MODULARIZING DATA MINING: A CASE STUDY FRAMEWORK

Jeremy T. Engle, Edward L. Robertson
Department of Computer Science, Indiana University University, Lindley Hall 215, Bloomington, USA
jtengle@indiana.edu, edrbtsn@indiana.edu

Keywords: Approximate Functional Dependencies, AFD, Data Mining Engineering, Data Mining Architecture, Lozenge Search, Powerset Lattice Search, MoLS.

Abstract: This paper presents the fundamental concepts underpinning MoLS, a framework for exploring and applying many variations of algorithms for one datamining problem: mining a database relation for Approximate Functional Dependencies (AFDs). An engineering approach to AFD mining suggests a framework which can be customized with plug-ins, yielding targetability and improved performance. This paper organizes familiar approaches for navigating a search spaces and introduces a new concepts to define and utilize variations of those spaces.

1 INTRODUCTION

This paper presents the fundamental concepts underpinning MoLS, a framework for exploring and applying many variations of algorithms for one data mining problem. While the implementation of this system targets just one problem, Approximate Functional Dependency (AFD) mining, we present the underlying concepts in the hope that these concepts will motivate similar developments concerning other data mining problems that share similar search concerns.

The use of mined information in applications has become both more prevalent and more diverse in recent years. In order to meet the diverse needs of these applications, developers often create *ad hoc* extensions of black box mining systems. We advocate that the solution should instead be a mining framework that allows the use of a family of algorithms that result from users customizing existing or novel modules. To truly provide flexibility, the search algorithm itself must be modularized. In this paper we undertake a conceptual approach to constructing an algorithm as a set of plug-ins and present the design of a framework, MoLS, in accordance with that approach. Preliminary results from this case study present customizations and evidence improved performance (Engle and Robertson, 2008).

There is no single algorithm which could meet all of the known or unforeseen future needs of developers, so instead a mining system must be designed to accommodate a family of algorithms and customizations. Accommodating a family of algorithms requires a deep understanding of how algorithms find and verify results. It is the combination of designing with customization in mind and understanding how customizations impact finding results that make a framework approach possible. The resulting framework is also an ideal testbed for efficiency-based customizations. There are three fundamental questions which characterize this family of algorithms:

(*i*): how are search spaces demarcated?
(*ii*): how does an algorithm move through spaces?
(*iii*): how is approximateness measured and how does the cost of measurement fare in the face of search space combinatorics?

The problem domain of this case study is the discovery of Approximate Functional Dependencies (AFD's) in an instance of a database relation. Extending the notion of Functional Dependency (FD), (see any database text, *e.g.* (Ramakrishnan and Gehrke, 2002)), an AFD is an FD which almost holds, where almost is defined by a parametrized measure and threshold. The search space in this domain is based on combinatorially huge powerset lattices.

Understanding the range of AFD mining algorithms of course began with existing techniques (see §3 below), but that range exhibited limited variation. Our own investigations broadened the range by meaningfully posing question (*i*) (Engle and Robertson, 2008). That paper answered question (*i*) by introducing Lozenge Search (LS), but did little to allow customizations in answering question (*ii*). It was these results which motivated the desire for a framework which encapsulated each question, thus allowing variations at every level of the algorithm in a clean and uniform way.

After establishing definitions and conventions (§2), we identify the salient features of other AFD miners (§3). Next we discuss the foundational background of the problem domain (§4) in order to understand the range of variations that a framework must handle, and then outline how the architecture provides the intended modularity (§5).

2 DEFINITIONS

The paper uses the following conventions:

- $\boldsymbol{R}$ is a relation schema and $\boldsymbol{r}$ is an instance of $\boldsymbol{R}$.
- $A, B, C, \cdots$ are individual attributes of $\boldsymbol{R}$.
- $X, Y, Z, \cdots$ are sets of attributes.
- $X \rightarrow Y$ is a rule, a pair of subsets of $\boldsymbol{R}$; henceforth Y has only one attribute.
- LHS abbreviates Left Hand Side; RHS, Right Hand Side; BU, Bottom UP; TD, Top Down; BFS, Breadth First Search; DFS, Depth First Search.

