whether it is moving slow or fast. Let's assume, 40 km per hour is a threshold, below which car is considered to have slow speed and above it, considered to be fast. If a car is moving at the speed of 10 km/hr. is definitely slow speed and pushing accelerator will be appropriate decision. On the other hand, if a car is moving at the speed of 60 km/hr is definitely moving with fast speed and applying brakes will be advisable. But think of a situation where the speed is 39.5 km/hr. as per traditional logic, accelerators should be pushed and as soon as speed becomes 40.5 /hr., brakes should be applied. In this way, a car will keep speeding up and suddenly stopping. The person inside the car will keep experiencing continuous jerks.

The only solution to handle such a situation was to consider speed of the car as a continuous imprecise data than fixed precise. Slow speed can be anywhere between 0 to 40, depending on how close it is to the threshold value, we can say that it is extremely slow, very slow, little slow, slow, little fast, very fast or extremely fast. Fuzzy logic helps in accepting such continuous data and further take actions based on such input.

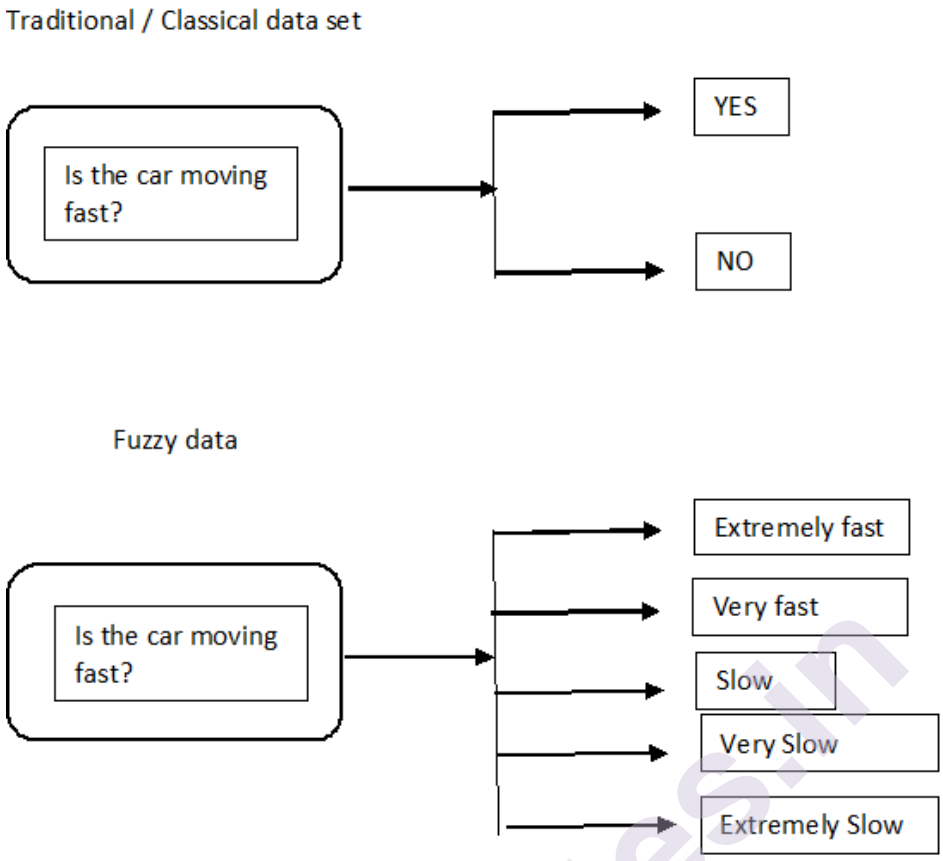


Figure 6.1 Extracting fuzzy models (rules) from data

a. Architecture of fuzzy logic based software systems

A software system based on fuzzy logic mainly has 4 components: Fuzzifier module, a rule base, Inference engine and De-Fuzzifier module.

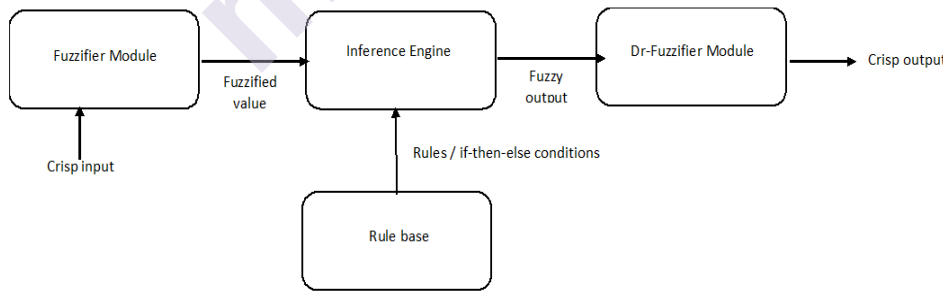


Figure 6.2: Architecture of Fuzzy Logic System

1. Fuzzifier module: A fuzzifier module accepts inputs in the form of crisp values. These values are further converted fuzzy data by applying membership function. E.g. consider an answer to a question – Is it hot today? The respondent, depending upon his / her perception, may answer differently- Very hot / extremely hot / hot / slightly hot / Not at all hot. Instead of plain YES or NO binary answer, there are varied answers. Such

answers are treated as LP, MP, SM, MN, and LN. Such an input is more like human like and realistic one.

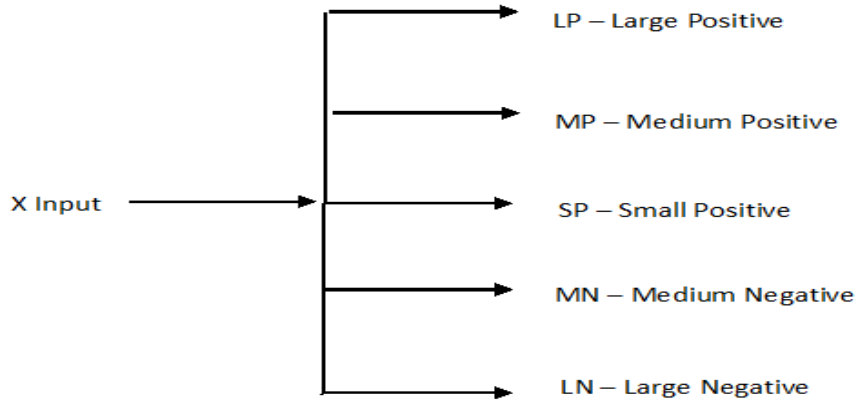


Figure 6.3: Fuzzification rules

2. Rule base: This is a collection of set of rules, which are applied on the fuzzy input received from fuzzifier module. The rules are in the form of if-then conditions and respective action, designed by experts. Such set of rules can be further updated to fine tune the system.

3. Inference engine: This is the most important component of the system. Based on inputs received from Fuzzifier module and rules base, inference engine is responsible for making decisions. After matching fuzzy inputs and selecting appropriate rules, the inference engine determines which rules to be applied for developing control actions.

4. De-Fuzzification module: This is responsible for output from inference engine to crisp values and present to user. Further, user can choose the best option to reduce the error.

De-Fuzzification methods – Lambda-Cut method, maxima method, weighted-sum method and centroid method are the methods which are used for converting fuzzy values to crisp values which are in the human understandable form.

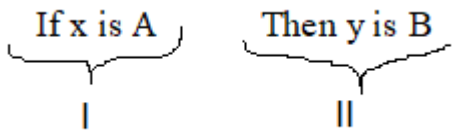
Let's consider the illustration of designing fuzzy logic system for a smart air conditioner. The system can detect temperature through a thermometer. This crisp value is taken as input for fuzzy system. Fuzzifier modules, using membership function, will convert it into fuzzy data set. These fuzzy values, combined with if-then rules base, inference engine will generate output, which is again fuzzy. Using defuzzifying techniques, output will be again converted into crisp value, based on which air conditioner will automatically adjust its value.

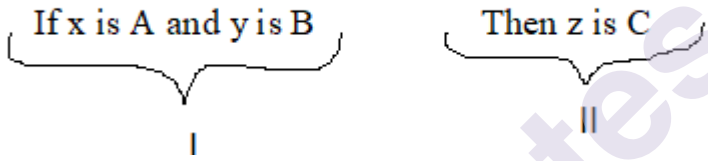
A fuzzy logic based system is one which can treat the input as a set of limited approximate values instead of precise values. All the values are

nothing but matter of degrees. Knowledge is nothing but a set of variables. Any logical system can be converted into fuzzy logic based system.

b. Membership function:

A membership function is one which can help in transforming crisp values to fuzzy sets. It was first put forth by Lotfi A Zadeh. Such a function helps in representing all the data in fuzzy set (discrete and continuous both). It helps in handling real world problems with the help of experts. It is possible to have one or many fuzzy rules with one or many antecedents and consequents. Consider Following If-Then rule, part I is called antecedent or premise and part II is called as consequence. In the following case there is only 1 antecedent and 1 consequents in the rule.





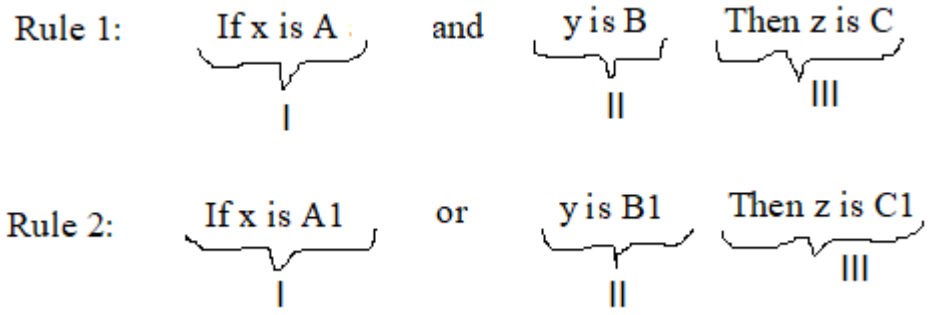
In the above example, there are 2 antecedents and 1 consequents in the rule.

Rules for defining fuzzy values are also fuzzy. In the similar way, it is possible to have multiple antecedent and multiple rules.

Here is an example with multiple rules with multiple antecedents.

Rule 1: If x is A and y is B Then z is C

Rule 2: If x is A1 or y is B1 Then z is C1.



Part I and part II in the above rules indicate antecedent I and II whereas III indicates consequents. Consequents of multiple rules in a rule base can be aggregated to generate defuzzified output.

The method of assigning membership values are as follows: 1. Intuition 2. Inference 3. Rank Ordering 4. Angular Fuzzy Sets 5. Neural Networks 6. Genetic Algorithm 7. Inductive Reasoning

One can represent a membership function with the help of a graph. **A membership value will always range between 0 and 1.**

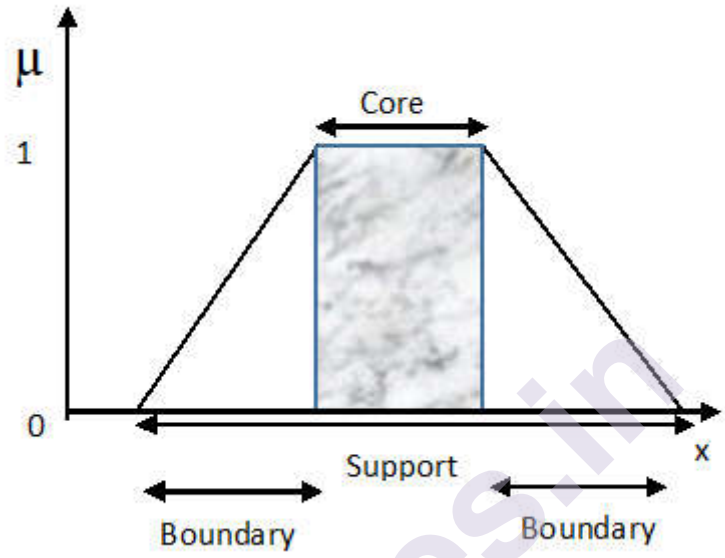


Figure 6.4: Graphical representation of membership function

Membership values lie between 0 and 1 and the one which are equal to full membership i.e. 1 are called core values. The values which are non-zero membership values are called support values. All the values which are greater than zero but have incomplete membership are called as boundary values. Membership values can be assigned based on intuition of experts, through referencing, by rank ordering, angular fuzzy sets, neural networks, genetic algorithm, and induction reasoning.

1.6.1 Extracting Fuzzy Models from Data

A Fuzzy rule consists of antecedent (also called as hypothesis), consequence (also known as conclusion). Multiple antecedents are possible in a rule and there can be many rules in a given scenario. Such an expression with antecedents and consequences, with optional AND, OR conjunctive/disjunctive operators If-Then rule. These are also called as **canonical form of rule base**.

When the two antecedents are conjunctive in nature i.e. joined by **AND** then the aggregated output **is intersection** of all membership values. In this case, all conditions that should be jointly satisfied, joined with AND. This is called as Conjunctive system of rules. It can be represented mathematically as

$\mu_x(x) = \min(\mu_{x1}(x_1), \mu_{x2}(x_2) \ldots\ldots \mu_{xn}(x_n))$

On the other hand, if antecedents are disjunctive in nature, i.e. joined by **OR**, then aggregated output is **union** of all membership values. In this case, at least one conditions that should be satisfied, joined with OR. This is called as Disjunctive system of rules. It can be mathematically represented as

$\mu_x(x) = \max(\mu_{x1}(x_1), \mu_{x2}(x_2) \ldots\ldots \mu_{xn}(x_n))$

There are well researched fuzzy methods that provide well defined systems for which can be used in inference system.

A. Mamdani system

Ebhasim Mamdani suggested this method in the year 1975. This method can accept crisp as well as fuzzy inputs for the purpose of inference. Consider a set of 2 rules

Rule 1 - If x is A and y is B Then z is C
Rule 2 - If x is A1 or y is B1 Then z is C1

There are two cases for 2 inputs methods in Mamdani system

a. **Max-Min inference method** – Considering above Rule 1 and Rule 2, with x=2.5 and y=3 as the inputs, minimum of membership values for different antecedent is considered. Let μ1 be the membership value for x=2.5 and μ2 be the membership value for y=3. Then minimum of μ1 and μ2 is considered for Rule 1. Same procedure is followed for Rule 2 (and up to Rule n if there are any) and maximum μ of all these rules is considered for final Defuzzification. This method is also called as **truncated membership method**.

Area covered under marked area is considered for finding out the final crisp value. Appropriate equation for area calculation is used based on the shape that is formed in the final output graph.

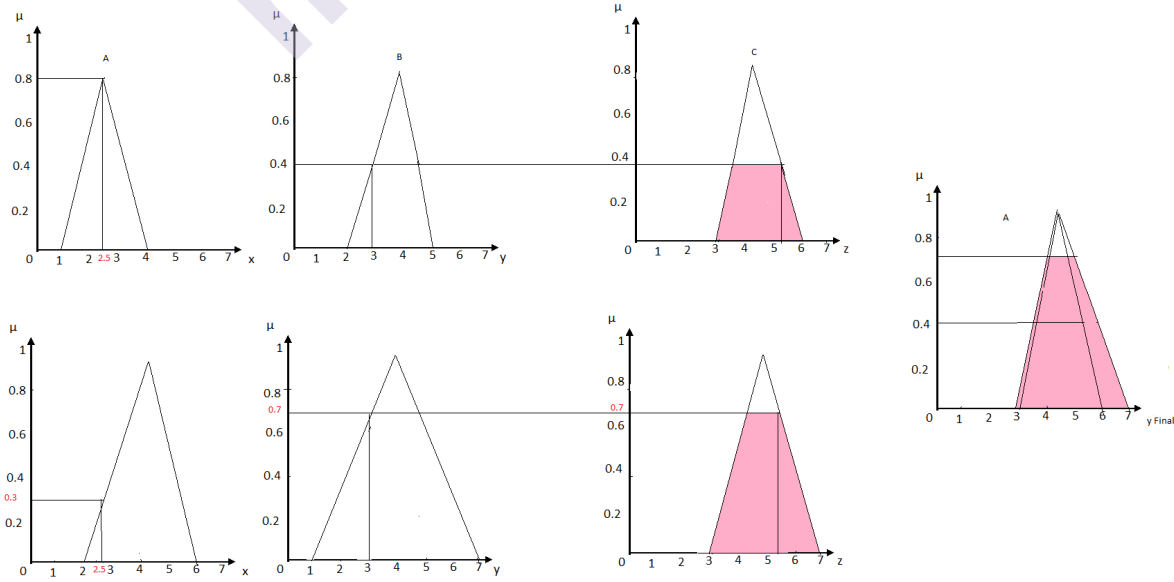


Figure 6.5: Max- Min Mamdani Method

b. **Max-product inference method** – Same procedure as Max-Min is followed, except, instead of considering minimum membership value for each rule, product is considered. For aggregation, maximum of all these products considered for final Defuzzification. Instead of truncated membership method, **scaled membership method** is used in Max-Product Method.

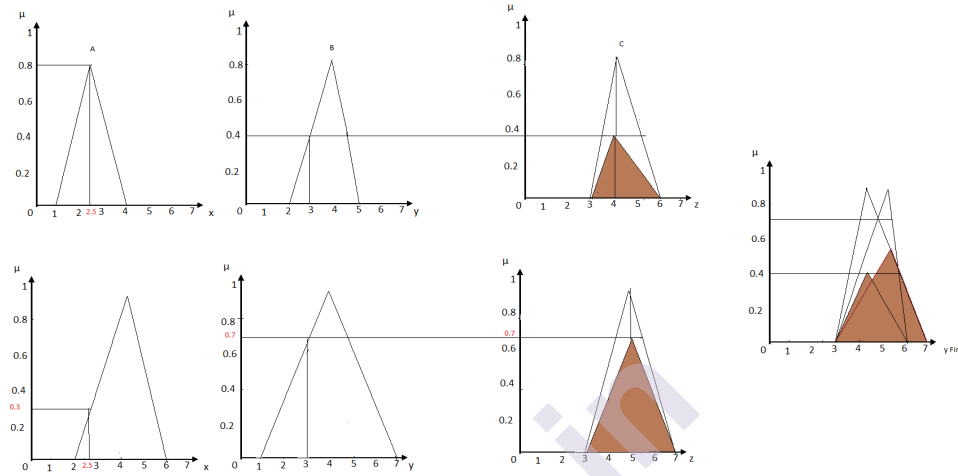


Figure 6.6: Max Product Mamdani Method

B. TSK / Sugeno system (Takagi Sugeno Kang / Sugeno)

This model was suggested in the year 1985. In case of Mamdani system, all the antecedents in If Then rule were in the fuzzy form and consequent is also fuzzy. But the consequent is a polynomial function represented as y=f(x, y), which is a crisp function.

Rule 1 - If x is small and y is small Then z1= (-x) +y+1
Rule 2 - If x is small and y is large Then z2= (-y) +3
Rule 3 - If x is large and y is small Then z3= (-x) +3
Rule 4 - If x is large and y is large Then z4= (-x) +y+2
Consider values of x=1.5 and y=2.5

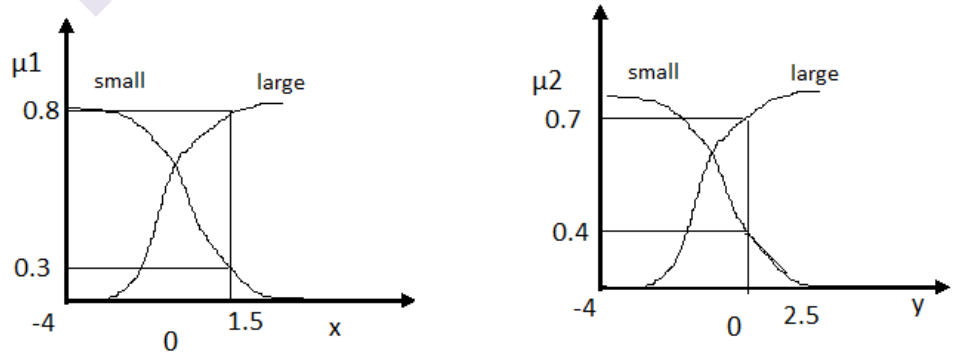


Figure 6.7: Graphical representatiopn of TSK / Sugeno method

Minimum of membership values for x=1.5 for small and large are 0.3 and 0.3 and that of y=2.5 are 0.4 and 0.7 respectively.

$$\text{Final output } y^* = \frac{w3.z \text{ (first case)} + w3.z \text{ (second case)}}{w3(\text{first case}) + w3 \text{ (second case)}}$$

$$y^* = \frac{(0.3 * 2) + (0.3 * 0.5) + (0.4 * 1.5) + (0.7 * 6)}{0.3 + 0.3 + 0.4 + 0.7}$$

y* = 3.264

Before discussing the third system i.e.Tsukamoto system, let's have brief discussion on comparison of Mamdani and Sugeno system.

a. As per Mamdani system, consequent is s fuzzy data set whereas according to Sugeno system, output membership function is a either linear or constant.
b. Sugeno system is more based on mathematical rules than Mamdani.
c. Mamdani system more suitable for human inputs
d. Sugeno controller has more adjustable parameters than Mamdani system.
e. Mamdani system is more intuitive and has widespread acceptance, but Sugeno method is more computationally efficient.
f. Sugeno system works better for optimization and adaptive techniques.

C. Tsukamoto system

In this system, antecedents as well as consequent is a fuzzy set, but the membership function of the consequent is a fuzzy set, based on monotonic function (which is also called shoulder function) whose successive values are increasing, decreasing or constant. Output of each rule is defined as a crisp value induced by membership value coming from antecedent rule.

Rule 1 – If x is A and y is B Then z is C

Rule 2 – If x is A1 and y is B1 Then z1 is C1

w1, w2 and w3 represent corresponding weights (based on membership value) for x, y and z.

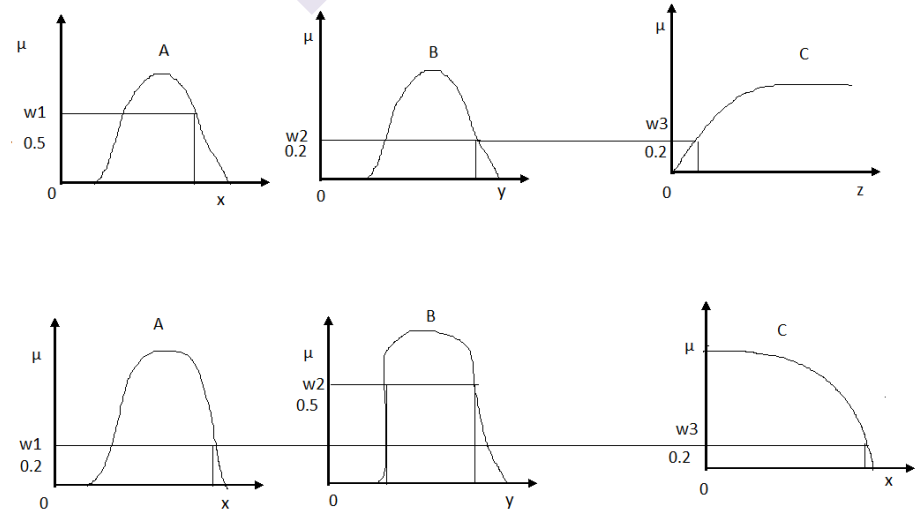


Figure 6.8: Graphical representation of Tsukamoto method

Based on rule 1 and rule 2, we find corresponding values for x and y and the membership value for output z. This is done using maximum or minimum depending on the rule (whether AND/OR condition used in the rule). In a given illustration, both the rules are connected with AND, hence we consider minimum of the two membership values. After extending w2 in first case and w2 in second case, corresponding values for membership value w3 can be obtained. (This is a crisp value). Overall output can be obtained by weighted average of each rule's output (i.e. w3 and z in both the rules).

$$\text{Final output } y^* = \frac{w3.z \text{ (first case)} + w3.z \text{ (second case)}}{w3(\text{first case}) + w3 \text{ (second case)}}$$

Consider values of x=2 and y=5 ,

$$y^* = \frac{(0.2 * 0.5) + (0.5 * 5)}{0.2 + 0.5}$$

y* = 3.714 (Final output)

Main advantage of Tsukamoto method is that it bypasses the long process of Defuzzification as each rule renders a crisp value, and overall output can be calculated with weighted average method.

Major lacuna of this method is that, it can be applied only when monotonic function used. In all other generic case Tsukamoto method cannot be used.

1.6.2 Fuzzy Decision Trees

Decision tree is a diagrammatic representation of decision rules and corresponding outcomes. A decision tree consists of 2 parts, decision node and branches. A decision tree of such kind helps the end user better design strategy in a complex situation where there are multiple decision rules and conditions. Let's consider the example of decision income tax percentage to be deducted in a given situation. Tax to be deducted will be decided upon following conditions

1. Whether a person under consideration is a salaried person or a business person
2. Age of the person
3. Gender of a person
4. Total amount of earning

To design a tree for this situation, Questions are designed in such a way that there are only two possible answers to a question. A condition is considered as a decision node and answers are like branches. All possible conditions and their answers are included in a single tree so that end-user can easily take decision. **It is important to note that all the decision rules and possible answers are clear and well-defined.**

Now, let's consider another situation, where person X has been interviewed by different companies and has received job offers from 4 different companies. X has to make a decision, which offer to accept, based on 3 criteria he has in mind. The criteria r\are salary, distance from home and growth opportunities. X is looking for salary in the range of 35-55 thousand per month. Distance from home should be between 5 to 30 km. Growth opportunities are indicated by number of ticks where more number of ticks indicate more opportunities. Unlike previous example of tax calculation, all the above conditions and possible criteria and unclear and fuzzy in this case.

Following table represents all the criteria and available job opportunities available for person X wherein J1, J2, J3 and J4 indicate job offers and C1, C2, C3 indicate criteria for selection of job offer.

Job offers Criteria	J1	J2	J3	J4
Salary C1	40 k	45 k	50 k	60 k
Distance C2	27	7.5	12	2.5
Growth C3	√√	√√√	√	√

After assigning membership function and assigning weights for salary criteria, we get a continuous line as shown below.

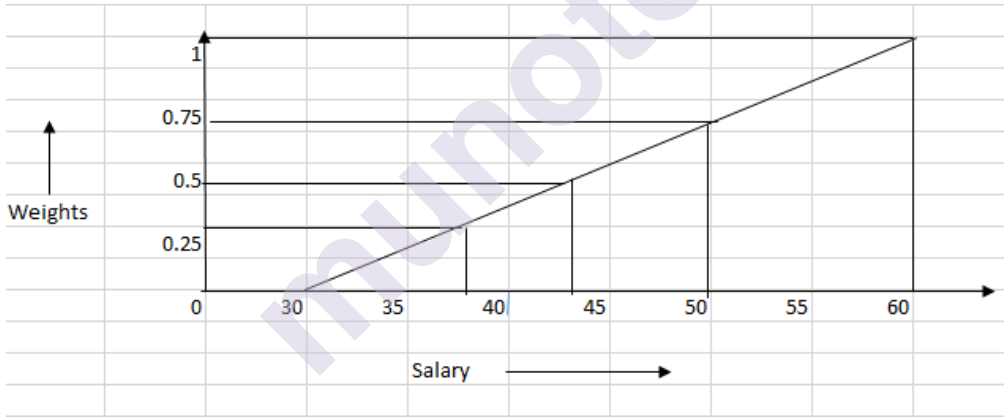


Figure 6.9: Graphical presentation of salary vs. membership values

After following similar procedure for all other criteria, we get a table as follows

Job offers Criteria	J1	J2	J3	J4
Salary C1	0.25	0.5	0.7	1
Distance C2	1	0.9	0.78	0.1
Growth C3	0.5	0.8	0.2	0.2

For finding out best possible option, following equation used:

D1=Min (C1 (J1), C1 (J2), C1 (J3), C1 (J4)) = 0.25

D2=Min (C2 (J1), C2 (J2), C2 (J3), C2 (J4)) = 0.5

D3=Min (C3 (J1), C3 (J2), C3 (J3), C3 (J4)) = 0.2

D4=Min (C4 (J1), C1 (J2), C1 (J3), C1 (J4)) = 0.1

D (Final) = Max (D1, D2, D3, D4) = Max (0.25, 0.5, 0.2, 0.1) = 0.5

Hence, best option is with the weight 0.5 i.e. option 2. Hence J2, job offer 2 is most advisable for X.

A decision tree based on fuzzy value will not have just 2 branches, but can have multiple branches. Experts' views do matter for designing weights giver to the values and membership function.

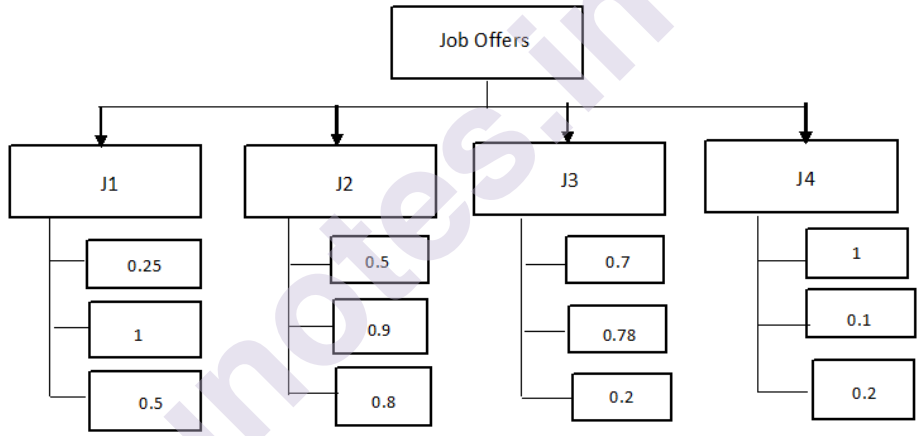


Figure 6.10: Decision tree (fuzzy values)

6.3 Stochastic Search Methods.

Since the advent of computers and software systems, they have undergone lot of evolution. In the recent days, software systems have reached a stage where one can expect them to imitate human intelligence. Needless to say, agility and adaptability is one of the most prominent feature of human intelligence. Incorporating changing environment to support decision making in most complex systems, machine learning, deep learning and neural networks have immensely aided in the development of appropriate artificially intelligent software.

An efficient adaptive, self-learning algorithm for speedy search in a large size database can give and edge over other traditional search algorithms.

Previously used deterministic and probabilistic models may not give expected intelligent output, nearer to human intelligence. Deterministic models as experiment based and with same set of initial conditions, will generate same output. Probabilistic works with certain degree of

randomness, but fails to work in an inherent highly random environment. Further, deterministic and probabilistic methods are not capable of handling time-dependent randomness. Consider an example of bacterial growth in a controlled environment. In spite of same set of initial condition and environment, final results may vary. Predicting stock prices at different points of time is also highly unpredictable process and asks for algorithms that can handle the nature of randomness in such a situation. Modeling efficient supply chain management from production facilities to warehouse, designing best red-yellow-green signal timings in various directions in a traffic-network, deciding time to administer a drug for its best therapeutic effects, Gaussian movement of particles are some more such areas with high degree of randomness. Stochastic methods can come handy in such situations.

What is stochastic search? : Most of the real-world problems need stochastic approach. Stochastic process is a set of random variables, which are time-dependent (time can be discrete-X_0, X_1, X_2, X_3… X_n or continuous – $\{X_t\}$ t>=0). Certain degree of uncertainty helps in improving ability in optimizing search processes. Natural world is full of stochasticity. Most of the machine learning algorithms are based on stochastic methods. Games do have certain level of stochasticity, such as rolling dice or shuffling cards. Following are some generic steps for building stochastic search model:

1. Creating a sample space (Ω) — which includes a list of all possible outcomes,
2. Assigning probabilities to all the elements in a sample space
3. Identifying different events of interest,
4. Calculating the probabilities for the events of interest.

Let's see a common example of this process in action: You are rolling a dice in a casino. If you roll a six or a one, you win Rs. 1000. The steps would be:

The sample space includes all possibilities for dice roll outcomes: Ω = {1, 2, 3, 4, 5, 6}.

The probability for any number being rolled is 1/6.

The event of interest is "roll a 6 or roll a 1".

The probability for "roll a 6 or 1" is 1/6 + 1/6 = 2/6 = 1/3.

Implementation of stochastic search is achieved through different algorithms and techniques. Such techniques are based on exploitation and exploration principles.

Following are some of the popular techniques for stochastic search:

a. Simulated annealing – The name simulated annealing come from the field of metallurgical engineering in which temperature is brought down in

a very slow manner, so that particles can settle down gradually while cooling (minimum lattice energy state, thus avoiding and crystal defects, final configuration results in a solid with such superior structural integrity). Simulation of this entire process of annealing is used in an algorithm, which can be represented as:

Simulated Annealing ()

Step 1 – Start with any random node and generate a solution

Step 2 - Using any cost function, calculate the cost of the solution

Step 3- Generate a new solution using a random neighbor

Step 4 – Calculate the cost of new function

Step 5 – Compare new solution cost against the cost of previous solution

Step 6- If new solution is better than the old one cost wise, move to new solution and move to one more iteration.

Step 7- Keep checking for termination condition, which may either maximum number of iteration or optimal solution resulted.

b. Genetic algorithms – Motivation for genetic algorithm come from nature and the way it has evolved. Genetic mutation is a common process that keep happening in animals as well as trees. In this process, a gene is replaced by another, for environmental reasons. The evolutionary fundamentals when applied for computation purpose, are called as **Evolutionary Computing** and one of the branch of Evolutionary Computing is Genetic algorithm. In GA, there is a population, which consists of all possible encoded solutions to a given problem, wherein, every single solution in the population is called as chromosome. Population in computational space is called as genotype, whereas population in real world is called phenotype. Genotype is basically encoded solution from real world population to computational space. On the other hand, phenotype is decoded solution from computational space to real world. In a given population with problem, which is random one or generated from other known heuristics problem, fit parent candidates are selected. Fitness function is used to select the fit parents, the function has to be very fast and is expected to quantitatively measure the fitness of the candidates selected as parent. Crossover and mutation is carried out to generate a new off-sprint, which in turn replace the one in original population. This process is repeated again and again till number of iterations are met or optimal solution is arrived. GA is widely used in robotic engineering as well as other search optimization techniques.

Process in the GA can be represented in the form of algorithm as follows:

Genetic Algorithm ()

Step 1 – Initialize the population

Step 2- Using fitness function, check the fitness of population

Step 3 – Select the parent

Step 4 – Probability of cross over is P1

Step 5 – Probability of mutation is P2

Step 6 – Decode the solution to real world and calculate for fitness

Step 7 – Select the survivor

Step 8 – Find the best one and return the same to real world population

Step 9 – Repeat the steps 3 to 8 till termination criteria is met

Before moving ahead with explanation for Hill climbing technique for stochastic search optimization, let's see the comparison of Simulated Annealing and Genetic Algorithm.

SA: Comes from metallurgy engineering

Uses cost function to compare two solutions

Uses only one population space

Keeps comparing one solution with the neighboring to reach optimal solution

Widely used in solving combinatorial problems

GA: Comes from human evolutionary concepts

Uses Best fit function for the comparison purpose

Uses 2 population spaces, Phenotype and Genotype.

Keeps combining two solutions to reach target best off-spring

Widely used in robotic application, production planning**.**

c. Hill climbing – Hill climbing algorithm starts with a random value and continues searching higher value, till it reaches peak. Then peak values of neighboring peaks are compared with each other for better optimization. TSP (Traveling salesman problem) is an area where hill-climbing algorithm is widely used. Hill-climbing algorithm is a variation of generate and test method, which helps to decide in which direction to move in a search space. The direction to move in a search space is decided based on cost function value. Steps for hill-climbing algorithm are as follows:

Hill Climbing ()

Step 1 – If existing state is equal to target state, then stop

Step 2 – If existing state is not the target, keep repeating the process of finding and comparing new states until target state is achieved or there is no new operator left to apply

Step 3- Select new operator and apply on the current state

Step 4- Keep swapping current state and new state if new state is better

Step 5- Exit the repetitive process the moment current state becomes the optimum state.

QUESTION BANK

Q-1. What is a Sample? Why to use Sample?

A-1. Sample is a subset of large population. It allow researcher to conduct their study in a timely manner as its size is small.

Q-2. What is a Sampling Distribution?

A-2. It is a probability distribution of a statistic from a large number of samples.

Q-3. Give one example of sampling distribution.

A-3. Any live example used by researcher for analysis.

Q-4. Define following terms.

Sample, Population, Sampled Population, Element and Frame.

A-4. A ***sample*** is a subset of the population.

A ***population*** is a collection of all the elements of interest.

The ***sampled population*** is the population from which the sample is drawn.

An ***element*** is the entity on which data are collected.

A ***frame*** is a list of the elements that the sample will be selected from.

