Chapter 2

Knowledge Discovery and Data Mining

In this chapter we recall the foundations of Data Mining. We distinguish the Knowledge Discovery *process*, a compound activity which requires user interaction, from its Data Mining *task*. Data Mining in this strict interpretation is a crucial step of the KDD process. We step through a more detailed analysis of Data Mining problems and techniques, by presenting a general break-down of the components of DM algorithms, and then showing its instantiation to some common cases. The systematic analysis of the algorithmic components of data mining techniques is the preliminary step to studying their parallel implementation, which is the matter of the next chapter.

The chapter has the following structure. Section 2.1 introduces the basic concept of knowledge extraction, section 2.2 discusses the duality among models and patters in encoding knowledge. Section 2.3 describes the overall structure of the knowledge discovery process, its component steps, and the consequences on KDD system architecture.

The impact of the KDD process structure on data management techniques is the topic of section 2.4. Section 2.5 is the main part of this chapter. We specify what is a data mining problem and define data mining algorithms, identifying their fundamental components. This taxonomy of the mining techniques is largely inspired by the work of Hand, Mannila and Smyth [HMS01]. To instantiate the abstract concept, the final part of the chapter describes three well-known data mining algorithms, k-means clustering (§ 2.6), Autoclass clustering (§ 2.7) and classification by induction of decision trees (§ 2.8).

2.1 Information and Knowledge

It is nowadays a commonplace to say that the ever growing size of electronic archives pushes the need to automate the interpretation of such an amount of data. The classical definition of Data Mining given in [FPSU96] is

> *Knowledge discovery in databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data.*

This is the commonly accepted definition of data mining, even if different researchers shift the attention on different aspects. Data analysis is intended to be automatic to a large extent, and *patterns* in this context are every kind of of unexpected data relationships and behaviors.

Large databases contain much more information than what is apparent. Trends, patterns, regularities, relationships among seemingly unrelated variables are implicit in the data. If properly made explicit, all this kinds of information can be a great advantage in the industrial, economic, scientific fields just to name three.

Knowledge in the form of descriptive models, which allow understanding and predicting phenomena, is the ultimate goal of science. On the other hand, a deeper insight in the behavior of a social or productive system can be a great advantage in commercial activities as much as finer control on production processes can give a competitive edge in the industrial field.

Earth observation systems, telecommunication systems, world-scale economic transfers are all examples of scientific, industrial and economic activities whose management systems can daily collect a data amount in the order of the Terabyte.

It is of course no longer feasible to rely only on human intuition and innate ability to recognize visual and symbolic patterns. Database size forces the analysis to be as much as possible automatic.

The field of Data Mining has originated from the application to this problem of methods from nearby research fields: statistics, machine learning, database theory are some examples. The tools and algorithms from these fields have since then undergone an intense investigation, because their application is bounded by a number of practical constraints.

Because of their size, datasets cannot be kept in-memory. Architectural limits impose that they are on secondary or even tertiary storage. The situation is not likely to change, as the size of the archives commercially used today is several orders of magnitude times bigger, and grows faster, than the size of main-memories.

Since datasets are out of core, the common intuition of algorithmically "tractable" problem fails. It impossible to exploit even moderately complex algorithms in the knowledge extraction from huge data sets, even $O(N^2)$ complexity is not affordable when speaking about Terabytes of data. Simple algorithms have often to be discarded, or recoded, to mitigate the penalty incurred when working on data that is at the bottom of a deep memory hierarchy (see section 1.6).

The problem of data mining is thus on one hand to develop algorithm that are effective in extracting knowledge. On the other hand we aim at performance as much as possible, to keep larger problems within the practical limits of computational resources.

2.2 Models and Patterns

The first question about data mining is "what is *knowledge*?".

The form of knowledge we are looking for in the data is some statement or generalization about the structure of the data, which can be in the form of a *model* or a set of *patterns*. We call model a global description of the structure of all the input data (a statistic distribution is an example of a model), while a pattern is a statement about some kind of regular structure that is present in a part of the input set (a periodic behavior of fixed length is an example of a pattern).

We will often use the word model to mean the answer we obtain by the mining process, including both global models and patterns, or sets of patterns, as most of the time there is no substantial difference among them.

Both models and patters are expressed in a given formal language. Choosing the language is the first step in defining the kind of results that we can obtain from data mining, because the language defines the set of all possible answers. Given a certain kind of description language, we call *solution space* the set of all possible models that can be represented with that formalism.

The next essential step in defining data mining is specifying a criterion to evaluate the different solutions. An *evaluation function* is a method to quantify how good a solution is, allowing us to compare different models and choose the best one. Model evaluation usually takes into account how well the model represents the data, its simplicity, or the fact that it can be generalized to other data. Not every aspect of knowledge evaluation can be made automatic, as we will see in the following. When working with a formally defined evaluation function, we regard the optimum in the solution space as the model which is most informative about the data.

Data Mining can thus be described as the process of searching the best model (or pattern) for a given set of data within a given solution space, where the best solution is the one that maximizes a certain evaluation function.

There are two abstract approaches to identify a model within a large space of feasible models.

- We can look at our solution space as a whole and apply a *refinement* strategy to prune it and progressively reduce it to a single element.
- We may choose to start with a very elementary (or empty) candidate model and gradually build the true solution by following a *generate-and-test* strategy.

It is well known [Pea84] that these two approaches are equivalent; implementation considerations (e.g. the need to explicitly store data structures in order to build intermediate solutions) can motivate the choice of one over the other approach for a data mining algorithm.

The duality of the approaches is more easily seen in data mining problems whose solution is a set of patterns: the refinement approach identifies subspaces of patterns which are not interesting, while the generate-and-test approach extend the intermediate solution with new (sets of) patterns. A classical example of pattern-based problem is Association Rule Mining, which we will discuss in chapter 5.

This is no complete taxonomy: the two approaches can be mixed together, and a wide range of optimization techniques can be applied to the search process, including exact and approximate ones, greedy as well as heuristic or randomized ones.

2.3 Knowledge Discovery in Databases

In the following we set forth a distinction among the two terms Knowledge Discovery in Databases (KDD) and Data Mining (DM). It is a distinction which is often overlooked in practice, especially in the industrial field, by using the common denomination of data mining to address the whole activity of extracting knowledge from the data. In this chapter, however, we will use the two terms KDD and DM as two different points of view of the same problem, which can be tackled with wide perspective or with a more focused, in-depth approach.

Knowledge Discovery in Databases is a process, i.e. an activity made of several different steps, and its process nature is *iterative* and *interactive*. As we explained in previous section, at the beginning of the KDD process we have a set of electronically collected data, which may exhibit a varying degree of organization and internal standardization, and our goal is to extract new, useful and understandable knowledge. The definition is essentially vague as there is often no simple and fixed criterion to define knowledge that is "useful" and "not trivial".

KDD is an interactive process because the human expert plays a fundamental supervision and validation role on the discovered information. Several parameters of the KDD process, and thus its results, are controlled by the human supervisor. Beside knowledge about the mining tools, its judgment exploits essentially its *domain knowledge*. This term refers to both formal and intuitive understanding of the nature of the input data and of the related problems, which allows the expert to tell apart true gained knowledge from irrelevant facts and misinterpretations.

Example 2.1 *Our computer-generated model $\mathcal{M}$ predicts the number of workers in a firm on the ground of economic and fiscal information. $\mathcal{M}$ incorrectly claims an exceptionally low number of workers for a whole class of firms. Is there a fraud of some kind? If all of the house-building firms in a certain region and year deviate from normal behavior, it will be apparent to human eyes, but not to any algorithm, that fiscal incentives to help rebuilding after previous year's earthquake have influenced our input data.*

The need for user's validation of the KDD results is also the reason why the KDD process is iterative: once first results are found, we evaluate their quality and then proceed to discard meaningless information and emphasize useful results. This may require retracing choices and parameters settings made in some of the steps of the process, and repeating those steps to enhance the final answers.