Definition 2.1
$X \rightarrow A$ is a *parent* of $W \rightarrow A$ when $W \subset X$ and $|X| = |W+1|$. Additionally, *child*, *descendant*, and *ancestor* have their usual interpretation with respect to *parent*.

The characterization of how close a rule is to an FD is done with an approximation measure φ evaluated on $\boldsymbol{r}$. All that is required of φ is (a) that it map rules into [0,1], with $\varphi(X \rightarrow Y) = 0$ iff the FD $X \rightarrow Y$ holds and (b) φ is monotone non-increasing as the LHS grows.

Since every rule has some approximation value, a threshold ε is specified as an upper bound for acceptable rules; henceforth ε is assumed to have a fixed (yet arbitrary) value. A rule $X \rightarrow Y$ is called a *pass* rule when $\varphi(X \rightarrow Y) \leq \varepsilon$ and a *fail* rule when $\varphi(X \rightarrow Y) > \varepsilon$. A *minimal pass* (*minP*) rules is a pass rule all of whose children fail; *maxF* is defined symmetrically.

The terms lattice and sub-lattice have standard meanings for the powerset of some collection of attributes. These meanings are augmented when applied to AFDs since a rule has both a LHS and RHS. A rule lattice is a lattice of LHS attribute sets which all share the same RHS attribute.

3 RELATED WORK

Mining for AFDs is a specialized subtopic within the broad range of data mining activities. There are two commonly cited AFD mining algorithms: TANE and an algorithm which we denote as B&B.

TANE, the most widely used AFD mining algorithm, was developed by a group at University of Helsinki. Their work included development of the g_3 measure (Kivinen and Mannila, 1995) and a partition-based data structure which facilitates efficient evaluation of g_3 (Huhtala et al., 1999). At its core, TANE uses a traditional bottom-up BFS. Other than specifying the parameter ε and of course the data set to be mined, TANE allows no customizations.

B&B, reported in (Bell and Brockhausen, 1995), originally mined for FDs and was extended in (Matos and Grasser, 2004) to mine for AFDs using the g_3 approximation measure. Lopes *et al.*combined B&B and TANE with formal concept analysis in a framework, though it is unclear how different plug-ins could be used in that framework.

The original motivation for AFD mining was to extend FD-based query optimization (Giannella et al., 2002; Ilyas et al., 2004). More recently, researchers have used AFDs to model probabilistic relationship for prediction (Andritsos et al., 2004; Aussem et al., 2007; Wolf et al., 2007a; Wolf et al., 2007b; Nambiar and Kambhampati, 2004).

4 FOUNDATIONAL CONCEPTS

A good modular implementation is only possible when it is based on an even better conceptual analysis. Performing such analysis is even more important when developing a framework that supports a broad family of algorithms. Hence, this section discerns significant fundamentals of the problem domain, then addresses the more familiar topic of moving through a lattice (question (*ii*)), and finally discusses the demarcation of the global search space (question (*i*)) in light of the discussion of question (*ii*).

Question (*iii)* is answered straight-forwardly by encapsulating the evaluation of φ. Encapsulating measure evaluation sharpens the focus on the other aspects of the framework design and facilitates the use of different measures. As the framework is independent of measures and the means by which measures are calculated, an exploration in measures can be found (Giannella and Robertson, 2004).

4.1 Monotonicity and Inferencing

Monotonicity is used in a variety of data mining algorithms to prune or terminate search since it guarantees that a threshold, once crossed, will not be recrossed. In the AFD context, the requirement that an approximation measure is monotonic (clause (b) in the characterization of φ), guarantees that, when a *pass* rule is found, it may be inferred that all its ancestors also *pass*. Symmetrically, descendants inherit failure.[1] Because of this inferencing, AFD miners return only *minP* rules, hopefully reducing a combinatorial explosion in output.

The inference of pass status is the basis upon which BFS AFD mining algorithms prune the search space in order to eliminate non-minimal results; the symmetric downward inference of failure is not used (or even relevant) in BFS. However, inferencing in both directions may be relevant in DFS algorithms. Inference is beneficial because it can help limit searching and because it may avoid the expensive step of visiting the data in order to evaluate the approximation measure. Traditional lattice mining algorithms combine these two facets of inferencing by pruning nodes which are neither evaluated nor explored; MoLS facilitates inferencing that explores but does not discard nodes.