Q-5. What is re-sampling?

A-5. We only have a single estimate of the population parameter. To avoid this situation, we can use estimating the population parameter multiple times from our data sample. This is called re-sampling.

Q-6. What are TWO commonly used re-sampling methods?

A-6. (1). Bootstrap

(2). K – fold Cross Validation

Q-7. Discuss Statistical Inference.

A-7. Statistical Inference makes propositions about a population. Statistical Inference consists of selecting a statistical model and process that generates data and deducing propositions from the model.

Q-8. Define Prediction error.

A-8. Prediction error is the failure of some expected event to occur.

Q-9. What is Regression Analysis?

A-9. It is a set of statistical processes for estimating relationships between a dependent variable and one or more independent variables.

Q-10. What are the types of Regressions models?

A-10. Linear

Logistics

Polynomial

Stepwise

Ridge

Lasso

Q-11. Case study I

A group of estate agents carried out a survey in Mumbai for predicting rent and deposit amounts for apartments in different locations. Rent and deposit amounts can vary upon variety of factors such as distance from railway station, locality of the flat, distance from airport, nearest school and mall, amenities and carpet area of the flat. Mr. and Mrs. Y are looking for an apartment on rent and approached group of property agents. Their criteria for selecting an apartment are proximity from a school (2-4 km), distance from nearest railway station (5-10 km), amenities and locality. Property agents have short listed 4 properties for them. For finalizing the best property for them, create a decision table and tree based on fuzzy logic. Refer following values to prepare the table: P1 to P4 are shortlisted properties and C1 to C4 are criteria.

Properties / Criteria	P1	P2	P3	P4
School distance C1	3.5 km	2 km	4 km	3 km
Railway station distance C2	8 km	6.5 km	10 km	5 km
Amenities C3	√√√	√√	√√	√
Locality C4	*	***	*	**

A-11. Refer the section 6.2, for solving the case. Using a membership function (1/5 for school distance criteria, we can plot the graph with the gap of 5), write the membership values along with school distance in a tabular form. Find out minimum of membership value for each short-listed property, after repeating this process for all the criteria. Consider the maximum membership value from all membership values for different criteria and corresponding property. This property is ideal for Mr. and Mrs. Y based on their criteria. Draw a decision tree based on membership values for all the criteria.

Q-12. Case study II

In class of 40, students are graded as poor, average and extraordinary based on their percentages in the examination. Consider the universe with percentage values as:
U={0,10,20,30,40,50,60,70,80,90,100} and students with percentage below 40 are considered as poor, above 40 till 70 percent are average and above 70 are extraordinary.

Assume 2 subsets A and B

A= {33, 56, 87, 96, 25, 66, 79}

B= {78, 42, 64, 86, 35, 27, 31}

Assign weights (membership values) to all the values in A and B, design a membership function for the same. Draw graphs and find out count of core, support and boundary values in subset A and B.

A-12. Consider the universe U, which has values 0 to 100, where the gap is of 10. Hence, membership function can be 1/10. For each member in A and B, apply membership function to find out membership values. Then plot membership values against original values of each element.

Q-13 Multiple choice questions

1. Membership functions are better represented with the help of
a. Tabular form b. Graphical form
c. Mathematical form d. Logical form

2. Which of the following are fuzzy operators?
a. AND b. OR c. NOT d. All of the above

3. How best can we define dry in terms of humidity of the weather?
a. Fuzzy set c. Crisp set
b. Fuzzy and Crisp d. None of the above

4. Values of X mapped to lie between 0 to 1 which is called as
a. Membership value c. Degree of membership
b. All of the above d. None of the above

5. Fuzzy systems can be implement with the help of
a. Hardware c. Software
b. Both of the above d. None of the above

6. For a given fuzzy set A, which of the following elements do not belong to A?
A={(a,0.5) , (b,0.2) , (c,0), (d,1) , (e,0.8), (f,0.3)}
a. c b. d c. None of the above d. All of the above

7. _____ is best used to represent fuzzy values in a graph.
a. Square c. Hexagon
b. Triangle d. All of the above

8. A fuzzy system architecture has ____ main components.
a. 2 b.4 c.5 d. None of the above

9. Which of the following logic is the form of Fuzzy logic?
a. Two-valued logic d. Crisp set logic
b. Binary set logic e. Many-valued logic
c. None of these

A-13. Answers in Red color above.

Q-14.What is a membership function used in fuzzy logic? What are different techniques for fuzzifying or defuzzifying data?

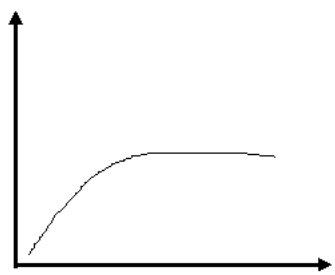
A-14. Definition and need for fuzzy logic with example. List down techniques for Defuzzification.

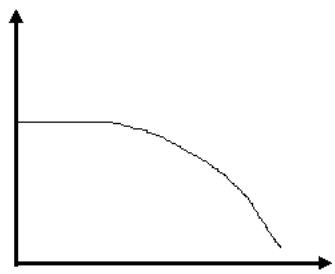
Q-15. Compare Mamdani and Sugeno model with their pros and cons.

A-15. Explain the concept of Stochastic Search methods. Mamdani and Sugeno method, Advantages and disadvantages of each.

Q-16. Explain the concept of 'Monotonic function"? Why it is alternatively called as shoulder function?

A-16. In Tsukamoto method, outcome is a polynomial function instead of fuzzy value. Monotonic function is one which takes increasing, decreasing or constant values. After plotting such successive values, we get a graph of following nature, which resembles human shoulder. Hence called as shoulder function.





Q-17. Explain architecture of Fuzzy system with appropriate diagram.

A-17. Draw a diagram with 4 important components of Fuzzy System: Fuzzifier, Rule Base, Inference System and De-fuzzifier. Explain each and list down any 2 techniques, each for Fuzzification and Defuzzification.

Q-18. Compare and contrast Simulated annealing, Genetic algorithm and Random walk techniques for stochastic search.

A-18. Explain SA and GA technique and steps for the same. Write one application where it can be used. Write advantages and disadvantages of each.

Q-19. What is Neural Network?

Q-20. What is generalization in Neural Network?

Q-21. List out various applications where we can use Neural Network

Q-22. What is Competitive Learning in Neural Network?

Q-23. What is need of Principal components analysis in Neural Network?

Q-24. List five characteristic of big data.

A-24. Volume, variety, veracity, variability and velocity are characteristics of big data.

Q-25. Name few unit of measurement for memory used in today's era of big data.

A-25. Terabytes, Petabytes, Zettabytes and Exabytes.

Q-26. Write various steps to carry out for analysis process in general.

A-26. The following steps has to carry out analysis process: Data collection, Data cleaning, Data preprocessing, Data analysis, Visualisation and Representation, Understanding results.

Q-27. Write 2 differences between analysis and reporting process.

A-27. (1) The goal of the analysis process is inspecting the data and transforming into useful meaningful information. Whereas, the goal of the reporting process is to transforming the output of process in to presentable format. (2) The main purpose of conducting analysis process is examining interpreting comparing and predicting about the data. Whereas reporting process is mainly focusing on highlighting organizing summarizing and formatting process.

Q-28. Write difference between structure and unstructured data.

A-28. Structure data can store with two dimensional structure like worksheets. The structure of data is predefined and fixed. Whereas

unstructured data do not have fixed data format. It is volatile in nature.

Q-29. Write examples of structure and unstructured data.

A-29. Structure data - Business data stored in RDBMS system, excel worksheet Unstructured data - text data, web data, images.

Q-30. Write any three reasons behind increasing volume of internet data in last few years.

A-30. The reasons behind increasing volume of internet data are as follow:

(1) Increase in number of internet users.

(2) Increasing popularity of social media websites and online shopping websites.

(3) IoT systems usage.

Q-31. Explain the term 'velocity' with reference to big data.

A-31. Velocity measures how fast the data is coming in. In some system data are come in in real-time, whereas in other systems data are come in batches. Depending on the velocity of data, data storage system has to manage the flow of the data.

Q-32. Name any three technology used for Big data analytics.

A-32. R language, Python language and Hadoop ecosystem are popular technology used for big data analytics.

Q-33. Differentiate between linear and non-linear time series data

Q-34. Explain various inherent components of time-series data, with suitable examples.

Q-35. Mention and briefly introduce algorithms available for rule induction process.

Q-36. Illustrate the statement “BDS is a litmus test for deciding non-linearity of time series data”.

Q-37. Compare and contrast Additive and Multiplicative methods.

Q-38. Explain steps to carry out Exponential Smoothing of time series data.

Q-39. Discuss pros and cons of ARIMA for the purpose of forecasting of time series data.

Text book:

- Mining of Massive Datasets, Anand Rajaraman and Jeffrey David Ullman, Cambridge University Press, 2012.

- Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends for Today's Businesses, Michael Minelli, Wiley, 2013

References:

- Big Data for Dummies, J. Hurwitz, et al., Wiley, 2013
- Understanding Big Data Analytics for Enterprise Class Hadoop and Streaming Data, Paul C. Zikopoulos, Chris Eaton, Dirk deRoos, Thomas Deutsch, George Lapis, McGraw-Hill, 2012.
- Big data: The next frontier for innovation, competition, and productivity, James Manyika ,Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers, McKinsey Global Institute May 2011.
- Big Data Glossary, Pete Warden, O'Reilly, 2011.
- Big Data Analytics: From Strategic Planning to Enterprise Integration with Tools, Techniques, NoSQL, and Graph, David Loshin, Morgan Kaufmann Publishers, 2013

2

MAP REDUCE

Unit Structure

2.0 OBJECTIVES

This chapter will make you to understand the following concepts:

- The requirement of the big data handling tool.
- The structure and working of the Distributed File System.
- Physical Organization of Compute Nodes
- Large-Scale File System Organization
- The importance of Apache Hadoop, MapReduce and parallel processing for mining large-scale data.
- The MapReduce Framework and its steps of execution.
- The features and working flow of the MapReduce system.
- The MapReduce execution of Matrix Multiplication algorithm and relational algebra operations.
- The input and output file format of MapReduce phases.
- The generalized form of MapReduce, a workflow system.
- The recursive extension of MapReduce and handling faults during execution of MapReduce.
- Designing the MapReduce algorithm for small tasks and large data.

2.1 INTRODUCTION

In modern applications the quick data insights or analysis require us to manage the immense amount of data quickly. In most of these applications, the data is extremely regular, and there is ample opportunity to exploit parallelism. Some of the Important examples are:

1. Importance wise ranking of Web pages, involves an iterated matrix-vector multiplication where the dimension is in the tens of billions.

2. At social networking sites, searches in “friends” networks involve graphs with hundreds of millions of nodes and many billions of edges.

In these applications, a new software stack has developed. These applications are using the new form of file system, which features much larger units than the disk blocks in a conventional operating system. This file system also provides the facility of replication of data to protect against the frequent media failures that occur when data is distributed over thousands of disks.

Now a day many of the higher-level programming languages support these file systems. The central component of these programming languages is MapReduce. The MapReduce implementation helps us to perform most common calculations on large-scale data on large collections of computers efficiently, that is tolerant of hardware failures during the computation.

Map-reduce systems are evolving and extending rapidly. In this chapter we will discuss the distributed file systems, MapReduce, generalizations of map-reduce, first to acyclic workflows and then to recursive algorithms. We will discuss some common algorithms of MapReduce as well.

2.2 DISTRIBUTED FILE SYSTEMS

Most computations are performed on a single processor that uses its own main memory, cache and local disk (a computing node).In such systems the files are managed by a file management system. The file management system is capable of handling the files that are stored on a single computer or cluster. In the past parallel processing applications, the parallel processing was done on special purpose computers with multiple processors and specialized hardware. The ever-increasing web services have created the demand to do huge computing independently and instantly on a large extensible cluster. As compared to the special-purpose parallel computers the Commodity hardware is cheap in cost.

The availability of cheap and faster hardware gives rise to a new generation of programming systems with the feature of parallelism. These systems take advantage of the power of parallelism and at the same time avoid the reliability problems that arise when the computing hardware consists of thousands of independent components, any of which could fail at any time.

In this chapter, we will discuss the characteristics of the computing installations and the specialized file systems that have been developed to take advantage of them.

2.2.1 Physical Organisation of Compute Nodes

The parallel-computing architecture or cluster computing comprises the computing nodes that are organised into the number of racks. The rack may contain 8 to 64 computing nodes that are connected by a network like gigabit ethernet. The racks are connected with each other through a switch or another level of network. In order to communicate with the nodes in other rack, the bandwidth of inter-rack communication should be greater than the bandwidth of intra-rack ethernet. Figure 2.1 shows the architecture of a large-scale computing system with multiple racks, each with multiple nodes.In this network, the principal modes are loss of a single node or loss of an entire rack. If any of the

nodes failed due to some reason, then the network will not be able to provide the data of this node or perform computations on this node. Or if any connection to a rack fails, then the network connecting its nodes to each other and the outside world fails.

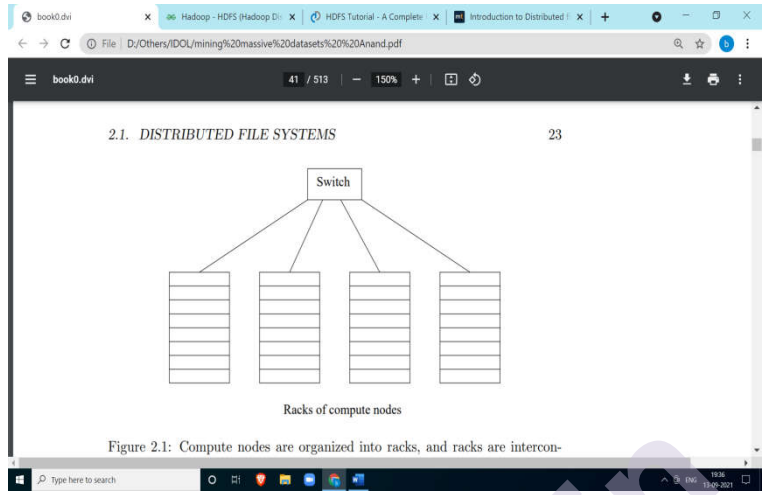


Figure 2.1: Computing nodes are organized into racks and racks are interconnected by a switch. The large computations may take minutes or even hours. During computation, if any one component failed, the abort or restart of computation may lead to failure.

To overcome this problem,

1. Files must be stored redundantly onto multiple nodes.
2. Computations must be divided into tasks and allocated to the multiple nodes.

2.2.2 Large-Scale File System Organization

To store the enormous file on multiple computers you need to use the distributed file system. The distributed File Systems (DFS) can handle the data stored across multiple clusters or nodes. The files that are stored on a distributed file system are rarely updated. The file is stored on multiple nodes by dividing it into a number of chunks.

For example, to store the file of 30 TB in a distributed file system with multiple nodes in a cluster (each of capacity 10 TB), needs to be divided into the blocks or chunks. The size of the chuck is defined by the user like 64 megabytes, 128 megabytes and so on.

The Fault tolerance is achieved by replicating the chunks three times, at three different compute nodes of different racks. This also helps us to get the copy of the chunk in case of rack failure. Usually, both the chunk size and the degree of replication can be decided by the user.

The metadata of the chunks of a file is stored on a name node which acts as a master node. The master node is itself replicated, and a directory for the file system as a whole knows where to find its copies. The directory itself can be

replicated, and all participants using the DFS know where the directory copies are.

DFS Implementations

There are several distributed file systems of the type. Some of these systems that are used in practice are:

1. The Google File System (GFS), the original of the class.
2. Hadoop Distributed File System (HDFS), an open-source DFS used with Hadoop, an implementation of map-reduce and distributed by the Apache Software Foundation.
3. Cloud Store, an open-source DFS originally developed by Kosmix.

2.3 APACHE HADOOP

Apache Hadoop is a collection of open-source utilities that allows us to use a network of many computers to solve problems involving massive amounts of Data and computation. Hadoop provides the software framework for distributed data storage and MapReduce programming model for processing big data. Hadoop is designed to scale up from a single server to a cluster of thousands of machines. Each of these machines in the cluster offers the local computation and storage.

Apache Hadoop was originally designed for computer clusters that are built from commodity hardware or even high-end hardware. The Hadoop framework distributes an analytical computation of massive data on many machines, each of which simultaneously operates on their own individual chunk of data.

For distributed computing, the distributed systems shall meet the following requirements -

1. **Fault Tolerance**: If any of the components fails, the entire system should not get fail. The system should gracefully degrade into a lower performing state. If a failed component recovers, it should be able to rejoin the system.
2. **Recoverability**: In case of failure, no data should be lost.
3. **Consistency**: The final result should not get affected due to failure of any component.
4. **Scalability**: Adding more data and more computation leads to a decline in performance but not fail; increasing resources should result in a proportional increase in capacity.

Hadoop addresses these requirements through the abstract concepts, as defined in the following list:

1. Data is distributed immediately when added to the cluster and stored on multiple nodes. Nodes prefer to process data that is stored locally in order to minimize traffic across the network.
2. Data is stored in blocks of a fixed size (usually 128 MB) and each block is duplicated multiple times across the system to provide redundancy and data safety.
3. A computation is usually referred to as a job; jobs are broken into tasks where each individual node performs the task on a single block of data.

4. Jobs are written at a high level without concern for network programming, time, or low-level infrastructure, allowing developers to focus on the data and computation rather than distributed programming details.
5. The amount of network traffic between nodes should be minimized transparently by the system. Each task should be independent and nodes should not have to communicate with each other during processing to ensure that there are no inter-process dependencies that could lead to deadlock.
6. Jobs are fault tolerant usually through task redundancy, such that if a single node or task fails, the final computation is not incorrect or incomplete.
7. Master programs allocate work to worker nodes such that many worker nodes can operate in parallel, each on their own portion of the larger dataset.

These basic concepts, while implemented slightly differ to various Hadoop systems, drive the core architecture and together ensure that the requirements for fault tolerance, recoverability, consistency, and scalability are met. These requirements also ensure that Hadoop is a data management system that behaves as expected for analytical data processing, which has traditionally been performed in relational databases or scientific data warehouses.

2.3.1 Elements of Hadoop Ecosystem

The Hadoop ecosystem is a platform that provides various services to solve the big data problems. This includes various Apache products, commercial tools and solutions. The four major elements of Hadoop Ecosystem are Hadoop Distributed File System (HDFS), MapReduce, YARN and Hadoop Common. Hadoop Ecosystem provides the tools that are used to perform tasks like load, analyse, and maintain data. Some of the components/tools of Hadoop Ecosystem are as follows:

1. Hadoop Distributed File System (HDFS)
2. Yet Another Resource Negotiator (YARN)
3. MapReduce - Programming based Data Processing
4. Spark for In-Memory data processing
5. PIG and HIVE - Query based processing of data services
6. HBase - NoSQL Database
7. Mahout and Spark MLLib - Machine Learning algorithm libraries
8. Solar and Lucene - Searching and Indexing
9. Zookeeper - Managing Cluster
10. Oozie - Job Scheduling

2.4 MAPREDUCE

Hadoop MapReduce is a Software framework. MapReduce is also referred to as a programming model that performs parallel and distributed processing on massive

datasets. The implementations of MapReduce can be used to manage large-scale computations in a way that is tolerant of hardware faults.

MapReduce is the processing component of Hadoop and is used to write the applications that process huge] amounts of data in parallel on large Hadoop clusters of commodity hardware. These clusters are scalable, reliable and fault tolerant.

The term 'MapReduce' specifies the two distinct tasks that are to be performed by Hadoop programs:

1. Map task which accepts the data and converts it into another set of data. Here each individual element of the data is split into Key-value pairs.
2. Reduce Task that takes the output of Map Task as input and combines them into a smaller set of tuples. So, the reducer task takes place after the completion of the map task.

In brief, a map-reduce computation executes as follows:

1. The Map tasks with given one or more chunks from a distributed file system turns the chunk into a sequence of key-value pairs. The way key-value pairs are produced from the input data is determined by the code written by the user for the Map function.
2. The key-value pairs from each Map task are collected by a master controller and sorted by key. The keys are divided among all the Reduce tasks, so all key-value pairs with the same key wind up at the same Reduce task.
3. The Reduce tasks work on one key at a time, and combine all the values associated with that key in some way. The manner of combination of values is determined by the code written by the user for the Reduce function.

The figure 2.2 shows the schematic of a MapReduce computation.

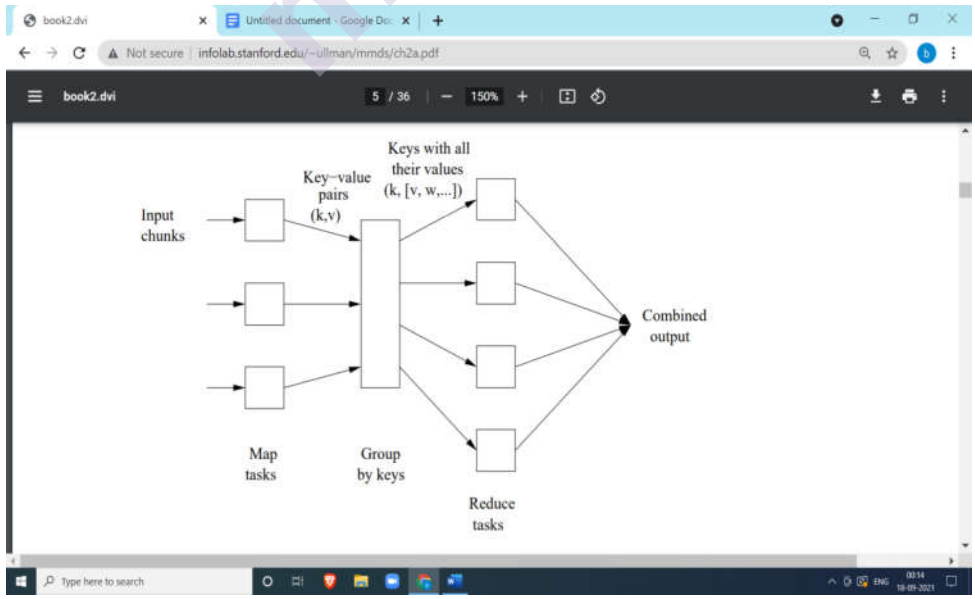


Figure 2.2: Schematic of a MapReduce Computation

The MapReduce programming is useful to gain valuable insights from the data. The advantages of MapReduce programming are as follows:

1. **Simple**: Developers can write code using any of the languages including Java, C++ and Python.

2. **Scalability**: Businesses can process petabytes of data stored in the Hadoop Distributed File System (HDFS).

3. **Flexibility**: Hadoop enables easier access to multiple sources of data and multiple types of data.

4. **Speed**: Due to parallel processing and minimal data movement, Hadoop offers fast processing of massive amounts of data.

The MapReduce programming model is developed using Java with three classes:

1. **Mapper class:** The Mapper class performs the Map task of splitting data and converting it into key-value pairs. The mapper class stores these resultant key-value pairs in HDFS.
2. **Reducer class:** The Reducer class reads the output of the mapper class from HDFS, processes it and generates the final output in the form of Key-value pairs. The reducer stores this output of the Reduce task in HDFS.
3. **Driver class:** The Driver class sets up the MapReduce job to run in Hadoop.

With the help of Mapper class and Reducer class, MapReduce processes the given input data and generates the output in form of key-value pairs. During this process the data undergoes the various MapReduce steps.

2.5 STEPS OF MAP REDUCE

The MapReduce programming model follows the following steps for solving the problem.

1. The Map Task
2. Grouping by Keys
3. Reduce Task
4. Combiner

2.5.1 The Map Task

The mapper accepts the user input file with the elements of any type like tuples or a document. The mapper will split the input into the number of chunks and distribute it over the network of map nodes. Here, a chunk is a collection of data elements and no element is stored across the two chunks. Each map node will process the data and will return the list of key-value pairs.

Technically, all inputs from Map tasks and outputs of Reduce tasks are of the key-value-pair form, but normally the keys of input elements are not relevant and we shall tend to ignore them. Insisting on this form for inputs and outputs is motivated by the desire to allow composition of several map-reduce processes.

A Map function is written to convert input elements to key-value pairs. The types of keys and values are each arbitrary. Here, the keys are not “keys” in the usual

sense; they do not have to be unique. Rather a Map task can produce several key-value pairs with the same key, even from the same element.

Example 2.1: Let us discuss a map-reduce computation with the standard Word count example application: counting the number of occurrences for each word in a collection of documents.

Here, in this example the input file is a repository of documents, and each document is an element. Here the Map function defines the key value pair with the document words as keys and the number of occurrences of words as integer values. The Map task reads a document and splits it into its sequence of words $w_1, w_2, \ldots, w_n$. After processing the Map task emits a sequence of key-value pairs where the value is always 1. That is, the output of the Map task for this document is the sequence of key-value pairs:

$(w_1,1), (w_2,1), \ldots, (w_n,1)$

A single Map task will typically process many documents where each of these documents is in one or more chunks. In such cases the output will be more than the sequence for the one document suggested above. If a word *w* appears *m* times among all the documents assigned to that process, then there will be *m* key-value pairs *(w, 1)* among its output.

2.5.2 Grouping by Keys

Grouping and aggregation task is performed independently by the master controller process. It is not related to Map and Reduce tasks. The master controller process knows how many Reduce tasks r, that are given by the user to the map-reduce system.

The master controller uses a hash function to group the keys. To do so it produces a bucket number from *0* to *r-1*. So, each key that is emitted by a Map task is hashed and its key-value pair is put in one of r local files. Each file is intended for one of the Reduce tasks.

After completing all the Map tasks successfully, the master controller merges the file from each Map task that are intended for a particular Reduce task and feeds the merged file to that process as a sequence of key-list-of-value pairs. For each key *k*, the input to the Reduce task that handles key k is a pair of the form *(k, [v1, v2, . . .,vn]), where (k, v1), (k, v2), . . ., (k, vn)* are all the key-value pairs with key *k* coming from all the Map tasks.

2.5.3 The Reduce Task

The Reduce function readsthe output of the Mapper function, which is in the key-value pairs and combines the values in some way. After reading these key-value pairs, the reducer function combines the list of values for each key. Once combined, the reducer function merges the output of all reduce tasks into a single file.

Example 2.2: In continuation to Example 2.1 of word-count. The Reduce function sums all the values and returns a sequence of *(w, m)* pairs, where *w* is a key or word that appears at least once in the input documents and *m* is the total number of occurrences of word *w* among all those documents.

2.5.4 Combiners

Usually, the Reduce function is associative and commutative, which helps to combine the values in any order, with the same result. In Example 2.2, the addition performed is an example of an associative and commutative operation. While grouping the values, the order of numbers in the list of values $v_1, v_2, \ldots, v_n$ does not affect the sum value.

Since the Reduce function is associative and commutative, it is possible to push some tasks of Reduce to the Map tasks. For example, in Example 2.1, instead of producing many pairs *(w, 1), (w, 1),* . . ., in map task, we could apply the Reduce function within the Map task before sending output of map task for grouping and aggregation. So, this list of key-value pairs would thus be replaced by one pair with key *w* and value equal to the sum of all the 1's in all those pairs. That means, the pairs with key *w* generated by a single Map task would be combined into a pair *(w, m)*, where m is the number of times that w appears among the documents handled by this Map task. Though the reduced task is applied in map tasks, there is still a need for grouping and aggregation operations for grouping the key-value pairs that are coming from map tasks of different map nodes.

2.5.5 Details of MapReduce Execution

While executing the MapReduce tasks, the various processes, tasks and files interact with each other as shown in Figure 2.3. With the help of Hadoop, a library provided by a MapReduce system, the user program forks a Master Controller process and some number of Worker processes at different compute nodes. Here each of the worker nodes can act as a Map worker or a Reduce worker but not both. The Map worker handles the Map task whereas the Reduce worker handles the Reduce task.

The Master creates some number of Map tasks and some number of Reduce tasks where this number is being selected by the user program. After creation, the Master assigns these tasks to worker processes. Depending on the size of the input file and the size of the chunk defined by the user, the Master creates one Map task for every chunk. For each Reduce task, the Map task needs to create an intermediate file. So, the number of reduced tasks should be less than the Map tasks, otherwise the number of intermediate files explodes.

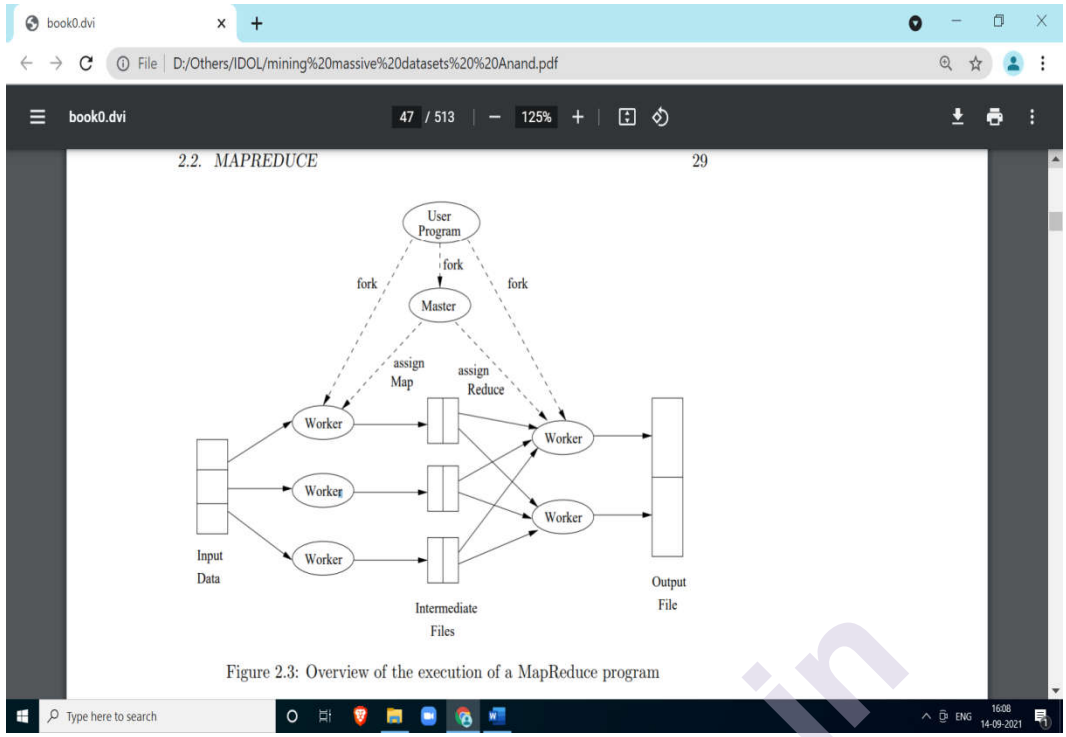


Figure 2.3: Overview of the execution of a MapReduce program

All the Map and Reduce nodes status is maintained and controlled by the Master node. The Master node keeps track of the execution process of the Map and Reduce nodes. If any of the Map or Reduce nodes finish the execution, the Master allocates the other task to this node. If any task execution fails at a particular node, the Master node reallocates that task to another node.

Every Map task is assigned one or more chunks of the input files. The Map node executes the mapper code, written by the user on these chunks. The Map tasks creates an output file for each reduce task and stores it on the local disk of Map node and sends information about size and location of this file to Master node.

The Master node assigns the Reduce task to worker nodes and provides the output files of Map tasks as an input. The reduce task executes code written by the user and sends output to the file that is stored in a distributed file system.

2.5.6 Coping with Node Failures

The Mater node is controlling the failure of the Map nodes and the Reduce nodes. But what if the Master node fails? The one node can bring the entire process down. In this case the entire MapReduce job needs to be restarted and completed eventually.

The Master node periodically pings the worker nodes, and hence the worker processes. In case of the failure of the worker node, the master node reallocates the complete process of this node to another node, since the output of this process needs to be assigned to the Reduce task. The Master must also inform each Reduce task that the location of its input from that Map task has changed.

Managing the failure of a Reduce worker is simpler. The Master simply sets the status of its currently executing Reduce tasks to idle. These will be rescheduled on another reduce worker later.

Example 2.3: Let us understand the steps of MapReduce with a word count example. The **Word count** example reads a text file and counts the total number of occurrences of each word.

Let us consider the text file sample.txt with the following text.

Bus. Car, Train, Ship, Ship, Train, Bus, Ship, Car

Figure 2.4 shows the steps of MapReduce tasks for the Wordcount example.

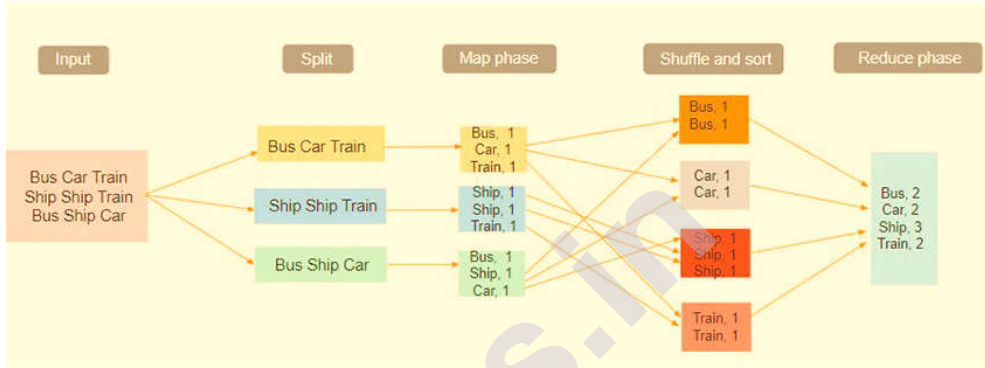


Figure 2.4: MapReduce steps of Wordcount example

The Map Task

For the above Example 2.3 mapper call read the input text file 'Sample.txt' and split it into the 3 chunks, (Bus. Car, Train), (Ship, Ship, Train), and (Bus, Ship, Car). After splitting the mapper assigns each of these chunks to the map nodes. Each map node then splits the chunk text into words and converts it into the key value pair, by adding a frequency of occurrence as 1. Here the key is the word and the value is the frequency of occurrence of that word.

Map Node 1: (Bus.1), (Car,1), (Train,1),

Map Node 2: (Ship,1), (Ship,1), (Train,1), and

Map Node 3: (Bus,1), (Ship,1), (Car,1)

Grouping by Key

In Example 2.3 after completing a mapper phase, the reducer will read the output of mapper, and partition it with the help of sorting and shuffling process for each of the keys in the data set. The partition process sent the tuples with the same key to a respective reducer. The sort and shuffle acts on these lists of <key, value> pairs and sends out unique keys and a list of values associated with this unique key <key, list(values)>.

Reducer Node 1: (Bus,1,1)

Reducer Node 2: (Car,1,1)

Reducer Node 3: (Ship,1,1,1)

Reducer Node 4: (Train,1,1)

The Reduce Task

In Example 2.3 the reducer will aggregate the values of intermediate tuples that are generated in sorting and shuffling step and will generate the list of unique key-value pairs with the total number of key occurrences by summing the list of values.

Reducer Node 1: (Bus,2)

Reducer Node 2: (Car,2)

Reducer Node 3: (Ship,3)

Reducer Node 4: (Train,2)

Combiner

In Example 2.3, the combiner will read the key-value pairs that are generated by reducer and combine it into a single set of key-value pairs and write it into the output file.

Final Output:

(Bus,2)

(Car,2)

(Ship,3)

(Train,2)

2.6 ALGORITHMS USING MAPREDUCE

MapReduce is growing rapidly and helps in parallel computing tasks like determining the price for products, yielding the highest profits, predicting and recommending analysis and so on. It allows programmers to run models over different data sets and uses advanced statistical techniques and machine learning techniques that help in predicting data.

MapReduce algorithms are not used for smaller tasks. Even every problem needs not to use the Distributed File Systems for storing data. For example, we would not expect to use either a DFS or an implementation of MapReduce for managing online retail sales, even though a large on-line retailer such as Amazon.com uses thousands of compute nodes when processing requests over the Web. The reason is that the principal operations on Amazon data involve responding to searches for products, recording sales, and so on. MapReduce algorithms are not advised to use for the processes that involve relatively little calculation and that need to update the database.

On the other hand, the MapReduce algorithms are used for large computations or processes. For example, Amazon uses MapReduce to perform analytic queries on large amounts of data, such as finding the customers pattern, who are buying the particular product.

MapReduce algorithm can be used with a variety of applications. It can be used for distributed pattern-based searching, distributed sorting, web link graph reversal, web access log stats. It can also help in creating and working on multiple clusters, desktop grids, volunteer computing environments. One can also create dynamic cloud environments, mobile environments and also high-performance computing environments.

