13
Dealing with Large Volumes of Data

13.1 Introduction

In the not too far distant past, datasets with a few hundred or a few thousand records would have been considered normal and those with tens of thousands of records would probably have been considered very large. The 'data explosion' that is so evident all around us has changed all that. In some fields only quite a small amount of data is available and that is unlikely to change very much (perhaps fossil data or data about patients with rare illnesses); in other fields (such as retailing, bioinformatics, branches of science such as chemistry, cosmology and particle physics, and the ever-growing area of mining data held by Internet applications such as blogs and social networking sites) the volume has greatly increased and seems likely to go on increasing rapidly.

Some of the best–known data mining methods were developed in those far-off days and were originally tested on datasets such as the UCI Repository [1]. It is certainly not self-evident that they will all scale up to much larger datasets with acceptable runtimes or memory requirements. The most obvious answer to this problem is to take a sample from a large dataset and use that for data mining. Taking a 1% sample chosen at random from a 100 million record dataset would leave 'only' a million records to analyse but that is itself a substantial number. Also, however random the 1% selection process itself may be, that does not guarantee that what results will be a random sample from the underlying (probably far larger) population of possible records for that task area, as that will depend on how the original data was collected. All that will be certain is that 99 million data records will have been discarded.

M. Bramer, *Principles of Data Mining*, Undergraduate Topics
in Computer Science, DOI 10.1007/978-1-4471-7307-6_13

In this chapter we will concentrate on classification rule induction as a particularly important and widely-used area of data mining, but many of the comments made will be more generally applicable.

Back in 1991, the Australian researcher Jason Catlett wrote a PhD thesis entitled *Megainduction: machine learning on very large databases* [2] in which he criticised the practice of sampling data before a classification rule induction algorithm was applied, showing that the accuracy of an induced classifier increases with an increasing size of training sample. The datasets that Catlett regarded as very large would now be considered small, or at most 'normal', but his warning remains a potent one. To add to this there is the consideration that some application areas (especially in science) are concerned with the discovery of new knowledge, where discarding a large proportion of the data is a very risky business. For other applications, even a small sample of the available data may still be massive.

For the purposes of this chapter we shall assume that you have a very large dataset (which may be a sample from an even larger one) and want to analyse it all. To tackle this problem, the methods of parallel and distributed computing are increasingly likely to be used. This is a large and complex field which goes far beyond data mining but in this chapter we will describe some of the issues and illustrate them by some recent work.

We will start by assuming that the approach adopted is to use a distributed local area network of personal computers (technically called a *loosely-coupled* architecture), as for many organisations this will be a much cheaper and more realistic option than the alternative of buying a high-performance supercomputer. Both 'desktop' and 'notebook' size computers are routinely sold in high-street stores at readily affordable prices. Organisations such as schools and university departments frequently throw away or give away 'out of date' models that are still perfectly usable. It is entirely realistic to think that even an individual working alone with a small budget could build up a network of say 20 machines at very low cost, each one of them with a speed and capacity which in past years would have qualified them to be called supercomputers.

In this chapter we will use the term *processor* to also include a local memory. It will be assumed that each classification (or other data mining) program is executed on a single processor using its local memory. Processors do not necessarily all have to have the same processing speed and memory capacity, but for simplicity we will generally assume that they do. We will sometimes use the term 'machine' to mean a processor plus its local memory.

With a network of processors it is tempting for the naïve newcomer to think that by dividing a task up to be performed by a network of say 100 identical processors it would be achievable in one hundredth of the time it would take for

one processor alone. A little experience will soon dispel this illusion. In reality it can easily be the case that 100 processors take considerably longer to do the job than just 10, because of communication and other overheads amongst them. We might invent the term 'the two many cooks principle' to describe this.

There are several ways in which a classification task could be distributed over a number of processors.

(1) If all the data is together in one very large dataset, we can distribute it on to p processors, run an identical classification algorithm on each one and combine the results.

(2) The data may inherently 'live' in different datasets on different processors, for example in different parts of a company or even in different cooperating organisations. As for (1) we could run an identical classification algorithm on each one and combine the results.

(3) An extreme case of a large data volume is *streaming data* arriving in effectively a continuous infinite stream in real time, e.g. from a CCTV. If the data is all coming to a single source, different parts of it could be processed by different processors acting in parallel. If it is coming into several different processors, it could be handled in a similar way to (2).

(4) An entirely different situation arises where we have a dataset that is not particularly large, but we wish to generate several or many different classifiers from it and then combine the results by some kind of 'voting' system in order to classify unseen instances. In this case we might have the whole dataset on a single processor, accessed by different classification programs (possibly identical or possibly different) accessing all or part of the data. Alternatively, we could distribute the data in whole or in part to each processor before running a set of either identical or different classification programs on it. This topic is discussed in Chapter 14 'Ensemble Classification'.