Example 2.2 *Looking at example 2.1, we can easily guess two possible solutions:*

- *to remove the problematic subset from the set of data in input,*
- *to add the relevant information and rebuild our model $\mathcal{M}$, in order to consider the new variables.*

Within this perspective, **Data Mining** is the application of a well specified algorithm to mine a certain kind of model from a set of data. Thus we momentarily get rid of all the details and complexities regarding the user interaction with the process, and concentrate on the computational techniques that are used to automatically execute the core task of finding patterns and model inside a large database.

We will return on the definition of data mining in section 2.5. Here we just add that when the term data mining is used in its more general meaning, the distinction between the knowledge discovery process and its mining step is somewhat transposed to the one between *exploratory mining* and *routine data mining.* Exploratory mining is applying data mining techniques to a new problem to find out entirely new knowledge, while *routine data mining* is based on a previously established knowledge model, and it looks at new data that confirm, update or conflict with the model. Routine mining is used for instance for monitoring and controlling industrial and economic processes.

Example 2.3 *Developing a model that predicts the chance of a credit card fraud is an example of exploratory mining; it is instead a routine data mining task to apply the model to transactions on a daily basis, to detect possible frauds and validate the model over new data.*

In the following we give a widely accepted taxonomy and a short description of the various steps of the knowledge discovery process [HK01, Joh97]. The various steps and their byproducts are graphically summarized in figure 2.1. We will also put in evidence some of the main interdependencies among them.

1. **Data cleaning**. In all databases, even those automatically collected, there are wrong and missing values, either due to the semantics of the data or to incorrect inputs. This first preprocessing step is devoted to remove all the "noisy data" which can be detected in the input.

2. **Data integration**. Our data will rarely come from a single source, so a careful work of integration is needed. Its purpose is to reconcile the format and representation of information from different sources, and detect ambiguous or mismatched information.

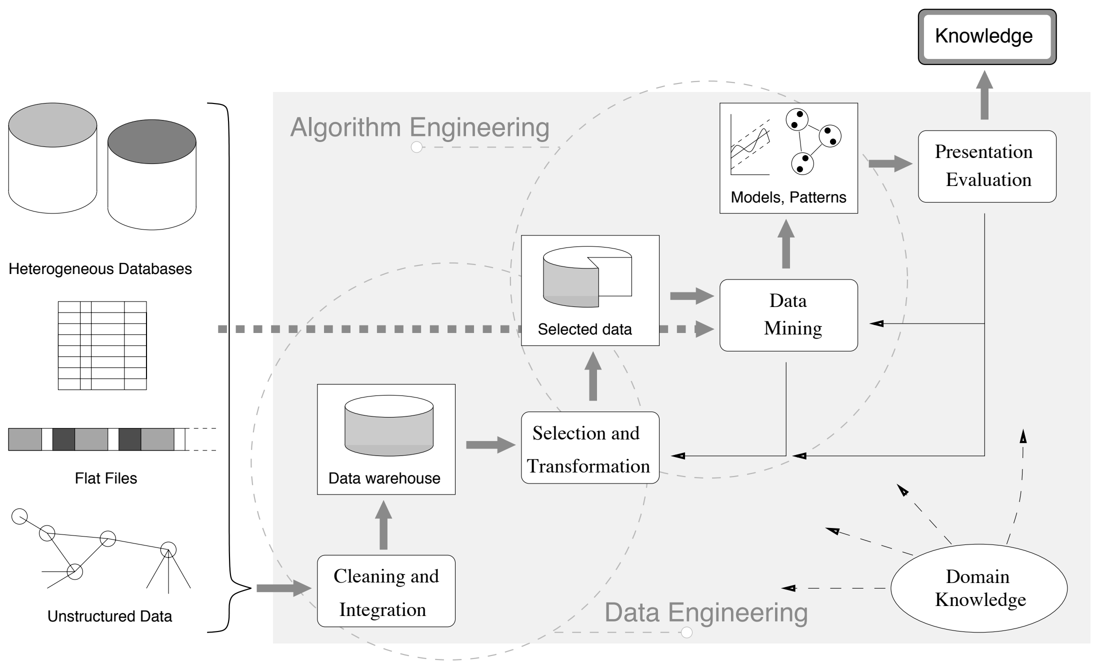

Figure 2.1: Structure and intermediate results of the knowledge discovery process.

3. **Data selection**. Real databases contain a lot of information, which is not always useful or exploitable by any data mining algorithm. The data selection step defines the subset of the data that is going to be actually mined. We may want to discard certain kind of information because they do not fit the search tools that we use, because they don't fit our current knowledge model, or because they are irrelevant or redundant with respect to the answers we are looking for.

4. **Data transformation**. Data can be transformed in several ways to make them more suitable to the mining process. We can perform individual data transformation (e.g. thresholding, scaling) or global transformations (e.g. equalization, normalization, summarization) on our data set to enhance the result of the data mining step.

5. **Data mining** is the essential process of detecting models within a large set of data. There are many different techniques, and algorithms that implement them. The choices of the algorithm and its running parameters as well as the accurate preparation of its input set heavily influence both the quality of the results that we get, and the computational effort required to achieve the answer.

6. **Pattern evaluation**. We evaluate the results of a mining algorithms to identify those models and pattern which contain useful information, and correct

the behavior of the KDD process to enhance the results.

7. **Knowledge presentation**. Presenting the discovered knowledge is not a trivial task, as by definition the discovered models are not obvious in the data. Sophisticate summarization and visualization techniques are needed to make apparent the complex patterns discovered from a large amount of data with several dimensions.

As we see in figure 2.1, these activities are logically ordered as they all use the results of a preceding a step. There are however other interactions among them which are essential to the data mining process. It is clear from example 2.1 that we may realize an error in a previous step only at the knowledge evaluation step, and in the general case any phase of the process can motivate a backtrack to a previous one.

The main interactions however have a more profound meaning due to the nature of the discovery process. The knowledge resulting from the first iteration of the process steps is seldom satisfying. Most mining algorithms are controlled by parameters which we can tune to enhance the quality of the model(see § 2.5). A more radical approach is to change the data mining algorithm with a different one that produces the same kind of knowledge model, or to change the kind of model, and hence the mining algorithm too. We call *algorithm engineering* the part of the KDD process which involves choosing and tuning the data mining algorithms.

Since data mining algorithms differ in the kind of input data they manage (e.g. real values, integer values, labels, strings) and in their sensitivity to noise, missing values, irrelevant or redundant data, the algorithm engineering phase also involves changing the data selection schemes, and possibly transforming the data. For instance, discretization is often used over real-valued data. More complex transformation are used to normalize data with odd distributions that impair model generation.

The denomination *data engineering* refers instead to the preliminary processing that is made in order to remove unwanted data and integrate various information sources. The cleaning process essentially applies domain knowledge to detect obviously wrong values. It is an important phase, as wrong data can cause deviations in the knowledge extraction process that is quite difficult to spot in the results, but excessive efforts toward "cleaning" the data can bias the mining process. Integration of data from different sources comprises merging data from similar sources (e.g. relational databases with the same purpose and schema from different agencies) as well as data with radically different layouts (e.g. legacy databases, or flat-file data). External information can be also be added in the integration phase, which is not contained in the data but it is likely to be useful for model generation.

Example 2.4 *In example 2.1 the error in the computer-generated model is recognized thanks to the background knowledge of the expert. A lot of information can be forgot, or not even be considered at first for inclusion, that can help the data mining algorithm to build a more accurate model. The task of the data engineer requires* *tuning the amount and kind of information provided to approximatively match the ability of the data mining algorithm(s) employed to exploit that information.*

Example 2.5 *Adding new, "unstructured" data to the input is usually done to help modeling the behavior of the original dataset.*

Consider the sell database of a set of different stores, which includes addresses of the stores and of customers' homes. We integrate the input with the average time that it takes to each customer to go to different shops by car (alternatively, the set of shops within twenty minutes of travel). This data, which does not relate with the sell database, and needs to be computed by other means, could help to predict customer behavior. For instance, we could gain a better understanding of which shops our customers are likely to visit on Saturdays, because they have to skip them during the week.