Google made use of MapReduce which regenerates Google Index of the World Wide Web. The original purpose for which the Google implementation of MapReduce was created is to execute very large matrix-vector multiplications as are needed in the calculation of Page Rank. Another important class of operations that can use MapReduce effectively are relational-algebra operations.

2.6.1Matrix-Vector Multiplication by MapReduce

Let us consider matrix *M of size n* × *n.* The location of the element of matrix M is referred to by row *i* and column *j*and will be denoted by m_{ij}. Let us have a vector *v* of length *n*, whose j^{th} element is v_j. The product of vector v and matrix M is the vector x of length n, whose i^{th} element x_i is given by

If the value of *n* is small, say 100, we do not want to use a DFS or MapReduce for this calculation. On the other hand, this method can be used when n is large. For example, in search engines for the ranking of Web pages, n is in the tens of billions.

When *n* is large, it should not be so large that vector *v* cannot fit in main memory and be part of the input to every Map task. It is observed that there is nothing in the definition of map-reduce that for bids providing the same input to more than one Map task.

Both the matrix *M* and the vector *v* each will be stored in a file of the Distributed File System. The elements of the Matrix are stored in rows and columns. The element m_{ij}, that is stored at the row *i* and column *j*, can be referred to by a triple(i, j, m_{ij}). In the same way the position of j^{th}element in the vector *v* is referred to by v_j.

The Map Function: Each Map task will take the entire vector v and a chunk of the matrix *M*. From each matrix element m_{ij} it produces the key-value pair $(i, m_{ij}v_j)$. Thus, all terms of the sum that make up the component x_i of the matrix-vector product will get the same key.

The Reduce Function: The Reduce task simply sums all the values associated with a given key i. The result will be a pair (i, x_i).

2.6.2 If the Vector v Cannot Fit in Main Memory

If the vector v is so large that it will not fit in main memory, then to perform the Matrix-Vector multiplication operation, we need to divide the vector into horizontal stripes of equal height. But in that case, we also need to divide the matrix into the vertical stripes of equal width. Here we need to use enough stripes so that the portion of the vector in one stripe can fit conveniently into the main memory at a compute node. Figure 2.5shows the matrix and vector, which are divided into five stripes.

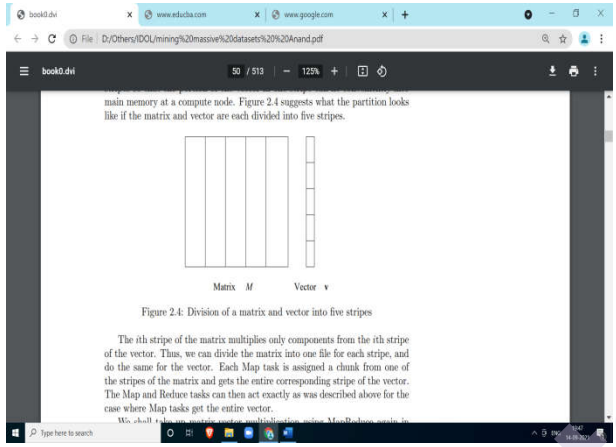


Figure 2.5: Division of matrix and vector into five stripes

The i^{th} stripe of the matrix multiplies only components from the i^{th} stripe of the vector. We can store each stripe of matrix and vector into individual files. Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector. The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector.

2.6.3 Relational Algebra Operations

In database queries, the number of operations needs to be performed on large-scale data. In many traditional database applications, the database is large but some of the queries need to retrieve a small amount of data. For example, in bank applications, the database is too large but the query for getting the balance of an account is too small. In all such applications, we need not to use MapReduce algorithms.

In fact, there are many operations on data that can be described easily in terms of the common database-query primitives, even if the queries themselves are not executed within a database management system. Thus, a good starting point for exploring applications of MapReduce is by considering the standard operations on relations.

In relational model, a relation is a table with column headers called attributes. Rows of the relation are called tuples. The set of attributes of a relation is called

its schema. We often write an expression for a Relation R like $R(A_1, A_2, \ldots, A_n)$where $A_1, A_2, \ldots, A_n$ *are the* attributes of it.

Example 2.4: The Figure 2.6shows the part of the relation Links that describes the structure of the Web. The relation has two attributes, From and To. In this relation, a row, or tuple is a pair of URLs, such that there is at least one link from the first URL to the second. For example, the first row of Figure 2.6 is the pair *(url1, url2)* that says the Web page *url1* has a link to page *url2*. Figure 2.6 shows only four tuples. But the typical search engine stores the billions of tuples that define the relation of url1 to url2. Such large files of rations are stored on DFS.

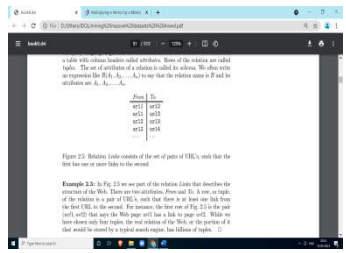


Figure 2.6: Relation Links consists of the set of pairs of URLs, such that the first has one or more links to the second

The relation algebra specifies several standard operations on relations that are used to implement queries. The queries are usually written in SQL. Some of the relational-algebra operations are:

1. **Selection**: Applying a condition *C* to each tuple in the relation and producing as output only those tuples that satisfy *C*. The result of the selection is denoted $\sigma_C(R)$.

2. **Projection**: For subset *S* of the attributes of the relation, produce from each tuple only the components for the attributes in *S*. The result of the projection is denoted $\pi_S(R)$.

3. **Union, Intersection, and Difference**: These set operations apply to the sets of tuples in two relations that have the same schema.

4. **Natural Join**: Given two relations, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. The natural join of relations *R* and *S* is denoted $R \bowtie S$. While we shall discuss executing only the natural join with map-reduce, all equijoins can be executed in the same manner.

5. **Grouping and Aggregation:** For a given relation *R*, partition its tuples according to their values in one set of attributes *G*, called the grouping attributes. Then, for each group, aggregate the values in certain other attributes.

The common aggregations are SUM, COUNT, AVG, MIN, and MAX. Here the MIN and MAX require the aggregated attributes of number or string type. The SUM and AVG require the numeric type attribute to perform arithmetic. The grouping-and-aggregation operation on a relation R is denoted by $\gamma X(R)$, where X is a list of elements that are either

a) A grouping attribute, or
b) An expression $\theta(A)$, where θ is one of the five aggregation operations such as SUM, and A is an attribute not among the grouping attributes.

The result of this operation is one tuple for each group. That tuple has a component for each of the grouping attributes, with the value common to tuples of that group, and a component for each aggregation, with the aggregated value for that group.

Example 2.5: For the relation in Figure 2.6, let us try to find the paths of length two in the Web. That is, we want to find the triples of URLs *(u, v, w)* such that there is a link from *u* to *v* and a link from *v* to *w*.

We need to take the natural join of Links with itself, but we first need to imagine that it is two relations, with different schemas, so we can describe the desired connection as a natural join. Thus, imagine that there are two copies of Links, namely *L1(U1, U2)* and *L2(U2, U3)*. Now, if we compute $L1 \rhd\lhd L2$, we shall have exactly what we want. That is, for each tuple *t1* of *L1* (i.e., each tuple of Links) and each tuple *t2* of *L2* (another tuple of Links, possibly even the same tuple), see if their *U2* components are the same. Note that these components are the second component of *t1* and the first component of *t2*. If these two components agree, then produce a tuple for the result, with schema *(U1, U2, U3)*. This tuple consists of the first component of *t1*, the second component of *t1* (which must equal the first component of *t2*), and the second component of *t2*.

We may not want the entire path of length two, but only want the pairs *(u, w)* of URLs such that there is at least one path from u to w of length two. If so, we can project out the middle components by computing $\pi_{U1,U3}(L1 \rhd\lhd L2)$.

2.6.4 Computing Selections by MapReduce

The Selection operations does not require the full power of map-reduce. They can be done most conveniently either by using Map portion or the Reduce portion. A map-reduce implementation of selection is denoted by $\sigma C(R)$.

The Map Function: For each tuple *t* in *R*, test if it satisfies *C*. If so, produce the key-value pair *(t, t)*. That is, both the key and value are *t*.

The Reduce Function: The Reduce function is the identity. It simply passes each key-value pair to the output.

Here, the output is not exactly a relation, since it has key-value pairs. However, a relation can be obtained by using only the value components (or only the key components) of the output.

2.6.5 Computing Projections by MapReduce

Projection is performed similarly to selection, because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates. We may compute π_S (R) as follows.

The Map Function: For each tuple *t* in *R*, construct a tuple *t′* by eliminating from *t* those components whose attributes are not in *S*. Output the key-value pair *(t′, t′)*.

The Reduce Function: For each key *t′* produced by any of the Map tasks, there will be one or more key-value pairs *(t′, t′)*. The Reduce function turns *(t′, [t′, t′, . . ., t′])* into *(t′, t′)*, so it produces exactly one pair *(t′, t′)* for this key *t′*.

The Reduce operation is duplicate elimination. This operation is associative and commutative, so a combiner associated with each Map task can eliminate whatever duplicates are produced locally. However, the Reduce tasks are still needed to eliminate two identical tuples coming from different Map tasks.

2.6.6 Union, Intersection and Difference by MapReduce

Union

Let us consider the union of two relations. Suppose relations *R* and *S* have the same schema. Map tasks will be assigned chunks from either *R* or *S*; it doesn't matter which. The Map tasks don't really do anything except pass their input tuples as key-value pairs to the Reduce tasks. The latter need only eliminate duplicates as for projection.

The Map Function: Turn each input tuple t into a key-value pair *(t, t)*.

The Reduce Function: Associated with each key t there will be either one or two values. Produce output *(t, t)* in either case.

Intersection

To compute the intersection, we can use the same Map function. However, the Reduce function must produce a tuple only if both relations have the tuple. If the key t has two values *[t, t]* associated with it, then the Reduce task for *t* should produce *(t, t)*. However, if the value associated with key *t* is just *[t],* then one of *R* and *S* is missing *t*, so we don't want to produce a tuple for the intersection. We need to produce a value that indicates "no tuple," such as the SQL value NULL. When the result relation is constructed from the output, such a tuple will be ignored.

The Map Function: Turn each tuple t into a key-value pair *(t, t)*.

The Reduce Function: If key *t* has value list *[t, t],* then produce *(t, t)*. Otherwise, produce *(t, NULL)*.

Difference

The Difference $R-S$ requires a bit more thought. The only way a tuple t can appear in the output is if it is in R but not in S. The Map function can pass tuples from R and S through, but must inform the Reduce function whether the tuple came from R or S. We shall thus use the relation as the value associated with the key t. Here is a specification for the two functions.

The Map Function: For a tuple t in R, produce key-value pair *(t, R),* and for a tuple t in S, produce key-value pair *(t, S)*. Note that the intent is that the value is the name of R or S, not the entire relation.

The Reduce Function: For each key t, do the following.

1. If the associated value list is *[R]*, then produce *(t, t)*.
2. If the associated value list is anything else, which could only be *[R, S], [S, R],* or *[S],* produce *(t, NULL)*.

2.6.7 Computing Natural Join by MapReduce

The idea behind implementing natural join via map-reduce can be seen if we look at the specific case of joining R(A, B) with S(B, C). We must find tuples that agree on their B components, that is the second component from tuples of R and the first component of tuples of S. We shall use the B-value of tuples from either relation as the key. The value will be the other component and the name of the relation, so the Reduce function can know where each tuple came from.

The Map Function: For each tuple (a, b) of R, produce the key-value pair (b,(R, a)). For each tuple (b, c) of S, produce the key-value pair (b,(S, c)).

The Reduce Function: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S, say (R, a) and (S, c). The output for key b is (b, [(a1, b, c1), (a2, b, c2), . . .]), that is, b associated with the list of tuples that can be formed from an R-tuple and an S-tuple with a common b value.

2.7 EXTENSIONS TO MAPREDUCE

The MapReduce method of computation gave rise to many systems with some extensions and modifications. Some of the common characteristics of these systems and MapReduce systems are as follows:

1. Both the extended systems and the MapReduce are built on a distributed file system.
2. Both of them manage large numbers of tasks, which are nothing but the instantiations of a small number of user-written functions.
3. Both of these provide the feature of fault tolerance, that handles the execution of a large job, without having to restart that job from the beginning.

In this topic, we will discuss "workflow" systems, that are nothing but the extension of MapReduce. The workflow system supports acyclic networks of functions, where each function is implemented by a collection of tasks. The Systems like UC Berkeley's Spark, Google's Tens or Flow have been implemented using the workflow system. The latest machine learning applications have workflow systems at heart.

2.7.1 Workflow Systems

The experimental system, Clustera from the University of Wisconsin and Hyracks from the University of California at Irvine has extended the map-reduce from the simple two-step workflow (the Map function feeds the Reduce function) to any collection of functions, with an acyclic graph representing workflow among the functions. The MapReduce is a two-step workflow in which the Map function feeds the Reduce function. Workflow systems extend MapReduce to any collection of functions, with an acyclic graph representing workflow among the functions. In Workflow systems, the workflow is represented with an acyclic flow graph, whose arcs a → b represents the fact that function a's output is an input to function b.

In the workflow system the data file containing the elements of one type is passed from one function to the next function. In case of single input, the function is applied to each input independently, same as that of the Map and Reduce functions are applied to their input elements individually. Each of these functions spits the output in the form of a file, that is generated after processing the input file. When a function has inputs from multiple files, elements from each of the files can be combined in various ways. But the function itself is applied to combinations of input elements, at most one from each input file.

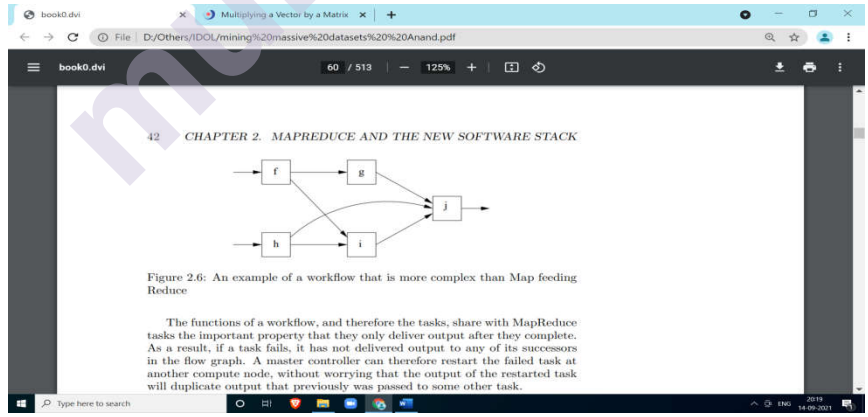


Figure 2.7: An example of a workflow that is more complex than Map feeding Reduce

Example2.6: The Figure 2.7 shows a workflow with five functions, f through j. Here the data is passed from left to right in such a way that the flow of data is acyclic and no tasks need to provide data out before getting its entire input. For example, the function h takes its input from a pre-existing file of the distributed file system. Then each output element of h is passed to the functions i and j. The function i takes the outputs of both f and h as inputs. The output of function j is

either passed to an application that invoked this dataflow or is stored in the distributed file.

The workflow systems are analogous to the MapReduce functions. So, in a workflow system each function of a workflow can be executed by many tasks where each of these functions is assigned a portion of the input. A master controller divides the work among the tasks that implement a function by hashing the input elements to decide on the proper task to receive an element. Same as that of the Map tasks, the workflow task that is implementing the function f has an output file of data, which is passed to each task implementing the successor function(s) f. After completing the task execution, the controller delivers these output files to the DFS.

Similar to MapReduce tasks, the workflow tasks follow the blocking property, in which they only deliver output after they complete. In case of task failure, it has not delivered output to any of its successors in the flow graph. To recover this failed task, a master controller restarts this task at another compute node, without worrying that the output of the restarted task will duplicate output that previously was passed to some other task

Some workflow systems applications effectively cascade the MapReduce jobs. For example, in the join of three relations, one MapReduce job joins the first two relations, and a second MapReduce job joins the third relation with the result of joining the first two relations.

The advantage of implementing cascades as a single workflow is that the master controller manages the flow of data among tasks, and its replication without storing the temporary file in the distributed file system whereas the MapReduce jobs stores output file in the distributed systems. By locating tasks at compute nodes that have a copy of their input, we can avoid much of the communication that would be necessary if we stored the result of one MapReduce job and then initiated a second MapReduce job. The Hadoop and other MapReduce systems also try to locate Map tasks where a copy of their input is already present.

The other popular extensions of MapReduce are Spark and Google's Tens or Flow, which has a workflow system at heart.

Spark

Spark uses a workflow system and provides the following advanced features:

1. A more efficient way to cope up with the failures.

2. A more efficient way of grouping tasks among compute nodes and scheduling execution of functions.

3. Integration of programming language features such as looping (which technically takes it out of the acyclic workflow class of systems) and function libraries.

Spark uses the central data abstraction, called the Resilient Distributed Dataset (RDD). RDD is a file of objects of one type. One of the examples of an RDD is the files of key-value pairs that are used in MapReduce systems or the files that get passed among functions of the workflow system as shown in Figure 2.7. The RDDs are normally broken into chunks that may be held at different compute nodes. The RDDs are "resilient" and are able to recover from the loss of any or all chunks of an RDD. Unlike the key-value-pair abstraction of MapReduce, there is no restriction on the type of the elements that comprise an RDD.

The Spark program performs the transformations and actions on the RDDs. A Sparks program consists of a sequence of steps. Each of these steps applies some function to an RDD to produce another RDD. These operations are also referred to as transformations. Some of the commonly used operations are Map, Flatmap, and Filter. Spark also allows to take data from the surrounding file system, such as Hadoop Distributed File System, and turn it into an RDD, and to take an RDD and return it to the surrounding file system or to produce a result that is passed back to an application that called a Spark program. Here the process of returning the RDD output to an application is also referred to as actions. In Spark, the Reduce operation is an action, not a transformation.

The Spark implementation differs from Hadoop or other MapReduce implementations. It uses lazy evaluation of RDD's and lineage for RDD's.

Tenso Flow

TensorFlow is an open-source system developed at Google to support machine-learning applications. Same as that of Spark, TensorFlow provides a programming interface in which one writes a sequence of steps. Programs are typically acyclic, although like Spark it is possible to iterate blocks of code.

One major difference between Spark and TensorFlow is the type of data that is passed between steps of the program. In place of the RDD, TensorFlow uses tensors; a tensor is simply a multidimensional matrix.

2.7.2 Recursive Extensions to MapReduce

Many large-scale computations like Google's search algorithm, Page Rank are recursive extensions of MapReduce. These are nothing but the computations of the fixed point of a matrix-vector multiplication that can be computed under MapReduce systems by the matrix-vector multiplication iterative algorithm. The iteration typically continues for an unknown number of steps, each step being a MapReduce job, until the results of two consecutive iterations are sufficiently close that we believe convergence has occurred.

Recursions present a problem for failure recovery. Recursive tasks inherently lack the blocking property necessary for independent restart of failed tasks. It is impossible for a collection of mutually recursive tasks, each of which has an output that is input to at least some of the other tasks, to produce output only at the end of the task. If they all followed that policy, no task would ever receive any input, and nothing could be accomplished. As a result, some mechanism

other than simple restart of failed tasks must be implemented in a system that handles recursive workflows (flow graphs that are not acyclic). We shall start by studying an example of a recursion implemented as a workflow, and then discuss approaches to dealing with task failures.

Example 2.7: Let us consider a directed graph with arcs, that are represented by the relation *E(X, Y)*. That means there is an arc from node *X* to node *Y*. Here we wish to compute the paths relation *P(X, Y),*that is a path from node *X* to node *Y*having of length 1 or more. The path *P* is the transitive closure of *E*. A simple recursive algorithm is:

1. Start with *P(X, Y) = E(X, Y).*
2. While changes to the relation *P* occur, add to *P* all tuples in

The above equation states that the pairs of nodes *X* and *Y* for some point *Z* are known to have the path from *X* to *Z* and from *Z* to *Y*.

Figure 2.8shows the organization of recursive tasks to be performed for this computation. There are two kinds of tasks: Join tasks and Dup-elim tasks. The figure 2.8 shows the some of the n tasks with the respective bucket of hash function h.

Once discovered, a path tuple *P(a, b)*, becomes input to two Join tasks that are numbered *h(a)* and *h(b)*. The job of the i^{th} Join task, when it receives input tuple *P(a, b)*, is to find certain other tuples seen previously (and stored locally by that task).

1. Store *P(a, b)* locally.

2. If h(a) = i then look for tuples P(x, a) and produce output tuple P(x, b).

3. If h(b) = i then look for tuples P(b, y) and produce output tuple P(a, y).

In rare cases, we have h(a) = h(b), so both steps (2) and (3) are executed. But generally, only one of these needs to be executed for a given tuple.

Also, Figure 2.8 shows m Dup-elim tasks with the corresponding bucket of hash function g with two arguments. The output of some Join task *P(c, d)* is then sent to Dup-elim task *j = g(c, d).*On receiving this tuple, the j^{th} Dup-elim task checks that it has not received this tuple before, since its job is duplicate elimination. If previously received, the tuple is ignored. But if this tuple is new, it is stored locally and sent to two Join tasks, those numbered h(c) and h(d).

Every Join task has m output files, one for each Dup-elim task. Every Dup-elim task has n output files, one for each Join task. These files are distributed according to any of several strategies. Initially, the *E(a, b)* tuples representing the arcs of the graph are distributed to the Dup-elim tasks, with *E(a, b)* being sent as *P(a, b)* to Dup-elim task *g(a, b).* The master controller waits until each Join task has processed its entire input for a round. Then, all output files are distributed to

the Dup-elim tasks, which create their own output. That output is distributed to the Join tasks and becomes their input for the next round.

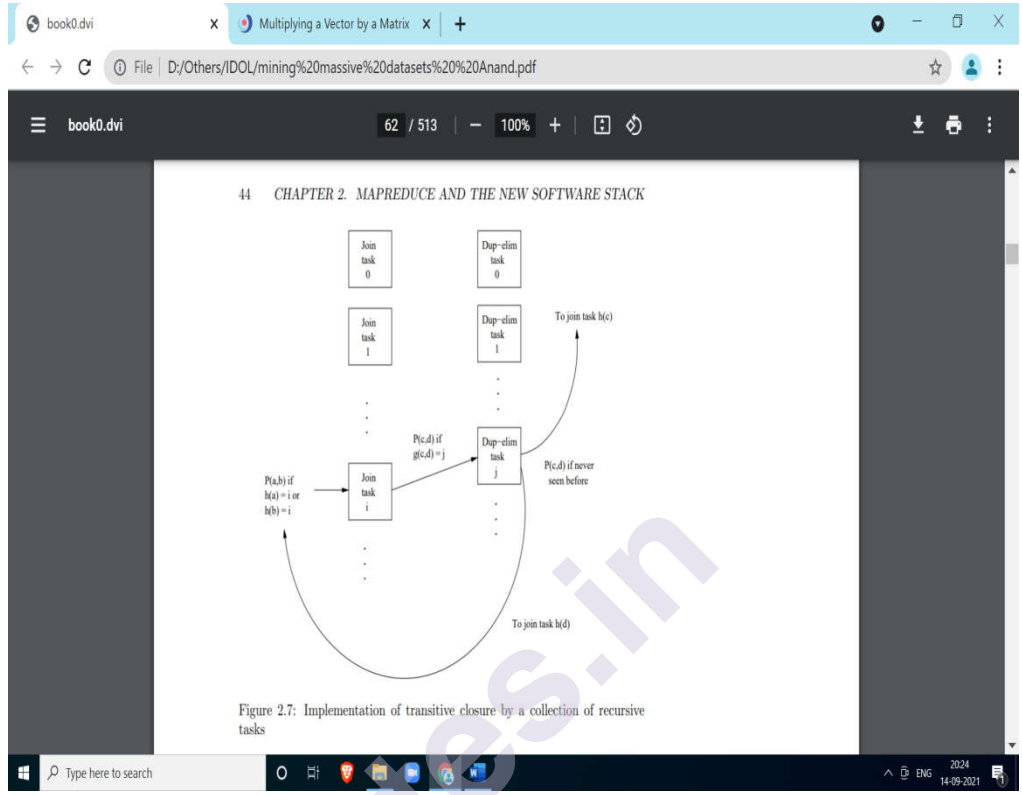


Figure 2.8: Implementation of transitive closure by a collection of recursive tasks

In Example 2.7 it is not necessary to have two kinds of tasks. Instead, Join tasks could eliminate duplicates as they are received, since they must store their previously received inputs anyway. This arrangement has an advantage when we must recover from a task failure. If each task stores all the output files it has ever created, and we place Join tasks on different racks from the Dup-elim tasks, then we can deal with any single compute node or single rack failure. That is, a Join task needing to be restarted can get all the previously generated inputs that it needs from the Dup-elim tasks, and vice versa.

In the specific case of computing transitive closure, it is not necessary to prevent a restarted task from generating outputs that the original task generated previously. In the computation of the transitive closure, the rediscovery of a path does not influence the eventual answer. However, many computations cannot tolerate a situation where both the original and restarted versions of a task pass the same output to another task. For example, if the final step of the computation were an aggregation, say a count of the number of nodes reached by each node in the graph, then we would get the wrong answer if we counted a path twice.

Let us discuss at least three different approaches that have been used to deal with failures while executing a recursive program.

1. Iterated MapReduce: Write the recursion as repeated execution of a MapReduce job or of a sequence of MapReduce jobs. In this case, to handle

failures at any step, we can then rely on the failure mechanism of the MapReduce implementation. The very first example of such a system was HaLoop.

2. The Spark Approach: The Spark language includes iterative statements, such as for-loops that allow the implementation of recursions. In Spark, failure management is implemented using the lazy-evaluation and lineage mechanisms. In addition to this the Spark programmer has options to store intermediate states of the recursion.

3. Bulk-Synchronous Systems: These systems use a graph-based model of computation. They typically use another resilience approach: periodic check pointing. One of the examples of bulk synchronous system is Pragel.

2.7.3 Pragel

Another approach that implements the recursive algorithms on a computing cluster is represented by Google's Pragel system. This System is the first example of a graph-based, bulk-synchronous system that processes massive amounts of data. This system views its data as a graph, where each node of the graph corresponds roughly to a task. Each graph node generates output messages that are destined for other nodes of the graph, and each graph node processes the inputs it receives from other nodes.

Example 2.8: Suppose our data is a collection of weighted arcs of a graph, and we want to find, for each node of the graph, the length of the shortest path to each of the other nodes. As the algorithm executes, each node a will store a set of pairs *(b, w),* where *w* is the length of the shortest path from node *a* to node *b* that is currently known.

Here first we need to store the set of pairs and weight for each graph node. For example, graph node a, stores the set of pairs *(b, w)* such that there is an arc from a to b of weight w. Then these facts are sent to all other nodes, as triples *(a, b, w)*, with the intended meaning that node *a* knows about a path of length *w* to node *b*. When the node a receives a triple (c, d, w), it must decide whether this fact implies a shorter path than *a* already knows about from itself to node *d*. Node a looks up its current distance to c; that is, it finds the pair (c, v) stored locally, if there is one. It also finds the pair (d, u) if there is one. If w + v < u, then the pair (d, u) is replaced by (d, w + v), and if there is no pair (d, u), then the pair (d, w + v) is stored at the node a. Also, the other nodes are sent the message (a, d, w + v) in either of these two cases.

In Pregel, the computations are organized into super steps. In one super step, all the messages that were received by any of the nodes at the previous super step are processed, and then all the messages generated by those nodes are sent to their destination. This approach of packaging many messages into one is referred to as "bulk-synchronous."

The bulk synchronous approach has reduced the overhead of sending many messages on the network. This is one of the very important advantages of the bulk synchronous approach.

Suppose that in Example 2.8 we sent a single new shortest-distance fact to the relevant node every time one was discovered. The number of messages sent would be enormous if the graph was large, and it would not be realistic to implement such an algorithm. However, in a bulk-synchronous system, a task that has the responsibility for managing many nodes of the graph can bundle together all the messages being sent by its nodes to any of the nodes being managed by another task. That choice typically saves orders of magnitude in the time required to send all the needed messages.

Failure Management in Pregel

In case of a compute-node failure, there is no attempt to restart the failed tasks at that compute node. Rather, Pregel checkpoints its entire computation after some of the super steps. A checkpoint consists of making a copy of the entire state of each task, so it can be restarted from that point if necessary. If any compute node fails, the entire job is restarted from the most recent checkpoint.

Although this recovery strategy causes many tasks that have not failed to redo their work, it is satisfactory in many situations. Recall that the reason MapReduce systems support restart of only the failed tasks is that we want assurance that the expected time to complete the entire job in the face of failures is not too much greater than the time to run the job with no failures. Any failure-management system will have that property as long as the time to recover from a failure is much less than the average time between failures. Thus, it is only necessary that Pregel checkpoints its computation after a number of super steps such that the probability of a failure during that number of super steps is low.

2.8 COMMON MAPREDUCE ALGORITHMS

The MapReduce implements the number of mathematical algorithms. Such algorithms divide a task into number of chunks and assign them to distributed nodes. These distributed nodes act as Map nodes and Reduce nodes, and executes the map and reduce tasks respectively. Some of the common mathematical algorithms are:

1. Sorting
2. Searching
3. Indexing
4. TF-IDF

2.8.1 Sorting

Sorting is one of the basic MapReduce algorithms, used to process and analyse data. MapReduce implements the sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys. The mapper class implements Sorting method. After tokenizing the values, during the Shuffle and Sort phase, the mapper class collects the matching valued keys as a collection. To collect similar intermediate key-value pairs, the Mapper class takes the help

of class to sort the key-value pairs. The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values (K2, {V2, V2, …}) before they are presented to the Reducer.

2.8.2 Searching

Searching plays an important role in the Map Reduce algorithm. It helps in the combiner phase and in the Reducer phase. The following example demonstrates the working of the searching algorithm.

Example 2.9: The example shows how MapReduce employs a Searching algorithm to find out the details of the employee who draws the highest salary in a given employee dataset.

Let us assume we have employee data in four different files A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly.

name, salary
santosh, 26000
Krish, 25000
Ajit, 15000
Jayant, 10000

name, salary
Harsh, 50000
Krish, 25000
Ajit, 15000
Jayant, 10000

name, salary
santosh, 26000
Tanmay, 45000
Ajit, 15000
Jayant, 10000

name, salary
santosh, 26000
Krish, 25000
Manasi, 45000
Jayant, 10000

Figure 2.9: Data of files A,B,C and D

The Map phase processes each input file and provides the employee data in key-value pairs (<k, v> :<emp name, salary>) as shown in Figure 2.10.

<santosh, 26000>
<Krish, 25000>
<Ajit, 15000>
<Jayant, 10000>

<Harsh, 50000>
<Krish, 25000>
<Ajit, 15000>
<Jayant, 10000>

<santosh, 26000>
<Tanmay, 45000>
<Ajit, 15000>
<Jayant, 10000>

<santosh, 26000>
<Krish, 25000>
<Manasi, 45000>
<Jayant, 10000>

Figure 2.10: Output of Map Process

The combiner phase (searching technique) will accept the input from the Map phase as a key-value pair with employee name and salary. Using searching technique, the combiner will check all the employee salary to find the highest salaried employee in each file. The expected result is as shown in figure 2.11.

<Satish, 26000>	<Harsh,50000>	<Tanmay, 45000>	<Mansi,45000>

Figure 2.11: Output of Combiner

Reducer phase - Form each file, you will find the highest salaried employee. To avoid redundancy, check all the <k, v> pairs and eliminate duplicate entries, if any. The same algorithm is used in between the four <k, v> pairs, which are coming from four input files. The final output should be as follows -

<Harsh, 50000>

2.8.3 Indexing

Normally indexing is used to point to a particular data and its address. It performs batch indexing on the input files for a particular Mapper.

The indexing technique that is normally used in MapReduce is known as **inverted index.** Search engines like Google and Bing use inverted indexing techniques. Let us try to understand how Indexing works with the help of a simple example.

Example 2.10 : The following text is the input for inverted indexing. Here T[0], T[1], and t[2] are the file names and their content are in double quotes.

T[0] = "it is what it is"

T[1] = "what is it"

T[2] = "it is a banana"

After applying the Indexing algorithm, we get the following output -

"a": {2}

"banana": {2}

"is": {0, 1, 2}

"it": {0, 1, 2}

"what": {0, 1}

Here "a": {2} implies the term "a" appears in the T[2] file. Similarly, "is": {0, 1, 2} implies the term "is" appears in the files T[0], T[1], and T[2].

2.8.4 TF-IDF

TF-IDF is a text processing algorithm which is short for Term Frequency − Inverse Document Frequency. It is one of the common web analysis algorithms. Here, the term 'frequency' refers to the number of times a term appears in a document.

Term Frequency (TF)

It measures how frequently a particular term occurs in a document. It is calculated by the number of times a word appears in a document divided by the total number of words in that document.

TF(the) = (Number of times term the ‘the’ appears in a document) / (Total number of terms in the document)

Inverse Document Frequency (IDF)

It measures the importance of a term. It is calculated by the number of documents in the text database divided by the number of documents where a specific term appears.

While computing TF, all the terms are considered equally important. That means, TF counts the term frequency for normal words like “is”, “a”, “what”, etc. Thus, we need to know the frequent terms while scaling up the rare ones, by computing the following -

IDF(the) = log_e(Total number of documents / Number of documents with term ‘the’ in it).

2.9 SUMMARY

- The common architecture, cluster of compute nodes, is used to process very large-scale applications.
- The Distributed File Systems architecture is used to store and process the large data files on distributed nodes.
- The MapReduce framework processes the data parallelly on the DFS with the help of cluster nodes like Master node, Map node and Reduce node and so on.
- The Map and Reduce functions are problem specific and need to be designed by the user.
- The Map and Reduce functions generate the output in Key-value pair formats. The Map function stores output in the intermediatory file whereas the Reduce function stores the final output file.
- Apache Hadoop is the open-source implementation of a Distributed File System also referred as HDFS.
- The MapReduce framework is fault tolerant and manages the faults of Master, Map and Reduce nodes.
- MapReduce is not suitable for all parallel algorithms. The Simple implementations like, Matrix-Vector multiplication, Matrix-Matrix Multiplications, Principal operators of linear algebra can be done in MapReduce.
- MapReduce is generalized to the systems, supporting any acyclic collection of functions, which are referred to as workflow systems. Each of these functions can be instantiated by any number of tasks that are responsible for executing that function on a portion of the data.
- In case of recursive workflows, it is not possible to restart the whole task again. Instead, a number of checkpointing parts of the computation allows restart of single task. You can also restart all tasks from a recent checkpoint has been proposed.
- The MapReduce algorithms can be implemented by using any of the programming languages like, Java, Python and so on. The MapReduce algorithms are generally written for large-scale data.

2.10 LIST OF REFERENCES AND BIBLIOGRAPHY

1. Mining of Massive Datasets, Anand Rajaraman and Jeffrey David Ullman, Cambridge University Press, 2012.
2. Big Data, Big Analytics: Emerging Business Intelligence and Analytic Trends forToday's Businesses, Michael Minelli, Wiley, 2013.
3. https://www.ibm.com/topics/mapreduce
4. https://www.dcs.bbk.ac.uk/~dell/teaching/cc/book/mmds/mmds_ch2n_4.pdf
5. https://www.tutorialspoint.com/map_reduce/map_reduce_algorithm. htm
6. https://stanford.edu/~rezab/amdm/notes/lecture4.pdf
7. https://www.cdac.in/index.aspx?id=ev_hpc_hadoop-map-reduce#hadoop-map-reduce-par-prog-id12

2.11 UNIT END EXERCISE

1. Distributed File System? How is DFS extended in the Hadoop Distributed File System?
2. What is a Distributed File System? How does the system store file of large size on DFS?
3. What is Apache Hadoop? What are the characteristics of a Distributed File System?
4. What is the Hadoop ecosystem? Discuss the various elements of the Hadoop Ecosystem?
5. What is MapReduce? What are the advantages of MapReduce?
6. Explain the steps of execution of MapReduce.
7. Describe the Map task and Reduce task with an example for each.
8. What is the role of mapper function and combiner function in MapReduce?
9. What is the role of a Master node? How does the master role control the failure of a task or a node?
10. Explain the steps of execution for word count algorithm with an example.
11. Explain the Matrix-Vector multiplication algorithm with an example.
12. How does the MapReduce algorithm handle the vector of large size?
13. What are relational algebra operations? Explain each operation in brief.
14. How does MapReduce handle the selection and projection operations computing? Explain the role of Map and Reduce tasks and an example for each.
15. Explain union, intersection and NaturalJoin computing operations of MapReduce.
16. What are the characteristics of the MapReduce System? How is the MapReduce framework extended to the workflow system?
17. Explain the function of the workflow system with an example.