A common feature of all these approaches is that there needs to be some kind of 'control module' to combine the results obtained on the p processors. Depending on the application, the control module may also need to distribute the data to different processors, initiate the processing on each processor and perhaps synchronise the p processors' work. The control module might be running on an additional processor or as a separate process on one of the p processors mentioned previously.

In the next section we will focus on the first category of application, i.e. all the data is together in one very large dataset, a part of which we can distribute on to each of p processors, then run an identical classification algorithm on each one and combine the results.

13.2 Distributing Data onto Multiple Processors

Large data volumes are generally large in one of two ways:

- There are far more instances (records) than attributes. We will call such datasets 'portrait style' and think about dividing them horizontally (called *horizontal partitioning*) on to different processors. This is illustrated in Figure 13.1 for a dataset with 17 instances × 4 attributes, divided into 5 parts.
- There are far more attributes than instances. We will call such datasets 'landscape style' and think about dividing them vertically (called *vertical partitioning*) on to different processors. This is illustrated in Figure 13.2 for a dataset with 3 instances × 25 attributes, divided into 7 parts.

Naturally a dataset can also be divided both horizontally and vertically depending on the circumstances.

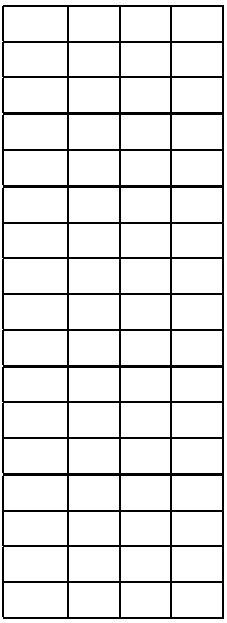

Figure 13.1 A Portrait-style Dataset with Horizontal Partitioning

This leads to a very rough outline for a possible way of distributing a classification task to a network of processors. For simplicity we will assume that the aim is to generate a set of classification rules corresponding to a given dataset, rather than some other form of classification model.

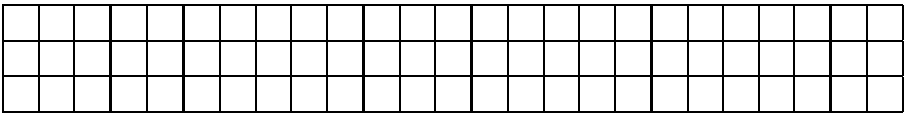

Figure 13.2 A Landscape-style Dataset with Vertical Partitioning

(a) The data is divided up either vertically or horizontally (or perhaps both) amongst the processors.

(b) The same algorithm is executed on each processor to analyse the corresponding portion of the data.

(c) Finally the results obtained by each processor are passed to a 'control module', which combines the results into a set of rules. It will also have been responsible for initiating steps (a) and (b) and for whatever action was necessary to keep the processors in step during step (b).

A general model for distributed data mining of this kind is provided by the Cooperating Data Mining (CDM) model introduced by Provost [3]. Figure 13.3 shows the basic architecture (reproduced from [4] with permission).

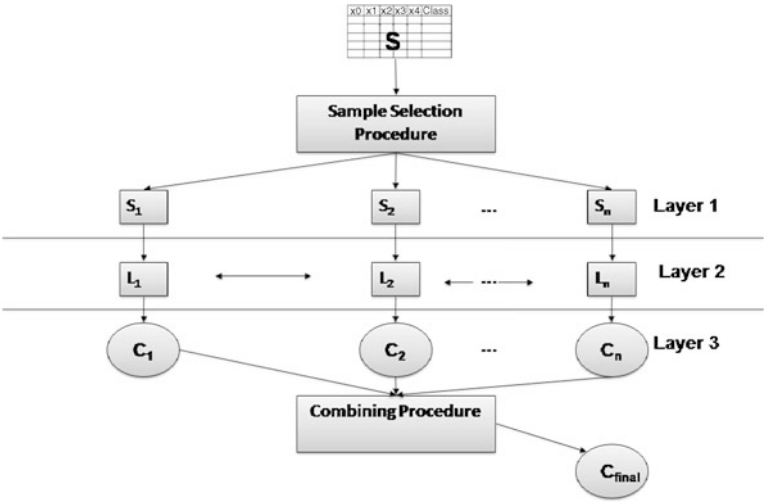

Figure 13.3 Cooperating Data Mining

The model has three *layers*:

– **Layer 1:** the *sample selection procedure*, which partitions the data sample S into subsamples (one for each of the processors available)

– **Layer 2:** For each processor there is a corresponding *learning algorithm* L_i which runs on the corresponding subsample S_i and generates a concept description C_i.

– **Layer 3:** the concept descriptions are then merged by a *combining procedure* to form a final concept description C_{final} (such as a set of classification rules).

The model allows for the learning algorithms L_i to communicate with each other but does not specify how.