2.3.1 KDD System Architecture

The structure of the KDD process, and the grouping of activities that we have described influence the architectural organization of system solutions for data mining. Most of the data engineering task is usually devoted to building a data warehouse, a repository of clean and integrated data which is a safe and sound support for subsequent phases. The data warehouse can be implemented as a virtual interface toward a set of database systems, or can be actually materialized and stored independently. Since real-time update of the data is not usually an issue for data mining[1] there are usually separate database systems devoted to data mining from those used for on-line transaction processing.

More selections and transformations are then performed on the information contained in the data warehouse in the process of knowledge extraction.

There are many other issues in data mining system architecture. One problem is how to help the human supervisor in the task of choosing and tuning the mining algorithms. Wrappers [Joh97] are optimization procedures that operate on algorithm parameters and data selection schemes until the mining results meet a certain user-specified criterion.

Another question is how to accumulate the results of different mining techniques and integrate different knowledge models. One approach is to store this knowledge in the data warehouse as additional information for subsequent mining. Meta-learning methods employ independent knowledge sub-models to develop higher-order, more accurate ones [Joh97, PCS00].

[1]This is not entirely true for routine data mining.

2.4 Data Management and Representation

All the operations described in sections 2.3 and 2.5 operate on large amounts of data, thus details of data management are critical for the efficiency of the KDD process.

The input is often hold within database management systems, which are designed to achieve the highest efficiency for a set of common operation. DBMS architecture is outside the scope of this thesis, it is relevant here to underline that almost any DBMS accepts queries in standard SQL, which are internally compiled and executed on the data. The low level implementation of the query engine and the SQL query optimizers are carefully crafted to take into account all problems of secondary memory data management we mentioned in section 1.6.

Often the reconciled data view of the data warehouse is materialized in a separate database, in other cases the warehouse is an interface toward a set of DBMS. In any case, data warehouses offer a high level interface at least as sophisticated as that of the DBMSs.

On the other hand, still many data mining applications read flat files, especially parallel implementations. This is due to the fact that it is not efficient to use the standard SQL interface for many data mining algorithms. Several extensions to the SQL standard have been proposed to help standardize the design and integration of data mining algorithms, in the form of single primitives (see for instance [SZZA01]) or as data mining (sub)languages [HFW$^+$96] which can be integrated within SQL. The book by Han and Kamber [HK01, chapter 4] cites several such proposals. The recent appearing of Microsoft's OLE-DB [Mic00] has given a stronger impulse to the standardization of commercial tools.

2.4.1 Input Representation for Data Mining

Regardless of the level of management support, a very simple **tabular view** of the data is sufficient to analyze most data mining algorithms. We do not claim that this is the only or most useful way to look at the data, mining of loosely structured knowledge (e.g. web mining) being an active research issue. On the other hand, most traditional mining algorithms adopt a tabular view. It distinguishes between two aspects of the data (the "objects" from the "characteristics") which have a different impact on knowledge models.

We assume that our database $\mathcal{D}$ is a huge table. The rows of the table will generally be records in the input database, thus being logically or physically a unit of knowledge, or an *instance*. Record, point, case will be synonymous of instance in different situations.

Example 2.6 *In market basket analysis, a row in the data represents the record of goods bought by a customer in a single purchase. For spatial databases, a row can be the set of coordinates of a single point, or of a single object. A set of physical measurements of the same object or event makes a row in a case database.*

The columns of the data are assumed to have uniform meaning, i.e. all the rows (at least in our data mining view) share the same record format and fields. We will often call the columns *attributes* of the data instances. As we anticipated in section 2.3, in the general settings some of the attributes will be missing (unspecified) from some of the records.

Example 2.7 *If our databases consists of anonymous points in a space of dimension d, the attributes are uniform and the i-th attribute is actually coordinate i in the space. A database of blood analysis for patients in an hospital will be much less uniform, each column having a different meaning and a different range of possible values. Most of the values will certainly be missing, as only a few basic analysis are done on all the patients.*

We will speak of **horizontal representation** of the data if the row of our tabular view are kept as units. Likewise, horizontal partitioning will mean that the whole database is split in several blocks, each block holding a subset of the records.

Vertical representation has the columns (the attributes) as its units. Each attribute corresponds to a unit in this case. Vertical partitioning is representing each attribute (or group of attributes) as a separate object.

The distinction applies to the layout of data on disk, to partitioning of the input set, and to compact representations. The disk layout (storing consecutively elements of a record or of a column) determines the access time to the data. Compact representations influence file occupation, and indirectly access time, as well as the computational complexity of the algorithm.

Partitioning is sometimes required to split a problem into sub problems, for algorithmic reasons or because of memory limitations. The influence of the choice between horizontal and vertical representation is deep, as the different score functions employed by data mining algorithms can result in either a dominantly horizontal access pattern or a dominantly vertical one.

Example 2.8 *Our database $\mathcal{D}$ is a set of records of boolean variables, i.e. all attributes are boolean flags. In this case horizontal and vertical representation are essentially the same, as a by-row or by-column Boolean matrix. We choose to represent each line of data in a compact way, e.g. as a list of all the 1s, with every other element assumed to be 0. The compact horizontal and compact vertical representation now will require a different amount of space. Moreover, accessing to the data in the "orthogonal" way now will result in a larger complexity both in term of instructions and of I/O operations.*

Example 2.9 *Some data transport layer specifically dedicated to mining systems, like the Data Space Transfer Protocol in [BCG$^+$99], explicitly maintain data in both formats to allow the fastest retrieval from disk, at the expense of doubling the space occupation.*

2.5 The Data Mining Step

We have informally defined the Data Mining step of the KDD process as the application of an algorithm to the data in order to extract useful information. The knowledge extraction has been modeled as a search within a given state space of a model that optimizes the value of a given evaluation function. We now add more details to this characterization to define what we mean with data mining algorithm.

As discussed in [HMS01], Data Mining algorithms can be specified as the combination of different characteristics and components.

1. **Task** — Each algorithm is targeted at a specific mining task or problem: classification, regression, clustering, visualization are all different activities accomplished by different sets of algorithms[2].

2. **Model or Pattern Structure** — To describe a given mining task we need first to state what kind of solution we are looking for. The nature of the model that we want to fit to the data, or the set of patterns we look for, are fixed for any given algorithm. The pattern language, the set of parameters and the general structure of our model define the solution space explored by the algorithm. This first restriction of the kind of knowledge that we can find is known as *description bias* [FL98]. In the KDD Process this inherent bias is dealt with by applying several different mining algorithms to the same problem.

3. **Score Function** — We use a function to evaluate the quality of fit of a specific model with respect to the input data. This function should actually allow us to measure at least the relative quality of different models with respect to the input data. In order for our algorithm to search the model that best fits the data in practice.

   A theoretically well-founded error measure can give a simple and understandable score function. The score function can be based on how well the model fits the observed data, or it can consider the expected performance when generalizing a model to different data. The score function is the implementation of the *search bias*, the preference that the algorithm should have for "better" solutions. In the general case, a good solution is a compromise between fitting the model to the data, keeping it simple and understandable, and ensuring that the model generalizes well.

   As we will see, good score functions are not always straightforward, and sometimes well-defined, sound functions are very expensive to compute. Choosing the right score function is thus a crucial issue in designing a Data Mining algorithm.

[2]Here we will concentrate on automatic knowledge discovery, so important tasks like visualization will not be discussed in full.

4. **Search Strategy and Optimization Method** — We use the score function as an objective function to search a good solution in the model space. A number of optimization methods and search strategies are used in data mining algorithms, and sometimes they can be usefully combined. The optimization method can vary from a simple steepest descent to a full combinatorial exploration, and be combined with partitioning of the solution space, and heuristic or exact pruning.