18. What is the purpose of the workflow system? Discuss any two examples of workflow systems.

19. What do mean by recursive extension of MapReduce? Describe the process of transitive closure for the number of recursive tasks.

20. Discuss the various approaches of handling the failure of recursive MapReduce tasks?

21. Describe the Bulk-Synchronous System - Pregel with an example.

22. Discuss any three common MapReduce algorithms.

23. Write a program to implement the matrix-multiplication algorithm using any one programming language.

3

SHINGLING OF DOCUMENTS

Unit Structure

3.0 OBJECTIVES

We will study how to define the distance between sets. To illustrate and motivate this study, we will focus on using Jaccard distance to measure the distance between documents. This uses the common "bag of words" model, which is simplistic, but is sufficient for many applications. We start with some big questions. This lecture will only begin to answer them.

• Given two homework assignments (reports) how can a computer detect if one is likely to have been plagiarized from the other without understanding the content?

• In trying to index webpages, how does Google avoid listing duplicates or mirrors?

• How does a computer quickly understand emails, for either detecting spam or placing effective advertisers? (If an ad worked on one email, how can we determine which others are similar?)

The key to answering these questions will be convert the data (homeworks, webpages, emails) into an object in an abstract space that we know how to measure distance, and how to do it efficiently.

3.1 INTRODUCTION

In data mining large number of dataset is finding similar items. As an example, finding similar documents can be recommended. In this case many methods are existed. For example, Shingling method and length based filtering are one of them.

In Shingling method, from each document, substrings have been selected with symbol name and, they are placed on one set. For finding similar documents, the similarities of sets that related with them have been calculated. In Length based filtering just documents which close these lengths have been compared. These methods don't consider repetition of symbols. With considering the repetition can calculate length of documents with more accurately.

In this paper we suggested a method for finding similar documents with considering the repetition of symbols. This method separated documents to better form. The main goal of this a method for finding similar documents with take fewer comparisons and time indeed.

3.2 FINDING SIMILAR ITEMS

A fundamental data-mining problem is to examine data for "similar" items. We shall take up applications in Section 3.1, but an example would be looking at a collection of Web pages and finding near-duplicate pages. These pages could beplagiarisms, for example, or they could be mirrors that have almost the same content but differ in information about the host and about other mirrors.

We begin by phrasing the problem of similarity as one of finding sets with a relatively large intersection. We show how the problem of finding textually similar documents can be turned into such a set problem by the technique known as "shingling." Then, we introduce a technique called "minhashing," which compresses large sets in such a way that we can still deduce the similarity of the underlying sets from their compressed versions. Other techniques that work when the required degree of similarity is very high are covered in Section 3.9.

Another important problem that arises when we search for similar items of any kind is that there may be far too many pairs of items to test each pair for their degree of similarity, even if computing the similarity of any one pair can be made very easy. That concern motivates a technique called "locality-sensitive hashing," for focusing our search on pairs that are most likely to be similar.

Finally, we explore notions of "similarity" that are not expressible as inter- section of sets. This study leads us to consider the theory of

distance measures in arbitrary spaces. It also motivates a general framework for locality-sensitive hashing that applies for other definitions of "similarity."

3.3 APPLICATIONS OF NEAR-NEIGHBOR SEARCH

We shall focus initially on a particular notion of "similarity": the similarity of sets by looking at the relative size of their intersection. This notion of similarity is called "Jaccard similarity," and will be introduced in Section 3.1.1. We then examine some of the uses of finding similar sets. These include finding textually similar documents and collaborative filtering by finding similar customers and similar products. In order to turn the problem of textual similarity of documents into one of set intersection, we use a technique called "shingling," which is introduced in Section 3.2.

3.3.1 Jaccard Similarity of Sets

The *Jaccard similarity* of sets S and T is $S \cap T / S \cup T$, that is, the ratio of the size of the intersection of S and T to the size of their union. We shall denote the Jaccard similarity of S and T by SIM (S, T).

Example 3.1: In Fig. 3.1 we see two sets S and T. There are three elements in their intersection and a total of eight elements that appear in S or T or both. Thus, SIM(S, T) = 3/8. ✷

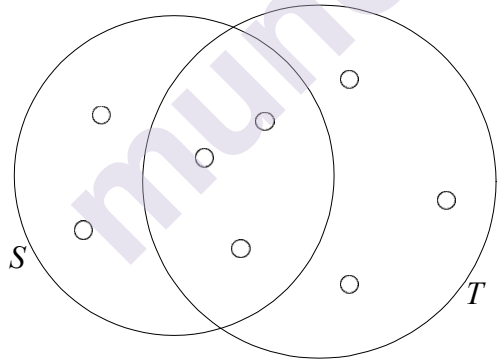


Figure 3.1: Two sets with Jaccard similarity 3/8

3.3.2 Similarity of Documents

An important class of problems that Jaccard similarity addresses well is that of finding textually similar documents in a large corpus such as the Web or a collection of news articles. We should understand that the aspect of similarity we are looking at here is character-level similarity, not "similar meaning," which requires us to examine the words in the documents and their uses. That problem is also interesting but is addressed by other techniques, which we hinted at in Section 1.3.1. However, textual similarity also has important uses. Many of these involve finding duplicates or near duplicates. First, let us observe

that testing whether two documents are exact duplicates is easy; just compare the two documents character-by-character, and if they ever differ then they are not the same. However, in many applications, the documents are not identical, yet they share large portions of their text. Here are some examples:

APPLICATIONS OF NEAR-NEIGHBOR SEARCH

Plagiarism

Finding plagiarized documents tests our ability to find textual similarity. The plagiarizer may extract only some parts of a document for his own. He may alter a few words and may alter the order in which sentences of the original appear. Yet the resulting document may still contain 50% or more of the original. No simple process of comparing documents character by character will detect a sophisticated plagiarism.

Mirror Pages

It is common for important or popular Web sites to be duplicated at a number of hosts, in order to share the load. The pages of these *mirror* sites will be quite similar, but are rarely identical. For instance, they might each contain information associated with their particular host, and they might each have links to the other mirror sites but not to themselves. A related phenomenon is the appropriation of pages from one class to another. These pages might include class notes, assignments, and lecture slides. Similar pages might change the name of the course, year, and make small changes from year to year. It is important to be able to detect similar pages of these kinds, because search engines produce better results if they avoid showing two pages that are nearly identical within the first page of results.

Articles from the Same Source

It is common for one reporter to write a news article that gets distributed, say through the Associated Press, to many newspapers, which then publish the article on their Web sites. Each newspaper changes the article somewhat. They may cut out paragraphs, or even add material of their own. They most likely will surround the article by their own logo, ads, and links to other articles at their site. However, the core of each newspaper's page will be the original article. News aggregators, such as Google News, try to find all versions of such an article, in order to show only one, and that task requires finding when two Web pages are textually similar, although not identical.[1]

3.3.3 Collaborative Filtering as a Similar-Sets Problem

Another class of applications where similarity of sets is very important is called *collaborative filtering*, a process whereby we recommend to users items that were liked by other users who have exhibited similar tastes. We shall investigate collaborative filtering in detail in Section

9.3, but for the moment let us see some common examples.

[1]News aggregation also involves finding articles that are about the same topic, even though not textually similar. This problem too can yield to a similarity search, but it requires techniques other than Jaccard similarity of sets.

On-Line Purchases

Amazon.com has millions of customers and sells millions of items. Its database records which items have been bought by which customers. We can say two cus- tomers are similar if their sets of purchased items have a high Jaccard similarity. Likewise, two items that have sets of purchasers with high Jaccard similarity will be deemed similar. Note that, while we might expect mirror sites to have Jaccard similarity above 90%, it is unlikely that any two customers have Jac- card similarity that high (unless they have purchased only one item). Even a Jaccard similarity like 20% might be unusual enough to identify customers with similar tastes. The same observation holds for items; Jaccard similarities need not be very high to be significant.

Collaborative filtering requires several tools, in addition to finding similar customers or items, as we discuss in Chapter 9. For example, two Amazon customers who like science-fiction might each buy many science-fiction books, but only a few of these will be in common. However, by combining similarity- finding with clustering (Chapter 7), we might be able to discover that science- fiction books are mutually similar and put them in one group. Then, we can get a more powerful notion of customer-similarity by asking whether they made purchases within many of the same groups.

Movie Ratings

Netflix records which movies each of its customers rented, and also the ratings assigned to those movies by the customers. We can see movies as similar if they were rented or rated highly by many of the same customers, and see customers as similar if they rented or rated highly many of the same movies. The same observations that we made for Amazon above apply in this situation: similarities need not be high to be significant, and clustering movies by genre will make things easier.

When our data consists of ratings rather than binary decisions (bought/did not buy or liked/disliked), we cannot rely simply on sets as representations of customers or items. Some options are:

3.3.3.1 Ignore low-rated customer/movie pairs; that is, treat these events as if the customer never watched the movie.

3.3.3.2 When comparing customers, imagine two set elements for each movie, “liked” and “hated.” If a customer rated a movie highly, put the “liked” for that movie in the customer’s set. If they gave a low rating to a movie, put “hated” for that movie in their set. Then, we can

look for high Jaccard similarity among these sets. We can do a similar trick when comparing movies.

3.3.3.3 If ratings are 1-to-5-stars, put a movie in a customer's set n times if they rated the movie n-stars. Then, use *Jaccard similarity for bags* when measuring the similarity of customers. The Jaccard similarity for bags B and C is defined by counting element n times in the intersection if n is the minimum of the number of times the element appears in B and C. In the union, we count the element the sum of the number of times it appears in B and in C.[2]

Example 3.2: The bag-similarity of bags {a, a, a, b} and {a, a, b, b, c} is 1/3. The intersection counts a twice and b once, so its size is 3. The size of the union of two bags is always the sum of the sizes of the two bags, or 9 in this case. Since the highest possible Jaccard similarity for bags is 1/2, the score of 1/3 indicates the two bags are quite similar, as should be apparent from an examination of their contents.

3.3.4 Exercises for Section 3.1

Exercise 3.1.1: Compute the Jaccard similarities of each pair of the following three sets: $\{1, 2, 3, 4\}$, $\{2, 3, 5, 7\}$, and $\{2, 4, 6\}$.

Exercise 3.1.2: Compute the Jaccard bag similarity of each pair of the fol- lowing three bags: $\{1, 1, 1, 2\}$, $\{1, 1, 2, 2, 3\}$, and $\{1, 2, 3, 4\}$.

!! Exercise 3.1.3: Suppose we have a universal set U of n elements, and we choose two subsets S and T at random, each with m of the n elements. What is the expected value of the Jaccard similarity of S and T?

3.4 SHINGLING OF DOCUMENTS

The most effective way to represent documents as sets, for the purpose of iden- tifying lexically similar documents is to construct from the document the set of short strings that appear within it. If we do so, then documents that share pieces as short as sentences or even phrases will have many common elements in their sets, even if those sentences appear in different orders in the two docu- ments. In this section, we introduce the simplest and most common approach, shingling, as well as an interesting variation.

3.4.1 *k*-Shingles

A document is a string of characters. Define a k-shingle for a document to be any substring of length k found within the document. Then, we may associate with each document the set of k-shingles that appear one or more times within that document.

Example 3.3: Suppose our document D is the string abcdabd, and we pick k = 2. Then the set of 2-shingles for D is {ab, bc, cd, da, and bd}.

Note that the substring ab appears twice within D, but appears only once as a shingle. A variation of shingling produces a bag, rather than a set, so each shingle would appear in the result as many times as it appears in the document. However, we shall not use bags of shingles here.

There are several options regarding how white space (blank, tab, newline, etc.) is treated. It probably makes sense to replace any sequence of one or more white-space characters by a single blank. That way, we distinguish shingles that cover two or more words from those that do not.

Example 3.4: If we use k = 9, but eliminate whitespace altogether, then we would see some lexical similarity in the sentences "The plane was ready for touch down" and "The quarterback scored a touchdown". However, if we retain the blanks, then the first has shingles touch dow and ouch down, while the second has touchdown. If we eliminated the blanks, then both would have touchdown.

3.4.2 Choosing the Shingle Size

We can pick k to be any constant we like. However, if we pick k too small, then we would expect most sequences of k characters to appear in most documents. If so, then we could have documents whose shingle-sets had high Jaccard simi- larity, yet the documents had none of the same sentences or even phrases. As an extreme example, if we use k = 1, most Web pages will have most of the common characters and few other characters, so almost all Web pages will have high similarity.

- How large k should be depends on how long typical documents are and how large the set of typical characters is. The important thing to remember is: k should be picked large enough that the probability of any given shingle appearing in any given document is low.

Thus, if our corpus of documents is emails, picking k = 5 should be fine. To see why, suppose that only letters and a general white-space character ap- pear in emails (although in practice, most of the printable ASCII characters can be expected to appear occasionally). If so, then there would be $27^5 = 14{,}348{,}907$ possible shingles. Since the typical email is much smaller than 14 million characters long, we would expect k = 5 to work well, and indeed it does. However, the calculation is a bit more subtle. Surely, more than 27 charac- ters appear in emails, However, all characters do not appear with equal proba- bility. Common letters and blanks dominate, while"z" and other letters that have high point-value in Scrabble are rare. Thus, even short emails will have many 5-shingles consisting of common letters, and the chances of unrelated emails sharing these common shingles are greater than would be implied by the calculation in the paragraph above. A good rule of thumb is to imagine that there are only 20 characters and estimate the number of k-shingles as 20^k. For large documents, such as research articles, choice k = 9 is considered safe.

3.4.3 Hashing Shingles

Instead of using substrings directly as shingles, we can pick a hash function that maps strings of length k to some number of buckets and treat the resulting bucket number as the shingle. The set representing a document is then the set of integers that are bucket numbers of one or more k-shingles that appear in the document. For instance, we could construct the set of 9-shingles for a document and then map each of those 9-shingles to a bucket number in the range 0 to 2^{32} -1. Thus, each shingle is represented by four bytes instead of nine. Not only has the data been compacted, but we can now manipulate (hashed) shingles by single-word machine operations.

Notice that we can differentiate documents better if we use 9-shingles and hash them down to four bytes than to use 4-shingles, even though the space used to represent a shingle is the same. The reason was touched upon in Section 3.2.2. If we use 4-shingles, most sequences of four bytes are unlikely or impossible to find in typical documents. Thus, the effective number of different shingles is much less than 2^{32} -1. If, as in Section 3.2.2, we assume only 20 characters are frequent in English text, then the number of different 4-shingles that are likely to occur is only $(20)^4 = 160{,}000$. However, if we use 9-shingles, there are many more than 2^{32} likely shingles. When we hash them down to four bytes, we can expect almost any sequence of four bytes to be possible, as was discussed in Section 1.3.2.

3.4.4 Shingles Built from Words

An alternative form of shingle has proved effective for the problem of identifying similar news articles, mentioned in Section 3.1.2. The exploitable distinction for this problem is that the news articles are written in a rather different style than are other elements that typically appear on the page with the article. News articles, and most prose, have a lot of stop words (see Section 1.3.1), the most common words such as “and,” “you,” “to,” and so on. In many applications, we want to ignore stop words, since they don’t tell us anything useful about the article, such as its topic.

However, for the problem of finding similar news articles, it was found that defining a shingle to be a stop word followed by the next two words, regardless of whether or not they were stop words, formed a useful set of shingles. The advantage of this approach is that the news article would then contribute more shingles to the set representing the Web page than would the surrounding ele-ments. Recall that the goal of the exercise is to find pages that had the same articles, regardless of the surrounding elements. By biasing the set of shingles in favor of the article, pages with the same article and different surrounding material have higher Jaccard similarity than pages with the same surrounding material but with a different article.

Example 3.5: An ad might have the simple text “Buy Sudzo.” However, a news article with the same idea might read something

like "*A* spokesperson *for the* Sudzo Corporation revealed today *that* studies *have* shown *it is* good *for* people *to* buy Sudzo products." Here, we have italicized all the likely stop words, although there is no set number of the most frequent words that should be considered stop words. The first three shingles made from a stop word and the next two following are:

A spokesperson for

for the Sudzo

the Sudzo Corporation

There are nine shingles from the sentence, but none from the "ad."

3.4.5 Exercises for Section 3.2

Exercise 3.2.1 : What are the first ten 3-shingles in the first sentence of Sec- tion 3.2?

Exercise 3.2.2 : If we use the stop-word-based shingles of Section 3.2.4, and we take the stop words to be all the words of three or fewer letters, then what are the shingles in the first sentence of Section 3.2?

Exercise 3.2.3 : What is the largest number of k-shingles a document of n bytes can have? You may assume that the size of the alphabet is large enough that the number of possible strings of length k is at least as n.

3.5 SIMILARITY-PRESERVING SUMMARIES OF SETS

Sets of shingles are large. Even if we hash them to four bytes each, the space needed to store a set is still roughly four times the space taken by the document. If we have millions of documents, it may well not be possible to store all the shingle-sets in main memory.[3]

Our goal in this section is to replace large sets by much smaller represen- tations called "signatures." The important property we need for signatures is that we can compare the signatures of two sets and estimate the Jaccard sim- ilarity of the underlying sets from the signatures alone. It is not possible that the similarity of each pair. We take up the solution to this problem in Section 3.4. the signatures give the exact similarity of the sets they represent, but the esti- mates they provide are close, and the larger the signatures the more accurate the estimates. For example, if we replace the 200,000-byte hashed-shingle sets that derive from 50,000-byte documents by signatures of 1000 bytes, we can usually get within a few percent.

3.5.1 Matrix Representation of Sets

Before explaining how it is possible to construct small signatures from large sets, it is helpful to visualize a collection of sets as their

characteristic matrix. The columns of the matrix correspond to the sets, and the rows correspond to elements of the universal set from which elements of the sets are drawn. There is a 1 in row r and column c if the element for row r is a member of the set forcolumn c. Otherwise the value in position (r, c) is 0.

Element	S_1	S_2	S_3	S_4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

Figure 3.2: A matrix representing four sets

Example 3.6: In Fig. 3.2 is an example of a matrix representing sets chosen from the universal set {a, b, c, d, e}. Here, S_1 = {a, d}, S_2 = {c}, S_3 = {b, d, e}, and S_4 = a, c, d . The top row and leftmost columns are not part of the matrix, but are present only to remind us what the rows and columns represent.

It is important to remember that the characteristic matrix is unlikely to be the way the data is stored, but it is useful as a way to visualize the data. For one reason not to store data as a matrix, these matrices are almost always *sparse* (they have many more 0's than 1's) in practice. It saves space to represent a sparse matrix of 0's and 1's by the positions in which the 1's appear. For another reason, the data is usually stored in some other format for other purposes.

As an example, if rows are products, and columns are customers, represented by the set of products they bought, then this data would really appear in a database table of purchases. A tuple in this table would list the item, the purchaser, and probably other details about the purchase, such as the date and the credit card used.

3.5.2 Minhashing

The signatures we desire to construct for sets are composed of the results of a large number of calculations, say several hundred, each of which is a "minhash" of the characteristic matrix. In this section, we shall learn how a minhash is computed in principle, and in later sections we shall see how a good approxi- mation to the minhash is computed in practice.

To *minhash* a set represented by a column of the characteristic matrix, pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1.

Example 3.7 : Let us suppose we pick the order of rows beadc for the matrix of Fig. 3.2. This permutation defines a minhash function h that maps sets to rows. Let us compute the minhash value of set S_1 according to h. The first column, which is the column for set S_1, has 0 in row b, so we proceed to row e, the second in the permuted order. There is again a 0 in the column for S_1, so we proceed to row a, where we find a 1. Thus. $h(S_1) = a$.

Element	S_1	S_2	S_3	S_4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

Figure 3.3: A permutation of the rows of Fig. 3.2

Although it is not physically possible to permute very large characteristic matrices, the minhash function h implicitly reorders the rows of the matrix of Fig. 3.2 so it becomes the matrix of Fig. 3.3. In this matrix, we can read off the values of h by scanning from the top until we come to a 1. Thus, we see that $h(S_2) = c$, $h(S_3) = b$, and $h(S_4) = a$.

3.5.3 Minhashing and Jaccard Similarity

There is a remarkable connection between minhashing and Jaccard similarity of the sets that are minhashed.

The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets.

To see why, we need to picture the columns for those two sets. If we restrict ourselves to the columns for sets S_1 and S_2, then rows can be divided into three classes:

3.5.3.1 Type X rows have 1 in both columns.

3.5.3.2 Type Y rows have 1 in one of the columns and 0 in the other.

3.5.3.3 Type Z rows have 0 in both columns.

Since the matrix is sparse, most rows are of type Z. However, it is the ratio of the numbers of type X and type Y rows that determine both $SIM(S_1, S_2)$ and the probability that $h(S_1) = h(S_2)$. Let there be x rows of type X and y rows of type Y . Then $SIM(S_1, S_2) = x/(x + y)$. The reason is that x is the size of $S_1 \cap S_2$ and x + y is the size of $S_1 \cup S_2$.

Now, consider the probability that $h(S_1) = h(S_2)$. If we imagine the rows permuted randomly, and we proceed from the top, the probability that we shall meet a type X row before we meet a type Y row is $x/(x + y)$. But if the first row from the top other than type Z rows is a type X row, then surely $h(S_1) = h(S_2)$. On the other hand, if the first row other than a type Z row that we meet is a type Y row, then the set with a 1 gets that row as its minhash value. However the set with a 0 in that row surely gets some row further down the permuted list. Thus, we know $h(S_1) /= h(S_2)$ if we first meet a type Y row. We conclude the probability that $h(S_1) = h(S_2)$ is $x/(x + y)$, which is also the Jaccard similarity of S_1 and S_2.

3.5.4 Minhash Signatures

Again think of a collection of sets represented by their characteristic matrix M. To represent sets, we pick at random some number n of permutations of the rows of M. Perhaps 100 permutations or several hundred permutations will do. Call the minhash functions determined by these permutations $h_1, h_2, \ldots, h_n$. From the column representing set S, construct the *minhash signature* for S, the vector $[h_1(S), h_2(S), \ldots, h_n(S)]$. We normally represent this list of hash-values as a column. Thus, we can form from matrix M a *signature matrix*, in which the ith column of M is replaced by the minhash signature for (the set of) the ith column.

Note that the signature matrix has the same number of columns as M but only n rows. Even if M is not represented explicitly, but in some compressed form suitable for a sparse matrix (e.g., by the locations of its 1's), it is normal for the signature matrix to be much smaller than M.

3.5.5 Computing Minhash Signatures

It is not feasible to permute a large characteristic matrix explicitly. Even picking a random permutation of millions or billions of rows is time-consuming, and the necessary sorting of the rows would take even more time. Thus, permuted matrices like that suggested by Fig. 3.3, while conceptually appealing, are not implementable.

Fortunately, it is possible to simulate the effect of a random permutation by a random hash function that maps row numbers to as many buckets as there are rows. A hash function that maps integers 0, $1, \ldots, k - 1$ to bucket numbers 0 through k 1 typically will map some pairs of integers to the same bucket and leave other buckets unfilled. However, the difference is unimportant as long as k is large and there are not too many collisions. We can maintain the fiction that our hash function h "permutes" row r to position h(r) in the permuted order.

Thus, instead of picking n random permutations of rows, we pick n randomly chosen hash functions $h_1, h_2, \ldots, h_n$ on the rows. We construct the signature matrix by considering each row in their given order. Let SIG(i, c) be the element of the signature matrix for the ith

hash function and column c. Initially, set SIG(i, c) to ∞ for all i and c. We handle row r by doing the following:

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$.

2. For each column c do the following:

(a) If c has 0 in row r, do nothing.

(b) However, if c has 1 in row r, then for each i = 1, 2, . . . , n set SIG(i, c)to the smaller of the current value of SIG(i, c) and $h_i(r)$.

Row	S_1	S_2	S_3	S_4	x + 1 mod 5	3x + 1 mod 5
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

Figure 3.4: Hash functions computed for the matrix of Fig. 3.2

Example 3.8 : Let us reconsider the characteristic matrix of Fig. 3.2, which we reproduce with some additional data as Fig. 3.4. We have replaced the letters naming the rows by integers 0 through 4. We have also chosen two hash functions: $h_1(x) = x+1 \mod 5$ and $h_2(x) = 3x+1 \mod 5$. The values of these two functions applied to the row numbers are given in the last two columns of Fig. 3.4. Notice that these simple hash functions are true permutations of the rows, but a true permutation is only possible because the number of rows, 5, is a prime. In general, there will be collisions, where two rows get the same hash value.

Now, let us simulate the algorithm for computing the signature matrix.

Initially, this matrix consists of all ∞'s:

	S_1	S_2	S_3	S_4
h_1	∞	∞	∞	∞
h_2	∞	∞	∞	∞

First, we consider row 0 of Fig. 3.4. We see that the values of $h_1(0)$ and $h_2(0)$ are both 1. The row numbered 0 has 1's in the columns for sets S_1 and S_4, so only these columns of the signature

matrix can change. As 1 is less than, we do in fact change both values in the columns for S_1 and S_4. The current estimate of the signature matrix is thus:

	S_1	S_2	S_3	S_4
h_1	1	∞	∞	1
h_2	1	∞	∞	1

Now, we move to the row numbered 1 in Fig. 3.4. This row has 1 only in S_3, and its hash values are $h_1(1) = 2$ and $h_2(1) = 4$. Thus, we set SIG(1, 3) to 2 and SIG(2, 3) to 4. All other signature entries remain as they are because their columns have 0 in the row numbered 1. The new signature matrix:

	S_1	S_2	S_3	S_4
h_1	1	∞	2	1
h_2	1	∞	4	1

The row of Fig. 3.4 numbered 2 has 1's in the columns for S_2 and S_4, and its hash values are $h_1(2) = 3$ and $h_2(2) = 2$. We could change the values in the signature for S_4, but the values in this column of the signature matrix, [1, 1], are each less than the corresponding hash values [3, 2]. However, since the column for S_2 still has ∞'s, we replace it by [3, 2], resulting in:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	1	2	4	1

Next comes the row numbered 3 in Fig. 3.4. Here, all columns but S_2 have 1, and the hash values are $h_1(3) = 4$ and $h_2(3) = 0$. The value 4 for h_1 exceeds what is already in the signature matrix for all the columns, so we shall not change any values in the first row of the signature matrix. However, the value 0 for h_2 is less than what is already present, so we lower SIG(2, 1), SIG(2, 3) and SIG(2, 4) to 0. Note that we cannot lower SIG(2, 2) because the column for S_2 in Fig. 3.4 has 0 in the row we are currently considering. The resulting signature matrix:

	S_1	S_2	S_3	S_4
h_1	1	3	2	1
h_2	0	2	0	0

Finally, consider the row of Fig. 3.4 numbered 4. $h_1(4) = 0$ and $h_2(4) = 3$. Since row 4 has 1 only in the column for S_3, we only compare the current signature column for that set, [2, 0] with the hash values [0, 3]. Since $0 < 2$, we change SIG(1, 3) to 0, but since $3 > 0$ we do not change SIG(2, 3). The final signature matrix is:

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

We can estimate the Jaccard similarities of the underlying sets from this signature matrix. Notice that columns 1 and 4 are identical, so we guess that $SIM(S_1, S_4) = 1.0$. If we look at Fig. 3.4, we see that the true Jaccard similarity of S_1 and S_4 is 2/3. Remember that the fraction of rows that agree in the signature matrix is only an estimate of the true Jaccard similarity, and this example is much too small for the law of large numbers to assure that the estimates are close. For additional examples, the signature columns for S_1 and S_3 agree in half the rows (true similarity 1/4), while the signatures of S_1 and S_2 estimate 0 as their Jaccard similarity (the correct value).

3.5.6 Exercises for Section 3.3

Exercise 3.3.1 : Verify the theorem from Section 3.3.3, which relates the Jac- card similarity to the probability of minhashing to equal values, for the partic- ular case of Fig. 3.2.

(a) Compute the Jaccard similarity of each of the pairs of columns in Fig. 3.2.

! (b) Compute, for each pair of columns of that figure, the fraction of the 120 permutations of the rows that make the two columns hash to the same value.

Exercise 3.3.2 : Using the data from Fig. 3.4, add to the signatures of the columns the values of the following hash functions:

(a) $h_3(x) = 2x + 4 \mod 5$.

(b) $h_4(x) = 3x - 1 \mod 5$.

Element	S_1	S_2	S_3	S_4
0	0	1	0	1
1	0	1	0	0
2	1	0	0	1
3	0	0	1	0
4	0	0	1	1
5	1	0	0	0

Figure 3.5: Matrix for Exercise 3.3.3

Exercise 3.3.3 : In Fig. 3.5 is a matrix with six rows.

(a) Compute the minhash signature for each column if we use the following three hash functions: $h_1(x) = 2x + 1 \mod 6$; $h_2(x) = 3x + 2 \mod 6$; $h_3(x) = 5x + 2 \mod 6$.

(b) Which of these hash functions are true permutations?

(c) How close are the estimated Jaccard similarities for the six pairs of columnsto the true Jaccard similarities?

! Exercise 3.3.4 : Now that we know Jaccard similarity is related to the proba- bility that two sets minhash to the same value, reconsider Exercise 3.1.3. Can you use this relationship to simplify the problem of computing the expected Jaccard similarity of randomly chosen sets?

! Exercise 3.3.5 : Prove that if the Jaccard similarity of two columns is 0, then minhashing always gives a correct estimate of the Jaccard similarity.

!! Exercise 3.3.6 : One might expect that we could estimate the Jaccard simi- larity of columns without using all possible permutations of rows. For example, we could only allow cyclic permutations; i.e., start at a randomly chosen row r, which becomes the first in the order, followed by rows r + 1, r + 2, and so on, down to the last row, and then continuing with the first row, second row, and so on, down to row r 1. There are only n such permutations if there are n rows. However, these permutations are not sufficient to estimate the Jaccard similarity correctly. Give an example of a two-column matrix where averaging over all the cyclic permutations does not give the Jaccard similarity.

! Exercise 3.3.7 : Suppose we want to use a MapReduce framework to compute minhash signatures. If the matrix is stored in chunks that correspond to some columns, then it is quite easy to exploit parallelism. Each Map task gets some of the columns and all the hash

functions, and computes the minhash signatures of its given columns. However, suppose the matrix were chunked by rows, so that a Map task is given the hash functions and a set of rows to work on. Design Map and Reduce functions to exploit MapReduce with data in this form.

3.6 LOCALITY-SENSITIVE HASHING FOR DOCUMENTS

Even though we can use minhashing to compress large documents into small signatures and preserve the expected similarity of any pair of documents, it still may be impossible to find the pairs with greatest similarity efficiently. The reason is that the number of pairs of documents may be too large, even if there are not too many documents.

Example 3.9 : Suppose we have a million documents, and we use signatures of length 250. Then we use 1000 bytes per document for the signatures, and the entire data fits in a gigabyte – less than a typical main memory of a laptop.

However, there are 1,000,000 or half a trillion pairs of documents. If it takes a microsecond to compute the similarity of two signatures, then it takes almost six days to compute all the similarities on that laptop.

If our goal is to compute the similarity of every pair, there is nothing we can do to reduce the work, although parallelism can reduce the elapsed time. However, often we want only the most similar pairs or all pairs that are above some lower bound in similarity. If so, then we need to focus our attention only on pairs that are likely to be similar, without investigating every pair. There is a general theory of how to provide such focus, called *locality-sensitive hashing* (LSH) or *near-neighbor search*. In this section we shall consider a specific form of LSH, designed for the particular problem we have been studying: documents, represented by shingle-sets, then minhashed to short signatures. In Section 3.6 we present the general theory of locality-sensitive hashing and a number of applications and related techniques.

3.6.1 LSH for Minhash Signatures

One general approach to LSH is to "hash" items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are. We then consider any pair that hashed to the same bucket for any of the hashings to be a *candidate pair*. We check only the candidate pairs for similarity. The hope is that most of the dissimilar pairs will never hash to the same bucket, and therefore will never be checked. Those dissimilar pairs that do hash to the same bucket are *false positives*; we hope these will be only a small fraction of all pairs. We also hope that most of the truly similar

pairs will hash to the same bucket under at least one of the hash functions. Those that do not are *false negatives*; we hope these will be only a small fraction of the truly similar pairs.

If we have minhash signatures for the items, an effective way to choose the hashings is to divide the signature matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers (the portion of one column within that band) and hashes them to some large number of buckets. We can use the same hash function for all the bands, but we use a separate bucket array for each band, so columns with the same vector in different bands will not hash to the same bucket.

Example 3.10 : Figure 3.6 shows part of a signature matrix of 12 rows divided into four bands of three rows each. The second and fourth of the explicitly shown columns each have the column vector [0, 2, 1] in the first band, so they will definitely hash to the same bucket in the hashing for the first band. Thus, regardless of what those columns look like in the other three bands, this pair of columns will be a candidate pair. It is possible that other columns, such as the first two shown explicitly, will also hash to the same bucket according to the hashing of the first band. However, since their column vectors are different, [1, 3, 0] and [0, 2, 1], and there are many buckets for each hashing, we expect the chances of an accidental collision to be very small. We shall normally assume that two vectors hash to the same bucket if and only if they are identical.

Two columns that do not agree in band 1 have three other chances to become a candidate pair; they might be identical in any one of these other bands.

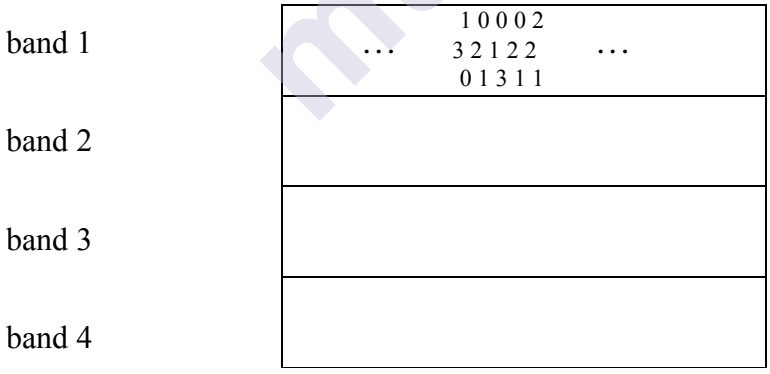


Figure 3.6: Dividing a signature matrix into four bands of three rows per band

However, observe that the more similar two columns are, the more likely it is that they will be identical in some band. Thus, intuitively the banding strategy makes similar columns much more likely to be candidate pairs than dissimilar pairs.

3.6.2 Analysis of the Banding Technique

Suppose we use b bands of r rows each, and suppose that a particular pair of documents have Jaccard similarity s. Recall from Section 3.3.3 that the prob- ability the minhash signatures for these documents agree in any one particular row of the signature matrix is s. We can calculate the probability that these documents (or rather their signatures) become a candidate pair as follows:

3.6.2.1 The probability that the signatures agree in all rows of one particular band is s^r.

3.6.2.2 The probability that the signatures disagree in at least one row of a par-ticular band is $1 - s^r$.

3.6.2.3 The probability that the signatures disagree in at least one row of each of the bands is $(1 - s^r)^b$.

3.6.2.4 The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$.

It may not be obvious, but regardless of the chosen constants b and r, this function has the form of an *S-curve*, as suggested in Fig. 3.7. The *threshold*, that is, the value of similarity s at which the probability of becoming a candidate is 1/2, is a function of b and r. The threshold is roughly where the rise is the steepest, and for large b and r there we find that pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates – exactly the situation we want.

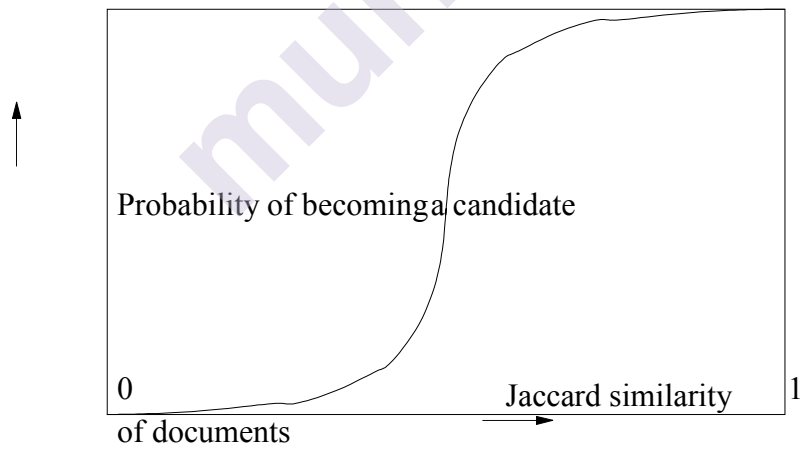


Figure 3.7: The S-curve

An approximation to the threshold is $(1/b)^{1/r}$. For example, if b = 16 and r = 4, then the threshold is approximately at s = 1/2, since the 4th root of 1/16 is 1/2.