13.3 Case Study: PMCRI

Some rule generation algorithms lend themselves considerably better to parallelisation than others. An early attempt to parallelise the TDIDT decision tree induction algorithm is described in [5]. The Prism algorithm for generating modular rules described in Chapter 11 is also one that lends itself well to this approach. The PMCRI (Parallel Modular Classification Rule Induction) framework [4, 6, 7] was developed by the German researcher Dr. Frederic Stahl in association with the present author as a distributed version of Prism. In this and the following section PMCRI will be used as a vehicle for explaining some general principles, but the algorithm itself will not be described in detail here. The account in these two sections draws heavily from [4]. Figures 13.5 to 13.7 are reproduced from [4] and Figures 13.4 and 13.8 are reproduced from [6] with permission.

PMCRI uses a variant of the Prism algorithm described in Chapter 11, called PrismTCS, but the differences are not important here. The important point is how the CDM model is used to control the rule generation process. Assuming that there are p processors, all roughly identical, the sample selection procedure at **Layer 1** divides the data up approximately evenly amongst them. If we focus on landscape-style data, that is achieved by giving each processor all the instances for $1/p$ th of the total number of attributes.

Without repeating the details of the original Prism algorithm here, the main point is that each classification rule is generated term-by-term. For example we may start with an outline rule

IF ..... THEN class = 1

with an empty left-hand side and expand it progressively to

IF X = large ..... THEN class = 1

IF X = large AND $Z < 124.7$ ..... THEN class = 1

IF X = large AND $Z < 124.7$ AND $Q < 12.0$ ..... THEN class = 1
IF X = large AND $Z < 124.7$ AND $Q < 12.0$ AND M = green
THEN class = 1

which is its final form.

As each term of each rule is generated at **Layer 2** there are a number of possible attribute/value pairs to consider, e.g. X = large or $Y < 23.4$ and we need to calculate the probability of each one. If we suppose that there are, say, 200 attributes and 10 processors it is straightforward to allocate 20 attributes to each of the ten processors. As each new term comes to be generated each processor looks at all possible attribute/value pairs for its group of 20 attributes, finds the one with highest probability as a 'locally best rule term' and notifies the probability (but not the term itself) to the control module by means of the *Blackboard* described below, as a kind of bid, essentially saying (for example) 'the best term processor 3 can find has a probability of 0.9'. It is easy for the control program to combine the 'bids' from all 10 processors to find the overall highest probability, corresponding to the 'globally best rule term', at each stage.

PMCRI implements communication amongst the learning algorithms in the second CDM layer by means of a distributed *blackboard* architecture, inspired by the DARBS distributed blackboard system [8].

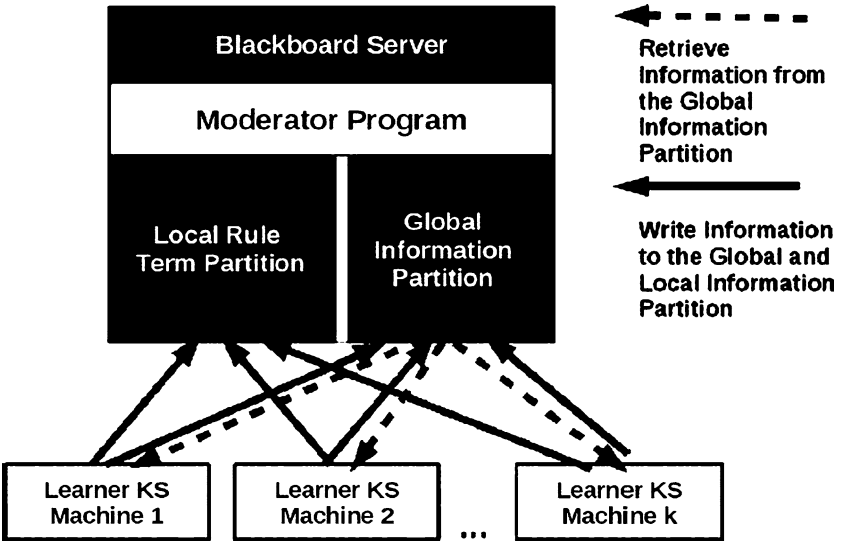

Figure 13.4 The architecture of the PMCRI framework using a distributed blackboard system

A Blackboard can be thought of as similar to a blackboard (on an easel) of the sort that teachers used to write on (and some still possibly do) with a piece of chalk in an old-fashioned classroom. A group of experts all work on a common problem, but the only way they can communicate with each other is by writing to or reading from the blackboard. Naturally, the 'experts' are not human ones and in the case of PMCRI the experts (described as 'Learner KS Machines' in Figure 13.4, for reasons that need not concern us) are just the processors referred to previously, each working out the probabilities for all possible attribute/value pairs for the attributes assigned to it. The Blackboard is just a reserved storage area on one of the processors, or perhaps some separate processor. There is a *Local Rule Term Partition* of the Blackboard to which the experts write the probabilities corresponding to their 'locally best rule terms' (although not the terms themselves). There is also a moderator program (previously called a control module) which can write to the Global Information Partition to tell the experts which one of them posted the highest probability (implying that the corresponding term was the 'globally best') and/or what to do next, e.g. start working on the next rule term or the next rule. The moderator can also read from the local rule term partition so that when all the probabilities (corresponding to the locally best term found by each expert) have been posted it can examine them and find the highest (corresponding to the globally best rule term).