   Different situations arise depending on the kind of score function: some models only have a small number of parameters to be assigned a value, but sophisticate models can have quite a complex structure. For models like decision trees, the structure is an essential component of the data model, which is entirely known only at the end of the algorithm execution (see section 2.8). General score functions dynamically depend on both the model structure and its associated set of parameters.

5. **Data Management Strategy** — Data mining algorithms need to store and access

   (a) their input data and

   (b) intermediate results.

   Traditional methods from machine learning and statistics disregarded the problem of data management, simply assuming that all relevant information could be held in memory. When moving to practical data mining application we find that amounts of data in input and within the model can be huge, and their impact on running time can easily outweigh the algorithm complexity.

   Example 2.10 *As an extreme example, if the input or the representation of our model reside in secondary memory, the issues of sections 1.6 rule out a whole class of in-core algorithms, which become unpractical because of a constant slowdown of several orders of magnitude. This is an actual problem for clustering algorithms, a field in which the exploitation of sophisticated external-memory aware techniques is nowadays common [KH99].*

The behavior of a data mining algorithm can usually be modified by means of a set of parameters. Score functions can often be changed or made parametric, as well as non trivial search strategies can be controlled by the user. Attention is also growing toward parameterizing the search space by adding constraints to the problem definition. Algorithm tuning is usually part of the KDD process, as it determines the quality of the produced information. Tuning the data management is also possible, but it is usually performed only for the sake of enhancing the program performance.

The different components of data mining algorithms will be evidenced in the following examples, and the impact of parallel computation on them will be the matter of chapter 3.

2.6 K-means Clustering

Clustering is the problem of identifying subsets (clusters) in the data, which express a high level of internal similarity, and low inter-cluster similarity. We essentially want to infer a set of classes in the data starting with no initial information about class memberships, so clustering is also called *unsupervised classification*.

The K-means algorithm is a *spatial clustering* method, i.e. its input points are distributed in a suitable space, and the measure of similarity is inherited by the properties of this space. For k-means we use a euclidean space R^d with its distance metric.

Input A set of N data points $X_1, X_2, \dots X_N$ in a space R^d of dimension d. The number k of clusters we look for is an input parameter too.

Model The cluster model is a set of k point $\{m_j\}_{j=1}^k$ in R^d, which are called *cluster centroids*. The points X_i are grouped into k clusters by the centroids, each X_i belonging to the cluster originated from the nearest centroid.

Score function The score function for k-means clustering is the sum over all the points of the square of the distance with their assigned centroid.

$$\frac{1}{N}\sum_{i=1}^{n}\left(\min_j d^2(X_i, m_j)\right) \tag{2.1}$$

To minimize this score function means ensuring that the set of centroids chosen makes our clusters more “compact”.

Search It is known that the problem of minimizing function 2.1 is NP-complete. The classical k-means algorithm uses a greedy gradient-descent search to update a given set of k initial centroids.

The search iteratively computes the value of function 2.1, thus assigning points to the nearest centroid, then it computes a new approximation of the centroids, where each centroid is the mean of all the points in its associated cluster. The search continues as long as the new centroid set improves the value of the score function.

The usual application of k-means employs randomly generated initial centroids. Since it is known that the search procedure converges to a local minimum, it is also a common solution to apply an outer-level random sampling search, by restarting the algorithm with new random centroids. The set of centroids with the minimum score from several runs is used as an approximation of the best clustering.

Data Management The data management issues in k-means clustering come from two sources.

- Computing distances at each step is expensive, and computing them for all X_i, m_j pairs is actually a waste of time, since only $1/k$ of them are added in the value of the score function.

  Techniques have been developed to infer constraints about which points in a cluster could actually change their memberships at the following steps, and to which cluster they could actually switch, in order to reduce the number of distance computations. Up to 96% of savings in the number of computed distances has been reported [JMJ98].

  These techniques can be regarded as an optimization in the computation of the score function, but they require to split the data into blocks. Partial results for (nested) blocks of points are recorded at each iteration in order to save on distance computations. This kind of data management of course has a cost, and it influences processor-level data locality.

- If the data are out of core, each iteration requires scanning the whole input data set. Even in this case the data is often reorganized to save both in computation and I/O time by grouping nearby points.

2.7 Autoclass

The Autoclass algorithm [CS96] is a clustering method based on a probabilistic definition of cluster and membership. Autoclass uses a Bayesian approach to probability, which means trying to choose the model with the highest posterior probability of generating the observed input data.

Input The input data is a set of cases with d measurable properties as attributes. Each attribute can be either numeric (real values are used, with a specified range and measurement error) or nominal (one label from a fixed sets of symbols for that attribute).

Model The clustering model in Autoclass is a set of k classes which are defined as probability distributions. Conversely, the *membership* property of each case is represented by a set of k probability values, which specify the likelihood that the case belongs to each one of the classes.

The probabilistic model of Autoclass has a two-level structure.

- A *class* specifies a distribution function (e.g. multinomial, Gaussian, multivariate) for each attribute. Multivariate distributions can be used to model sets of parameters. All the separate distributions in a class are assumed to be independent, and each distribution has its own parameters (e.g. average, variance, covariance matrix).

  The overall structure of the class, having fixed the parameters of its distribution, gives the distribution of attribute values for cases in that class.

- A *classification* is a set of k classes with associated weights. Weights give the probability that a generic case belongs to each class. The classification thus expressed assigns a probabilistic membership to the cases in the input set, and it is a distribution of the values of the attributes.

Search The search strategy in Autoclass is made up of two essential levels, the inner search and the outer search.

The inner search is used to optimize the parameter of a given class structure. Since we don't know either the true memberships of cases or the right values of distribution parameters, and a pure combinatorial search is unfeasible, the expectation maximization (EMax) method is applied, which can deal with hidden variables (see [HMS01, chapter 8]).

The starting point of the inner search in Autoclass is a set of classes with randomly chosen parameters. Then the EMax method iteratively executes the following steps.

1. It evaluates the probability of membership of each item in input to each one of the classes.

2. Using the new membership information, class statistics are evaluated. The question now is "what is the *marginal likelihood*", i.e. what is the likelihood that this model could generate the input. Revised parameters are computed for the classification.

3. Check convergence conditions, and repeat if they are not met. Several user-definable criterion include absolute convergence, speed of convergence, time elapsed and number of iterations.

Since under the hypothesis of the method the EMax algorithm converges always to a local maximum, but in the parameter space there are an unknown number of local maximums, an outer level of searching is employed.

The outer search strategy is essetially a randomized sampling of the parameter space for the inner search (number and distribution of classes, kind and initial parameters of their density functions). In autoclass, the probability distribution for the new trials depends on the results of previous searches.

Score function The outermost evaluation function is a Bayesian measure of the probability of the given input with respect to the model found. To cite the on-line documentation of the Autoclass code [CST02] :

> *This includes the probability that the "world" would have chosen this number of classes, this set of relative weights, and this set of parameters for each class, and the likelihood that such a set of classes would have generated this set of values for the attributes in the data set.*

The values for each classification are computed within the inner search process and the numerical approximation of such a complex measure is not an easy task, thus the computational cost of the Autoclass algorithm is quite high.

In the EMax inner cycle, in order to compute the global probability value, we alternate among the computation of the probability of cases and the evaluation of the most probable parameters. The operations of numerical integration involved essentially require a great amount of sums and products while scanning the attribute and membership arrays.

Data Management As the Autoclass method is computation-intensive and requires a lengthy optimization process to converge, which accesses the whole data at each iteration, it is unpractical to apply it to data sets which don't fit in main memory.

2.8 Decision Tree Induction

Decision trees are a recursively structured knowledge model. We describe the general model, and the implementation of the C4.5 algorithm [Qui93], which will be discussed again in chapter 6.