Example 3.11 : Let us consider the case b = 20 and r = 5. That is, we suppose we have signatures of length 100, divided into twenty bands

of five rows each. Figure 3.8 tabulates some of the values of the function $1 - (1 - s^5)^{20}$. Notice that the threshold, the value of s at which the curve has risen halfway, is just slightly more than 0.5. Also notice that the curve is not exactly the ideal step function that jumps from 0 to 1 at the threshold, but the slope of the curve in the middle is significant. For example, it rises by more than 0.6 going from $s = 0.4$ to $s = 0.6$, so the slope in the middle is greater than 3.

s	$1 - (1 - s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Figure 3.8: Values of the S-curve for $b = 20$ and $r = 5$

For example, at $s = 0.8$, $1 - (0.8)^5$ is about 0.672. If you raise this number to the 20th power, you get about 0.00035. Subtracting this fraction from 1 yields 0.99965. That is, if we consider two documents with 80% similarity, then in any one band, they have only about a 33% chance of agreeing in all five rows and thus becoming a candidate pair. However, there are 20 bands and thus 20 chances to become a candidate. Only roughly one in 3000 pairs that are as high as 80% similar will fail to become a candidate pair and thus be a false negative.

3.6.3 Combining the Techniques

We can now give an approach to finding the set of candidate pairs for similar documents and then discovering the truly similar documents among them. It must be emphasized that this approach can produce false negatives – pairs of similar documents that are not identified as such because they never become a candidate pair. There will also be false positives – candidate pairs that are evaluated, but are found not to be sufficiently similar.

3.6.3.1 Pick a value of k and construct from each document the set of k-shingles. Optionally, hash the k-shingles to shorter bucket numbers.

3.6.3.2 Sort the document-shingle pairs to order them by shingle.

3.6.3.3 Pick a length n for the minhash signatures. Feed the sorted list to the algorithm of Section 3.3.5 to compute the minhash signatures for all the documents.

3.6.3.4 Choose a threshold t that defines how similar documents have to be in order for them to be regarded as a desired "similar pair." Pick

a number of bands b and a number of rows r such that br = n, and the threshold t is approximately $(1/b)^{1/r}$. If avoidance of false negatives is important, you may wish to select b and r to produce a threshold lower than t; if speed is important and you wish to limit false positives, select b and r to produce a higher threshold.

3.6.3.5 Construct candidate pairs by applying the LSH technique of Section 3.4.1.

3.6.3.6 Examine each candidate pair's signatures and determine whether the frac-tion of components in which they agree is at least t.

3.6.3.7 Optionally, if the signatures are sufficiently similar, go to the documents themselves and check that they are truly similar, rather than documents that, by luck, had similar signatures.

3.6.4 Exercises for Section 3.4

Exercise 3.4.1 : Evaluate the S-curve $1 - (1 - s^r)^b$ for s = 0.1, 0.2, . . . , 0.9, for the following values of r and b:

- r = 3 and b = 10.
- r = 6 and b = 20.
- r = 5 and b = 50.

! Exercise 3.4.2 : For each of the (r, b) pairs in Exercise 3.4.1, compute the threshold, that is, the value of s for which the value of $1 - (1 - s^r)^b$ is exactly 1/2. How does this value compare with the estimate of $(1/b)^{1/r}$ that was suggested in Section 3.4.2?

! Exercise 3.4.3 : Use the techniques explained in Section 1.3.5 to approximate the S-curve $1 - (1 - s^r)^b$ when s^r is very small.

! Exercise 3.4.4 : Suppose we wish to implement LSH by MapReduce. Specifi- cally, assume chunks of the signature matrix consist of columns, and elements are key-value pairs where the key is the column number and the value is the signature itself (i.e., a vector of values).

(a) Show how to produce the buckets for all the bands as output of a single MapReduce process. *Hint* : Remember that a Map function can produce several key-value pairs from a single element.

(b) Show how another MapReduce process can convert the output of (a) to a list of pairs that need to be compared. Specifically, for each column i, there should be a list of those columns j > i with which i needs to be compared.

3.7 DISTANCE MEASURES

We now take a short detour to study the general notion of distance measures. The Jaccard similarity is a measure of how close sets are, although it is not really a distance measure. That is, the closer sets are,

the higher the Jaccard similarity. Rather, 1 minus the Jaccard similarity is a distance measure, as we shall see; it is called the *Jaccard distance*.

However, Jaccard distance is not the only measure of closeness that makes sense. We shall examine in this section some other distance measures that have applications. Then, in Section 3.6 we see how some of these distance measures also have an LSH technique that allows us to focus on nearby points without comparing all points. Other applications of distance measures will appear when we study clustering in Chapter 7.

3.7.1 Definition of a Distance Measure

Suppose we have a set of points, called a *space*. A *distance measure* on this space is a function d(x, y) that takes two points in the space as arguments and produces a real number, and satisfies the following axioms:

3.7.1.1 d(x, y) ≥ 0 (no negative distances).

3.7.1.2 d(x, y) = 0 if and only if x = y (distances are positive, except for the distance from a point to itself).

3.7.1.3 d(x, y) = d(y, x) (distance is symmetric).

3.7.1.4 d(x, y) $\leq$ d(x, z) + d(z, y) (the *triangle inequality*).

The triangle inequality is the most complex condition. It says, intuitively, that to travel from x to y, we cannot obtain any benefit if we are forced to travel via some particular third point z. The triangle-inequality axiom is what makes all distance measures behave as if distance describes the length of a shortest path from one point to another.

3.7.2 Euclidean Distances

The most familiar distance measure is the one we normally think of as "dis- tance." An n-*dimensional Euclidean space* is one where points are vectors of n real numbers. The conventional distance measure in this space, which we shall refer to as the L_2-*norm*, is defined:

$$d([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

That is, we square the distance in each dimension, sum the squares, and take the positive square root.

It is easy to verify the first three requirements for a distance measure are satisfied. The Euclidean distance between two points cannot be negative, be- cause the positive square root is intended.

Since all squares of real numbers are nonnegative, any i such that $x_i \neq y_i$ forces the distance to be strictly positive. On the other hand, if $x_i = y_i$ for all i, then the distance is clearly 0. Symmetry follows because $(x_i - y_i)^2 = (y_i - x_i)^2$. The triangle inequality requires a good deal of algebra to verify. However, it is well understood to be a property of Euclidean space: the sum of the lengths of any two sides of a triangle is no less than the length of the third side.

There are other distance measures that have been used for Euclidean spaces. For any constant r, we can define the L_r-*norm* to be the distance measure d defined by:

$$d([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) = \left(\sum_{i=1}^{n} |x_i - y_i|^r\right)^{1/r}$$

The case r = 2 is the usual L_2-norm just mentioned. Another common distance measure is the L_1-norm, or *Manhattan distance*. There, the distance between two points is the sum of the magnitudes of the differences in each dimension. It is called "Manhattan distance" because it is the distance one would have to travel between points if one were constrained to travel along grid lines, as on the streets of a city such as Manhattan.

Another interesting distance measure is the L_∞-norm, which is the limit as r approaches infinity of the L_r-norm. As r gets larger, only the dimension with the largest difference matters, so formally, the L_∞-norm is defined as the maximum of $|x_i - y_i|$ over all dimensions i.

Example 3.12 : Consider the two-dimensional Euclidean space (the customary plane) and the points (2, 7) and (6, 4). The L_2-norm gives a distance of $\sqrt{(2-6)^2 + (7-4)^2} = \sqrt{4^2 + 3^2} = 5$. The L_1-norm gives a distance of $|2-6| + |7-4| = 4 + 3 = 7$. The L_∞-norm gives a distance of $\max(|2-6|, |7-4|) = \max(4, 3) = 4$

3.7.3 Jaccard Distance

As mentioned at the beginning of the section, we define the *Jaccard distance* of sets by d(x, y) = 1 SIM(x, y). That is, the Jaccard distance is 1 minus the ratio of the sizes of the intersection and union of sets x and y. We must verify that this function is a distance measure.

3.7.3.1 d(x, y) is nonnegative because the size of the intersection cannot exceed the size of the union.

3.7.3.2 d(x, y) = 0 if x = y, because $x \cap x = x \cup x = x$. However, if $x \neq y$, then the size of $x \cap y$ is strictly less than the size of $x \cup y$, so d(x, y) is strictly positive.

3.7.3.3 d(x, y) = d(y, x) because both union and intersection are symmetric; i.e., $x \cup y = y \cup x$ and $x \cap y = y \cap x$.

3.7.3.4 For the triangle inequality, recall from Section 3.3.3 that SIM(x, y) is the probability a random minhash function maps x and y to the same value. Thus, the Jaccard distance d(x, y) is the probability that a random min- hash function *does not* send x and y to the same value. We can therefore translate the condition $d(x, y) \leq d(x, z) + d(z, y)$ to the statement that if h is a random minhash function, then the probability that $h(x) = h(y)$ is no greater than the sum of the probability that $h(x) = h(z)$ and the probability that $h(z) = h(y)$. However, this statement is true because whenever $h(x) = h(y)$, at least one of h(x) and h(y) must be different from h(z). They could not both be h(z), because then h(x) and h(y) would be the same.

3.7.4 Cosine Distance

The *cosine distance* makes sense in spaces that have dimensions, including Eu- clidean spaces and discrete versions of Euclidean spaces, such as spaces where points are vectors with integer components or Boolean (0 or 1) components. In such a space, points may be thought of as directions. We do not distinguish be- tween a vector and a multiple of that vector. Then the cosine distance between two points is the angle that the vectors to those points make. This angle will be in the range 0 to 180 degrees, regardless of how many dimensions the space has.

We can calculate the cosine distance by first computing the cosine of the angle, and then applying the arc-cosine function to translate to an angle in the 0-180 degree range. Given two vectors x and y, the cosine of the angle between them is the dot product x.y divided by the L_2-norms of x and y (i.e., their

Euclidean distances from the origin). Recall that the dot product of vectors $[x_1, x_2, \ldots, x_n].[y_1, y_2, \ldots, y_n]$ is $\sum_{i=1}^{n} x_i y_i$.

Example 3.13 : Let our two vectors be x = [1, 2, −1] and = [2, 1, 1]. The dot product x.y is $1 \times 2 + 2 \times 1 + (-1) \times 1 = 3$. The L_2-norm of both vectors is $\sqrt{6}$. For example, x has L_2-norm $\sqrt{1^2 + 2^2 + (-1)^2} = \sqrt{6}$. Thus, the cosine of the angle between x and y is $3/(\sqrt{6}\sqrt{6})$ or 1/2. The angle whose cosine is ½ is 60 degrees, so that is the cosine distance between x and y.

We must show that the cosine distance is indeed a distance measure. We have defined it so the values are in the range 0 to 180, so no negative distances are possible. Two vectors have angle 0 if and only if they are the same direction.[4] Symmetry is obvious: the angle between x and y is the same as the angle between y and x. The triangle inequality is best argued by physical reasoning. One way to rotate from x to y is to rotate to z and thence to y. The sum of those two rotations cannot be less than the rotation directly from x to y.

3.7.5 Edit Distance

This distance makes sense when points are strings. The distance between two strings $x = x_1x_2 \quad x_n$ and $y = y_1y_2 \quad y_m$ is the smallest number of insertions and deletions of single characters that will convert x to y.

Example 3.14 : The edit distance between the strings x = abcde and y = acfdeg is 3. To convert x to y:

3.7.5.1 Delete b.
3.7.5.2 Insert f after c.
3.7.5.3 Insert g after e.

No sequence of fewer than three insertions and/or deletions will convert x to y. Thus, d(x, y) = 3.

Another way to define and calculate the edit distance d(x, y) is to compute a *longest common subsequence* (LCS) of x and y. An LCS of x and y is a string that is constructed by deleting positions from x and y, and that is as long as any string that can be constructed that way. The edit distance d(x, y) can be calculated as the length of x plus the length of y minus twice the length of their LCS.

Example 3.15 : The strings x = abcde and y = acfdeg from Example 3.14 have a unique LCS, which is acde. We can be sure it is the longest possible, because it contains every symbol appearing in both x and y. Fortunately, these common symbols appear in the same order in both strings, so we are able to use them all in an LCS. Note that the length of x is 5, the length of y is 6, and the length of their LCS is 4. The edit distance is thus 5 + 6 2 4 = 3, which agrees with the direct calculation in Example 3.14.

For another example, consider x = aba and y = bab. Their edit distance is 2. For example, we can convert x to y by deleting the first a and then inserting b at the end. There are two LCS's: ab and ba. Each can be obtained by deleting one symbol from each string. As must be the case for multiple LCS's of the same pair of strings, both LCS's have the same length. Therefore, we may compute the edit distance as $3 + 3 - 2 \times 2 = 2$.

Edit distance is a distance measure. Surely no edit distance can be negative, and only two identical strings have an edit distance of 0. To see that edit distance is symmetric, note that a sequence of insertions and deletions can be reversed, with each insertion becoming a deletion, and vice versa. The triangle inequality is also straightforward. One way to turn a string s into a string t is to turn s into some string u and then turn u into t. Thus, the number of edits made going from s to u, plus the number of edits made going from u to t cannot be less than the smallest number of edits that will turn s into t.

3.7.6 Hamming Distance

Given a space of vectors, we define the *Hamming distance* between two vectors to be the number of components in which they differ. It should be obvious that Hamming distance is a distance measure. Clearly the Hamming distance cannot be negative, and if it is zero, then the vectors are identical. The dis- tance does not depend on which of two vectors we consider first. The triangle inequality should also be evident. If x and z differ in m components, and z and y differ in n components, then x and y cannot differ in more than m + n components. Most commonly, Hamming distance is used when the vectors are Boolean; they consist of 0's and 1's only. However, in principle, the vectors can have components from any set.

> Non-Euclidean Spaces
>
> Notice that several of the distance measures introduced in this section are not Euclidean spaces. A property of Euclidean spaces that we shall find important when we take up clustering in Chapter 7 is that the average of points in a Euclidean space always exists and is a point in the space. However, consider the space of sets for which we defined the Jaccard dis- tance. The notion of the "average" of two sets makes no sense. Likewise, the space of strings, where we can use the edit distance, does not let us take the "average" of strings.
>
> Vector spaces, for which we suggested the cosine distance, may or may not be Euclidean. If the components of the vectors can be any real num- bers, then the space is Euclidean. However, if we restrict components to be integers, then the space is not Euclidean. Notice that, for instance, we cannot find an average of the vectors [1, 2] and [3, 1] in the space of vectors with two integer components, although if we treated them as members of the two-dimensional Euclidean space, then we could say that their average was [2.0, 1.5].

Example 3.16 : The Hamming distance between the vectors 10101 and 11110 is 3. That is, these vectors differ in the second, fourth, and fifth components, while they agree in the first and third components.

3.7.7 Exercises for Section 3.5

! Exercise 3.5.1 : On the space of nonnegative integers, which of the following functions are distance measures? If so, prove it; if not, prove that it fails to satisfy one or more of the axioms.

(a) max(x, y) = the larger of x and y.

(b) diff(x, y) = $|x - y|$ (the absolute magnitude of the difference between x and y).

(c) sum(x, y) = x + y.

Exercise 3.5.2 : Find the L_1 and L_2 distances between the points (5, 6, 7) and (8, 2, 4).

!! Exercise 3.5.3 : Prove that if i and j are any positive integers, and i < j, then the L_i norm between any two points is greater than the L_j norm between those same two points.

Exercise 3.5.4 : Find the Jaccard distances between the following pairs of sets:

(a) $\{1, 2, 3, 4\}$ and $\{2, 3, 4, 5\}$.

(b) $\{1, 2, 3\}$ and $\{4, 5, 6\}$.

Exercise 3.5.5 : Compute the cosines of the angles between each of the fol- lowing pairs of vectors.[5]

(a) $(3, -1, 2)$ and $(-2, 3, 1)$.

(b) $(1, 2, 3)$ and $(2, 4, 6)$.

(c) $(5, 0, -4)$ and $(-1, -6, 2)$.

(d) $(0, 1, 1, 0, 1, 1)$ and $(0, 0, 1, 0, 0, 0)$.

! Exercise 3.5.6 : Prove that the cosine distance between any two vectors of 0's and 1's, of the same length, is at most 90 degrees.

Exercise 3.5.7 : Find the edit distances (using only insertions and deletions) between the following pairs of strings.

(a) abcdef and bdaefc.

(b) abccdabc and acbdcab.

(c) abcdef and baedfc.

! Exercise 3.5.8 : There are a number of other notions of edit distance available. For instance, we can allow, in addition to insertions and deletions, the following operations:

i. *Mutation*, where one symbol is replaced by another symbol. Note that a mutation can always be performed by an insertion followed by a deletion, but if we allow mutations, then this change counts for only 1, not 2, when computing the edit distance.

ii. *Transposition*, where two adjacent symbols have their positions swapped. Like a mutation, we can simulate a transposition by one insertion followed by one deletion, but here we count only 1 for these two steps.

Repeat Exercise 3.5.7 if edit distance is defined to be the number of insertions, deletions, mutations, and transpositions needed to transform one string into another.

! Exercise 3.5.9 : Prove that the edit distance discussed in Exercise 3.5.8 is indeed a distance measure.

Exercise 3.5.10 : Find the Hamming distances between each pair of the fol- lowing vectors: 000000, 110011, 010101, and 011100.

[5]Note that what we are asking for is not precisely the cosine distance, but from the cosine of an angle, you can compute the angle itself, perhaps with the aid of a table or library function.

3.8 THE THEORY OF LOCALITY-SENSITIVE FUNCTIONS

The LSH technique developed in Section 3.4 is one example of a family of func- tions (the minhash functions) that can be combined (by the banding technique) to distinguish strongly between pairs at a low distance from pairs at a high dis- tance. The steepness of the S-curve in Fig. 3.7 reflects how effectively we can avoid false positives and false negatives among the candidate pairs.

Now, we shall explore other families of functions, besides the minhash func- tions, that can serve to produce candidate pairs efficiently. These functions can apply to the space of sets and the Jaccard distance, or to another space and/or another distance measure. There are three conditions that we need for a family of functions:

1. They must be more likely to make close pairs be candidate pairs than distant pairs. We make this notion precise in Section 3.6.1.

2. They must be statistically independent, in the sense that it is possible to estimate the probability that two or more functions will all give a certain response by the product rule for independent events.

3. They must be efficient, in two ways:

(a) They must be able to identify candidate pairs in time much less than the time it takes to look at all pairs. For example, minhash functions have this capability, since we can hash sets to minhash values in time proportional to the size of the data, rather than the square of the number of sets in the data. Since sets with common values are colocated in a bucket, we have implicitly produced the candidate pairs for a single minhash function in time much less than the number of pairs of sets.

(b) They must be combinable to build functions that are better at avoid- ing false positives and negatives, and the combined functions must also take time that is much less than the number of pairs. For ex- ample, the banding technique of Section 3.4.1 takes single minhash functions, which satisfy condition 3a but do not, by themselves have the S-curve behavior we want, and produces from a number of min- hash functions a combined function that has the S-curve shape.

Our first step is to define "locality-sensitive functions" generally. We then see how the idea can be applied in several applications.

Finally, we discuss how to apply the theory to arbitrary data with either a cosine distance or a Euclidean distance measure.

3.8.1 Locality-Sensitive Functions

For the purposes of this section, we shall consider functions that take two items and render a decision about whether these items should be a candidate pair.

In many cases, the function f will "hash" items, and the decision will be based on whether or not the result is equal. Because it is convenient to use the notation f(x) = f(y) to mean that f(x, y) is "yes; make x and y a candidate pair," we shall use f(x) = f(y) as a shorthand with this meaning. We also use f(x) = f(y) to mean "do not make x and y a candidate pair unless some other function concludes we should do so."

A collection of functions of this form will be called a *family* of functions. For example, the family of minhash functions, each based on one of the possible permutations of rows of a characteristic matrix, form a family.

Let $d_1 < d_2$ be two distances according to some distance measure d. A family F of functions is said to be (d_1, d_2, p_1, p_2)-*sensitive* if for every f in F:

3.8.1.1 If $d(x, y) \leq d_1$, then the probability that f(x) = f(y) is at least p_1.

3.8.1.2 If $d(x, y) \geq d_2$, then the probability that f(x) = f(y) is at most p_2.

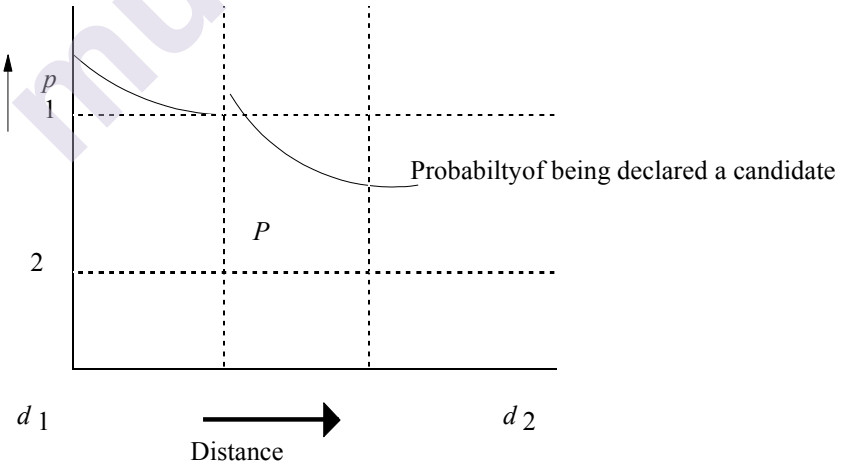


Figure 3.9: Behavior of a (d_1, d_2, p_1, p_2)-sensitive function

Figure 3.9 illustrates what we expect about the probability that a given function in a (d_1, d_2, p_1, p_2)-sensitive family will declare two items to be a can- didate pair. Notice that we say nothing about what happens when the distance between the items is strictly between d_1 and d_2, but we can make d_1 and d_2 as close as we wish. The penalty is that

typically p_1 and p_2 are then close as well. As we shall see, it is possible to drive p_1 and p_2 apart while keeping d_1 and d_2 fixed.

3.8.2 Locality-Sensitive Families for Jaccard Distance

For the moment, we have only one way to find a family of locality-sensitive functions: use the family of minhash functions, and assume that the distance measure is the Jaccard distance. As before, we interpret a minhash function h to make x and y a candidate pair if and only if h(x) = h(y).

The family of minhash functions is a $(d_1, d_2, 1 - d_1, 1 - d_2)$-sensitive family for any d_1 and d_2, where $0 \leq d_1 < d_2 \leq 1$.

The reason is that if $d(x, y) \leq d_1$, where d is the Jaccard distance, then $SIM(x, y) = 1 - d(x, y) \geq 1 - d_1$. But we know that the Jaccard similarity of x and y is equal to the probability that a minhash function will hash x and y to the same value. A similar argument applies to d_2 or any distance.

Example 3.17 : We could let $d_1 = 0.3$ and $d_2 = 0.6$. Then we can assert that the family of minhash functions is a (0.3, 0.6, 0.7, 0.4)-sensitive family. That is, if the Jaccard distance between x and y is at most 0.3 (i.e., $SIM(x, y) \geq 0.7$) then there is at least a 0.7 chance that a minhash function will send x and y to the same value, and if the Jaccard distance between x and y is at least 0.6 (i.e., $SIM(x, y) \leq 0.4$), then there is at most a 0.4 chance that x and y will be sent to the same value. Note that we could make the same assertion with another choice of d_1 and d_2; only $d_1 < d_2$ is required.

3.8.3 Amplifying a Locality-Sensitive Family

Suppose we are given a (d_1, d_2, p_1, p_2)-sensitive family F. We can construct a new family F' by the *AND-construction* on F, which is defined as follows. Each member of F' consists of r members of F for some fixed r. If f is in F', and f is constructed from the set $\{f_1, f_2, \ldots, f_r\}$ of members of F, we say f(x) = f(y) if and only if $f_i(x) = f_i(y)$ for all $i = 1, 2, \ldots, r$. Notice that this construction mirrors the effect of the r rows in a single band: the band makes x and y a candidate pair if every one of the r rows in the band say that x and y are equal (and therefore a candidate pair according to that row).

Since the members of F are independently chosen to make a member of F', we can assert that F' is a $d_1, d_2, (p_1)^r, (p_2)^r$ -sensitive family. That is, for any p, if p is the probability that a member of F will declare (x, y) to be a candidate pair, then the probability that a member of F' will so declare is p^r.

There is another construction, which we call the *OR-construction*, that turns a (d_1, d_2, p_1, p_2)-sensitive family F into a $d_1, d_2, 1 - (1 - p_1)^b, 1 - (1 - p_2)^b$ - sensitive family F'. Each member f of F' is constructed from b members of F, say $f_1, f_2, \ldots, f_b$. We define f(x) = f(y) if

and only if $f_i(x) = f_i(y)$ for one or more values of i. The OR-construction mirrors the effect of combining several bands: x and y become a candidate pair if any band makes them a candidate pair.

If p is the probability that a member of F will declare (x, y) to be a candidate pair, then 1−p is the probability it will not so declare. $(1-p)^b$ is the probability that none of $f_1, f_2, \ldots, f_b$ will declare (x, y) a candidate pair, and $1-(1-p)^b$ is the probability that at least one f_i will declare (x, y) a candidate pair, and therefore that f will declare (x, y) to be a candidate pair.

Notice that the AND-construction lowers all probabilities, but if we choose F and r judiciously, we can make the small probability p_2 get very close to 0, while the higher probability p_1 stays significantly away from 0. Similarly, the OR- construction makes all probabilities rise, but by choosing F and b judiciously, we can make the larger probability approach 1 while the smaller probability remains bounded away from 1. We can cascade AND- and OR-constructions in any order to make the low probability close to 0 and the high probability close to 1. Of course the more constructions we use, and the higher the values of r and b that we pick, the larger the number of functions from the original family that we are forced to use. Thus, the better the final family of functions is, the longer it takes to apply the functions from this family.

Example 3.18 : Suppose we start with a family F. We use the AND-construc- tion with r = 4 to produce a family F_1. We then apply the OR-construction to F_1 with b = 4 to produce a third family F_2. Note that the members of F_2 each are built from 16 members of F, and the situation is analogous to starting with 16 minhash functions and treating them as four bands of four rows each.

p	$1 - (1 - p^4)^4$
0.2	0.0064
0.3	0.0320
0.4	0.0985
0.5	0.2275
0.6	0.4260
0.7	0.6666
0.8	0.8785
0.9	0.9860

Figure 3.10: Effect of the 4-way AND-construction followed by the 4-way OR-construction

The 4-way AND-function converts any probability p into p^4. When we follow it by the 4-way OR-construction, that probability

p	$1 - (1 - p)^4$ ^{4}
0.1	0.0140
0.2	0.1215
0.3	0.3334
0.4	0.5740
0.5	0.7725
0.6	0.9015
0.7	0.9680

is further converted into $1 (1 p^4)^4$. Some values of this transformation are indicated in Fig. 3.10. This function is an S-curve, staying low for a while, then rising steeply (although not too steeply; the slope never gets much higher than 2), and then leveling off at high values. Like any S-curve, it has a *fixedpoint*, the value of p that is left unchanged when we apply the function of the S-curve. In this case, the fixedpoint is the value of p for which $p = 1 (1 \quad p^4)^4$. We can see that the fixedpoint is somewhere between 0.7 and 0.8. Below that value, probabilities are decreased, and above it they are increased. Thus, if we pick a high probability above the fixedpoint and a low probability below it, we shall have the desired effect that the low probability is decreased and the high probability is increased. Suppose F is the minhash functions, regarded as a (0.2, 0.6, 0.8, 0.4)-sensitive family. Then F_2, the family constructed by a 4-way AND followed by a 4-way OR, is a (0.2, 0.6, 0.8785, 0.0985)-sensitive family, as we can read from the rows for 0.8 and 0.4 in Fig. 3.10. By replacing F by F_2, we have reduced both the false-negative and false-positive rates, at the cost of making application of the functions take 16 times as long.

Figure 3.11: Effect of the 4-way OR-construction followed by the 4-way AND-construction

Example 3.19 : For the same cost, we can apply a 4-way OR-construction followed by a 4-way AND-construction. Figure 3.11 gives the transformation on probabilities implied by this construction. For instance, suppose that F is a (0.2, 0.6, 0.8, 0.4)-sensitive family. Then the constructed family is a (0.2, 0.6, 0.9936, 0.5740)-sensitive family. This choice is not necessarily the best. Although the higher probability has moved much closer to 1, the lower probability has also raised, increasing the number of false positives.

Example 3.20 : We can cascade constructions as much as we like. For exam- ple, we could use the construction of Example 3.18 on the family of minhash functions and then use the construction of Example 3.19 on the resulting family. The constructed family would then have functions each built from 256 minhash functions. It would, for instance transform a (0.2, 0.8, 0.8, 0.2)-sensitive family into a (0.2, 0.8, 0.9991285, 0.0000004)-sensitive family.

3.8.4 Exercises for Section 3.6

Exercise 3.6.1 : What is the effect on probability of starting with the family of minhash functions and applying:

(a) A 2-way AND construction followed by a 3-way OR construction.

(b) A 3-way OR construction followed by a 2-way AND construction.

(c) A 2-way AND construction followed by a 2-way OR construction, followed by a 2-way AND construction.

(d) A 2-way OR construction followed by a 2-way AND construction, followed by a 2-way OR construction followed by a 2-way AND construction.

Exercise 3.6.2 : Find the fixedpoints for each of the functions constructed in Exercise 3.6.1.

! Exercise 3.6.3 : Any function of probability p, such as that of Fig. 3.10, has a slope given by the derivative of the function. The maximum slope is where that derivative is a maximum. Find the value of p that gives a maximum slope for the S-curves given by Fig. 3.10 and Fig. 3.11. What are the values of these maximum slopes?

!! Exercise 3.6.4 : Generalize Exercise 3.6.3 to give, as a function of r and b, the point of maximum slope and the value of that slope, for families of functions defined from the minhash functions by:

(a) An r-way AND construction followed by a b-way OR construction.

(b) A b-way OR construction followed by an r-way AND construction.

3.9 LSH FAMILIES FOR OTHER DISTANCE MEASURES

There is no guarantee that a distance measure has a locality-sensitive family of hash functions. So far, we have only seen such families for the Jaccard distance. In this section, we shall show how to construct locality-sensitive families for Hamming distance, the cosine distance and for the normal Euclidean distance.

3.9.1 LSH Families for Hamming Distance

It is quite simple to build a locality-sensitive family of functions for the Ham- ming distance. Suppose we have a space of d-dimensional vectors, and h(x, y) denotes the Hamming distance between vectors x and y. If we take any one position of the vectors, say the ith position, we can define the function $f_i(x)$ to be the ith bit of vector

x. Then $f_i(x) = f_i(y)$ if and only if vectors x and y agree in the ith position. Then the probability that $f_i(x) = f_i(y)$ for a ran- domly chosen i is exactly 1 h(x, y)/d; i.e., it is the fraction of positions in which x and y agree.

This situation is almost exactly like the one we encountered for minhashing.

Thus, the family F consisting of the functions $\{f_1, f_2, \ldots, f_d\}$ is $a(d_1, d_2, 1 - d_1/d, 1 - d_2/d)$-sensitive family of hash functions, for any $d_1 < d_2$. There are only two differences between this family and the family of minhash functions.

3.9.1.1 While Jaccard distance runs from 0 to 1, the Hamming distance on a vector space of dimension d runs from 0 to d. It is therefore necessary to scale the distances by dividing by d, to turn them into probabilities.

3.9.1.2 While there is essentially an unlimited supply of minhash functions, the size of the family F for Hamming distance is only d.

The first point is of no consequence; it only requires that we divide by d at appropriate times. The second point is more serious. If d is relatively small, then we are limited in the number of functions that can be composed using the AND and OR constructions, thereby limiting how steep we can make the S-curve be.

3.9.2 Random Hyperplanes and the Cosine Distance

Recall from Section 3.5.4 that the cosine distance between two vectors is the angle between the vectors. For instance, we see in Fig. 3.12 two vectors x and y that make an angle θ between them. Note that these vectors may be in a space of many dimensions, but they always define a plane, and the angle between them is measured in this plane. Figure 3.12 is a "top-view" of the plane containing x and y.

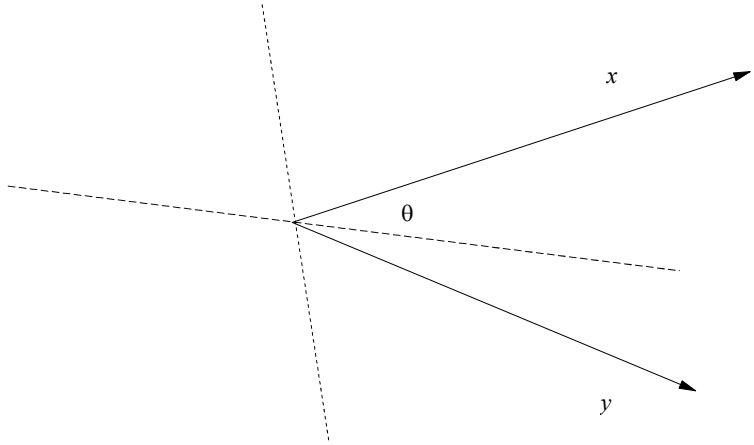


Figure 3.12: Two vectors make an angle θ

Suppose we pick a hyperplane through the origin. This hyperplane intersects the plane of x and y in a line. Figure 3.12 suggests two possible hyperplanes, one whose intersection is the dashed line and the other's intersection is the dotted line. To pick a random hyperplane, we actually pick the normal vector to the hyperplane, say v. The hyperplane is then the set of points whose dot product with v is 0.

First, consider a vector v that is normal to the hyperplane whose projection is represented by the dashed line in Fig. 3.12; that is, x and y are on different sides of the hyperplane. Then the dot products v.x and v.y will have different signs. If we assume, for instance, that v is a vector whose projection onto the plane of x and y is above the dashed line in Fig. 3.12, then v.x is positive, while v.y is negative. The normal vector v instead might extend in the opposite direction, below the dashed line. In that case v.x is negative and v.y is positive, but the signs are still different.

On the other hand, the randomly chosen vector v could be normal to a hyperplane like the dotted line in Fig. 3.12. In that case, both v.x and v.y have the same sign. If the projection of v extends to the right, then both dot products are positive, while if v extends to the left, then both are negative.

What is the probability that the randomly chosen vector is normal to a hyperplane that looks like the dashed line rather than the dotted line? All angles for the line that is the intersection of the random hyperplane and the plane of x and y are equally likely. Thus, the hyperplane will look like the dashed line with probability θ/180 and will look like the dotted line otherwise.

Thus, each hash function f in our locality-sensitive family F is built from a randomly chosen vector v_f. Given two vectors x and y, say $f(x) = f(y)$ if and only if the dot products $v_f.x$ and $v_f.y$ have the same sign. Then F is a locality-sensitive family for the cosine distance. The parameters are essentially the same as for the Jaccard-distance family described in Section 3.6.2, except the scale of distances is 0–180 rather than 0–1. That is, F is a $(d_1, d_2, (180 - d_1)/180, (180 - d_2)/180)$-sensitive family of hash functions. From this basis, we can amplify the family as we wish, just as for the minhash-based family.

3.9.3 Sketches

Instead of chosing a random vector from all possible vectors, it turns out to be sufficiently random if we restrict our choice to vectors whose components are +1 and 1. The dot product of any vector x with a vector v of +1's and 1's is formed by adding the components of x where v is +1 and then subtracting the other components of x – those where v is −1.

If we pick a collection of random vectors, say $v_1, v_2, \ldots, v_n$, then we can apply them to an arbitrary vector x by computing $v_1.x, v_2.x, \ldots,$

v_n.x and then replacing any positive value by +1 and any negative value by 1. The result is called the *sketch* of x. You can handle 0's arbitrarily, e.g., by chosing a result +1 or 1 at random. Since there is only a tiny probability of a zero dot product, the choice has essentially no effect.

Example 3.21 : Suppose our space consists of 4-dimensional vectors, and we pick three random vectors: $v_1 = [+1, -1, +1, +1]$, $v_2 = [-1, +1, -1, +1]$, and $v_3 = [+1, +1, -1, -1]$. For the vector x = [3, 4, 5, 6], the sketch is [+1, +1, −1].