The advantage of the PMCRI approach is that the workloads on the processors stay in the same proportion as the rule generation process goes on.

Once the rule generation process is finished, each expert will hold zero, one or more of the constituent terms for each of the rules in its memory. These are the terms corresponding to the probabilities it placed on the Blackboard that turned out to be the highest 'bids'. As an example, for expert number 3 the terms might be $z < 48.3$ and $q =$ green for rule 2, $x < 99.1$, $w < 62.3$ and $j < 82.67$ for rule 9 and $z < 112.9$ for rule 17.

Next the 'Combining Procedure' in **Layer 3** is started. Each expert submits its rule terms to the Global Information Partition, the moderator reads the submitted terms (rule fragments) and constructs the full ruleset from them.

Full details of the PMCRI algorithm are given in [4]. The aim of this chapter is not to describe PMCRI in detail, but to sketch out a general approach.

13.4 Evaluating the Effectiveness of a Distributed System: PMCRI

A distributed data mining system such as PMCRI can be evaluated in terms of three kinds of performance: its *scale-up*, its *size-up* and its *speed-up*. We will consider each of these in turn.

In what follows we will assume that all the processors in the distributed system are identical. We will use the term *runtime* to refer to the elapsed time taken by the entire system to complete a specified data mining task, excluding the time taken to load the data (Layer 1), which is a fixed overhead on any system of this kind.

We will use the term the *workload* of a processor to mean the number of instances held in its associated memory. Note however that a value of, say, 10,000 may mean 10,000 instances with all their attributes, or 20,000 instances with half of the attributes each, or 100,000 instances with one tenth of the attributes each, etc. We will assume that the workload is the same for each processor that is in use in the network.

Finally we will use the term *total workload of the system* to mean the sum of the workloads for each of the processors in use in the network, again measured as a number of instances.

Scale-Up

Scale-up experiments evaluate the performance of the system with respect to the number of processors for a fixed workload per processor. We keep the workload per processor constant and measure the runtime as additional processors are added. Ideally the runtime measured this way would remain constant, as for example, doubling the number of processors would double the amount of data to be processed by the system as a whole but there would be twice the number of processors to do it. A constant runtime would be indicated by a horizontal line on a graph of runtime against the number of processors.

Figure 13.5 is one of several showing results obtained for PMCRI. The runtime is plotted against the number of processors, increasing from 2 to 10, for three values of the workload per processor: 130K, 300K and 850K instances. We can see that rather than remaining horizontal, each plot increases as the number of processors increases. This is caused by an additional communications overhead in the network as more processors need to communicate information via the blackboard. Unsurprisingly, the runtime even for just two processors is greater when the workload per processor is larger. It is easier to see what

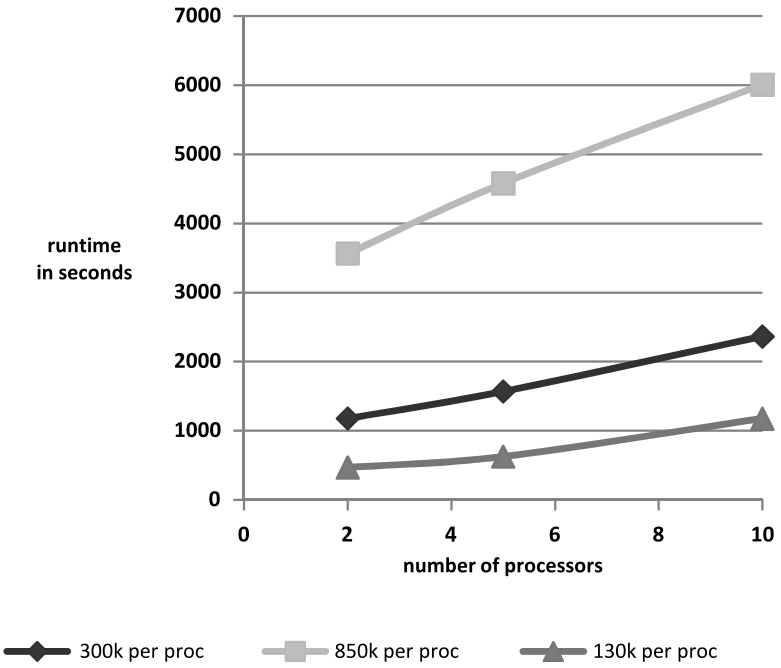

Figure 13.5 Scale-up of PMCRI

is happening if we plot on the vertical axis not runtime but relative runtime, i.e. (for each of the three plots) the runtime divided by the runtime for just 2 processors. This gives us Figure 13.6. Now each plot starts with a relative runtime of one (for two processors) and we have added the 'ideal' situation of a horizontal line of height one to the graph accordingly.