4.2 Boundary Rules

The notion of *boundary* captures how monotonicity divides the lattice into regions (technically, into semi-lattices). The boundary is fundamental because desired result lie along it. This notion has appeared in certain specialized applications (*e.g.* (Cong and Liu, 2002) or (Yan et al., 2004)) but rarely as a general lattice-based concept. In the AFD context, the boundary separates the pass region of a lattice from the fail region. A *boundary rule* is either a pass rule with at least one failing child or a fail rule with at least one passing parent. Not only can the status of any rule be inferred from the boundary but the boundary itself may be inferred from the sets of *minP* and *maxF* rules.

[1]These inferences are not the inferences associated with FDs and Armstrong's Axioms (Ramakrishnan and Gehrke, 2002), although properties of the InD measure generalize Armstrong's approach (Giannella, 2002).

There are three corollaries of the characterization of the boundary. First, the minimum effort algorithm evaluates the measure φ on exactly the *minP* and *maxF* rule sets, assuming that these sets are provided by some oracle. Second, any algorithm that evaluates φ on at least these sets is sound and complete with respect to an exhaustive exploration of the space. Third, any sound and complete algorithm must evaluate φ on these sets.

These corollaries determine an optimal (but likely unachievable) benchmark, suggesting how the framework should be instrumented for performance evaluation. They also suggest that a good algorithm should find the boundary as quickly as possible and then explore that boundary in the most expeditions fashion.

4.3 Navigating a Lattice

Navigation from node to node within a lattice takes many forms: traditional BFS, DFS, and many nuanced variations motivated by boundary considerations. The major question, (*ii*), breaks into two: what are the candidate rules that should be considered for visitation in the future and what is the order in which these candidate rules should be visited.

An example of the framework's application is provided by the well-known BU-BFS. BU-BFS answers the first part by having fail rules add their parents as candidates and the second part by ordering according to level in the lattice. More simply, a FIFO queue suffices, building the order into a data structure; BU-DFS uses a LIFO stack in a similar manner.

From a software engineering perspective, providing a mechanism to specify visit order encapsulates a complex decision process with a simple module interface. Customization determines whether and when, the framework carries out these decisions.

In another domain, Frequent Itemset Mining (FIM), candidate generation has received considerable attention and is often a major factor differentiating algorithms. We mention FIM only to highlight a component which has received little attention in AFD mining but is considered highly significant in a different domain. A more elaborate example of candidate generation is found in the jumping algorithm (Dexters et al., 2006) which, as its name implies, can generate rules several levels away.

4.4 Demarcating Global Search Space

This section addresses the possible answers to question (*i*) above. Demarcation forms portions of the global search space to be considered separately.

Although question (*i*) is generally applicable to many data mining algorithms, it is natural that it was first meaningfully asked in the context of AFD mining, where the search space has inherent structure: n independent power-sets over $n-1$ attributes rather than a single powerset of n attributes. B&B use 4n4 independent power-sets while TANE use only one.

Unlike the wide variations in ways to navigate a lattice, there is limited flexibility available in the ways to demarcate a space, always involving an iteration though attributes. At each iteration, a *demarcated* subspace is constructed. For example, when the demarcated space is the set of all rules with a fixed attribute on the RHS and when the navigation is BU-BFS, the framework implements B&B.

The approach which demarcates a space by adding one attribute at a time to the set of of *active* attributes was proposed in (Engle and Robertson, 2008), with the LS algorithm. In this approach, the order in which attributes are added is highly significant. Preliminary tests show that some orders tend to work much more quickly. Furthermore, if the attributes were ordered according to some aspect of the user's interest, the algorithm that user could terminate the algorithm when a significant set of rules were discovered.

Within this fixed pattern of active space growth and relying on domain considerations discussed above, the framework can provide optimizations and guarantees which the user need not consider. Inferencing considerations can be used to to ensure irredundant searches as the active space grows. In LS, a projection of the boundary onto the active space leads to to an effective and efficient way to carry forward partial results as attributes are added. Finally, the facts concerning the boundary allow the framework to guarantee soundness and, if the search is sufficiently extensive, completeness.