That is, $v_1.x = 3-4+5+6 = 10$. Since the result is positive, the first component of the sketch is +1. Similarly, $v_2.x = 2$ and $v_3.x = 4$, so the second component of the sketch is +1 and the third component is 1.

Consider the vector y = [4, 3, 2, 1]. We can similarly compute its sketch to be [+1, 1, +1]. Since the sketches for x and y agree in 1/3 of the positions, we estimate that the angle between them is 120 degrees. That is, a randomly chosen hyperplane is twice as likely to look like the dashed line in Fig. 3.12 than like the dotted line.

The above conclusion turns out to be quite wrong. We can calculate the cosine of the angle between x and y to be x.y, which is

$$6 \times 1 + 5 \times 2 + 4 \times 3 + 3 \times 4 = 40$$

divided by the magnitudes of the two vectors. These magnitudes are

$$\sqrt{6^2 + 5^2 + 4^2 + 3^2} = 9.274$$

and $\sqrt{1^2 + 2^2 + 3^2 + 4^2} = 5.477$. Thus, the cosine of the angle between x and y is 0.7875, and this angle is about 38 degrees. However, if you look at all 16 different vectors v of length 4 that have +1 and 1 as components, you find that there are only four of these whose dot products with x and y have a different sign, namely v_2, v_3, and their complements [+1, 1, +1, 1] and [1, 1, +1, +1]. Thus, had we picked all sixteen of these vectors to form a sketch, the estimate of the angle would have been 180/4 = 45 degrees.

3.9.4 LSH Families for Euclidean Distance

Now, let us turn to the Euclidean distance (Section 3.5.2), and see if we can develop a locality-sensitive family of hash functions for this distance. We shall start with a 2-dimensional Euclidean space. Each hash function f in our family F will be associated with a randomly chosen line in this space. Pick a constant a and divide the line into segments of length a, as suggested by Fig. 3.13, where the "random" line has been oriented to be horizontal.

The segments of the line are the buckets into which function f hashes points. A point is hashed to the bucket in which its projection onto the line lies. If the distance d between two points is small compared with a, then there is a good chance the two points hash to the same bucket, and thus the hash function f will declare the two points equal. For example, if d = a/2, then there is at least a 50% chance the two points will fall in the same bucket. In fact, if the angle θ between the randomly chosen line and the line connecting the points is large, then there is an even greater chance that the two points will fall in the same bucket. For instance, if θ is 90 degrees, then the two points are certain to fall in the same bucket.

However, suppose d is larger than a. In order for there to be any chance of the two points falling in the same bucket, we need $d \cos \theta \leq a$. The diagram of Fig. 3.13 suggests why this requirement holds. Note that even if $d \cos \theta \ll a$ it Bucket width *a*

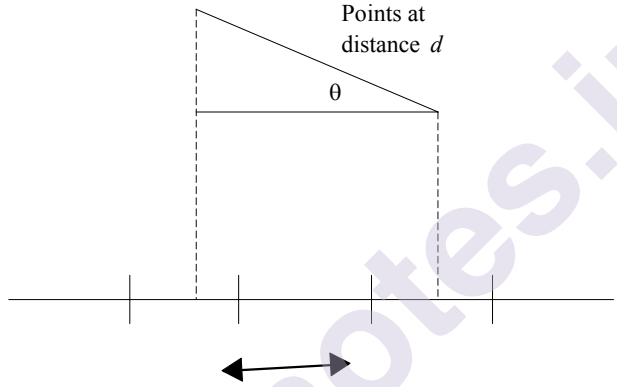


Figure 3.13: Two points at distance d a have a small chance of being hashed to the same bucket is still not certain that the two points will fall in the same bucket. However, we can guarantee the following. If $d \geq 2a$, then there is no more than a 1/3 chance the two points fall in the same bucket. The reason is that for cos θ to be less than 1/2, we need to have θ in the range 60 to 90 degrees. If θ is in the range 0 to 60 degrees, then cos θ is more than 1/2. But since θ is the smaller angle between two randomly chosen lines in the plane, θ is twice as likely to be between 0 and 60 as it is to be between 60 and 90.

We conclude that the family F just described forms a (a/2, 2a, 1/2, 1/3)- sensitive family of hash functions. That is, for distances up to a/2 the proba- bility is at least 1/2 that two points at that distance will fall in the same bucket, while for distances at least 2a the probability points at that distance will fall in the same bucket is at most 1/3. We can amplify this family as we like, just as for the other examples of locality-sensitive hash functions we have discussed.

3.9.5 More LSH Families for Euclidean Spaces

There is something unsatisfying about the family of hash functions developed in Section 3.7.4. First, the technique was only described for

two-dimensional Euclidean spaces. What happens if our data is points in a space with many dimensions? Second, for Jaccard and cosine distances, we were able to develop locality-sensitive families for any pair of distances d_1 and d_2 as long as $d_1 < d_2$. In Section 3.7.4 we appear to need the stronger condition $d_1 < 4d_2$.

However, we claim that there is a locality-sensitive family of hash func- tions for any $d_1 < d_2$ and for any number of dimensions. The family's hash functions still derive from random lines through the space and a bucket size a that partitions the line. We still hash points by projecting them onto the line. Given that $d_1 < d_2$, we may not know what the probability p_1 is that two points at distance d_1 hash to the same bucket, but we can be certain that it is greater than p_2, the probability that two points at distance d_2 hash to the same bucket. The reason is that this probability surely grows as the distance shrinks. Thus, even if we cannot calculate p_1 and p_2 easily, we know that there is a (d_1, d_2, p_1, p_2)-sensitive family of hash functions for any $d_1 < d_2$ and any given number of dimensions.

Using the amplification techniques of Section 3.6.3, we can then adjust the two probabilities to surround any particular value we like, and to be as far apart as we like. Of course, the further apart we want the probabilities to be, the larger the number of basic hash functions in F we must use.

3.9.6 Exercises for Section 3.7

Exercise 3.7.1 : Suppose we construct the basic family of six locality-sensitive functions for vectors of length six. For each pair of the vectors 000000, 110011, 010101, and 011100, which of the six functions makes them candidates?

Exercise 3.7.2 : Let us compute sketches using the following four "random" vectors:

$v_1 = [+1, +1, +1, -1]$ $v_2 = [+1, +1, -1, +1]$

$v_3 = [+1, -1, +1, +1]$ $v_4 = [-1, +1, +1, +1]$

Compute the sketches of the following vectors. (a) [2, 3, 4, 5].

(b) [−2, 3, −4, 5].

(c) [2, −3, 4, −5].

For each pair, what is the estimated angle between them, according to the sketches? What are the true angles?

Exercise 3.7.3 : Suppose we form sketches by using all sixteen of the vectors of length 4, whose components are each +1 or 1. Compute the sketches of the three vectors in Exercise 3.7.2. How do the estimates of the angles between each pair compare with the true angles?

Exercise 3.7.4 : Suppose we form sketches using the four vectors from Exer- cise 3.7.2.

! (a) What are the constraints on a, b, c, and d that will cause the sketch ofthe vector [a, b, c, d] to be [+1, +1, +1, +1]?

!! (b) Consider two vectors [a, b, c, d] and [e, f, g, h]. What are the conditions on

a, b, . . . , h that will make the sketches of these two vectors be the same?

Exercise 3.7.5 : Suppose we have points in a 3-dimensional Euclidean space: $p_1 = (1, 2, 3)$, $p_2 = (0, 2, 4)$, and $p_3 = (4, 3, 2)$. Consider the three hash functions defined by the three axes (to make our calculations very easy). Let buckets be of length a, with one bucket the interval [0, a) (i.e., the set of points x such that $0 \leq x < a$), the next [a, 2a), the previous one [−a, 0), and so on.

(a) For each of the three lines, assign each of the points to buckets, assuming

$a = 1$.

(b) Repeat part (a), assuming $a = 2$.

(c) What are the candidate pairs for the cases $a = 1$ and $a = 2$?

(d) For each pair of points, for what values of a will that pair be a candidatepair?

3.10 APPLICATIONS OF LOCALITY-SENSITIVE HASHING

In this section, we shall explore three examples of how LSH is used in practice. In each case, the techniques we have learned must be modified to meet certain constraints of the problem. The three subjects we cover are:

1. *Entity Resolution*: This term refers to matching data records that refer to the same real-world entity, e.g., the same person. The principal problem addressed here is that the similarity of records does not match exactly either the similar-sets or similar-vectors models of similarity on which the theory is built.

2. *Matching Fingerprints*: It is possible to represent fingerprints as sets. However, we shall explore a different family of locality-sensitive hash func-tions from the one we get by minhashing.

3. *Matching Newspaper Articles*: Here, we consider a different notion of shingling that focuses attention on the core article in an on-line news- paper's Web page, ignoring all the extraneous material such as ads and newspaper-specific material.

3.10.1 Entity Resolution

It is common to have several data sets available, and to know that they refer to some of the same entities. For example, several different bibliographic sources provide information about many of the same books or papers. In the general case, we have records describing entities of some type, such as people or books. The records may all have the same format, or they may have different formats, with different kinds of information.

There are many reasons why information about an entity may vary, even if the field in question is supposed to be the same. For example, names may be expressed differently in different records because of misspellings, absence of a middle initial, use of a nickname, and many other reasons. For example, "Bob S. Jomes" and "Robert Jones Jr." may or may not be the same person. If records come from different sources, the fields may differ as well. One source's records may have an "age" field, while another does not. The second source might have a "date of birth" field, or it may have no information at all about when a person was born.

3.10.2 An Entity-Resolution Example

We shall examine a real example of how LSH was used to deal with an entity- resolution problem. Company A was engaged by Company B to solicit cus- tomers for B. Company B would pay A a yearly fee, as long as the customer maintained their subscription. They later quarreled and disagreed over how many customers A had provided to B. Each had about 1,000,000 records, some of which described the same people; those were the customers A had provided to B. The records had different data fields, but unfortunately none of those fields was "this is a customer that A had provided to B." Thus, the problem was to match records from the two sets to see if a pair represented the same person.

Each record had fields for the name, address, and phone number of the person. However, the values in these fields could differ for many reasons. Not only were there the misspellings and other naming differences mentioned in Section 3.8.1, but there were other opportunities to disagree as well. A customer might give their home phone to A and their cell phone to B. Or they might move, and tell B but not A (because they no longer had need for a relationship with A). Area codes of phones sometimes change.

The strategy for identifying records involved scoring the differences in three fields: name, address, and phone. To create a *score* describing the likelihood that two records, one from A and the other from B, described the same per- son, 100 points was assigned to each of the three fields, so records with exact matches in all three fields got a score of 300. However, there were deductions for mismatches in each of the three fields. As a first approximation, edit-distance (Section 3.5.5) was used, but the penalty grew quadratically with the distance. Then,

certain publicly available tables were used to reduce the penalty in appropriate situations. For example, "Bill" and "William" were treated as if they differed in only one letter, even though their edit-distance is 5.

However, it is not feasible to score all one trillion pairs of records. Thus, a simple LSH was used to focus on likely candidates. Three "hash functions" were used. The first sent records to the same bucket only if they had identical names; the second did the same but for identical addresses, and the third did the same for phone numbers. In practice, there was no hashing; rather the records were sorted by name, so records with identical names would appear consecutively and get scored for overall similarity of the name, address, and phone. Then the records were sorted by address, and those with the same

When Are Record Matches Good Enough?

While every case will be different, it may be of interest to know how the experiment of Section 3.8.3 turned out on the data of Section 3.8.2. For scores down to 185, the value of x was very close to 10; i.e., these scores indicated that the likelihood of the records representing the same person was essentially 1. Note that a score of 185 in this example represents a situation where one field is the same (as would have to be the case, or the records would never even be scored), one field was completely different, and the third field had a small discrepancy. Moreover, for scores as low as 115, the

address were scored. Finally, the records were sorted a third time by phone, and records with identical phones were scored.

This approach missed a record pair that truly represented the same person but none of the three fields matched exactly. Since the goal was to prove in a court of law that the persons were the same, it is unlikely that such a pair would have been accepted by a judge as sufficiently similar anyway.

3.10.3 Validating Record Matches

What remains is to determine how high a score indicates that two records truly represent the same individual. In the example at hand, there was an easy way to make that decision, and the technique can be applied in many similar situations. It was decided to look at the creation-dates for the records at hand, and to assume that 90 days was an absolute maximum delay between the time the service was bought at Company A and registered at B. Thus, a proposed match between two records that were chosen at random, subject only to the constraint that the date on the B-record was between 0 and 90 days after the date on the A-record, would have an average delay of 45 days.

It was found that of the pairs with a perfect 300 score, the average delay was 10 days. If you assume that 300-score pairs are surely correct matches, then you can look at the pool of pairs with any given score s, and compute the average delay of those pairs. Suppose that the average delay is x, and the fraction of true matches among those pairs with score s is f. Then $x = 10f + 45(1 - f)$, or $x = 45 - 35f$. Solving for f, we find that the fraction of the pairs with score s that are truly matches is $(45 - x)/35$.

The same trick can be used whenever:

3.10.3.1 There is a scoring system used to evaluate the likelihood that two records represent the same entity, and

3.10.3.2 There is some field, not used in the scoring, from which we can derive a measure that differs, on average, for true pairs and false pairs.

For instance, suppose there were a "height" field recorded by both companies A and B in our running example. We can compute the average difference in height for pairs of random records, and we can compute the average difference in height for records that have a perfect score (and thus surely represent the same entities). For a given score s, we can evaluate the average height difference of the pairs with that score and estimate the probability of the records representing the same entity. That is, if h_0 is the average height difference for the perfect matches, h_1 is the average height difference for random pairs, and h is the average height difference for pairs of score s, then the fraction of good pairs with score s is $(h_1 - h)/(h_1 - h_0)$.

3.10.4 Matching Fingerprints

When fingerprints are matched by computer, the usual representation is not an image, but a set of locations in which *minutiae* are located. A minutia, in the context of fingerprint descriptions, is a place where something unusual happens, such as two ridges merging or a ridge ending. If we place a grid over a fingerprint, we can represent the fingerprint by the set of grid squares in which minutiae are located.

Ideally, before overlaying the grid, fingerprints are normalized for size and orientation, so that if we took two images of the same finger, we would find minutiae lying in exactly the same grid squares. We shall not consider here the best ways to normalize images. Let us assume that some combination of techniques, including choice of grid size and placing a minutia in several adjacent grid squares if it lies close to the border of the squares enables us to assume that grid squares from two images have a significantly higher probability of agreeing in the presence or absence of a minutia than if they were from images of different fingers.

Thus, fingerprints can be represented by sets of grid squares – those where their minutiae are located – and compared like any sets, using the Jaccard sim- ilarity or distance. There are two versions of fingerprint comparison, however.

- The *many-one* problem is the one we typically expect. A fingerprint has been found on a gun, and we want to compare it with all the fingerprints in a large database, to see which one matches.
- The *many-many* version of the problem is to take the entire database, andsee if there are any pairs that represent the same individual.

While the many-many version matches the model that we have been following for finding similar items, the same technology can be used to speed up the many-one problem.

3.10.5 A LSH Family for Fingerprint Matching

We could minhash the sets that represent a fingerprint, and use the standard LSH technique from Section 3.4. However, since the sets are chosen from a relatively small set of grid points (perhaps 1000), the need to minhash them into more succinct signatures is not clear. We shall study here another form of locality-sensitive hashing that works well for data of the type we are discussing. Suppose for an example that the probability of finding a minutia in a random grid square of a random fingerprint is 20%. Also, assume that if two fingerprints come from the same finger, and one has a minutia in a given grid square, then the probability that the other does too is 80%. We can define a locality-sensitive family of hash functions as follows. Each function f in this family F is defined by three grid squares. Function f says "yes" for two fingerprints if both have minutiae in all three grid squares, and otherwise f says "no." Put another way, we may imagine that f sends to a single bucket all fingerprints that have minutiae in all three of f's grid points, and sends each other fingerprint to a bucket of its own. In what follows, we shall refer to the first of these buckets as "the" bucket for f and ignore the buckets that are required to be singletons.

If we want to solve the many-one problem, we can use many functions from the family F and precompute their buckets of fingerprints to which they answer "yes." Then, given a new fingerprint that we want to match, we determine which of these buckets it belongs to and compare it with all the fingerprints found in any of those buckets. To solve the many-many problem, we compute the buckets for each of the functions and compare all fingerprints in each of the buckets.

Let us consider how many functions we need to get a reasonable probability of catching a match, without having to compare the fingerprint on the gun with each of the millions of fingerprints in the database. First, the probability that two fingerprints from different fingers would be in the bucket for a function f in F is $(0.2)^6 = 0.000064$. The reason is that they will both go into the bucket only if

they each have a minutia in each of the three grid points associated with f, and the probability of each of those six independent events is 0.2.

Now, consider the probability that two fingerprints from the same finger wind up in the bucket for f. The probability that the first fingerprint has minutiae in each of the three squares belonging to f is $(0.2)^3 = 0.008$. However, if it does, then the probability is $(0.8)^3 = 0.512$ that the other fingerprint will as well. Thus, if the fingerprints are from the same finger, there is a 0.008 0.512 = 0.004096 probability that they will both be in the bucket of f. That is not much; it is about one in 200. However, if we use many functions from F, but not too many, then we can get a good probability of matching fingerprints from the same finger while not having too many false positives – fingerprints that must be considered but do not match.

Example 3.22 : For a specific example, let us suppose that we use 1024 functions chosen randomly from F. Next, we shall construct a new fam- ily F_1 by performing a 1024-way OR on F. Then the probability that F_1 will put fingerprints from the same finger together in at least one bucket is

1 (1 0.004096)^{1024} = 0.985. On the other hand, the probability that two fingerprints from different fingers will be placed in the same bucket is (1 (1 0.000064)^{1024} = 0.063. That is, we get about 1.5% false negatives and about 6.3% false positives.

The result of Example 3.22 is not the best we can do. While it offers only a 1.5% chance that we shall fail to identify the fingerprint on the gun, it does force us to look at 6.3% of the entire database. Increasing the number of functions from F will increase the number of false positives, with only a small benefit of reducing the number of false negatives below 1.5%. On the other hand, we can also use the AND construction, and in so doing, we can greatly reduce the probability of a false positive, while making only a small increase in the false-negative rate. For instance, we could take 2048 functions from F in two groups of 1024. Construct the buckets for each of the functions. However, given a fingerprint P on the gun:

1. Find the buckets from the first group in which P belongs, and take the union of these buckets.

2. Do the same for the second group.

3. Take the intersection of the two unions.

4. Compare P only with those fingerprints in the intersection.

Note that we still have to take unions and intersections of large sets of finger- prints, but we compare only a small fraction of those. It is the comparison of fingerprints that takes the bulk of the time; in steps

(1) and (2) fingerprints can be represented by their integer indices in the database.

If we use this scheme, the probability of detecting a matching fingerprint is $(0.985)^2 = 0.970$; that is, we get about 3% false negatives. However, the probability of a false positive is $(0.063)^2 = 0.00397$. That is, we only have to examine about 1/250th of the database.

3.10.6 Similar News Articles

Our last case study concerns the problem of organizing a large repository of on-line news articles by grouping together Web pages that were derived from the same basic text. It is common for organizations like The Associated Press to produce a news item and distribute it to many newspapers. Each newspaper puts the story in its on-line edition, but surrounds it by information that is special to that newspaper, such as the name and address of the newspaper, links to related articles, and links to ads. In addition, it is common for the newspaper to modify the article, perhaps by leaving off the last few paragraphs or even deleting text from the middle. As a result, the same news article can appear quite different at the Web sites of different newspapers.

The problem looks very much like the one that was suggested in Section 3.4: find documents whose shingles have a high Jaccard similarity. Note that this problem is different from the problem of finding news articles that tell about the same events. The latter problem requires other techniques, typically examining the set of important words in the documents (a concept we discussed briefly in Section 1.3.1) and clustering them to group together different articles about the same topic.

However, an interesting variation on the theme of shingling was found to be more effective for data of the type described. The problem is that shingling as we described it in Section 3.2 treats all parts of a document equally. However, we wish to ignore parts of the document, such as ads or the headlines of other articles to which the newspaper added a link, that are not part of the news article. It turns out that there is a noticeable difference between text that appears in prose and text that appears in ads or headlines. Prose has a much greater frequency of stop words, the very frequent words such as "the" or "and." The total number of words that are considered stop words varies with the application, but it is common to use a list of several hundred of the most frequent words.

Example 3.23 : A typical ad might say simply "Buy Sudzo." On the other hand, a prose version of the same thought that might appear in an article is "I recommend that you buy Sudzo for your laundry." In the latter sentence, it would be normal to treat "I," "that," "you," "for," and "your" as stop words.

Suppose we define a *shingle* to be a stop word followed by the next two words. Then the ad "Buy Sudzo" from Example 3.23 has no shingles and would not be reflected in the representation of the Web page containing that ad. On the other hand, the sentence from Example 3.23 would be represented by five shingles: "I recommend that," "that you buy," "you buy Sudzo," "for your laundry," and "your laundry x," where x is whatever word follows that sentence.

Suppose we have two Web pages, each of which consists of half news text and half ads or other material that has a low density of stop words. If the news text is the same but the surrounding material is different, then we would expect that a large fraction of the shingles of the two pages would be the same. They might have a Jaccard similarity of 75%. However, if the surrounding material is the same but the news content is different, then the number of common shingles would be small, perhaps 25%. If we were to use the conventional shingling, where shingles are (say) sequences of 10 consecutive characters, we would expect the two documents to share half their shingles (i.e., a Jaccard similarity of 1/3), regardless of whether it was the news or the surrounding material that they shared.

3.10.7 Exercises for Section 3.8

Exercise 3.8.1 : Suppose we are trying to perform entity resolution among bibliographic references, and we score pairs of references based on the similar- ities of their titles, list of authors, and place of publication. Suppose also that all references include a year of publication, and this year is equally likely to be any of the ten most recent years. Further, suppose that we discover that among the pairs of references with a perfect score, there is an average difference in the publication year of 0.1.[6] Suppose that the pairs of references with a certain score s are found to have an average difference in their publication dates of 2. What is the fraction of pairs with score s that truly represent the same pub- lication? *Note*: Do not make the mistake of assuming the average difference in publication date between random pairs is 5 or 5.5. You need to calculate it exactly, and you have enough information to do so.

Exercise 3.8.2 : Suppose we use the family F of functions described in Sec- tion 3.8.5, where there is a 20% chance of a minutia in an grid square, an 80% chance of a second copy of a fingerprint having a minutia in a grid square where the first copy does, and each function in F being formed from three grid squares. In Example 3.22, we constructed family F_1 by using the OR construction on 1024 members of F. Suppose we instead used family F_2 that is a 2048-way OR of members of F.

(a) Compute the rates of false positives and false negatives for F_2.

(b) How do these rates compare with what we get if we organize the same 2048 functions into a 2-way AND of members of F_1, as was discussed at the end of Section 3.8.5?

Exercise 3.8.3 : Suppose fingerprints have the same statistics outlined in Ex- ercise 3.8.2, but we use a base family of functions F′ defined like F, but using only two randomly chosen grid squares. Construct another set of functions F′ from F′ by taking the n-way OR of functions from F′. What, as a function ofn, are the false positive and false negative rates for F′ ?

Exercise 3.8.4 : Suppose we use the functions F_1 from Example 3.22, but we want to solve the many-many problem.

(a) If two fingerprints are from the same finger, what is the probability that they will not be compared (i.e., what is the false negative rate)?

(b) What fraction of the fingerprints from different fingers will be compared (i.e., what is the false positive rate)?

! Exercise 3.8.5 : Assume we have the set of functions F as in Exercise 3.8.2, and we construct a new set of functions F_3 by an n-way OR of functions in

F. For what value of n is the sum of the false positive and false negative ratesminimized?

[6]We might expect the average to be 0, but in practice, errors in publication year do occur.

3.11 METHODS FOR HIGH DEGREES OF SIMILARITY

LSH-based methods appear most effective when the degree of similarity we accept is relatively low. When we want to find sets that are almost identical, there are other methods that can be faster. Moreover, these methods are exact, in that they find every pair of items with the desired degree of similarity. There are no false negatives, as there can be with LSH.

3.11.1 Finding Identical Items

The extreme case is finding identical items, for example, Web pages that are identical, character-for-character. It is straightforward to compare two docu- ments and tell whether they are identical, but we still must avoid having to compare every pair of documents. Our first thought would be to hash docu- ments based on their first few characters, and compare only those documents that fell into the same bucket. That scheme should work well, unless all the documents begin with the same characters, such as an HTML header.

Our second thought would be to use a hash function that examines the entire document. That would work, and if we use enough buckets, it would be very rare that two documents went into the same bucket, yet were not identical. The downside of this approach is that we must

examine every character of every document. If we limit our examination to a small number of characters, then we never have to examine a document that is unique and falls into a bucket of its own.

A better approach is to pick some fixed random positions for all documents, and make the hash function depend only on these. This way, we can avoid a problem where there is a common prefix for all or most documents, yet we need not examine entire documents unless they fall into a bucket with another document. One problem with selecting fixed positions is that if some documents are short, they may not have some of the selected positions. However, if we are looking for highly similar documents, we never need to compare two documents that differ significantly in their length. We exploit this idea in Section 3.9.3.

3.11.2 Representing Sets as Strings

Now, let us focus on the harder problem of finding, in a large collection of sets, all pairs that have a high Jaccard similarity, say at least 0.9. We can represent a set by sorting the elements of the universal set in some fixed order, and representing any set by listing its elements in this order. The list is essentially a string of "characters," where the characters are the elements of the universal set. These strings are unusual, however, in that:

3.11.2.1 No character appears more than once in a string, and

3.11.2.2 If two characters appear in two different strings, then they appear in the same order in both strings.

Example 3.24 : Suppose the universal set consists of the 26 lower-case letters, and we use the normal alphabetical order. {Then the set d, a, b is represented by the string abd.

In what follows, we shall assume all strings represent sets in the manner just described. Thus, we shall talk about the Jaccard similarity of strings, when strictly speaking we mean the similarity of the sets that the strings represent. Also, we shall talk of the length of a string, as a surrogate for the number of elements in the set that the string represents.

Note that the documents discussed in Section 3.9.1 do not exactly match this model, even though we can see documents as strings. To fit the model, we would shingle the documents, assign an order to the shingles, and represent each document by its list of shingles in the selected order.

3.11.3 Length-Based Filtering

The simplest way to exploit the string representation of Section 3.9.2 is to sort the strings by length. Then, each string s is compared with those strings t that follow s in the list, but are not too long. Suppose the lower bound on Jaccard similarity between two strings is J. For

any string x, denote its length by L_x. Note that $L_s \leq L_t$. The intersection of the sets represented by s and t cannot have more than L_s members, while their union has at least L_t members. Thus, the Jaccard similarity of s and t, which we denote SIM(s, t), is at most L_s/L_t. That is, in order for s and t to require comparison, it must be that $J \leq L_s/L_t$, or equivalently, $L_t \leq L_s/J$.

Example 3.25 : Suppose that s is a string of length 9, and we are looking for strings with at least 0.9 Jaccard similarity. Then we have only to compare s with strings following it in the length-based sorted order that have length at most $9/0.9 = 10$. That is, we compare s with those strings of length 9 that follow it in order, and all strings of length 10. We have no need to compare s with any other string.

Suppose the length of s were 8 instead. Then s would be compared with following strings of length up to $8/0.9 = 8.89$. That is, a string of length 9 would be too long to have a Jaccard similarity of 0.9 with s, so we only have to compare s with the strings that have length 8 but follow it in the sorted order.

3.11.4 Prefix Indexing

In addition to length, there are several other features of strings that can be exploited to limit the number of comparisons that must be made to identify all pairs of similar strings. The simplest of these options is to create an index for each symbol; recall a symbol of a string is any one of the elements of the universal set. For each string s, we select a prefix of s consisting of the first p symbols of s. How large p must be depends on L_s and J, the lower bound on Jaccard similarity. We add string s to the index for each of its first p symbols. In effect, the index for each symbol becomes a bucket of strings that must be compared. We must be certain that any other string t such that SIM(s, t) will have at least one symbol in its prefix that also appears in the prefix of s.

> A Better Ordering for Symbols
>
> Instead of using the obvious order for elements of the universal set, e.g., lexicographic order for shingles, we can order symbols rarest first. That is, determine how many times each element appears in the collection of sets, and order them by this count, lowest first. The advantage of doing so is that the symbols in prefixes will tend to be rare. Thus, they will cause that string to be placed in index buckets that have relatively few members. Then, when we need to examine a string for possible matches, we shall find few other strings that are candidates for comparison.

Suppose not; rather SIM(s,t) J, but t has none of the first p symbols of s. Then the highest Jaccard similarity that s and t can have occurs

when t is a suffix of s, consisting of everything but the first p symbols of s. The Jaccard similarity of s and t would then be $(L_s - p)/L_s$. To be sure that we do not have to compare s with t, we must be certain that $J > (L_s - p)/L_s$. That is, p must be at least $(1 - J)L_s + 1$. Of course we want p to be as small as possible, so we do not index string s in more buckets than we need to. Thus, we shall hereafter take $p = (1 - J)L_s + 1$ to be the length of the prefix that gets indexed.

Example 3.26 : Suppose $J = 0.9$. If $L_s = 9$, then $p = \lfloor 0.1 \times 9 \rfloor + 1 = 0.9 + 1 = 1$. That is, we need to index s under only its first symbol. Any string t that does not have the first symbol of s in a position such that t is indexed by that symbol will have Jaccard similarity with s that is less than 0.9. Suppose s is bcdefghij. Then s is indexed under b only. Suppose t does not begin with b. There are two cases to consider.

1. If t begins with a, and SIM(s, t) 0.9, then it can only be that t is abcdefghij. But if that is the case, t will be indexed under both a and

b. The reason is that $L_t = 10$, so t will be indexed under the symbols ofits prefix of length $\lfloor 0.1 \times 10 \rfloor + 1 = 2$.

2. If t begins with c or a later letter, then the maximum value of SIM(s, t) occurs when t is cdefghij. But then $SIM(s, t) = 8/9 < 0.9$.

In general, with $J = 0.9$, strings of length up to 9 are indexed by their first symbol, strings of lengths 10–19 are indexed under their first two symbols, strings of length 20–29 are indexed under their first three symbols, and so on.

We can use the indexing scheme in two ways, depending on whether we are trying to solve the many-many problem or a many-one problem; recall the distinction was introduced in Section 3.8.4. For the many-one problem, we create the index for the entire database. To query for matches to a new set S, we convert that set to a string s, which we call the *probe* string. Determine the length of the prefix that must be considered, that is, $(1 - J)L_s + 1$. For each symbol appearing in one of the prefix positions of s, we look in the index bucket for that symbol, and we compare s with all the strings appearing in that bucket.

If we want to solve the many-many problem, start with an empty database of strings and indexes. For each set S, we treat S as a new set for the many-one problem. We convert S to a string s, which we treat as a probe string in the many-one problem. However, after we examine an index bucket, we also add s to that bucket, so s will be compared with later strings that could be matches.

3.11.5 Using Position Information

Consider the strings s = acdefghijk and t = bcdefghijk, and assume J = 0.9. Since both strings are of length 10, they are indexed under their first two symbols. Thus, s is indexed under a and c, while t is indexed under b and c. Whichever is added last will find the other in the bucket for c, and they will be compared. However, since c is the second symbol of both, we know there will be two symbols, a and b in this case, that are in the union of the two sets but not in the intersection. Indeed, even though s and t are identical from c to the end, their intersection is 9 symbols and their union is 11; thus SIM(s, t) = 9/11, which is less than 0.9.

If we build our index based not only on the symbol, but on the position of the symbol within the string, we could avoid comparing s and t above. That is, let our index have a bucket for each pair (x, i), containing the strings that have symbol x in position i of their prefix. Given a string s, and assuming J is the minimum desired Jaccard similarity, we look at the prefix of s, that is, the positions 1 through $(1 - J)L_s + 1$. If the symbol in position i of the prefix is x, add s to the index bucket for (x, i).

Now consider s as a probe string. With what buckets must it be compared? We shall visit the symbols of the prefix of s from the left, and we shall take advantage of the fact that we only need to find a possible matching string t if none of the previous buckets we have examined for matches held t. That is, we only need to find a candidate match once. Thus, if we find that the ith symbol of s is x, then we need look in the bucket (x, j) for certain small values of j.

To compute the upper bound on j, suppose t is a string none of whose first j − 1 symbols matched anything in s, but the ith symbol of s is the same as the jth symbol of t. The highest value of SIM(s, t) occurs if s and t are identical Symbols definitely appearing in only one string

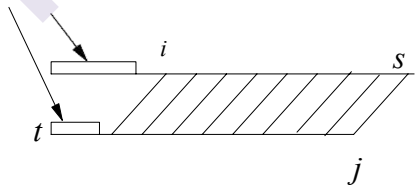


Figure 3.14: Strings s and t begin with i − 1 and j − 1 unique symbols, respec- tively, and then agree beyond that beyond their ith and jth symbols, respectively, as suggested by Fig. 3.14. If that is the case, the size of their intersection is $L_s - i + 1$, since that is the number of symbols of s that could possibly be in t. The size of their union is at least $L_s + j - 1$. That is, s surely contributes L_s symbols to the union, and there are also at least j − 1 symbols of t that are not in s. The ratio of the sizes of the intersection and union must be at least J, so we must have:

$\underline{L_s - i + 1}$ $\mathbf{J}$

$L_s + j - 1 \geq$

If we isolate j in this inequality, we have $j \leq$

$L_s(1 - J) - i + 1 + J$

/J.

Example 3.27 : Consider the string s = acdefghijk with J = 0.9 discussed at the beginning of this section. Suppose s is now a probe string. We already established that we need to consider the first two positions; that is, i can be 1 or 2. Suppose i = 1. Then j (10 0.1 1 + 1 + 0.9)/0.9. That is, we only have to compare the symbol a with strings in the bucket for (a, j) if j 2.11. Thus, j can be 1 or 2, but nothing higher.

Now suppose i = 2. Then we require j (10 0.1 2 + 1 + 0.9)/0.9, Or j 1. We conclude that we must look in the buckets for (a, 1), (a, 2), and (c, 1), but in no other bucket. In comparison, using the buckets of Section 3.9.4, we would look into the buckets for a and c, which is equivalent to looking to all buckets (a, j) and (c, j) for any j.

3.11.6 Using Position and Length in Indexes

When we considered the upper limit on j in the previous section, we assumed that what follows positions i and j were as in Fig. 3.14, where what followed these positions in strings s and t matched exactly. We do not want to build an index that involves every symbol in the strings, because that makes the total work excessive. However, we can add to our index a summary of what follows the positions being indexed. Doing so expands the number of buckets, but not beyond reasonable bounds, and yet enables us to eliminate many candidate matches without comparing entire strings. The idea is to use index buckets corresponding to a symbol, a position, and the *suffix length*, that is, the number of symbols following the position in question.

Example 3.28 : The string s = acdefghijk, with J = 0.9, would be indexed in the buckets for (a, 1, 9) and (c, 2, 8). That is, the first position of s has symbol a, and its suffix is of length 9. The second position has symbol c and its suffix is of length 8.

Figure 3.14 assumes that the suffixes for position i of s and position j of t have the same length. If not, then we can either get a smaller upper bound on the size of the intersection of s and t (if t is shorter) or a larger lower bound on the size of the union (if t is longer). Suppose s has suffix length p and t has suffix length q.

Case 1: p ≥q. Here, the maximum size of the intersection is $L_s - i + 1 - (p - q)$

Since $L_s = i + p$, we can write the above expression for the intersection size as $q + 1$. The minimum size of the union is $L_s + j - 1$, as it was when we did not take suffix length into account. Thus, we require

$$\frac{q+1}{s}$$

whenever $p \geq q$.

$$L + j - 1 \geq J$$

Case 2: $p < q$. Here, the maximum size of the intersection is $L_s - i + 1$, as when suffix length was not considered. However, the minimum size of the union is now $L_s + j - 1 + q - p$. If we again use the relationship $L_s = i + p$, we can replace $L_s - p$ by i and get the formula $i + j - 1 + q$ for the size of the union. If the Jaccard similarity is at least J, then

$$\frac{L_s - i + 1}{} \geq J \quad i + j - 1 + q$$

whenever $p < q$.

Example 3.29 : Let us again consider the string s = acdefghijk, but to make the example show some details, let us choose J = 0.8 instead of 0.9. We know that $L_s = 10$. Since $(1 - J)L_s + 1 = 3$, we must consider prefix positions i = 1, 2, and 3 in what follows. As before, let p be the suffix length of s and q the suffix length of t.