Input We use the common tabular view of the data of section 2.4.1, where each attribute can be either numerical or nominal (i.e. its possible values are finite and unordered). A distinguished one of the nominal attributes is the class of the case. Since we know in advance the set of classes and the membership of the cases, decision tree induction is a problem of *supervised classification.*

Model The *decision tree* model is a tree structure where each node makes a decision based on a test over the attribute values. Starting from the root, the cases in the initial data set are recursively split according to the outcome of the tests. Following a path from the root to any leaf we find smaller and smaller subsets of the input, which satisfy all the tests along that path. Each node thus is associated to a subset of the input, and the set of leaves to a partition of the data[3].

[3]We omit here the management of missing attribute values, which requires replication of a part of the cases.

When all the leaves are homogeneous with respect to the class attribute, the tree contains a model of the attributes for all the classes in the input. A decision tree is both a descriptive model and a predictive one. The tree can also be converted into a set of classification rules[4] in order to have a simpler, decomposable representation of the classes.

With respect to the model, tree-based classifiers use different kind of split tests, which vary in the number of attributes involved in the test and in the number of possible outcomes[5].

The tests used in C4.5 work on a single attribute at each node. Two-way comparisons are used for the numerical attributes, of the form $x < t$ and $x \geq t$, where the threshold t is specified in the decision node. Nominal values give rise to a multi-way split, where each different value of the attribute generates a new son.

Search The search procedure is greedy, it builds the decision tree step by step adding new nodes, and it is recursively decomposed. A step in the search corresponds to selecting a test for a node, and splitting the associated data into two (or more) son nodes.

We call *open* a leaf which is not homogeneous. The elements of our greedy search are thus the following.

Intermediate solution. An intermediate solution is partially built tree (which has some open leaves).

Neighborhood. A neighbor solution is reached from the current one by choosing an open leaf, selecting an attribute to test and replacing the leaf with a decision subtree of height one. As we will see in a while, we can restrict the neighborhood definition by choosing arbitrarily an open node, with no loss of generality

To determine the best attribute and test at an open node, a score function is evaluated over all the attributes for the subset of the data of that node. Each single search step is thus an exhaustive enumeration. On the other hand, no backtrack is ever done of an already chosen test, so the overall search is greedy. It is not guaranteed to find the “best” classification tree.

With these hypotesis, the problem of building a decision tree decomposes into smaller subproblems, with independent model representation and disjoint datasets, thus C4.5 is a divide and conquer algorithm[6]. This is why the choice of the open node to expand is irrelevant for the final result. The C4.5 uses a depth-first tree expansion.

[4]This is essentially done by performing a simplification of the set of rules corresponding to all the paths to the leaves.

[5]E.g.: we can test for each different value of a nominal attribute. Separate subtrees can be started at each node for every different outcome of the test, or for groups of outcomes.

[6]In the original implementation the split operation over numeric attributes is performed in a non decomposable way.

Score Function The score function which guides the choice of the split test is the gini index, which is a measure of the gain in entropy by splitting with the given test.

The *information gain* of the test is computed in term of the entropy of the generated partition. We define the entropy of a set S of data with respect to the set C_k of classes

$$info(S) = -\sum_{j=1}^{k} \left(\frac{freq(C_j, S)}{|S|} \times log_2 \frac{freq(C_j, S)}{|S|} \right) \tag{2.2}$$

where $freq(C_j, S)$ is the number of cases with class C_j in the given set S. Information gain is a combination of the entropy of node T_0 to be expanded with that of the resulting partitions T_i. Its definition is

$$gain(T_0, T_1 \ldots T_b) = info(T_0) - \sum_{i=1}^{b} \left(\frac{|T_i|}{|T_0|} \times info(T_i) \right) \tag{2.3}$$

Evaluation of the split on a nominal attribute requires building histograms of all the couples (attribute value, class). This can be done with a scan of the current subset of instances. On the other hand, when evaluating a test on a numerical attribute, the two classes T_1 and T_2 are defined by a split point t. The computation of the best split point (which gives the lowest value of the score function for that node) requires to access the column of each numerical attribute in sorted order.

Data Management In the original formulation of C4.5 [Qui93] data management issues are not taken into account. Since a great deal of sorting and scan operations happen on varying subsets of the input, the algorithm is designed to work in-core and simply use large arrays and index tables to access the data.

Chapter 3

Parallel Data Mining

In this chapter we discuss the advantages and issues of parallel programming for data mining algorithms and systems. While it is rather obvious that parallel computation can boost the performance of data mining algorithms, it is less clear how parallelism would be exploited at best. Starting from the decomposition of the algorithms given in chapter 2, we examine the opportunities for parallelism exploitation in the different components.

In section 3.1 we begin our discussion by confronting the issues of parallel and distributed data mining. There is a growing opportunity of interaction between the two approaches, as hierarchical system architectures (SMP clusters and computational grids) become commonplace in HPC.

Section 3.2 is the main part of the chapter. Parallel decomposition of the algorithm requires structural changes that can impact several different levels of the algorithm. We now look at which components of the mining algorithm need to be changed, what are the parallel design options for each one, and their reciprocal interactions.

We distinguish the two interdependent classes of *data components* (section 3.2.1) and *functional components* (section 3.2.2) in data mining algorithms. The elementary components described in the previous chapter can be seen as more related to the functional nature of the algorithm (the score function, see section 3.3, and the search strategy, section 3.4) or to the organization of its data (section 3.5 is about data management).

Section 3.6 compares our results with related work about common patterns for parallelism exploitation in data mining algorithms. The chapter ends with section 3.7 about the issues of KDD system integration for parallel data mining algorithms.

3.1 Parallel and Distributed Data Mining

The problem of exploiting parallelism in data mining algorithms (*Parallel Data Mining*, PDM) is closely related to the issues of *distributed data mining* (DDM). Both research fields aim at the solution of scalability and performance problems for data mining by means of parallel algorithms, but in different architectural settings. As it generally happens with the fields of parallel and distributed computing, part of the theory is common, but the actual solutions and tools are often different.

PDM essentially deals with parallel systems that are tightly coupled. Among the architectures in this class we find shared memory multiprocessors, distributed memory architectures, clusters of SMP machines, or large clusters with high-speed interconnection networks.

DDM, on the contrary, concentrates on loosely coupled systems such as clusters of workstations connected by a slow LAN, geographically distributed sites over a wide area network, or even computational grid resources.

Both parallel and distributed data mining offer the common advantages that come from the removal of sequential architecture bottlenecks. Higher I/O bandwidth, larger memory and computational power result than those of existing sequential systems, all these factors leading to lower response times and improved scalability to larger data sets.

The common drawback is that algorithm and application design becomes more complex. We need to devise algorithms and techniques that distribute the I/O and the computation in parallel, minimizing communication and data transfers to avoid wasting resources.

In this view PDM has its central target in the exploitation of massive and fine-grain parallelism, paying closer attention to synchronization and load balancing, and exploiting high-performance I/O subsystems where available. PDM applications deal with large and hard problems, and they are typically designed for intensive mining of centralized archives.

By contrast, DDM techniques use a coarser computation grain and loose assumptions on interconnection networks. DDM techniques are often targeted at distributed databases, where data transfers are minimized or replaced by moving results in the form of intermediate or final knowledge models. A widespread approach is independent learning integrated with summarization and meta-learning techniques such as voting, arbitration and combining. The work [PCS00] is a excellent survey about meta-learning techniques and on other issues of distributed data mining.

Distributed data mining comes into play because our data and computing resources are often naturally distributed. Some databases are distributed over several different sites and cannot be centralized to perform the DM tasks. There may be cost reasons for that, because moving huge datasets from several different sites to a centralized repository is a costly and lengthy operation despite broad-band data connections. Moreover, moving a whole database can be a security issue if the data are valuable. Many countries have regulations that restrict from freely transferring certain kinds of data, for instance in order to protect personal privacy rights. It may then even be illegal for a multinational firm to mine its databases in a centralized fashion. If the data transfer is not a problem, other resources might be. Centralized mining of the whole data may require too much computational power or storage space for any of our computing facilities. In this cases, a distributed algorithm is the only way to go.