We can now see that the relative runtime is greatest for the smallest workload per processor (130K) and smallest for the largest workload (850K). So with this algorithm, the effect of the communication overhead in increasing the runtime above the ideal is lower as the workload per processor increases. As we wish to be able to deal with very large datasets this is a most desirable result.

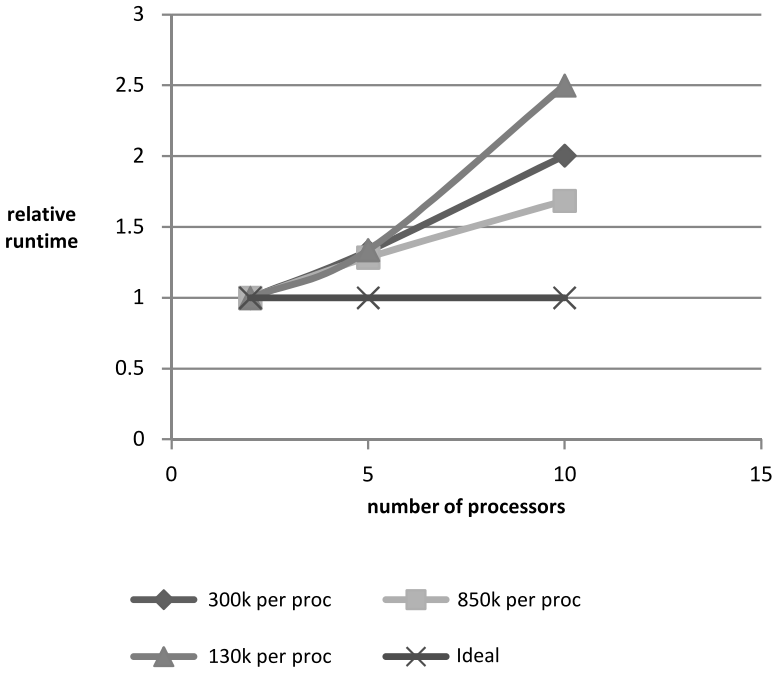

Figure 13.6 Scale-up of PMCRI Using Relative Runtimes

Size-Up

Size-up experiments evaluate the performance of the system with respect to its total workload for a fixed configuration of processors. We keep the number of processors constant and measure the runtime as the total number of training instances is increased.

Figure 13.7 shows a graph of relative runtime against number of instances, increasing from 17K to 8,000K, plotted for 1, 2, 5 and 10 processors. (Relative runtime is the runtime divided by the runtime for 17K instances.) Each plot shows an approximately linear size-up, i.e. the runtime is approximately a linear function of the size of the training data.

We have added a plot of the 'ideal' size-up where increasing the number of instances by a factor of N increases the relative runtime by a factor of N. It can be seen that the serial (i.e. one processor) plot is worse (i.e. has a greater runtime) than the ideal size-up, but the 2, 5 and 10 processor plots are all appreciably better than the 'ideal' size-up. This is possible because of

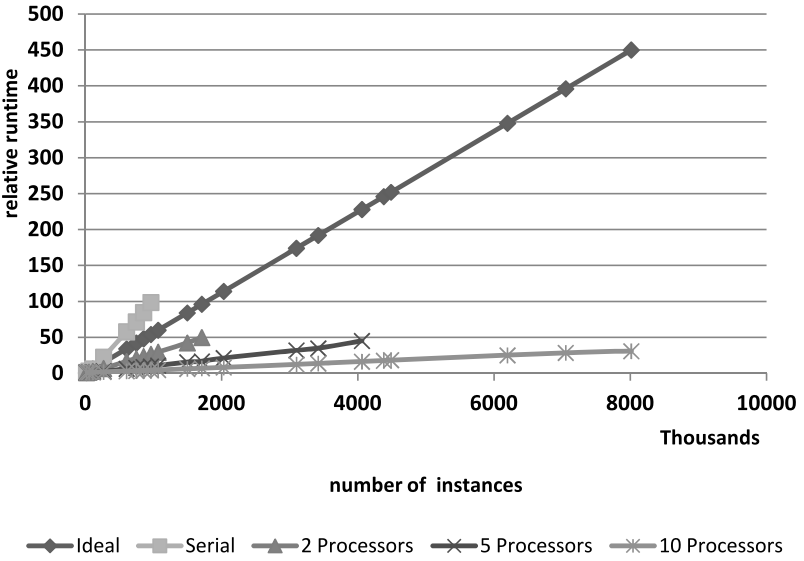

Figure 13.7 Size-up of PMCRI Using Relative Runtimes

the way the system handles the communication overheads. This is a very good result.