First, consider the case $p \geq q$. The additional constraint we have on q and j is $(q + 1)/(9 + j) \geq 0.8$. We can enumerate the pairs of values of j and q for each i between 1 and 3, as follows.

i = 1: Here, p = 9, so $q \leq 9$. Let us consider the possible values of q:

q = 9: We must have $10/(9 + j) \geq 0.8$. Thus, we can have j = 1, j = 2, or j = 3. Note that for j = 4, 10/13 > 0.8.

q = 8: We must have $9/(9 + j) \geq 0.8$. Thus, we can have j = 1 or j = 2.

For j = 3, 9/12 > 0.8.

q = 7: We must have $8/(9 + j) \geq 0.8$. Only j = 1 satisfies this inequality. q = 6: There are no possible values of j, since $7/(9 + j) > 0.8$ for every positive integer j. The same holds for every smaller value of q.

i = 2: Here, p = 8, so we require $q \leq 8$. Since the constraint $(q+1)/(9+j) \geq 0.8$ does not depend on i,[7] we can use the analysis from the above case, but exclude the case q = 9. Thus, the only possible values of j and q when i = 2 are

1. q = 8; j = 1.

2. q = 8; j = 2.

3. q = 7; j = 1.

i = 3: Now, p = 7 and the constraints are q ≥ 7 and (q + 1)/(9 + j) ≥ 0.8. The only option is q = 7 and j = 1.

Next, we must consider the case p < q. The additional constraint is

$$\frac{11-i}{i+j+q-1} \geq 0.8$$

Again, consider each possible value of i.

i = 1: Then p = 9, so we require q ≥ 10 and 10/(q + j) ≥ 0.8. The possible values of q and j are

1. q = 10; j = 1.

2. q = 10; j = 2.

3. q = 11; j = 1.

i = 2: Now, p = 8, so we require q ≥ 9 and 9/(q + j + 1) ≥ 0.8. Since j must be a positive integer, the only solution is q = 9 and j = 1, a possibility that we already knew about.

i = 3: Here, p = 7, so we require q ≥ 8 and 8/(q + j + 2) ≥ 0.8. There are no solutions.

When we accumulate the possible combinations of i, j, and q, we see that the set of index buckets in which we must look forms a pyramid. Figure 3.15 shows the buckets in which we must search. That is, we must look in those buckets (x, j, q) such that the ith symbol of the string s is x, j is the position associated with the bucket and q the suffix length.

[7]Note that *i* does influence the value of *p*, and through *p*, puts a limit on *q*.

	q	j = 1	j = 2	j = 3
i = 1	7	x		
	8	x	x	
	9	x	x	x
	10	x	x	
	11	x		
	7	x		

i = 2	8	x x
	9	x
i = 3	7	x

Figure 3.15: The buckets that must be examined to find possible matches for the string s = acdefghijk with **J** = 0.8 are marked with an x

3.11.7 Exercises for Section 3.9

Exercise 3.9.1 : Suppose our universal set is the lower-case letters, and the order of elements is taken to be the vowels, in alphabetic order, followed by the consonants in reverse alphabetic order. Represent the following sets as strings.

a {q, w, e, r, t, y}.

(b) {a, s, d, f, g, h, j, u, i}.

Exercise 3.9.2 : Suppose we filter candidate pairs based only on length, as in Section 3.9.3. If s is a string of length 20, with what strings is s compared when J, the lower bound on Jaccard similarity has the following values: (a) J = 0.85 (b) J = 0.95 (c) **J** = 0.98?

Exercise 3.9.3 : Suppose we have a string s of length 15, and we wish to index its prefix as in Section 3.9.4.

(a) How many positions are in the prefix if J = 0.85?

(b) How many positions are in the prefix if J = 0.95?

! (c) For what range of values of J will s be indexed under its first four symbols, but no more?

Exercise 3.9.4 : Suppose s is a string of length 12. With what symbol-position pairs will s be compared with if we use the indexing approach of Section 3.9.5, and (a) **J** = 0.75 (b) **J** = 0.95?

! Exercise 3.9.5 : Suppose we use position information in our index, as in Sec- tion 3.9.5. Strings s and t are both chosen at random from a universal set of 100 elements. Assume **J** = 0.9. What is the probability that s and t will be compared if

(a) s and t are both of length 9.

(b) s and t are both of length 10.

Exercise 3.9.6 : Suppose we use indexes based on both position and suffix length, as in Section 3.9.6. If s is a string of length 20, with what symbol- position-length triples will s be compared with, if (a) **J** = 0.8 (b) **J** = 0.9?

3.12 SUMMARY OF UNIT 3

✦ *Jaccard Similarity*: The Jaccard similarity of sets is the ratio of the size of the intersection of the sets to the size of the union. This measure of similarity is suitable for many applications, including textual similarity of documents and similarity of buying habits of customers.

✦ *Shingling*: A k-shingle is any k characters that appear consecutively in a document. If we represent a document by its set of k-shingles, then the Jaccard similarity of the shingle sets measures the textual similarity of documents. Sometimes, it is useful to hash shingles to bit strings of shorter length, and use sets of hash values to represent documents.

✦ *Minhashing*: A minhash function on sets is based on a permutation of the universal set. Given any such permutation, the minhash value for a set is that element of the set that appears first in the permuted order.

✦ *Minhash Signatures*: We may represent sets by picking some list of per- mutations and computing for each set its minhash signature, which is the sequence of minhash values obtained by applying each permutation on the list to that set. Given two sets, the expected fraction of the permutations that will yield the same minhash value is exactly the Jaccard similarity of the sets.

✦ *Efficient Minhashing*: Since it is not really possible to generate random permutations, it is normal to simulate a permutation by picking a random hash function and taking the minhash value for a set to be the least hash value of any of the set's members.

✦ *Locality-Sensitive Hashing for Signatures*: This technique allows us to avoid computing the similarity of every pair of sets or their minhash sig- natures. If we are given signatures for the sets, we may divide them into bands, and only measure the similarity of a pair of sets if they are identi- cal in at least one band. By choosing the size of bands appropriately, we can eliminate from consideration most of the pairs that do not meet our threshold of similarity.

✦ *Distance Measures*: A distance measure is a function on pairs of points in a space that satisfy certain axioms. The distance between two points is 0 if the points are the same, but greater than 0 if the points are different. The distance is symmetric; it does not matter in which order we consider the two points. A distance measure must satisfy the triangle inequality: the distance between two points is never more than the sum of the distances between those points and some third point.

✦ *Euclidean Distance*: The most common notion of distance is the Euclidean distance in an n-dimensional space. This distance, sometimes called the L_2-norm, is the square root of the sum of the

squares of the differences between the points in each dimension. Another distance suitable for Eu- clidean spaces, called Manhattan distance or the L_1-norm is the sum of the magnitudes of the differences between the points in each dimension.

✦ *Jaccard Distance*: One minus the Jaccard similarity is a distance measure, called the Jaccard distance.

✦ *Cosine Distance*: The angle between vectors in a vector space is the cosine distance measure. We can compute the cosine of that angle by taking the dot product of the vectors and dividing by the lengths of the vectors.

✦ *Edit Distance*: This distance measure applies to a space of strings, and is the number of insertions and/or deletions needed to convert one string into the other. The edit distance can also be computed as the sum of the lengths of the strings minus twice the length of the longest common subsequence of the strings.

✦ *Hamming Distance*: This distance measure applies to a space of vectors. The Hamming distance between two vectors is the number of positions in which the vectors differ.

✦ *Generalized Locality-Sensitive Hashing*: We may start with any collection of functions, such as the minhash functions, that can render a decision as to whether or not a pair of items should be candidates for similarity checking. The only constraint on these functions is that they provide a lower bound on the probability of saying "yes" if the distance (according to some distance measure) is below a given limit, and an upper bound on the probability of saying "yes" if the distance is above another given limit. We can then increase the probability of saying "yes" for nearby items and at the same time decrease the probability of saying "yes" for distant items to as great an extent as we wish, by applying an AND construction and an OR construction.

✦ *Random Hyperplanes and LSH for Cosine Distance*: We can get a set of basis functions to start a generalized LSH for the cosine distance measure by identifying each function with a list of randomly chosen vectors. We apply a function to a given vector v by taking the dot product of v with each vector on the list. The result is a sketch consisting of the signs (+1 or −1) of the dot products. The fraction of positions in which the sketches of two vectors agree, multiplied by 180, is an estimate of the angle between the two vectors.

✦ *LSH For Euclidean Distance*: A set of basis functions to start LSH for Euclidean distance can be obtained by choosing random lines and project- ing points onto those lines. Each line is broken into fixed-length intervals, and the function answers "yes" to a pair of points that fall into the same interval.

✦ *High-Similarity Detection by String Comparison*: An alternative approach to finding similar items, when the threshold of Jaccard similarity is close to 1, avoids using minhashing and LSH. Rather, the universal set is ordered, and sets are represented by strings, consisting their elements in order. The simplest way to avoid comparing all pairs of sets or their strings is to note that highly similar sets will have strings of approximately the same length. If we sort the strings, we can compare each string with only a small number of the immediately following strings.

✦ *Character Indexes*: If we represent sets by strings, and the similarity threshold is close to 1, we can index all strings by their first few characters. The prefix whose characters must be indexed is approximately the length of the string times the maximum Jaccard distance (1 minus the minimum Jaccard similarity).

✦ *Position Indexes*: We can index strings not only on the characters in their prefixes, but on the position of that character within the prefix. We reduce the number of pairs of strings that must be compared, because if two strings share a character that is not in the first position in both strings, then we know that either there are some preceding characters that are in the union but not the intersection, or there is an earlier symbol that appears in both strings.

✦ *Suffix Indexes*: We can also index strings based not only on the characters in their prefixes and the positions of those characters, but on the length of the character's suffix – the number of positions that follow it in the string. This structure further reduces the number of pairs that must be compared, because a common symbol with different suffix lengths implies additional characters that must be in the union but not in the intersection.

❖❖❖❖

4

MINING DATA STREAMS

Unit Structure

4.1 INTRODUCTION TO STREAMS CONCEPT:

In this chapter, we shall make another assumption: data arrives in a stream or streams, and if it is not processed immediately or stored, then it is lost forever. Moreover, we shall assume that the data arrives so rapidly that it is not feasible to store it all in active storage (i.e., in a conventional database), and then interact with it at the time of our choosing.

The algorithms for processing streams each involve summarization of the stream in some way. We shall start by considering how to make a useful sample of a stream and how to filter a stream to eliminate most of the "undesirable" elements. We then show how to estimate the number of different elements in a stream using much less storage than would be required if we listed all the elements we have seen.

Another approach to summarizing a stream is to look at only a fixed-length "window" consisting of the last n elements for some (typically large) n. We then query the window as if it were a relation in a database. If there are many streams and/or n is large, we may not be able to store the entire window for every stream, so we need to summarize even the windows. We address the fundamental problem of maintaining an approximate count on the number of 1's in the window of a bit stream, while using much less space than would be needed to store the entire window itself. This technique generalizes to approximating various kinds of sums.

Let us begin by discussing the elements of streams and stream processing. We explain the difference between streams and databases and the special problems that arise when dealing with streams. Some typical applications where the stream model applies will be examined.

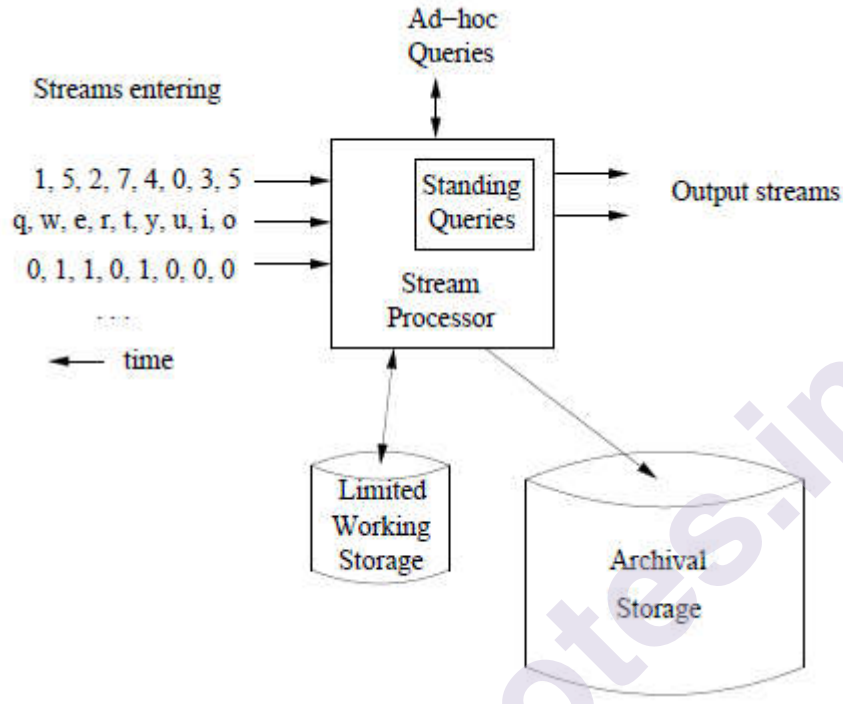


Figure 4.1: A data-stream-management system

4.2.1 A Data-Stream-Management System

In analogy to a database-management system, we can view a stream processor as a kind of data-management system, the high-level organization of which is suggested in Fig. 4.1. Any number of streams can enter the system. Each stream can provide elements at its own schedule; they need not have the same data rates or data types, and the time between elements of one stream need not be uniform. The fact that the rate of arrival of stream elements is not under the control of the system distinguishes stream processing from the processing of data that goes on within a database-management system. The latter system controls the rate at which data is read from the disk, and therefore never has to worry about data getting lost as it attempts to execute queries.

Streams may be archived in a large *archival store*, but we assume it is not possible to answer queries from the archival store. It could be examined only under special circumstances using time-consuming retrieval processes. There is also a *working store*, into which summaries or parts of streams may be placed, and which can be used for answering queries. The working store might be disk, or it might be main memory, depending on how fast we need to process queries. But

either way, it is of sufficiently limited capacity that it cannot store all the data from all the streams.

4.2.2 Examples of Stream Sources

Before proceeding, let us consider some of the ways in which stream data arises naturally.

Sensor Data

Imagine a temperature sensor bobbing about in the ocean, sending back to a base station a reading of the surface temperature each hour. The data produced by this sensor is a stream of real numbers. It is not a very interesting stream, since the data rate is so low. It would not stress modern technology, and the entire stream could be kept in main memory, essentially forever.

Now, give the sensor a GPS unit, and let it report surface height instead of temperature. The surface height varies quite rapidly compared with tempera- ture, so we might have the sensor send back a reading every tenth of a second. If it sends a 4-byte real number each time, then it produces 3.5 megabytes per day. It will still take some time to fill up main memory, let alone a single disk. But one sensor might not be that interesting. To learn something about ocean behavior, we might want to deploy a million sensors, each sending back a stream, at the rate of ten per second. A million sensors isn't very many; there would be one for every 150 square miles of ocean. Now we have 3.5 terabytes arriving every day, and we definitely need to think about what can be kept in working storage and what can only be archived.

Image Data

Satellites often send down to earth streams consisting of many terabytes of images per day. Surveillance cameras produce images with lower resolution than satellites, but there can be many of them, each producing a stream of images at intervals like one second. London is said to have six million such cameras, each producing a stream.

Internet and Web Traffic

A switching node in the middle of the Internet receives streams of IP packets from many inputs and routes them to its outputs. Normally, the job of the switch is to transmit data and not to retain it or query it. But there is a tendency to put more capability into the switch, e.g., the ability to detect denial-of-service attacks or the ability to reroute packets based on information about congestion in the network.

Web sites receive streams of various types. For example, Google receives sev- eral hundred million search queries per day. Yahoo! accepts billions of "clicks" per day on its various sites. Many

interesting things can be learned from these streams. For example, an increase in queries like "sore throat" enables us to track the spread of viruses. A sudden increase in the click rate for a link could indicate some news connected to that page, or it could mean that the link is broken and needs to be repaired.

4.2.3 Stream Queries

There are two ways that queries get asked about streams. We show in Fig. 4.1 a place within the processor where *standing queries* are stored. These queries are, in a sense, permanently executing, and produce outputs at appropriate times.

Example 4.1 : The stream produced by the ocean-surface-temperature sen- sor mentioned at the beginning of Section 4.1.2 might have a standing query to output an alert whenever the temperature exceeds 25 degrees centigrade. This query is easily answered, since it depends only on the most recent stream element.

Alternatively, we might have a standing query that, each time a new reading arrives, produces the average of the 24 most recent readings. That query also can be answered easily, if we store the 24 most recent stream elements. When a new stream element arrives, we can drop from the working store the 25th most recent element, since it will never again be needed (unless there is some other standing query that requires it).

Another query we might ask is the maximum temperature ever recorded by that sensor. We can answer this query by retaining a simple summary: the maximum of all stream elements ever seen. It is not necessary to record the entire stream. When a new stream element arrives, we compare it with the stored maximum, and set the maximum to whichever is larger. We can then answer the query by producing the current value of the maximum. Similarly, if we want the average temperature over all time, we have only to record two values: the number of readings ever sent in the stream and the sum of those readings. We can adjust these values easily each time a new reading arrives, and we can produce their quotient as the answer to the query.

The other form of query is *ad-hoc*, a question asked once about the current state of a stream or streams. If we do not store all streams in their entirety, as normally we can not, then we cannot expect to answer arbitrary queries about streams. If we have some idea what kind of queries will be asked through the ad-hoc query interface, then we can prepare for them by storing appropriate parts or summaries of streams as in Example 4.1.

If we want the facility to ask a wide variety of ad-hoc queries, a common approach is to store a *sliding window* of each stream in the working store. A sliding window can be the most recent n elements of a stream, for some n, or it can be all the elements that arrived within

the last t time units, e.g., one day. If we regard each stream element as a tuple, we can treat the window as a relation and query it with any SQL query. Of course the stream-management system must keep the window fresh, deleting the oldest elements as new ones come in.

Example 4.2 : Web sites often like to report the number of unique users over the past month. If we think of each login as a stream element, we can maintain a window that is all logins in the most recent month. We must associate the arrival time with each login, so we know when it no longer belongs to the window. If we think of the window as a relation Logins(name, time), then it is simple to get the number of unique users over the past month. The SQL query is:

```
SELECT COUNT(DISTINCT(name))

FROM Logins WHERE time >= t;
```

Here, t is a constant that represents the time one month before the current time.

Note that we must be able to maintain the entire stream of logins for the past month in working storage. However, for even the largest sites, that data is not more than a few terabytes, and so surely can be stored on disk.

4.2.4 Issues in Stream Processing

Before proceeding to discuss algorithms, let us consider the constraints under which we work when dealing with streams. First, streams often deliver elements very rapidly. We must process elements in real time, or we lose the opportunity to process them at all, without accessing the archival storage. Thus, it often is important that the stream-processing algorithm is executed in main memory, without access to secondary storage or with only rare accesses to secondary storage. Moreover, even when streams are "slow," as in the sensor-data example of Section 4.1.2, there may be many such streams. Even if each stream by itself can be processed using a small amount of main memory, the requirements of all the streams together can easily exceed the amount of available main memory.

Thus, many problems about streaming data would be easy to solve if we had enough memory, but become rather hard and require the invention of new techniques in order to execute them at a realistic rate on a machine of realistic size. Here are two generalizations about stream algorithms worth bearing in mind as you read through this chapter:

4.2.4.1 Often, it is much more efficient to get an approximate answer to our problem than an exact solution.

4.2.4.2 As in Chapter 3, a variety of techniques related to hashing turn out to be useful. Generally, these techniques introduce useful randomness into the algorithm's behavior, in order to produce an approximate answer that is very close to the true result.

4.3 SAMPLING DATA IN A STREAM

As our first example of managing streaming data, we shall look at extracting reliable samples from a stream. As with many stream algorithms, the "trick" involves using hashing in a somewhat unusual way.

4.3.1 A Motivating Example

The general problem we shall address is selecting a subset of a stream so that we can ask queries about the selected subset and have the answers be statistically representative of the stream as a whole. If we know what queries are to be asked, then there are a number of methods that might work, but we are looking for a technique that will allow ad-hoc queries on the sample. We shall look at a particular problem, from which the general idea will emerge.

Our running example is the following. A search engine receives a stream of queries, and it would like to study the behavior of typical users.[1] We assume the stream consists of tuples (user, query, time). Suppose that we want to answer queries such as "What fraction of the typical user's queries were repeated over the past month?" Assume also that we wish to store only 1/10th of the stream elements.

The obvious approach would be to generate a random number, say an integer from 0 to 9, in response to each search query. Store the tuple if and only if the random number is 0. If we do so, each user has, on average, 1/10th of their queries stored. Statistical fluctuations will introduce some noise into the data, but if users issue many queries, the law of large numbers will assure us that most users will have a fraction quite close to 1/10th of their queries stored.

However, this scheme gives us the wrong answer to the query asking for the average number of duplicate queries for a user. Suppose a user has issued s search queries one time in the past month, d search queries twice, and no search queries more than twice. If we have a 1/10th sample, of queries, we shall see in the sample for that user an expected $s/10$ of the search queries issued once. Of the d search queries issued twice, only $d/100$ will appear twice in the sample; that fraction is d times the probability that both occurrences of the query will be in the 1/10th sample. Of the queries that appear twice in the full stream, $18d/100$ will appear exactly once. To see why, note that 18/100 is the probability that one of the two occurrences will be in the 1/10th of the stream that is selected, while the other is in the 9/10th that is not selected.

The correct answer to the query about the fraction of repeated searches is $d/(s+d)$. However, the answer we shall obtain from the sample is $d/(10s+19d)$. To derive the latter formula, note that $d/100$ appear twice, while $s/10+18d/100$ appear once. Thus, the fraction appearing twice in the sample is $d/100$ divided by $d/100 + s/10 + 18d/100$. This ratio is $d/(10s + 19d)$. For no positive values of s and d is $d/(s + d) = d/(10s + 19d)$.

4.3.2 Obtaining a Representative Sample

The query of Section 4.2.1, like many queries about the statistics of typical users, cannot be answered by taking a sample of each user's search queries. Thus, we must strive to pick 1/10th of the users, and take all their searches for the sample, while taking none of the searches from other users. If we can store a list of all users, and whether or not they are in the sample, then we could do the following. Each time a search query arrives in the stream, we look up the user to see whether or not they are in the sample. If so, we add this search query to the sample, and if not, then not. However, if we have no record of ever having seen this user before, then we generate a random integer between 0 and 9. If the number is 0, we add this user to our list with value "in," and if the number is other than 0, we add the user with the value "out."

That method works as long as we can afford to keep the list of all users and their in/out decision in main memory, because there isn't time to go to disk for every search that arrives. By using a hash function, one can avoid keeping the list of users. That is, we hash each user name to one of ten buckets, 0 through 9. If the user hashes to bucket 0, then accept this search query for the sample, and if not, then not.

Note we do not actually store the user in the bucket; in fact, there is no data in the buckets at all. Effectively, we use the hash function as a random- number generator, with the important property that, when applied to the same user several times, we always get the same "random" number. That is, without storing the in/out decision for any user, we can reconstruct that decision any time a search query by that user arrives.

More generally, we can obtain a sample consisting of any rational fraction a/b of the users by hashing user names to b buckets, 0 through $b - 1$. Add the search query to the sample if the hash value is less than a.

4.3.3 The General Sampling Problem

The running example is typical of the following general problem. Our stream consists of tuples with n components. A subset of the components are the *key* components, on which the selection of the sample will be based. In our running example, there are three components – user, query, and time – of which only *user* is in the key. However, we could also take a sample of queries by making *query* be the key, or even take a sample of user-query pairs by making both those components form the key.

To take a sample of size a/b, we hash the key value for each tuple to b buckets, and accept the tuple for the sample if the hash value is less than a. If the key consists of more than one component, the hash function needs to combine the values for those components to make a single hash-value. The result will be a sample consisting of all

tuples with certain key values. The selected key values will be approximately *a/b* of all the key values appearing in the stream.

4.3.4 Varying the Sample Size

Often, the sample will grow as more of the stream enters the system. In our running example, we retain all the search queries of the selected 1/10th of the users, forever. As time goes on, more searches for the same users will be accumulated, and new users that are selected for the sample will appear in the stream.

If we have a budget for how many tuples from the stream can be stored as the sample, then the fraction of key values must vary, lowering as time goes on. In order to assure that at all times, the sample consists of all tuples from a subset of the key values, we choose a hash function h from key values to a very large number of values $0, 1, \ldots, B-1$. We maintain a *threshold* t, which initially can be the largest bucket number, $B - 1$. At all times, the sample consists of those tuples whose key K satisfies $h(K) \leq t$. New tuples from the stream are added to the sample if and only if they satisfy the same condition.

If the number of stored tuples of the sample exceeds the allotted space, we lower t to $t-1$ and remove from the sample all those tuples whose key K hashes to t. For efficiency, we can lower t by more than 1, and remove the tuples with several of the highest hash values, whenever we need to throw some key values out of the sample. Further efficiency is obtained by maintaining an index on the hash value, so we can find all those tuples whose keys hash to a particular value quickly.

4.3.5 Exercises for Section 4.2

Exercise 4.2.1 : Suppose we have a stream of tuples with the schema Grades(university, courseID, studentID, grade) Assume universities are unique, but a courseID is unique only within a uni- versity (i.e., different universities may have different courses with the same ID, e.g., "CS101") and likewise, studentID's are unique only within a university (different universities may assign the same ID to different students). Suppose we want to answer certain queries approximately from a 1/20th sample of the data. For each of the queries below, indicate how you would construct the sample. That is, tell what the key attributes should be.

(a) For each university, estimate the average number of students in a course.

(b) Estimate the fraction of students who have a GPA of 3.5 or more.

(c) Estimate the fraction of courses where at least half the students got "A."

4.4 FILTERING STREAMS

Another common process on streams is selection, or filtering. We want to accept those tuples in the stream that meet a criterion. Accepted tuples are passed to another process as a stream, while other tuples are dropped. If the selection criterion is a property of the tuple that can be calculated (e.g., the first component is less than 10), then the selection is easy to do. The problem becomes harder when the criterion involves lookup for membership in a set. It is especially hard, when that set is too large to store in main memory. In this section, we shall discuss the technique known as "Bloom filtering" as a way to eliminate most of the tuples that do not meet the criterion.

4.4.1 A Motivating Example

Again let us start with a running example that illustrates the problem and what we can do about it. Suppose we have a set S of one billion allowed email addresses – those that we will allow through because we believe them not to be spam. The stream consists of pairs: an email address and the email itself. Since the typical email address is 20 bytes or more, it is not reasonable to store S in main memory. Thus, we can either use disk accesses to determine whether or not to let through any given stream element, or we can devise a method that requires no more main memory than we have available, and yet will filter most of the undesired stream elements.

Suppose for argument's sake that we have one gigabyte of available main memory. In the technique known as *Bloom filtering*, we use that main memory as a bit array. In this case, we have room for eight billion bits, since one byte equals eight bits. Devise a hash function h from email addresses to eight billion buckets. Hash each member of S to a bit, and set that bit to 1. All other bits of the array remain 0.

Since there are one billion members of S, approximately 1/8th of the bits will be 1. The exact fraction of bits set to 1 will be slightly less than 1/8th, because it is possible that two members of S hash to the same bit. We shall discuss the exact fraction of 1's in Section 4.3.3. When a stream element arrives, we hash its email address. If the bit to which that email address hashes is 1, then we let the email through. But if the email address hashes to a 0, we are certain that the address is not in S, so we can drop this stream element.

Unfortunately, some spam email will get through. Approximately 1/8th of the stream elements whose email address is not in S will happen to hash to a bit whose value is 1 and will be let through. Nevertheless, since the majority of emails are spam (about 80% according to some reports), eliminating 7/8th of the spam is a significant benefit. Moreover, if we want to eliminate every spam, we need only check for membership in S those good and bad emails that get through the filter. Those checks will require the use of secondary memory to access S itself. There are also other options, as

we shall see when we study the general Bloom-filtering technique. As a simple example, we could use a cascade of filters, each of which would eliminate 7/8th of the remaining spam.

4.4.2 The Bloom Filter

A *Bloom filter* consists of:

1. An array of n bits, initially all 0's.
2. A collection of hash functions $h_1, h_2, \ldots, h_k$. Each hash function maps "key" values to n buckets, corresponding to the n bits of the bit-array.
3. A set S of m key values.

The purpose of the Bloom filter is to allow through all stream elements whose keys are in S, while rejecting most of the stream elements whose keys are not in S.

To initialize the bit array, begin with all bits 0. Take each key value in S and hash it using each of the k hash functions. Set to 1 each bit that is $h_i(K)$ for some hash function h_i and some key value K in S.

To test a key K that arrives in the stream, check that all of $h_1(K)$, $h_2(K)$, . . . , $h_k(K)$ are 1's in the bit-array. If all are 1's, then let the stream element through. If one or more of these bits are 0, then K could not be in S, so reject the stream element.

4.4.3 Analysis of Bloom Filtering

If a key value is in S, then the element will surely pass through the Bloom filter. However, if the key value is not in S, it might still pass. We need to understand how to calculate the probability of a *false positive*, as a function of n, the bit-array length, m the number of members of S, and k, the number of hash functions.

The model to use is throwing darts at targets. Suppose we have x targets and y darts. Any dart is equally likely to hit any target. After throwing the darts, how many targets can we expect to be hit at least once? The analysis is similar to the analysis in Section 3.4.2, and goes as follows:

4.4.3.1 The probability that a given dart will not hit a given target is $(x-1)/x$.

4.4.3.2 The probability that none of the y darts will hit a given target is $(\frac{x-1}{x})^y$.

We can write this expression as $(1-\frac{1}{x})^{x(\frac{y}{x})}$.

4.4.3.3 Using the approximation $(1-\epsilon)^{1/\epsilon} = 1/e$ for small ϵ (recall Section 1.3.5), we conclude that the probability that none of the y darts hit a given target is $e^{-y/x}$.

Example 4.3 : Consider the running example of Section 4.3.1. We can use the above calculation to get the true expected number of 1's in the bit array. Think of each bit as a target, and each member of S as a dart. Then the probability that a given bit will be 1 is the probability that the corresponding target will be hit by one or more darts. Since there are one billion members of S, we have $y = 10^9$ darts. As there are eight billion bits, there are $x = 8 \times 10^9$ targets. Thus, the probability that a given target is not hit is $e^{-y/x} = e^{-1/8}$ and the probability that it *is* hit is $1 - e^{-1/8}$. That quantity is about 0.1175. In Section 4.3.1 we suggested that 1/8 = 0.125 is a good approximation, which it is, but now we have the exact calculation.

We can apply the rule to the more general situation, where set S has m members, the array has n bits, and there are k hash functions. The number of targets is $x = n$, and the number of darts is $y = km$. Thus, the probability that a bit remains 0 is $e^{-km/n}$. We want the fraction of 0 bits to be fairly large, or else the probability that a nonmember of S will hash at least once to a 0 becomes too small, and there are too many false positives. For example, we might choose k, the number of hash functions to be n/m or less. Then the probability of a 0 is at least e^{-1} or 37%. In general, the probability of a false positive is the probability of a 1 bit, which is $1 - e^{-km/n}$, raised to the kth power, i.e., $(1 - e^{-km/n})^k$.

Example 4.4 : In Example 4.3 we found that the fraction of 1's in the array of our running example is 0.1175, and this fraction is also the probability of a false positive. That is, a nonmember of S will pass through the filter if it hashes to a 1, and the probability of it doing so is 0.1175.

Suppose we used the same S and the same array, but used two different hash functions. This situation corresponds to throwing two billion darts at eight billion targets, and the probability that a bit remains 0 is $e^{-1/4}$. In order to be a false positive, a nonmember of S must hash twice to bits that are 1, and this probability is $(1 - e^{-1/4})^2$, or approximately 0.0493. Thus, adding a second hash function for our running example is an improvement, reducing the false-positive rate from 0.1175 to 0.0493.

4.4.4 Exercises for Section 4.3

Exercise 4.3.1 : For the situation of our running example (8 billion bits, 1 billion members of the set S), calculate the false-positive rate if we use three hash functions? What if we use four hash functions?

! Exercise 4.3.2 : Suppose we have n bits of memory available, and our set S has m members. Instead of using k hash functions, we could divide the n bits into k arrays, and hash once to each array. As a function of n, m, and k, what is the probability of a false positive? How does it compare with using k hash functions into a single array?

!! Exercise 4.3.3 : As a function of n, the number of bits and m the number of members in the set S, what number of hash functions minimizes the false-positive rate?

4.5 COUNTING DISTINCT ELEMENTS IN A STREAM

In this section we look at a third simple kind of processing we might want to do on a stream. As with the previous examples – sampling and filtering – it is somewhat tricky to do what we want in a reasonable amount of main memory, so we use a variety of hashing and a randomized algorithm to get approximately what we want with little space needed per stream.

4.5.1 The Count-Distinct Problem

Suppose stream elements are chosen from some universal set. We would like to know how many different elements have appeared in the stream, counting either from the beginning of the stream or from some known time in the past.

Example 4.5 : As a useful example of this problem, consider a Web site gath- ering statistics on how many unique users it has seen in each given month. The universal set is the set of logins for that site, and a stream element is generated each time someone logs in. This measure is appropriate for a site like Amazon, where the typical user logs in with their unique login name.

A similar problem is a Web site like Google that does not require login to issue a search query, and may be able to identify users only by the IP address from which they send the query. There are about 4 billion IP addresses,[2] sequences of four 8-bit bytes will serve as the universal set in this case.

The obvious way to solve the problem is to keep in main memory a list of all the elements seen so far in the stream. Keep them in an efficient search structure such as a hash table or search tree, so one can quickly add new elements and check whether or not the element that just arrived on the stream was already seen. As long as the number of distinct elements is not too great, this structure can fit in main memory and there is little problem obtaining an exact answer to the question how many distinct elements appear in the stream.

However, if the number of distinct elements is too great, or if there are too many streams that need to be processed at once (e.g., Yahoo! wants to count the number of unique users viewing each of its pages in a month), then we cannot store the needed data in main memory. There are several options. We could use more machines, each machine handling only one or several of the streams. We could store most of the data structure in secondary memory and batch stream elements so whenever we brought a disk block to main memory there would be many tests and updates to be performed on the data in that block. Or

we could use the strategy to be discussed in this section, where we only estimate the number of distinct elements but use much less memory than the number of distinct elements.

4.5.2 The Flajolet-Martin Algorithm

It is possible to estimate the number of distinct elements by hashing the ele- ments of the universal set to a bit-string that is sufficiently long. The length of the bit-string must be sufficient that there are more possible results of the hash function than there are elements of the universal set. For example, 64 bits is sufficient to hash URL's. We shall pick many different hash functions and hash each element of the stream using these hash functions. The important property of a hash function is that when applied to the same element, it always produces the same result. Notice that this property was also essential for the sampling technique of Section 4.2.

The idea behind the Flajolet-Martin Algorithm is that the more different elements we see in the stream, the more different hash-values we shall see. As we see more different hash-values, it becomes more likely that one of these values will be "unusual." The particular unusual property we shall exploit is that the value ends in many 0's, although many other options exist.

Whenever we apply a hash function h to a stream element a, the bit string $h(a)$ will end in some number of 0's, possibly none. Call this number the *tail length* for a and h. Let R be the maximum tail length of any a seen so far in the stream. Then we shall use estimate 2^R for the number of distinct elements seen in the stream.

This estimate makes intuitive sense. The probability that a given stream element a has $h(a)$ ending in at least r 0's is 2^{-r}. Suppose there are m distinct elements in the stream. Then the probability that none of them has tail length at least r is $(1 - 2^{-r})^m$. This sort of expression should be familiar by now.We can rewrite it as $(1 - 2^{-r})^{2^r m}$. Assuming r is reasonably large, the inner expression is of the form $(1 - \epsilon)^{1/\epsilon}$, which is approximately $1/e$. Thus, the probability of not finding a stream element with as many as r 0's at the end of its hash value is $e^{-m2^{-r}}$. We can conclude:

1. If m is much larger than 2^r, then the probability that we shall find a tail of length at least r approaches 1.

2. If m is much less than 2^r, then the probability of finding a tail length at least r approaches 0.

We conclude from these two points that the proposed estimate of m, which is 2^R (recall R is the largest tail length for any stream element) is unlikely to be either much too high or much too low.

4.5.3 Combining Estimates

Unfortunately, there is a trap regarding the strategy for combining the estimates of m, the number of distinct elements, that we obtain by using many different hash functions. Our first assumption would be that if we take the average of the values 2^R that we get from each hash function, we shall get a value that approaches the true m, the more hash functions we use. However, that is not the case, and the reason has to do with the influence an overestimate has on the average.