5 IMPLEMENTATION

A framework provides a skeleton independent of the combination of plug-ins. We propose a design for how conceptual elements of algorithms map to framework plug-ins and discuss what aspects of search are universal and therefore should occur in the skeleton.

The intent of proposing MoLS is to describe the responsibilities of the framework and of the different plug-ins. The skeleton allow users full control of search customization without an excessive burden from mechanics. The goal is to make plug-ins easier to write and reusable so that proven portions of code can bootstrap the writing of new algorithms.

5.1 The Invariant Framework

The framework architecture is shown in Fig. 1. The figure indicates where the skeleton of the framework accepts plug-ins (single-line boxes) and where it takes on a more meaningful role (double-line boxes). The dashed lines correspond to the bodies of respective nested iterations.

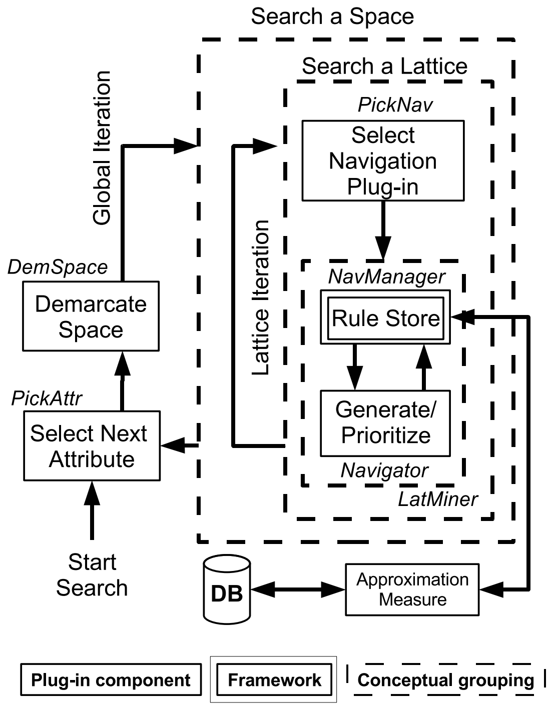

Figure 1: High Level Flow of MoLS

Global Iteration handles question (*i*). The plug-in *PickAttr* returns the next attribute added to the active set; *DemSpace* constructs the demarcated subspace in the current iteration. This simple interface allows the customization to dynamically choose the order in which attributes are added. At the level of Global Iteration, the framework only does set-up; the significant work actually happens at the next level down, on a per-lattice basis.

The most complex framework/plug-in interaction occurs in the innermost block. As the figure shows, there is a coroutine-like interaction between *Navigator* and *NavManager*, the portion of the framework that manages searching a specific lattice. Similar to the other portions of the framework, *NavManager* has a number of bookkeeping tasks, such as removing rules from the search queue for visitation and making calls to the approximation measure.

No matter the implementation, every AFD mining algorithm determines *minP* rules in a similar manner.

Except for bottom-up BFS, determining which rules are *minP* is a significant task, as a different navigation is needed. For example, a top-down algorithm must continue searching below *minP* rules to find *fail* rules and then somehow pass that information back to parent rules to determine whether the *minP* definition is met. As we will discuss later, making the framework responsible for this determination makes customization much easier.

5.1.1 Rule Store

Recognizing that space costs may be as problematic as time costs in the face of a combinatorial explosion, *NavManager* has a single mechanism for storing information for both itself and for use by *Navigator*. This mechanism is called the *rule store*. Rule store information used by *NavManager* includes whether a rule has been visited and the *pass*/*fail* status of a visited rule. This information is used to determine *minP* rules. The rule store and effects of using inferred information can be found in (Engle and Robertson, 2009).

As is often the case in algorithm implementation, there is a "save or recompute" tradeoff pertaining to the contents of the rule store. For example, should *pass*/*fail* status information remain in the rule store until the end of searching the lattice and then be used to determine all *minP* rules in a single final pass or should determination of *minP* rules be done incrementally, allowing information to be discarded earlier at the cost of rediscovering some of that information.