Speed-Up

Speed-up experiments evaluate the performance of the system with respect to the number of processors for a fixed total workload.

We keep the total workload of the system constant and measure the runtime as the number of processors is increased. This shows how much a distributed algorithm is faster than the serial (one processor) version, as a large dataset is distributed to more and more processors.

We can define two performance metrics associated with speed-up.

- The *speedup factor* S_p is defined by $S_p = R_1/R_p$, where R_1 and R_p are the runtimes of the algorithm on a single processor and on p processors, respectively. This measures how much the runtime is faster using p processors than just one. The ideal case is that $S_p = p$, but the usual situation is that $S_p < p$ because of communication or other overheads in the system.
- The *efficiency* E_p of using p processors rather than one is defined by $E_p = S_p/p$ (i.e. the speedup factor divided by the number of processors). E_p is

usually a number between 0 and 1 but can occasionally be a value greater than one, in the case of what is known as a *superlinear* speedup.

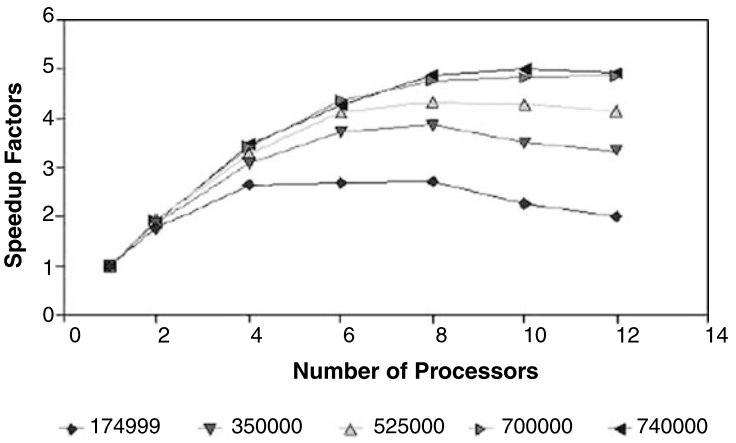

Figure 13.8 Speed-up of PMCRI

Figure 13.8 shows a graph of speedup factor against number of processors, increasing from 1 to 12, plotted for a total workload from 174,999 to 740,000 instances. This form of display is often preferred to the more obvious plot of runtime versus number of processors, as it makes it straightforward to see the largest number of processors that has a positive impact on the runtime, for a fixed workload.

We can see from Figure 13.8 that having more than four processors either does not increase or reduces the speedup factor for the smallest workload (174,999 instances) but using a larger number of processors (up to at least 10) is beneficial for the two largest workloads. Thus the PMCRI approach appears to be of most value with larger numbers of instances, which is clearly desirable.

13.5 Revising a Classifier Incrementally

In this book we have generally assumed that all the data needed to generate a classifier has already been collected and is available in a training set, possibly one that is so large that it needs to be sampled and/or distributed to a number of processors.

A very different situation arises when a classifier has been constructed and then a large volume of additional data comes in, for example data about

customer choices in a retailing application. We may have a classifier constructed using a training set of 100,000 instances and then receive an additional 10,000 instances of classified data every evening about that day's transactions. After a few weeks the amount of additional data will be much greater than the amount in the training set from which the classifier was constructed, but even a small number of additional instances (in an extreme case, even just one) can make a considerable difference to a classifier such as a decision tree. In the interests of reliable classification, we should take advantage of the availability of the additional data by generating a new classifier but how often should we do this? Once a day? Once a week? However often we do it, we would certainly not want to have to re-process all the data that has already been used to generate the classifier, starting 'from scratch' each time with an ever-growing volume of data.

To deal with the frequent arrival of new training data we need to use a classification algorithm that is *incremental*, i.e. where a classifier already constructed can be updated using new data without needing to re-process data already used. Once processed the training data can then be discarded, if it is not needed for other purposes.

An extreme version of this situation arises with *streaming* data, i.e. data that arrives in real time as effectively an infinite stream, e.g. images from CCTV or messages from telemetry devices or a news or information feed (such as the latest share prices) or the transactions from a high-volume application such as purchases made in a supermarket or by credit card.

Given an incremental classification algorithm, it is not realistic to update the classifier for each single new instance that arrives, so we will generally batch incoming instances into groups of N and update the classifier as each batch is completed. Two important questions about this approach are:

1. How accurately will a classifier produced in this way approximate the classifier that would have been constructed if all the data had been available for processing at the beginning as a single job?
2. To what extent does the choice of batch size N affect the answer to (1)?

A collection of algorithms and tools for mining streaming data is described in [9] for those with a knowledge of the Java programming language.

In the remainder of this section we will consider a method of classification which lends itself very well to an incremental approach: the Naïve Bayes classifier which was described in Chapter 3. In this case processing the data as batches of whatever size involves no loss of accuracy compared with collecting

a potentially vast amount of data together and processing it all as a single job. This is a highly desirable property.