The two fields of PDM and DDM are not rigidly separated, however. Depending on the relative characteristics of the problem and of the computational architecture, the distinction between fine grain, highly synchronized parallelism and coarse grain parallelism may be not too sharp.

Also in view of recent architectural trends, massively parallel architectures and large, loosely coupled clusters of sequential workstations can be seen as the extremes of the range of large-scale High-Performance platforms for Computational Grids at the cluster, Intranet and Internet level [FK98a].

High performance computer architectures are more and more often based on a large number of commodity processors. Scaling SMP architectures is difficult and costly, so large clusters of SMPs are becoming increasingly popular resource for parallel computation at high degrees of parallelism. Beside the use of dedicated, fast interconnection networks, the technology of distributed systems and of parallel ones is nearly the same. On the other hand, the enhancement of local area networks makes possible to use distributed computing resources in a more coordinated way than before. The building of Beowulf-class parallel clusters, the exploitation of the Internet for grid computing or for distributed computing are all expressions of the diminishing gap among massively parallel and distributed computing.

As already noticed in [Zak99] it is thus definitely realistic to devise the feasibility of geographically distributed data mining algorithms, where the local mining task is performed by a parallel algorithm, eventually on massively parallel hardware.

3.2 Parallelism Exploitation in DM Algorithms

We want exploit parallelism to remove the processing speed bottleneck, main memory size bottleneck and I/O bandwidth bottleneck of sequential architectures, thus solving the performance and scalability issues of data mining algorithms.

In order to achieve these results, we have to confront with the design of parallel algorithms, which adds a new dimension to the design space of programs. Mixing the implementation details of parallelism and data access management with those of the search strategy makes the design of a parallel data mining algorithm a complex task.

We have seen in the previous chapter that a data mining algorithm can be specified in term of five abstract characteristics:

- the data mining task that it solves;

- the knowledge model that the algorithm uses;
- the score function that guides the model search;
- the search strategy employed;
- the data management techniques employed in the algorithm.

In this chapter we look at the possible decomposition of these components, exploring the options for parallelism exploitation in order to identify problems, and interactions among the components. We will also try to separate as much as possible the consequences of parallelism exploitation on different levels of the data mining application.

Having chosen a specific data mining technique, we can split the algorithm components in two main classes that obviously interact with each other, the *data components* and the *functional components*. The data components will have to be partitioned, replicated or distributed in some way in order to decompose their memory requirements, and to distribute in parallel the workload due to the functional components of the algorithm. On the other hand, the functional components will require modification in order to be parallelized, leading in general to a parallel overhead which depends on the data decomposition and on the characteristics of the original functions.

Our design choices depend on how much effective is the partitioning of data in inducing independent computations, compared to how high is the cost for splitting problems and recomposing results at the end of the parallel algorithm.

3.2.1 Data Components

Data components are the two main data structures that the algorithm works on, the input and the model representation. We continue to use the tabular view: since the meaning and the management of the data are quite different, and have a deep impact on the parallelization of the functional parts of the algorithm, we will deal with instances and attributes as different data components, which are decomposed along two independent axes.

Data decompositions are often found in sequential data mining too, as enforced by the issues of memory hierarchy management (section 1.6). Thus considering splits of the input along two directions, we distinguish three main parallelization options with respect to the data structures.

Instances The records in the input $\mathcal{D}$ usually represent different elements. Horizontal partitioning into blocks does not change the meaning of the data items, and it is often simply called *data partitioning* in the literature. In the following we will use both terms to generally mean partitioning the records into subsets, not necessarily in consecutive blocks. Horizontal partitions usually produce smaller instances of the same data mining problem.

Attributes Separate vertical partitions (subsets of attributes in the input data) represent a projection of the input in a narrower data space. The meaning of data is now changed, as any knowledge resulting from correlation among attributes belonging to different vertical partitions is disregarded. This may not be a problem if we want to compute score functions which do not depend on attribute correlation (e.g. attribute statistic properties), or if the separate sub-models can be efficiently exploited to recompose a solution to the original problem.

Model Since data mining employs a wide range of different knowledge models, we do not try to specify in a general way how a model can be partitioned, we just underline a few issues. One issue is the kind of dependence of the model from the data, if all the data concur in specifying each part of the model, or if it is possible to devise a decomposition of the model which corresponds to one of the input set.

Knowledge models consisting of pattern sets are a peculiar issue, as partitioning of the pattern space usually induces some easily understandable (but unknown) kind of partitioning on the input data[1]. This leads to a vicious circle, since to exploit the data partitioning we need information about the model structure, which is exactly what we expect to get from the algorithm. Some pattern-mining algorithms thus exploit knowledge gained from sampling to optimize the main search process, or intermediate, partial knowledge gained so far to optimize data placement for the following steps.

While we have been talking about partitioning, in some cases we can exploit instead (partial) replication of data, attributes or model components. There is an additional tradeoff involving how much parallelism we can introduce, how much information we replicate, and the overhead of additional computation to synchronize or merge the different copies.

A main choice about data components is if data partitioning has to be static or dynamic. The tradeoff here is among data access time at run time and parallel overhead in the functional components due to the additional computation and data management.

Example 3.1 *The induction of decision trees from numeric values described in section 2.8 is an example of a difficult choice among static and dynamic partitioning. Using the gini index score function leads to a vertical access pattern to the data columns in sorted order, while the underlying knowledge model requires horizontal, unbalanced partitioning of the data at each search step. Data partitioning schemes thus need to balance access locality and load balancing. We will discuss the problem in chapter 6.*

[1]The projection of a partitioning in the pattern space is often a set of subsets in the data, not a real data partition. An example is found in association rule mining, described in chapter 5.

3.2.2 Functional Components

Functional Components operate on the data, so their parallelization is linked to that of the data components. Functional decomposition implies a parallel overhead that can sometimes be negligible, but can also dominate the computational cost.

Score Function The score function we use to evaluate a model or a sub-model over the data can exhibit several properties which can ease some of the decomposition strategies. A first issue is that score functions may have a prevalent horizontal or vertical access pattern to the data, according to their instance-oriented or attribute-oriented nature. Functions with unordered access patterns can exploit different data structures to reorganize the data. Score functions defined on large arrays can lead to data-parallel formulation, if the load balancing and synchronization issues can be properly addressed.

Search Strategy Depending on the kind of search strategy employed, several parallelism opportunities can be considered

- sub-model parallelism
- state space parallelism
- search-step level parallelism

Sub-model parallelism means splitting the model in several independent sub-models, where separate searches can aim for the locally best sub-model. The results are then composed into a solution for the larger problem. The state space is partitioned in this case.

State-space parallelism in our view is running in parallel several searches in parallel in the same state space.

Search step parallelism is exploited by evaluating in parallel several candidates for the next step, to select the best one. It essentially requires exploiting parallelism among more invocations of the evaluation function within a neighborhood of the current solution.

The parallel decomposition of the search process interacts with the knowledge model decomposition. Another issue for the search strategy is exploiting static or dynamic partitioning of the model and the state space.

Other data management tasks Data management tasks may be inherited by the sequential formulation of the algorithm (DBMS-resident or disk-resident data, complex data structures like external memory spatial indexes). However, the hardest management tasks for parallel algorithms often come from the choices made in partitioning the data to exploit parallelism. An example can be the need to reorganize the data when employing dynamic partitioning of the dataset. It is a significant cost in the computation, which has to be balanced with the saving in exploiting a higher degree of locality.

For each functional component at least three cases are possible with respect to the overhead of parallel decomposition.