The rule store changes this tradeoff by providing a central point for inferencing, treating inferred knowledge the same as knowledge from evaluation. When information is added to the rule store, *NavManager* can infer the status of other rules and store that knowledge additionally, providing the benefits of inferencing without *Navigator* ever needing to understand how it happens.

5.2 The *Navigator* Plugin

As noted above, navigation is a matter of generating and ordering candidate rules. MoLS's mechanism for this is a priority queue, with distinct interfaces for enqueing and for setting priorities. Having a priority queue, as opposed to a data structure implementing FIFO or LIFO order, is itself innovative in data mining systems; allowing the priority to change dynamically is highly so.

Candidate generation is expected to change little from current practice, *i.e.* bottom-up algorithms generate parent rules and top-down would generate children rules. Other generation tactics, such as jumping to a more distant ancestor or descendant, require a more significant use of the rule store and interactions between *Navigator* and *NavManager*; details of this interaction have not yet been fully determined.

The second, less commonly explored, part of navigation is the priority indicating the order in which candidate rules are considered. The use of priority ordering merely makes explicit what other algorithms do implicitly, such as a breadth-first navigation using a simple queue to obtain a FIFO order. Guiding exploration using priority ordering allows easy switching between navigation modes. For example, bottom-up BFS gives higher priority to rules with fewer attributes while bottom-up DFS prioritizes rule along a path. By merely flipping the priority, a different mode is implemented.

The final part of navigation is determining when to stop searching. As with other in other algorithmic facets discussed above, MoLS makes explicit what is implicit in other approaches. In particular, *NavManager* terminates the exploration of work on one lattice when the queue for that lattice becomes empty or the priority falls to zero. That is, the framework allows *NavManager* to indicate that a candidate should not be evaluated merely by setting the priority to zero.

5.2.1 The *PickNav* Plugin

The ability to easily adjust the navigation mode naturally raises the issue of determining the proper mode for the current context; to this end, the navigation mode can be chosen on a lattice-by-lattice basis. It was shown in (Engle and Robertson, 2008) that combinations of algorithms could improve performance. The *PickNav* plug-in implements this mode selection. While it is possible that the navigation mode could be changed in mid-lattice, this seems an unlikely way to develop a correct and coherent implementation. Thus *PickNav* is evoked only to initiate the exploration of a lattice.

The idea of selecting the best algorithm for a problem has received attention in the AI world (Smith-Miles, 2008) and remains an open problem. In AFD mining, AI could assist assist a particular *PickNav* since higher-level reasoning is particularly applicable.

5.3 The *PickAttr* and *DemSpace* Plugins

The space demarcation requirements deriving from Section 4.4 are implemented with the *PickAttr* and the *DemSpace* plug-ins. We split question(i) from Section 1 into *PickAttr* and *DemSpace* because the functionalities which each implements are expected to exhibit very different change patterns. Search space demarcation, implemented by *DemSpace*, may rarely change, while the order of the global iteration, controlled by *PickAttr*, is likely to be customized.

6 CONCLUSION

This paper describes the application of a software engineering paradigm to the design and implementation of data mining software, resulting in a framework for implementing AFD search algorithms. Considerations of modularity were a primary factor in framework design.

Data mining algorithms, particularly those mining for AFDs, have always been designed as black box systems. Though data mining algorithms are often compared (particularly according to performance factors), heretofore there has not been a general analysis of the generic commonalities and differences between these algorithms. Hence this paper first presents and discusses a a series of questions that create a conceptual breakdown for different tasks that vary across different algorithms. This conceptual breakdown has led to a framework that facilitates customization of algorithms at many levels of granularity.

The development of a highly customizable framework has two benefits deriving from the original motivation for building a framework. The first allows users of AFDs to build customized systems based on a common framework. The resulting design also makes it easier to write algorithms, which in turn can allows more variations of algorithms to be explored and tested.