We will only briefly summarise the description of the Naïve Bayes classification algorithm here, using an example from Chapter 3. Given a training set such as Figure 13.9:

day	season	wind	rain	class
weekday	spring	none	none	on time
weekday	winter	none	slight	on time
weekday	winter	none	slight	on time
weekday	winter	high	heavy	late
saturday	summer	normal	none	on time
weekday	autumn	normal	none	very late
holiday	summer	high	slight	on time
sunday	summer	normal	none	on time
weekday	winter	high	heavy	very late
weekday	summer	none	slight	on time
saturday	spring	high	heavy	cancelled
weekday	summer	high	slight	on time
saturday	winter	normal	none	late
weekday	summer	high	none	on time
weekday	winter	normal	heavy	very late
saturday	autumn	high	slight	on time
weekday	autumn	none	heavy	on time
holiday	spring	normal	slight	on time
weekday	spring	normal	none	on time
weekday	spring	normal	slight	on time

Figure 13.9 The *train* Dataset

We construct a probability table giving conditional probabilities (in the body of the table) and prior probabilities (in the bottom row) corresponding to the training data (Figure 13.10).

	class			
	on time	late	very late	cancelled
day = weekday *	9/14 = 0.64	1/2 = 0.5	3/3 = 1	0/1 = 0
day = saturday	2/14 = 0.14	1/2 = 0.5	0/3 = 0	1/1 = 1
day = sunday	1/14 = 0.07	0/2 = 0	0/3 = 0	0/1 = 0
day = holiday	2/14 = 0.14	0/2 = 0	0/3 = 0	0/1 = 0
season = spring	4/14 = 0.29	0/2 = 0	0/3 = 0	1/1 = 1
season = summer *	6/14 = 0.43	0/2 = 0	0/3 = 0	0/1 = 0
season = autumn	2/14 = 0.14	0/2 = 0	1/3 = 0.33	0/1 = 0
season = winter	2/14 = 0.14	2/2 = 1	2/3 = 0.67	0/1 = 0
wind = none	5/14 = 0.36	0/2 = 0	0/3 = 0	0/1 = 0
wind = high *	4/14 = 0.29	1/2 = 0.5	1/3 = 0.33	1/1 = 1
wind = normal	5/14 = 0.36	1/2 = 0.5	2/3 = 0.67	0/1 = 0
rain = none	5/14 = 0.36	1/2 = 0.5	1/3 = 0.33	0/1 = 0
rain = slight	8/14 = 0.57	0/2 = 0	0/3 = 0	0/1 = 0
rain = heavy *	1/14 = 0.07	1/2 = 0.5	2/3 = 0.67	1/1 = 1
Prior Probability	14/20 = 0.70	2/20 = 0.10	3/20 = 0.15	1/20 = 0.05

Figure 13.10 Probability Table for the *train* Dataset

Then the score for each class for an unseen instance such as

weekday	summer	high	heavy	????

can be calculated from the values in the rows shown above that are marked with asterisks.

class = on time $0.70 * 0.64 * 0.43 * 0.29 * 0.07 = 0.0039$
class = late $0.10 * 0.5 * 0 * 0.5 * 0.5 = 0$
class = very late $0.15 * 1 * 0 * 0.33 * 0.67 = 0$
class = cancelled $0.05 * 0 * 0 * 1 * 1 = 0$

The class with the largest score is selected, in this case class = *on time*. (There are complications with zero values which will be ignored here.)

First we note that there is no need to store all the values shown above. All that needs to be stored for each of the attributes is a frequency table showing the number of instances with each possible combination of the attribute value and classification. For attribute *day* the table would be as shown in Figure 13.11.

	class			
	on time	late	very late	cancelled
weekday	9	1	3	0
saturday	2	1	0	1
sunday	1	0	0	0
holiday	2	0	0	0

Figure 13.11 Frequency Table for Attribute *day*

Together with a table for each attribute there needs to be a row showing the frequencies of each of the four classes, as shown for this example in Figure 13.12.

	class			
	on time	late	very late	cancelled
TOTAL	14	2	3	1

Figure 13.12 Class Frequencies

The values in the TOTAL row are used as the denominators when the values in the frequency table for each attribute are used in calculations, e.g. for the frequency table for attribute *day*, the value used for *weekday*/*on time* is 9/14. The Prior Probability row in Figure 13.10 does not need to be stored at all as in each case the value is the frequency of the corresponding class divided by the total number of instances (20 in this example).

Even when the volume of data is very large the number of classes is often small and even when there are a very large number of categorical attributes, the number of possible attribute values for each one is likely to be quite small, so overall it seems entirely practical to store a frequency table such as Figure 13.11 for each attribute plus a single table of class frequencies.

With this tabular representation for the probability model generated by the Naïve Bayes algorithm, incrementally updating a classifier becomes trivial. Suppose that based on 100,000 instances we have a frequency table for attribute A as shown in Figure 13.13.