- The overhead can be inexistent, because the operations performed are actually independent.
- The overhead is low, it is due to simple summarization operations, which produce aggregate results, and/or to broadcast communications, which spread the parameters of the current model. Here we assume that the computation and the communication volume are negligible, but we must keep into account the need to synchronize different execution units in order to exchange the data. Load balancing, for instance, is an issue in this case.
- We can have a moderate overhead in decomposing a part of the algorithm. For instance, we have to join different parts of the model, or to access data that is not local, which results in additional computation with respect to the sequential case. The overhead will influence the speed-up and scalability of the algorithm, but the parallelization may still be effective.
- In the worst case, the amount of work needed to split the computation is comparable to the computation itself, so there is no easy way to parallelize this function. An example is the need to sort data for the sake of tree induction, which accounts for the largest part of the computational cost of the algorithm (see chapter 6). We can apply the best possible techniques. It may also be possible to parallelize other components of the algorithm.

The hard issue is that these behaviors are often interdependent among the different functional components, either directly or because of the decomposition of the data structures.

3.3 Parallelism and Score Functions

Computing in parallel the score function that evaluates our model over the input set is the first and easy way to exploit parallelism in a data mining algorithm. If the computation of the score function over the whole input can be decomposed to exploit data parallelism, we can increase the speedup and scalability of the algorithm. The limiting assumptions are that the computation involves all the data set and the parallel overhead (the cost of communicating and composing the partial results) is bounded.

Several problems from optimization theory have been confronted this way [CZ97]. The data and the model in this case are expressed by matrices, and evaluation of the current solution is defined by algebraic operations on rows and blocks of data.

The formalization of the problems leads easily to the exploitation of FORTRAN-like data parallel programming models, where parallelism is essentially expressed by distributing large arrays and applying the owner-compute rule.

Most of the efforts in the field are put in developing decomposition of the score function, and of the algorithm, that make efficient use of this formalism. The main difficulties are achieving efficient operation on sparse matrices, and computational workload balancing.

Data parallel evaluation functions can be exploited in some data mining problems, but the general framework is different from numerical optimization. In optimization theory we want to find a good solution to a hard problem, thus we use algorithms that spend a large number of iterations in improving a solution. Exponential-complexity problems are common, and improved solutions are often valuable, so performance and algorithmic gains aim at finding better solutions and making larger problem instances computationally tractable. The parallel design effort is put in achieving near peak performance on in-memory (eventually shared memory) computation.

While we find worst-case exponential problems in data mining too, the simple size of the data restrain us from using sophisticate algorithms, and we have to settle for simpler algorithms. It is much less likely that the data can fit in memory, so secondary memory issues have to be taken into account even when the data is fully distributed across the machine. We more often aim at distributing the data and at keeping low the number of iterations, because the simple cost of out-of-core data access can easily dominate the algorithm behavior.

Example 3.2 *An algorithm of the former kind is the EMax (expectation maximization) method employed in several numerical optimization problems, like in 3D image reconstruction [CZ97], and also in the Autoclass algorithm of section 2.7. The amount of computation required by Autoclass is quite high, and the score function is essentially data parallel, so it is impractical to execute it on out-of core datasets.*

Data parallel schemes like those discussed in [BT02] allow to decompose the score function computation, and to execute the algorithm on larger datasets, at the expense of a certain communication overhead.

Generic score functions that are not easily implemented as data parallel, may still be usefully decomposable into sub-functions that can be computed on subsets of the input.

Example 3.3 *Functions that compute statistics about the value of a small set of attributes are well suited to vertical partitioning. As an example, the information gain used in C4.5 and in other decision tree classifiers (see section 2.8) is computed independently on each attribute of a data partition. Vertically partitioning the data is a good solution to speed up the computation, but it is difficult to match it with the recursive nature of the search, which implies horizontally splits to generate subproblems. The speed-up in the score function by attribute partitioning is furthermore* *limited by the number of the attributes, and by poor load balancing (the columns require different computing resources, and many of them are no longer used as the search proceeds to smaller subproblems).*

Example 3.4 *The score function in association rule mining is the number of occurrences of a pattern in the database. By using a vertical representation for the data and for the patterns (which is known as TID-list representation, see chapter 5) the score function for each pattern is the length of its TID-list, which is trivially computed when generating the lists during the algorithm.*

3.4 Parallelism and Search Strategies

Several different search strategies are employed in data mining algorithm, with some of them being easily parallelizable.

Usually the outermost structure of the search is a loop[2], and often more strategies are mixed or nested inside each other. Two nested levels of search are commonly found, which have different characteristics from the point of view of parallelization. Moreover, several of these strategies can be guided by heuristics in order to prune the space and gain better performance.

We summarize the most commonly employed search strategies.

Local search. This is the general form of all state-space searches which cannot use global information about the state space. Local searches move across points in the state space (the intermediate solutions) by exploiting information about a neighborhood of the current solution. The neighborhood may be explicitly defined, or implicitly defined by means of operators which produce all neighbor state representations. Local searches may exploit a different degree of memory, as well as exact or heuristic properties of the problem to improve and speed up the search.

Greedy search. It is an example of a local search which selects the most promising point in the neighborhood and never backtracks. While they exploit all the available information in the neighborhood, greedy methods in general stop at a local optimum, not at the global optimum.

Branch and bound. It is an example of local search which aims at exploring the whole state space. It backtracks after finding a local optimum and looks for unexplored subspaces. Branch and bound implicitly enumerates large portions of the state space by exploiting the additive property of the cost function to rule out unproductive subspaces.

[2]In some cases a loop is a simple implementation of a more complex strategy. See for instance the implementation of the C4.5 algorithm, whose *D&C* structure is mapped in the sequential implementation to a depth-first tree visit. We used a data-flow loop to implement *D&C* in parallel.

Optimization methods. Several numerical methods exist which exploit analytic properties of the score function to explore the state space.

Steepest descent. It is the equivalent of greedy local search, as it selects the next step by following the gradient of the score function at the current solution (the most promising direction, which approximatively selects the best solution in a small neighborhood) and it never explicitly backtracks. In the general case, steepest descent may converge to a local optimum, and is not always granted to converge.

EMax algorithm. The expectation-maximization strategy, briefly described in section 2.7, iterates alternating the optimization of two different score functions, looking for the best point in the state space which explains the input data. As steepest descent, EMax can converge toward a local optimum.

Randomized techniques. By simply generating a random point in the solution space, we can obtain information about that point alone. The dimension of the parameter space prevents from applying purely randomized strategies to data mining. However, randomized strategies are commonly employed to reduce the likelihood that other search strategies get stuck in a local optimum.

Exhaustive search. This kind of search simply tries every combination of parameters to produce all possible states. Given the size of the input and of the state space, it is never used alone in data mining problems. It is applied to particular subproblems (e.g. exploring the neighborhood of a solution).

Collective strategies. We group under this name all search strategies which employ a (possibly large) set of cooperating entities which interact in the process of finding a solution. Examples of collective search are genetic algorithms or ant-colony-like optimization methods [CDG99].

From the point of view of parallel execution or decomposition, we must look at the feasibility of splitting the solution space and partitioning the model. As we mentioned before in the chapter, we can exploit

- sub-model parallelism (split state space)
- state space parallelism (shared state space)
- search-step level parallelism. (shared state space, same current solution)

Collective strategies implicitly use a shared model space. However, it has been observed that some collective strategies exhibit a “double speed-up” phenomenon, discussed in section 3.6, which makes them well-suited for parallelization.

Randomized strategies also impose no particular problem, as sampling the state space can easily be done in parallel. However, replication of all the data is required, which can be an issue. When used as an outer loop strategy, it may happen that the randomization is guided by the preceding attempts (this is the case of Autoclass, for instance). It may well happen that the results of applying several randomized searches independently is not equivalent to running them in sequence. The performance and the quality of the solution may thus not improve as expected.

It requires analysis of the model to understand if a local search can be exploited on sub-models. As a general rule, it is easier to build patterns than models this way. It is often the case that the sub-models are induced by a split in the input data (an example is the effect on the pattern space of association rules by careful partitioning of the input) and are thus not entirely disjoint.

Complex strategies like the EMax one give an edge to the parallelization of their lower-level tasks, like the computation of the score function.