The second benefit is the framework as a testbed for experimenting with a variety of AFD mining algorithms. The ease with which *Navigator* plug-ins can be written allow algorithms to be prototyped easily and correctly. Furthermore, MoLS allows comparing new search approaches to traditional ones using the same evaluation metrics. As a testbed, the framework has succeeded far beyond our expectations. We are continuing to understand new facets to the AFD mining problem and develop algorithms (Engle and Robertson, 2009). This strongly suggests that comparable frameworks would facilitate research in other areas of data mining.

REFERENCES

Andritsos, P., Miller, R. J., and Tsaparas, P. (2004). Information-theoretic tools for mining database structure from large data sets. In *SIGMOD Proceedings*, pages 731–742, New York, NY, USA. ACM.

Aussem, A., Morais, S. R., and Corbex, M. (2007). Nasopharyngeal carcinoma data analysis with a novel bayesian network skeleton learning algorithm. In *AIME '07: Proceedings of the 11th conference on Artificial Intelligence in Medicine*, pages 326–330, Berlin, Heidelberg. Springer-Verlag.

Bell, S. and Brockhausen, P. (1995). Discovery of constraints and data dependencies in databases (extended abstract). In *European Conference on Machine Learning*, pages 267–270.

Cong, G. and Liu, B. (2002). Speed-up iterative frequent itemset mining with constraint changes. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining*, page 107, Washington, DC, USA. IEEE Computer Society.

Dexters, N., Purdom, P., and Van Gucht, D. (2006). Peak-jumping frequent itemset mining algorithms. In *PKDDproc*. Springer.

Engle, J. T. and Robertson, E. L. (2008). HLS: Tunable mining of approximate functional dependencies. In *BNCOD*, pages 28–39.

Engle, J. T. and Robertson, E. L. (2009). Depth first algorithms and inferencing for afd mining. In *IDEAS '09: Proceedings of the 2009 international symposium on Database engineering &#38; applications*, New York, NY, USA. ACM.

Giannella, C. (2002). An axiomatic approach to defining approximation measures for functional dependencies. In *ADBIS Proceedings*, pages 37–50, London, UK. Springer-Verlag.

Giannella, C., Dalkilic, M., Groth, D., and Robertson, E. (2002). Improving query evaluation with approximate functional dependency based decompositions. In *BNCOD*, volume 2405 of *Lecture Notes in Computer Science*, pages 26–41. Springer.

Giannella, C. and Robertson, E. (2004). On approximation measures for functional dependencies. *Inf. Syst.*, 29(6):483–507.

Huhtala, Y., Kärkkäinen, J., Porkka, P., and Toivonen, H. (1999). TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111.

Ilyas, I. F., Markl, V., Haas, P., Brown, P., and Aboulnaga, A. (2004). Cords: automatic discovery of correlations and soft functional dependencies. In *SIGMOD Proceedings*, pages 647–658, New York, NY, USA. ACM.

Kivinen, J. and Mannila, H. (1995). Approximate inference of functional dependencies from relations. In *ICDT*, pages 129–149, Amsterdam, The Netherlands, The Netherlands. Elsevier Science Publishers B. V.

Matos, V. and Grasser, B. (2004). Sql-based discovery of exact and approximate functional dependencies. In *ITiCSE Working Group Reports*, pages 58–63, New York, NY, USA. Association for Computing Machinery.

Nambiar, U. and Kambhampati, S. (2004). Mining approximate functional dependencies and concept similarities to answer imprecise queries. In *WebDB Proceedings*, pages 73–78, New York, NY, USA. Association for Computing Machinery.

Ramakrishnan, R. and Gehrke, J. (2002). *Database Management Systems*. McGraw-Hill Higher Education.

Smith-Miles, K. A. (2008). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1):1–25.

Wolf, G., Khatri, H., Chen, Y., and Kambhampati, S. (2007a). Quic: A system for handling imprecision & incompleteness in autonomous databases (demo). In *CIDR*, pages 263–268.

Wolf, G., Khatri, H., Chokshi, B., Fan, J., Chen, Y., and Kambhampati, S. (2007b). Query processing over incomplete autonomous databases. In *VLDB Proceedings*, pages 651–662. VLDB Endowment.

Yan, P., Chen, G., Cornelis, C., Cock, M. D., and Kerre, E. E. (2004). Mining positive and negative fuzzy association rules. In *KES*, pages 270–276.