The frequency counts for the four classes are 50120, 19953, 14301 and 15626 making a grand total of 100,000.

Suppose that we now want to process a batch of 50,000 more instances with a frequency table for attribute A as shown in Figure 13.14.

	class = c1	class = c2	class = c3	class = c4
a1	8201	8412	5907	8421
a2	34202	7601	6201	5230
a3	7717	3940	2193	1975

Figure 13.13 Frequency table for attribute A (first 100,000 instances)

	class = c1	class = c2	class = c3	class = c4
a1	4017	5412	2907	6421
a2	15002	2601	4201	2230
a3	2289	1959	2208	753

Figure 13.14 Frequency table for attribute A (next 50,000 instances)

For these new instances the frequency counts of the classes are 21308, 9972, 9316 and 9404, making a total of 50,000.

In order to obtain the same classification for any unseen instance with the training data received in two parts as if all 150,000 instances had been used together to generate the classifier as a single job, it is only necessary to add the two frequency tables for each attribute together element-by-element and to add together the frequency totals for each class. This is simple to do with no loss of accuracy involved.

Returning to the topic of distributing data to a number of processors by vertical partitioning, i.e. allocating a portion of the attributes to each processor, that approach fits well with the Naïve Bayes algorithm. All that each processor would have to do is to count the frequency of each attribute value/class combination for each of the attributes allocated to it and pass a small table for each one to the 'control module' whenever requested.

Experiments have shown that the classification accuracy of Naïve Bayes is generally competitive with that of other methods. Its main drawbacks are that it only applies when the attribute values are all categorical and that the probability model generated is not as explicit as a decision tree, say. Depending on the application, the explicitness of the model may or may not be a significant issue.

13.6 Chapter Summary

This chapter is concerned with issues relating to large volumes of data, in particular the ability of classification algorithms to scale up to be usable for such volumes.

Some of the ways in which a classification task could be distributed over a local area network of personal computers are described and a case study using an extended version of the Prism rule induction algorithm known as PMCRI is presented. Techniques for evaluating a distributed system of this kind are then illustrated.

The issue of streaming data is also considered, leading to a discussion of a classification algorithm that lends itself well to an incremental approach: the Naïve Bayes classifier.

13.7 Self-assessment Exercises for Chapter 13

After the data in the *train* dataset given in Figure 13.9 has been collected records for another 10 days are collected, as shown in the table below.

day	season	wind	rain	class
weekday	summer	none	none	cancelled
weekday	winter	none	none	on time
weekday	winter	none	none	on time
weekday	summer	high	heavy	late
saturday	summer	normal	none	on time
weekday	summer	normal	slight	very late
holiday	summer	high	slight	on time
sunday	summer	normal	none	on time
weekday	winter	high	heavy	very late
weekday	summer	none	slight	on time

1. Construct a frequency table for each of the four attributes and a class frequency table, using the data in the two *train* datasets combined.
2. Using these new tables find the most likely classification for the unseen instance given below.

weekday	summer	high	heavy	????

References

[1] Blake, C. L., & Merz, C. J. (1998). *UCI repository of machine learning databases*. Irvine: University of California, Department of Information and Computer Science. http://www.ics.uci.edu/~mlearn/MLRepository.html.

[2] Catlett, J. (1991). *Megainduction: machine learning on very large databases*. Sydney: University of Technology.

[3] Provost, F. (2000). Distributed data mining: scaling up and beyond. In H. Kargupta & P. Chan (Eds.), *Advances in distributed data mining*. San Mateo: Morgan Kaufmann.

[4] Stahl, F., Bramer, M., & Adda, M. (2009). PMCRI: a parallel modular classification rule induction framework. In *LNAI: Vol. 5632. Machine learning and data mining in pattern recognition* (pp. 148–162). Berlin: Springer.

[5] Shafer, J. C., Agrawal, R., & Mehta, M. (1996). SPRINT: a scalable parallel classifier for data mining. In *Twenty-second international conference on very large data bases*.

[6] Stahl, F. T., Bramer, M. A., & Adda, M. (2010). J-PMCRI: a methodology for inducing pre-pruned modular classification rules. In *Artificial intelligence in theory and practice III* (pp. 47–56). Berlin: Springer.

[7] Stahl, F., & Bramer, M. (2013). *Computationally efficient induction of classification rules with the PMCRI and J-PMCRI frameworks. Knowledge based systems*. Amsterdam: Elsevier.

[8] Nolle, L., Wong, K. C. P., & Hopgood, A. (2002). DARBS: a distributed blackboard system. In M. A. Bramer, F. Coenen, & A. Preece (Eds.), *Research and development in intelligent systems XVIII*. Berlin: Springer.

[9] Bifet, A., Holmes, G., Kirkby, R., & Pfahringer, B. (2010). MOA: massive online analysis, a framework for stream classification and clustering. *Journal of Machine Learning Research*, *99*, 1601–1604.