While we have been discussing the options of parallelization of the search, we have not yet addressed a key point, the *structure* of the model. If the model is not flat, i.e. it has a recursive structure, then there are chances that the optimization tasks on the parameters of the sub-models can be separated. This happens for different branches of decision trees, and in some hierarchical clustering algorithms (this is the case of CLIQUE [AGGR98]). As we turn sequential sub-model computations into parallel, we also map a sequential visit of the model structure, which accomplishes the search in the sequential case, into a parallel expansion scheme.

Example 3.5 *For decision trees, the tree expansion schemes (depth first search, breadth first search) tend to be mapped into breadth first expansions which operate on all the nodes of a level in parallel. The actual expansion process is however constrained by other issues, like load balancing, communication and synchronization costs. More complex search strategies are thus developed, which modify their behavior at run time, based on problem and performance measurements.*

3.5 Parallel Management of Data

The data management issues are a concern already for sequential data mining algorithms. The parallel version of an algorithm inherits these issues (in-memory data management, out-of-core management, DBMS exploitation). Among them, exploiting a DBMS for parallel mining is usually the hardest problem, as a single DBMS may not be able to service the amount of queries generated by a parallel program.

Parallel I/O exploitation is easily realized by means of local resources in each processing node if we know how to partition the data, and if the partitions are balanced.

There are however other issues that result from partitioning the input data. Splitting the data horizontally or vertically, according to the decomposition of our model and search, may require to reorganize the data before running the algorithm.

Two main problems surface.

- Complex data structures used to optimize data management may not be easily turned into parallel ones.
- It happens that there is no static partitioning that allows strictly local computation, or that such a partitioning cannot be computed in advance.

Example 3.6 *Many clustering algorithm, including k-means, can employ tree-like data structures to keep close points together and exploit memory locality. Clustering algorithms avoid useless computation by maintaining* clustering features *and* bounding boxes *of sets of point to avoid repeated computations. These data structures however are hard to decompose in parallel, as their internal organization changes during the algorithm. The same problem is faced in several numerical simulation which aim at dynamically exploiting the evolving spatial properties of data (e.g. adaptive multigrid methods).*

Note that in the case of k-means clustering, what actually needs data management is the amount intermediate information that is used to direct the search of our nicely compact model (the set of centroids).

Example 3.7 *In decision tree induction, the subdivision of the initial problem into subproblems is inherently dynamic and unpredictable. Reorganizing the input data at each search step (or bookkeeping additional information to allow efficient access anyway) accounts for the the largest part of the data management task.*

3.6 Related Work on Parallelism Exploitation

The analysis of parallel mining algorithm set forth in this chapter is an extension of the *reductionist viewpoint* presented in [HMS01] to take into account the parallelism-related issues.

Some of the work is based on an article by Skillicorn [Ski99] which distinguishes three classes of data mining algorithms. A high-level cost model is also proposed there for these classes, which is based on the BSP cost model. We have not yet developed our analysis to produce a cost model of this kind, which is undoubtedly a useful tool in designing algorithms. We believe that extensions of the BSP model, which are capable to deal with hierarchical memory issues (see section 1.6.1), will play a major role in developing more portable parallel algorithms.

In [Ski99] three classes of parallel mining algorithms are characterized. All of them are actually generalized to an iterative form of more consecutive parallel phases, which in the following we omit for the sake of conciseness.

Independent search is based on replicated input data and completely independent searches.

Algorithms in this class exploit the simplest form of parallel search, where all the input data is replicated and parallel execution of the sequential algorithm independently explore the same state-space, which defines the whole model. This class of parallel algorithms is important, as it can be extended to include complex algorithms employing randomization as their outer-level search strategy (e.g. clustering algorithms like Autoclass and k-means, or multiple-window supervised classification approaches).

Parallelized sequential is based on partitioning the input (instances or attributes) and employ the subdivision of the space-state that results in multiple parallel executions. The results are then joined to produce the true answer.

The attribute/instance-based partitioning projects the original state space into possibly smaller spaces for each partition. This is natural when doing vertical partitioning, as the model loses some of its parameters. In the case of horizontal partitioning the projected state space may coincide with the initial one. This class of parallelizations thus contains two subclasses, depending on the kind of join operation needed to reconcile the partial models.

- Models in the same state space need to be confronted to integrate knowledge. It is often the case that the resulting model is simply some kind of "sum" of the components which requires a small computational effort and/or a global validation.
- Models belonging to different state spaces usually need a more complex recomposition operation.

The parallelized algorithm class actually covers a wide range of algorithms, which exhibit a range of different recomposition issues for the knowledge gained in the parallel part of the computation. Estimating the computational cost of the joining phase requires a more in-depth analysis of the data mining algorithm. Looking from a different point of view, this class includes algorithms that obliviously partition the input and others that try to exploit informed partitioning scheme to enhance the independence of the resulting paralle computations.

Replicated sequential applies an instance-based (horizontal) partitioning to exploit independent sequential search for *approximate concepts*.

This could be actually seen as a subclass of the Parallelized-sequential one, but we use the specific assumption that the kind of model allows to generate complete models from horizontal partitions, and that these partial answers are all approximations of the final one. We assume that they can be joined with a relatively simple algorithm, and then verified or integrated in parallel over the data partitions. Algorithms in this class tend to parallelize well, as the parallel overhead is fairly bounded. Some of them also enjoy the benefit of a "double speedup" phenomenon. The exchange at each phase of highly concentrated knowledge[3] allows each independent search to quickly converge in the state space, like if it had available a much larger amount of data than its own partition.

3.7 Integration Issues

It is now recognized that a crucial issue in the effectiveness of DM tools is the degree of interoperability with conventional databases, data warehouses and OLAP services. Maniatty and Zaki [MZ00] state several requirements for parallel DM systems, and the issues related to the integration are clearly underlined.

System Transparency is the ability to easily exploit file system access as well as databases and data warehouses. This feature is not only a requirement of tool interoperability, but also an option to exploit the best software support available in different situations.

Most mining algorithms, especially the parallel ones, are designed for *flat-file mining* (exploiting data-mining on flat files with an algorithm-specific layout). This simplification eases initial code development, and it is often justified by the access to DBMS being too slow for high-performance computation. On the other hand, being restricted to flat-file mining imposes an overhead when working with higher-level data management supports (e.g. we need to dump the data to a specific format and to fully materialize DBMS views).

Industry standards are being developed to address this issue in the sequential settings, as noted in section 2.4, and research is ongoing about the parallel case (see for instance the book by Freitas and Lavington [FL98]). We can distinguish three main approaches.

- Development of data mining algorithms based on existing standard interfaces (e.g. SQL). Many algorithms have been rewritten or designed to work by means of DBMS primitives. DBSCAN (chapter 7) is designed assuming the use of a spatial database system.

- Development of a few new database-mining primitives within the frame of standard DBMS languages (a crucial issue is the expressive power that we may gain or lose).

- Design of dedicated, possibly parallel mining interfaces to allow tight integration of the mining process with the data management.

Pushing more of the computational effort into the data management support means exploiting the internal parallelism of modern database servers. On the other hand, scalability of such servers to massive parallelism is still a matter of research.

[3]Here we view each knowledge searching process as trying to compress information about its data partition.

While integration solutions are now emerging for sequential DM, this is not the case for parallel algorithms yet.

The bandwidth of I/O subsystems in parallel architectures is theoretically much higher than that of sequential ones, but a conventional file system or DBMS interface cannot easily exploit it. We need to use new software supports that are still far from being standards, and sometimes are architecture-specific.

Parallel file systems, high performance interfaces to parallel database servers are important resources to exploit for PDM. DDM must also take into account remote data servers, data transport layers, computational Grid resources, and all the issues about security, availability, and fault tolerance that are commonplace for large distributed systems.

Our approach is to develop a parallel programming environment that addresses the problem of parallelism exploitation within algorithms, while offering uniform interfacing characteristics with respect to different software and hardware resources for data management. Structured parallelism will be used to express the algorithmic parallelism, while an object-like interface will allow accessing a number of useful services in a portable way, including other applications and CORBA-operated software.