Consider a value of r such that 2^r is much larger than m. There is some probability p that we shall discover r to be the largest number of 0's at the end of the hash value for any of the m stream elements. Then the probability of finding $r + 1$ to be the largest number of 0's instead is at least $p/2$. However, if we do increase by 1 the number of 0's at the end of a hash value, the value of 2^R doubles. Consequently, the contribution from each possible large R to the expected value of 2^R grows as R grows, and the expected value of 2^R is actually infinite.[3]

Another way to combine estimates is to take the median of all estimates. The median is not affected by the occasional outsized value of 2^R, so the worry described above for the average should not carry over to the median. Unfortu- nately, the median suffers from another defect: it is always a power of 2. Thus, no matter how many hash functions we use, should the correct value of m be between two powers of 2, say 400, then it will be impossible to obtain a close estimate.

There is a solution to the problem, however. We can combine the two methods. First, group the hash functions into small groups, and take their average. Then, take the median of the averages. It is true that an occasional outsized 2^R will bias some of the groups and make them too large. However, taking the median of group averages will reduce the influence of this effect almost to nothing. Moreover, if the groups themselves are large enough, then the averages can be essentially any number, which enables us to approach the true value m as long as we use enough hash functions. In order to guarantee that any possible average can be obtained, groups should be of size at least a small multiple of $\log_2 m$.

4.5.4 Space Requirements

Observe that as we read the stream it is not necessary to store the elements seen. The only thing we need to keep in main memory is one integer per hash function; this integer records the largest tail length seen so far for that hash function and any stream element. If we are processing only one stream, we could use millions of hash functions, which is far more than we need to get a close estimate. Only if we are trying to process many streams at the same time would main memory constrain the number of hash functions we could associate with any one stream. In practice, the time it takes to compute hash values for each

stream element would be the more significant limitation on the number ofhash functions we use.

4.5.5 Exercises for Section 4.4

Exercise 4.4.1 : Suppose our stream consists of the integers 3, 1, 4, 1, 5, 9, 2,6, 5. Our hash functions will all be of the form $h(x) = ax + b \mod 32$ for some a and b. You should treat the result as a 5-bit binary integer. Determine the tail length for each stream element and the resulting estimate of the number of distinct elements if the hash function is:

(a) $h(x) = 2x + 1 \mod 32$.

(b) $h(x) = 3x + 7 \mod 32$.

(c) $h(x) = 4x \mod 32$.

! Exercise 4.4.2 : Do you see any problems with the choice of hash functions in Exercise 4.4.1? What advice could you give someone who was going to use a hash function of the form $h(x) = ax + b \mod 2^k$?

4.6 ESTIMATING MOMENTS

In this section we consider a generalization of the problem of counting distinct elements in a stream. The problem, called computing "moments," involves the distribution of frequencies of different elements in the stream. We shall define moments of all orders and concentrate on computing second moments, from which the general algorithm for all moments is a simple extension.

4.6.1 Definition of Moments

Suppose a stream consists of elements chosen from a universal set. Assume the universal set is ordered so we can speak of the ith element for any i. Let m_i be the number of occurrences of the ith element for any i. Then the *kth-order moment* (or just kth moment) of the stream is the sum over all i of $(m_i)^k$.

Example 4.6 : The 0th moment is the sum of 1 for each m_i that is greater than 0.[4] That is, the 0th moment is a count of the number of distinct elements in the stream. We can use the method of Section 4.4 to estimate the 0th moment of a stream.

The 1st moment is the sum of the m_i's, which must be the length of the stream. Thus, first moments are especially easy to compute; just count the length of the stream seen so far.

The second moment is the sum of the squares of the m_i's. It is some- times called the *surprise number*, since it measures how uneven the distribu- tion of elements in the stream is. To see the distinction, suppose we have a stream of length 100, in which

eleven different elements appear. The most even distribution of these eleven elements would have one appearing 10 times and the other ten appearing 9 times each. In this case, the surprise number is $10^2 + 10 \times 9^2 = 910$. At the other extreme, one of the eleven elements could appear 90 times and the other ten appear 1 time each. Then, the surprise number would be $90^2 + 10 \times 1^2 = 8110$.

As in Section 4.4, there is no problem computing moments of any order if we can afford to keep in main memory a count for each element that appears in the stream. However, also as in that section, if we cannot afford to use that much memory, then we need to estimate the kth moment by keeping a limited number of values in main memory and computing an estimate from these values. For the case of distinct elements, each of these values were counts of the longest tail produced by a single hash function. We shall see another form of value that is useful for second and higher moments.

4.6.2 The Alon-Matias-Szegedy Algorithm for Second Moments

For now, let us assume that a stream has a particular length n. We shall show how to deal with growing streams in the next section. Suppose we do not have enough space to count all the m_i's for all the elements of the stream. We can still estimate the second moment of the stream using a limited amount of space; the more space we use, the more accurate the estimate will be. We compute some number of *variables*. For each variable X, we store:

1. A particular element of the universal set, which we refer to as *X.element* , and

2. An integer *X.value*, which is the *value* of the variable. To determine the value of a variable X, we choose a position in the stream between 1 and n, uniformly and at random. Set *X.element* to be the element found there, and initialize *X.value* to 1. As we read the stream, add 1 to *X.value* each time we encounter another occurrence of *X.element* .

Example 4.7 : Suppose the stream is *a, b, c, b, d, a, c, d, a, b, d, c, a, a, b*. The length of the stream is $n = 15$. Since *a* appears 5 times, *b* appears 4 times, and *c* and *d* appear three times each, the second moment for the stream is $5^2 + 4^2 + 3^2 + 3^2 = 59$. Suppose we keep three variables, X_1, X_2, and X_3. Also, assume that at "random" we pick the 3rd, 8th, and 13th positions to define these three variables.

When we reach position 3, we find element *c*, so we set X_1.*element* $= c$ and X_1.*value* $= 1$. Position 4 holds *b*, so we do not change X_1. Likewise, nothing happens at positions 5 or 6. At position 7, we see *c* again, so we set X_1.*value* $= 2$.

At position 8 we find *d*, and so set X_2.*element* $= d$ and X_2.*value* $= 1$. Positions 9 and 10 hold *a* and *b*, so they do not affect X_1 or X_2. Position 11 holds *d* so we set X_2.*value* $= 2$, and position 12 holds *c* so

we set $X_1.value = 3$. At position 13, we find element a, and so set $X_3.element = a$ and $X_3.value = 1$. Then, at position 14 we see another a and so set $X_3.value = 2$. Position 15, with element b does not affect any of the variables, so we are done, with final values $X_1.value = 3$ and $X_2.value = X_3.value = 2$.

We can derive an estimate of the second moment from any variable X. This estimate is $n(2X.value - 1)$.

Example 4.8 : Consider the three variables from Example 4.7. From X_1 we derive the estimate $n(2X_1.value - 1) = 15 \times (2 \times 3 - 1) = 75$. The other two variables, X_2 and X_3, each have value 2 at the end, so their estimates are $15 \times (2 \times 2 - 1) = 45$. Recall that the true value of the second moment for this stream is 59. On the other hand, the average of the three estimates is 55, a fairly close approximation.

Why the Alon-Matias-Szegedy Algorithm Works

We can prove that the expected value of any variable constructed as in Sec- tion 4.5.2 is the second moment of the stream from which it is constructed. Some notation will make the argument easier to follow. Let $e(i)$ be the stream element that appears at position i in the stream, and let $c(i)$ be the number of times element $e(i)$ appears in the stream among positions $i, i + 1, \ldots, n$.

Example 4.9 : Consider the stream of Example 4.7. $e(6) = a$, since the 6th position holds a. Also, $c(6) = 4$, since a appears at positions 9, 13, and 14, as well as at position 6. Note that a also appears at position 1, but that fact does not contribute to $c(6)$. Q

The expected value of $n(2X.value - 1)$ is the average over all positions i between 1 and n of $n(2c(i) - 1)$, that is

$$E(n(2X.value - 1)) = \frac{1}{n}\sum_{i=1}^{n} n(2c(i) - 1)$$

We can simplify the above by canceling factors $1/n$ and n, to get

$$E(n(2X.value - 1)) = \sum_{i=1}^{n} (2c(i) - 1)$$

However, to make sense of the formula, we need to change the order of summation by grouping all those positions that have the same element. For instance, concentrate on some element a that appears m_a times in the stream. The term for the last position in which a appears must be $2 \times 1 - 1 = 1$. The term for the next-to-last position in which a appears is $2 \times 2 - 1 = 3$. The positions with a before that yield terms 5, 7, and so on, up to $2m_a - 1$, which is the term for the first position in which a appears. That is, the formula for the expected value of $2X.value - 1$ can be written:

$$E(n(2X.value - 1)) = \sum_{a} 1 + 3 + 5 + \cdots + (2m_a - 1)$$

Note that $1 + 3 + 5 + \cdots + (2m_a - 1) = (m_a)^2$. The proof is an easy induction on the number of terms in the sum. Thus, E(n(2X.value − 1))= $\Sigma_a (m_a)^2$, which is the definition of the second moment.

Higher-Order Moments

We estimate *k*th moments, for $k > 2$, in essentially the same way as we estimate second moments. The only thing that changes is the way we derive an estimate from a variable. In Section 4.5.2 we used the formula $n(2v - 1)$ to turn a value v, the count of the number of occurrences of some particular stream element a, into an estimate of the second moment. Then, in Section 4.5.3 we saw why this formula works: the terms $2v - 1$, for $v = 1, 2, \ldots, m$ sum to m^2, where m is the number of times a appears in the stream.

Notice that $2v - 1$ is the difference between v^2 and $(v - 1)^2$. Suppose we wanted the third moment rather than the second. Then all we have to do is replace $2v-1$ by $v^3-(v-1)^3 = 3v^2-3v+1$. Then $m\Sigma_{v=1}^{m} 3v^2-3v+1 = m^3$, so we can use as our estimate of the third moment the formula $n(3v^2 - 3v + 1)$, where $v = X.value$ is the value associated with some variable X. More generally, we can estimate kth moments for any $k \geq 2$ by turning value $v = X.value$ into $n(v^k - (v - 1)^k)$

4.6.3 Dealing With Infinite Streams

Technically, the estimate we used for second and higher moments assumes that n, the stream length, is a constant. In practice, n grows with time. That fact, by itself, doesn't cause problems, since we store only the values of variables and multiply some function of that value by n when it is time to estimate the moment. If we count the number of stream elements seen and store this value, which only requires log n bits, then we have n available whenever we need it.

A more serious problem is that we must be careful how we select the positions for the variables. If we do this selection once and for all, then as the stream gets longer, we are biased in favor of early positions, and the estimate of the moment will be too large. On the other hand, if we wait too long to pick positions, then early in the stream we do not have many variables and so will get an unreliable estimate.

The proper technique is to maintain as many variables as we can store at all times, and to throw some out as the stream grows. The discarded variables are replaced by new ones, in such a way that at all times, the probability of picking any one position for a variable is the same as that of picking any other position. Suppose we have space to store s variables. Then the first s positions of the stream are each picked as the position of one of the s variables.

Inductively, suppose we have seen n stream elements, and the

probability of any particular position being the position of a variable is uniform, that is s/n. When the $(n+1)$st element arrives, pick that position with probability $s/(n+1)$. If not picked, then the s variables keep their same positions. However, if the $(n + 1)$st position is picked, then throw out one of the current s variables, with equal probability. Replace the one discarded by a new variable whose element is the one at position $n + 1$ and whose value is 1.

Surely, the probability that position $n + 1$ is selected for a variable is what it should be: $s/(n + 1)$. However, the probability of every other position also is $s/(n + 1)$, as we can prove by induction on n. By the inductive hypothesis, before the arrival of the $(n + 1)$st stream element, this probability was s/n. With probability $1 - s/(n + 1)$ the $(n + 1)$st position will not be selected, and the probability of each of the first n positions remains s/n. However, with probability $s/(n + 1)$, the $(n + 1)$st position is picked, and the probability for each of the first n positions is reduced by factor $(s - 1)/s$. Considering the two cases, the probability of selecting each of the first n positions is

Thus, we have shown by induction on the stream length n that all positions have equal probability s/n of being chosen as the position of a variable.

4.6.4 Exercises for Section 4.5

Exercise 4.5.1 : Compute the surprise number (second moment) for the stream 3, 1, 4, 1, 3, 4, 2, 1, 2. What is the third moment of this stream?

> A General Stream-Sampling Problem
>
> Notice that the technique described in Section 4.5.5 actually solves a more general problem. It gives us a way to maintain a sample of s stream elements so that at all times, all stream elements are equally likely to be selected for the sample.
>
> As an example of where this technique can be useful, recall that in Section 4.2 we arranged to select all the tuples of a stream having key value in a randomly selected subset. Suppose that, as time goes on, there are too many tuples associated with any one key. We can arrange to limit the number of tuples for any key K to a fixed constant s by using the technique of Section 4.5.5 whenever a new tuple for key K arrives.

! Exercise 4.5.2 : If a stream has n elements, of which m are distinct, what are the minimum and maximum possible surprise number, as a function of m and n?

Exercise 4.5.3 : Suppose we are given the stream of Exercise 4.5.1, to which we apply the Alon-Matias-Szegedy Algorithm to estimate the surprise number. For each possible value of i, if X_i is a variable

starting position i, what is the value of $X_i.value$?

Exercise 4.5.4 : Repeat Exercise 4.5.3 if the intent of the variables is to com- pute third moments. What is the value of each variable at the end? What estimate of the third moment do you get from each variable? How does the average of these estimates compare with the true value of the third moment?

Exercise 4.5.5 : Prove by induction on m that $1 + 3 + 5 + \cdots + (2m - 1) = m^2$.

Exercise 4.5.6 : If we wanted to compute fourth moments, how would we convert $X.value$ to an estimate of the fourth moment?

4.6 COUNTING ONES IN A WINDOW

We now turn our attention to counting problems for streams. Suppose we have a window of length N on a binary stream. We want at all times to be able to answer queries of the form "how many 1's are there in the last k bits?" for any $k \leq N$. As in previous sections, we focus on the situation where we cannot afford to store the entire window. After showing an approximate algorithm for the binary case, we discuss how this idea can be extended to summing numbers.

4.6.1 The Cost of Exact Counts

To begin, suppose we want to be able to count exactly the number of 1's in the last k bits for any $k \leq N$. Then we claim it is necessary to store all N bits of the window, as any representation that used fewer than N bits could not work. In proof, suppose we have a representation that uses fewer than N bits to represent the N bits in the window. Since there are 2^N sequences of Nbits, but fewer than 2^N representations, there must be two different bit strings w and x that have the same representation. Since $w /= x$, they must differ in at least one bit. Let the last $k - 1$ bits of w and x agree, but let them differ on the kth bit from the right end.

Example 4.10 : If $w = 0101$ and $x = 1010$, then $k = 1$, since scanning from the right, they first disagree at position 1. If $w = 1001$ and $x = 0101$, then $k = 3$, because they first disagree at the third position from the right.

Suppose the data representing the contents of the window is whatever se- quence of bits represents both w and x. Ask the query "how many 1's are in the last k bits?" The query-answering algorithm will produce the same an- swer, whether the window contains w or x, because the algorithm can only see their representation. But the correct answers are surely different for these two bit-strings. Thus, we have proved that we must use at least N bits to answer queries about the last k bits for any k.

In fact, we need N bits, even if the only query we can ask is "how

many 1's are in the entire window of length N?" The argument is similar to that used above. Suppose we use fewer than N bits to represent the window, and therefore we can find w, x, and k as above. It might be that w and x have the same number of 1's, as they did in both cases of Example 4.10. However, if we follow the current window by any $N - k$ bits, we will have a situation where the true window contents resulting from w and x are identical except for the leftmost bit, and therefore, their counts of 1's are unequal. However, since the representations of w and x are the same, the representation of the window must still be the same if we feed the same bit sequence to these representations. Thus, we can force the answer to the query "how many 1's in the window?" to be incorrect for one of the two possible window contents.

4.6.2 The Datar-Gionis-Indyk-Motwani Algorithm

We shall present the simplest case of an algorithm called DGIM. This version of the algorithm uses $O(\log^2 N)$ bits to represent a window of N bits, and allows us to estimate the number of 1's in the window with an error of no more than 50%. Later, we shall discuss an improvement of the method that limits the error to any fraction $\square > 0$, and still uses only $O(\log^2 N)$ bits (although with a constant factor that grows as shrinks).

To begin, each bit of the stream has a *timestamp*, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on.

Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by $\log_2 N$ bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is.

We divide the window into *buckets*,[5] consisting of:

1. The timestamp of its right (most recent) end.
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the *size* of the bucket.

To represent a bucket, we need $\log_2 N$ bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need $\log_2 \log_2 N$ bits. The reason is that we know this number i is a power of 2, say 2^j, so we can represent i by coding j in binary. Since j is at most $\log_2 N$, it requires $\log_2 \log_2 N$ bits. Thus, $O(\log N)$ bits suffice to represent a bucket.

There are six rules that must be followed when representing a stream by buckets.

- The right end of a bucket is always a position with a 1.

- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

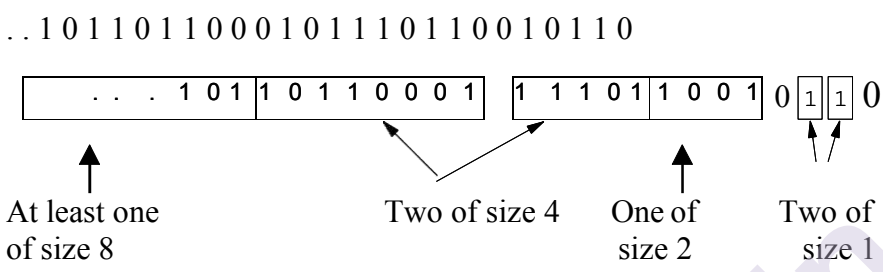


Figure 4.2: A bit-stream divided into buckets following the DGIM rules

Example 4.11 : Figure 4.2 shows a bit stream divided into buckets in a way that satisfies the DGIM rules. At the right (most recent) end we see two buckets of size 1. To its left we see one bucket of size 2. Note that this bucket covers four positions, but only two of them are 1. Proceeding left, we see two buckets of size 4, and we suggest that a bucket of size 8 exists further left.

Notice that it is OK for some 0's to lie between buckets. Also, observe from Fig. 4.2 that the buckets do not overlap; there are one or two of each size up to the largest size, and sizes only increase moving left. Q

In the next sections, we shall explain the following about the DGIM algo- rithm:

1. Why the number of buckets representing a window must be small.
2. How to estimate the number of 1's in the last k bits for any k, with an error no greater than 50%.
3. How to maintain the DGIM conditions as new bits enter the stream.

4.6.3 Storage Requirements for the DGIM Algorithm

We observed that each bucket can be represented by $O(\log N)$ bits. If the window has length N, then there are no more than N 1's, surely. Suppose the largest bucket is of size 2^j. Then j cannot exceed $\log_2 N$, or else there are more 1's in this bucket than there are 1's in the entire window. Thus, there are at most two buckets of all sizes from $\log_2 N$ down to 1, and no buckets of larger sizes.

We conclude that there are $O(\log N)$ buckets. Since each bucket can be represented in $O(\log N)$ bits, the total space required for all the buckets repre- senting a window of size N is $O(\log^2 N)$.

4.6.4 Query Answering in the DGIM Algorithm

Suppose we are asked how many 1's there are in the last k bits of the window, for some $1 \leq k \leq N$. Find the bucket b with the earliest timestamp that includes at least some of the k most recent bits. Estimate the number of 1's to be the sum of the sizes of all the buckets to the right (more recent) than bucket b, plus half the size of b itself.

Example 4.12 : Suppose the stream is that of Fig. 4.2, and $k = 10$. Then the query asks for the number of 1's in the ten rightmost bits, which happen to be 0110010110. Let the current timestamp (time of the rightmost bit) be t. Then the two buckets with one 1, having timestamps $t - 1$ and $t - 2$ are completely included in the answer. The bucket of size 2, with timestamp $t - 4$, is also completely included. However, the rightmost bucket of size 4, with timestamp $t - 8$ is only partly included. We know it is the last bucket to contribute to the answer, because the next bucket to its left has timestamp less than $t - 9$ and thus is completely out of the window. On the other hand, we know the buckets to its right are completely inside the range of the query because of the existence of a bucket to their left with timestamp $t - 9$ or greater.

Our estimate of the number of 1's in the last ten positions is thus 6. This number is the two buckets of size 1, the bucket of size 2, and half the bucket of size 4 that is partially within range. Of course the correct answer is 5.

Suppose the above estimate of the answer to a query involves a bucket b of size 2^j that is partially within the range of the query. Let us consider how far from the correct answer c our estimate could be. There are two cases: the estimate could be larger or smaller than c.

Case 1 : The estimate is less than c. In the worst case, all the 1's of b are actually within the range of the query, so the estimate misses half bucket b, or 2^{j-1} 1's. But in this case, c is at least 2^j; in fact it is at least $2^{j+1} - 1$, since there is at least one bucket of each of the sizes 2^{j-1}, 2^{j-2}, . . . , 1. We conclude that our estimate is at least 50% of c.

Case 2 : The estimate is greater than c. In the worst case, only the rightmost bit of bucket b is within range, and there is only one bucket of each of the sizes smaller than b. Then $c = 1 + 2^{j-1} + 2^{j-2} + \cdots + 1 = 2^j$ and the estimate we give is $2^{j-1} + 2^{j-1} + 2^{j-2} + \cdots + 1 = 2^j + 2^{j-1} - 1$. We see that the estimate is no more than 50% greater than c.

4.6.5 Maintaining the DGIM Conditions

Suppose we have a window of length N properly represented by buckets that satisfy the DGIM conditions. When a new bit comes in, we may need to modify the buckets, so they continue to represent the window and continue to satisfy the DGIM conditions. First, whenever a new bit enters:

- Check the leftmost (earliest) bucket. If its timestamp has now reached the current timestamp minus N, then this bucket no longer has any of its 1's in the window. Therefore, drop it from the list of buckets.

Now, we must consider whether the new bit is 0 or 1. If it is 0, then no further change to the buckets is needed. If the new bit is a 1, however, we may need to make several changes. First:

- Create a new bucket with the current timestamp and size 1.

If there was only one bucket of size 1, then nothing more needs to be done. However, if there are now three buckets of size 1, that is one too many. We fix this problem by combining the leftmost (earliest) two buckets of size 1.

- To combine any two adjacent buckets of the same size, replace them by one bucket of twice the size. The timestamp of the new bucket is the timestamp of the rightmost (later in time) of the two buckets.

Combining two buckets of size 1 may create a third bucket of size 2. If so, we combine the leftmost two buckets of size 2 into a bucket of size 4. That, in turn, may create a third bucket of size 4, and if so we combine the leftmost two into a bucket of size 8. This process may ripple through the bucket sizes, but there are at most $\log_2 N$ different sizes, and the combination of two adjacent buckets of the same size only requires constant time. As a result, any new bit can be processed in $O(\log N)$ time.

Example 4.13 : Suppose we start with the buckets of Fig. 4.2 and a 1 enters. First, the leftmost bucket evidently has not fallen out of the window, so we do not drop any buckets. We create a new bucket of size 1 with the current timestamp, say t. There are now three buckets of size 1, so we combine the leftmost two. They are replaced with a single bucket of size 2. Its timestamp is $t - 2$, the timestamp of the bucket on the right (i.e., the rightmost bucket that actually appears in Fig. 4.2.

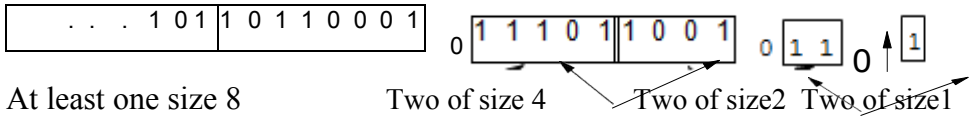


Figure 4.3: Modified buckets after a new 1 arrives in the stream

There are now two buckets of size 2, but that is allowed by the DGIM rules. Thus, the final sequence of buckets after the addition of the 1

is as shown in Fig. 4.3.

4.6.6 Reducing the Error

Instead of allowing either one or two of each size bucket, suppose we allow either $r-1$ or r of each of the exponentially growing sizes 1, 2, 4, . . ., for some integer $r > 2$. In order to represent any possible number of 1's, we must relax this condition for the buckets of size 1 and buckets of the largest size present; there may be any number, from 1 to r, of buckets of these sizes.

The rule for combining buckets is essentially the same as in Section 4.6.5. If we get $r+1$ buckets of size 2^j, combine the leftmost two into a bucket of size 2^{j+1}. That may, in turn, cause there to be $r+1$ buckets of size 2^{j+1}, and if so we continue combining buckets of larger sizes.

The argument used in Section 4.6.4 can also be used here. However, because there are more buckets of smaller sizes, we can get a stronger bound on the error. We saw there that the largest relative error occurs when only one 1 from the leftmost bucket b is within the query range, and we therefore overestimate the true count. Suppose bucket b is of size 2^j. Then the true count is at least

> Bucket Sizes and Ripple-Carry Adders
>
> There is a pattern to the distribution of bucket sizes as we execute the basic algorithm of Section 4.6.5. Think of two buckets of size 2^j as a "1" in position j and one bucket of size 2^j as a "0" in that position. Then as 1's arrive in the stream, the bucket sizes after each 1 form consecutive binary integers. The occasional long sequences of bucket combinations are analogous to the occasional long rippling of carries as we go from an integer like 101111 to 110000.

$1 + (r-1)(2^{j-1} + 2^{j-2} + \cdots + 1) = 1 + (r-1)(2^j - 1)$. The overestimate is $2^{j-1} - 1$. Thus, the fractional error is

$$\frac{2^{j-1} - 1}{1 + (r-1)(2^j - 1)}$$

No matter what j is, this fraction is upper bounded by $1/(r-1)$. Thus, by picking r sufficiently large, we can limit the error to any desired $\epsilon > 0$.

4.6.7 Extensions to the Counting of Ones

It is natural to ask whether we can extend the technique of this section to handle aggregations more general than counting 1's in a binary stream. An obvious direction to look is to consider streams of integers and ask if we can estimate the sum of the last k integers for any $1 \leq k \leq N$, where N, as usual, is the window size.

It is unlikely that we can use the DGIM approach to streams containing both positive and negative integers. We could have a stream containing both very large positive integers and very large negative integers, but with a sum in the window that is very close to 0. Any imprecision in estimating the values of these large integers would have a huge effect on the estimate of the sum, and so the fractional error could be unbounded.

For example, suppose we broke the stream into buckets as we have done, but represented the bucket by the sum of the integers therein, rather than the count of 1's. If b is the bucket that is partially within the query range, it could be that b has, in its first half, very large negative integers and in its second half, equally large positive integers, with a sum of 0. If we estimate the contribution of b by half its sum, that contribution is essentially 0. But the actual contribution of that part of bucket b that is in the query range could be anything from 0 to the sum of all the positive integers. This difference could be far greater than the actual query answer, and so the estimate would be meaningless.

On the other hand, some other extensions involving integers do work. Sup- pose that the stream consists of only positive integers in the range 1 to 2^m for some m. We can treat each of the m bits of each integer as if it were a separate stream. We then use the DGIM method to count the 1's in each bit. Suppose the count of the ith bit (assuming bits count from the low-order end, starting at 0) is c_i. Then the sum of the integers is

$$\sum_{i=0}^{m-1} c_i 2^i$$

If we use the technique of Section 4.6.6 to estimate each c_i with fractional error at most □, then the estimate of the true sum has error at most. The worst case occurs when all the c_i's are overestimated or all are underestimated by the same fraction.

4.6.8 Exercises for Section 4.6

Exercise 4.6.1 : Suppose the window is as shown in Fig. 4.2. Estimate the number of 1's the the last k positions, for k = (a) 5 (b) 15. In each case, how far off the correct value is your estimate?

! Exercise 4.6.2 : There are several ways that the bit-stream 1001011011101 could be partitioned into buckets. Find all of them.

Exercise 4.6.3 : Describe what happens to the buckets if three more 1's enter the window represented by Fig. 4.3. You may assume none of the 1's shown leave the window.

4.7 DECAYING WINDOWS

We have assumed that a sliding window held a certain tail of the

stream, either the most recent N elements for fixed N, or all the elements that arrived after some time in the past. Sometimes we do not want to make a sharp distinction between recent elements and those in the distant past, but want to weight the recent elements more heavily. In this section, we consider "exponentially decaying windows," and an application where they are quite useful: finding the most common "recent" elements.

4.7.1 The Problem of Most-Common Elements

Suppose we have a stream whose elements are the movie tickets purchased all over the world, with the name of the movie as part of the element. We want to keep a summary of the stream that is the most popular movies "currently." While the notion of "currently" is imprecise, intuitively, we want to discount the popularity of a movie like *Star Wars–Episode 4*, which sold many tickets, but most of these were sold decades ago. On the other hand, a movie that sold n tickets in each of the last 10 weeks is probably more popular than a movie that sold $2n$ tickets last week but nothing in previous weeks.

One solution would be to imagine a bit stream for each movie. The ith bit has value 1 if the ith ticket is for that movie, and 0 otherwise. Pick a window size N, which is the number of most recent tickets that would be considered in evaluating popularity. Then, use the method of Section 4.6 to estimate the number of tickets for each movie, and rank movies by their estimated counts. This technique might work for movies, because there are only thousands of movies, but it would fail if we were instead recording the popularity of items sold at Amazon, or the rate at which different Twitter-users tweet, because there are too many Amazon products and too many tweeters. Further, it only offers approximate answers.

4.7.2 Definition of the Decaying Window

An alternative approach is to redefine the question so that we are not asking for a count of 1's in a window. Rather, let us compute a smooth aggregation of all the 1's ever seen in the stream, with decaying weights, so the further back in the stream, the less weight is given. Formally, let a stream currently consist of the elements $a_1, a_2, \ldots, a_t$, where a_1 is the first element to arrive and a_t is the current element. Let c be a small constant, such as 10^{-6} or 10^{-9}. Define the *exponentially decaying window* for this stream to be the sum

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i$$

The effect of this definition is to spread out the weights of the stream el- ements as far back in time as the stream goes. In contrast, a fixed window with the same sum of the weights, $1/c$, would put equal weight 1 on each of the most recent $1/c$ elements to arrive and weight 0 on all previous elements. The distinction is suggested by Fig. 4.4.

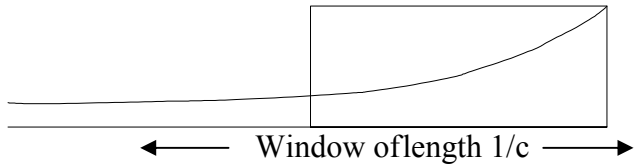


Figure 4.4: A decaying window and a fixed-length window of equal weight It is much easier to adjust the sum in an exponentially decaying window than in a sliding window of fixed length. In the sliding window, we have to worry about the element that falls out of the window each time a new element arrives. That forces us to keep the exact elements along with the sum, or to use an approximation scheme such as DGIM. However, when a new element a_{t+1} arrives at the stream input, all we need to do is:

1. Multiply the current sum by $1 - c$.
2. Add a_{t+1}.

The reason this method works is that each of the previous elements has now moved one position further from the current element, so its weight is multiplied by $1 - c$. Further, the weight on the current element is $(1 - c)^0 = 1$, so adding a_{t+1} is the correct way to include the new element's contribution.

4.7.3 Finding the Most Popular Elements

Let us return to the problem of finding the most popular movies in a stream of ticket sales.[6] We shall use an exponentially decaying window with a constant c, which you might think of as 10^{-9}. That is, we approximate a sliding window holding the last one billion ticket sales. For each movie, we imagine a separate stream with a 1 each time a ticket for that movie appears in the stream, and a 0 each time a ticket for some other movie arrives. The decaying sum of the 1's measures the current popularity of the movie.

We imagine that the number of possible movies in the stream is huge, so we do not want to record values for the unpopular movies. Therefore, we establish a threshold, say 1/2, so that if the popularity score for a movie goes below this number, its score is dropped from the counting. For reasons that will become obvious, the threshold must be less than 1, although it can be any number less than 1. When a new ticket arrives on the stream, do the following:

1. For each movie whose score we are currently maintaining, multiply its score by $(1 - c)$.
2. Suppose the new ticket is for movie M. If there is currently a score for M, add 1 to that score. If there is no score for M, create one and initialize it to 1.
3. If any score is below the threshold 1/2, drop that score.

It may not be obvious that the number of movies whose scores are main- tained at any time is limited. However, note that the sum of all scores is $1/c$. There cannot be more than $2/c$ movies with score of 1/2

or more, or else the sum of the scores would exceed 1/*c*. Thus, 2/*c* is a limit on the number of movies being counted at any time. Of course in practice, the ticket sales would be concentrated on only a small number of movies at any time, so the number of actively counted movies would be much less than 2/*c*.

4.8 SUMMARY OF CHAPTER

✦ *The Stream Data Model* : This model assumes data arrives at a processing engine at a rate that makes it infeasible to store everything in active storage. One strategy to dealing with streams is to maintain summaries of the streams, sufficient to answer the expected queries about the data. A second approach is to maintain a sliding window of the most recently arrived data.

✦ *Sampling of Streams*: To create a sample of a stream that is usable for a class of queries, we identify a set of key attributes for the stream. By hashing the key of any arriving stream element, we can use the hash value to decide consistently whether all or none of the elements with that key will become part of the sample.

✦ *Bloom Filters*: This technique allows us to filter streams so elements that belong to a particular set are allowed through, while most nonmembers are deleted. We use a large bit array, and several hash functions. Members of the selected set are hashed to buckets, which are bits in the array, and those bits are set to 1. To test a stream element for membership, we hash the element to a set of bits using each of the hash functions, and only accept the element if all these bits are 1.

✦ *Counting Distinct Elements*: To estimate the number of different elements appearing in a stream, we can hash elements to integers, interpreted as binary numbers. 2 raised to the power that is the longest sequence of 0's seen in the hash value of any stream element is an estimate of the number of different elements. By using many hash functions and combining these estimates, first by taking averages within groups, and then taking the median of the averages, we get a reliable estimate.

✦ *Moments of Streams*: The *k*th moment of a stream is the sum of the *k*th powers of the counts of each element that appears at least once in the stream. The 0th moment is the number of distinct elements, and the 1st moment is the length of the stream.

✦ *Estimating Second Moments*: A good estimate for the second moment, or surprise number, is obtained by choosing a random position in the stream, taking twice the number of times this element appears in the stream from that position onward, subtracting 1, and multiplying by the length of the stream. Many random variables of this type can be combined like the estimates for counting the number of distinct elements, to produce a reliable estimate of the second moment.

✦ *Estimating Higher Moments*: The technique for second moments works for *k*th moments as well, as long as we replace the formula $2x - 1$ (where x is the number of times the element appears at or after the selected position) by $x^k - (x-1)^k$.

4.9. REFERENCES FOR CHAPTER 4

✦ *Estimating the Number of 1's in a Window* : We can estimate the number of 1's in a window of 0's and 1's by grouping the 1's into buckets. Each bucket has a number of 1's that is a power of 2; there are one or two buckets of each size, and sizes never decrease as we go back in time. If we record only the position and size of the buckets, we can represent the contents of a window of size N with $O(\log^2 N)$ space.

✦ *Answering Queries About Numbers of 1's*: If we want to know the approx- imate numbers of 1's in the most recent k elements of a binary stream, we find the earliest bucket B that is at least partially within the last k positions of the window and estimate the number of 1's to be the sum of the sizes of each of the more recent buckets plus half the size of B. This estimate can never be off by more that 50% of the true count of 1's.

✦ *Closer Approximations to the Number of 1's*: By changing the rule for how many buckets of a given size can exist in the representation of a binary window, so that either r or $r-1$ of a given size may exist, we can assure that the approximation to the true number of 1's is never off by more than $1/r$.

✦ *Exponentially Decaying Windows*: Rather than fixing a window size, we can imagine that the window consists of all the elements that ever arrived in the stream, but with the element that arrived t time units ago weighted by e^{-ct} for some time-constant c. Doing so allows us to maintain certain summaries of an exponentially decaying window easily. For instance, the weighted sum of elements can be recomputed, when a new element arrives, by multiplying the old sum by $1 - c$ and then adding the new element.

✦ *Maintaining Frequent Elements in an Exponentially Decaying Window* : We can imagine that each item is represented by a binary stream, where 0 means the item was not the element arriving at a given time, and 1 means that it was. We can find the elements whose sum of their binary stream is at least 1/2. When a new element arrives, multiply all recorded sums by 1 minus the time constant, add 1 to the count of the item that just arrived, and delete from the record any item whose sum has fallen below 1/2.

❖❖❖❖