FINE-GRAINED PERFORMANCE EVALUATION AND MONITORING USING ASPECTS
A Case Study on the Development of Data Mining Techniques

Fernando Berzal, Juan-Carlos Cubero and Aída Jiménez
Dept. Computer Science and A.I., University of Granada, Granada 18071, Spain

Keywords: Aspect-oriented programming, performance monitoring, data mining techniques.

Abstract: This paper illustrates how aspect-oriented programming techniques support the I/O performance evaluation and monitoring of alternative data mining techniques. Without having to modify the source code of the system under analysis, aspects provide an unintrusive mechanism to perform this kind of analysis, letting us probe a system implementation so that we can identify potential bottlenecks.

1 INTRODUCTION

All programming methodologies provide some kind of support for separation of concerns, which entails breaking down a program into distinct parts that overlap in functionality as little as possible. The structured and object-oriented programming paradigms resort to procedures and classes, respectively, to encapsulate concerns into single entities and thus achieve some separation of concerns. However, some concerns defy these forms of encapsulation and lead to tangled, difficult-to-maintain code, since they cut across multiple modules in a program. Aspect-oriented programming overcomes this problem by enabling developers to express these cross-cutting concerns separately (Kiczales et al., 1997).

In this paper, we employ aspect-oriented software development techniques for solving a common problem programmers must face in the development of complex systems; namely, the fine-grained evaluation and monitoring of system performance. Since aspects provide an unintrusive way to tuck probes into their system, developers do not have to tweak their underlying system implementation for enabling system monitoring. As keen observers, they can study system performance without inadvertently introducing subtle errors nor degrading actual system performance in a production environment (aspects can easily be removed once the performance evaluation has been performed).

Our paper is organized as follows. Section 2 describes how cross-cutting concerns, or aspects, can be specified using AspectJ. Section 3 describes how aspects can be incorporated into a component-based framework. Section 4 presents a case study on the evaluation of the I/O performance of data mining techniques. Finally, Section 5 concludes our paper by summarizing the results of our study.

2 SPECIFYING CROSS-CUTTING CONCERNS WITH ASPECTJ

The main idea behind AOP is to capture the structure of crosscutting concerns explicitly, since these concerns are inherent to complex software systems but their implementation using conventional programming techniques leads to poorly-structured software.

Gregor Kiczales started and led the Xerox PARC team that eventually developed AspectJ (Kiczales et al., 2001), an aspect-oriented extension for the Java programming language.

AspectJ encapsulates crosscutting concerns in special classes, which are declared using the `aspect` keyword:

```
public aspect TMinerAspect
{
  // ... aspect implementation details ...
}
```

AspectJ aspects can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points specified by pointcuts. For those already familiar with current relational database management systems, we

Berzal F., Cubero J. and Jiménez A. (2008).
FINE-GRAINED PERFORMANCE EVALUATION AND MONITORING USING ASPECTS - A Case Study on the Development of Data Mining Techniques.
In *Proceedings of the Third International Conference on Software and Data Technologies - ISDM/ABF*, pages 259-262
DOI: 10.5220/0001874902590262

could say that AspectJ provides for object-oriented programs what triggers do for relational databases.

The complete details of the AspectJ language are covered in several programming handbooks (Laddad, 2003) (Gradecki and Lesiecki, 2003).

3 INSTRUMENTING A COMPONENT-BASED DATA MINING FRAMEWORK

We will now proceed to describe how we can equip a data mining framework written in Java with the necessary instruments for measuring and recording the number of I/O operations performed by a data mining algorithm. Later, we will show some experimental results we have obtained with the help of this instrumentation.

First of all, we must intercept component creation calls. In a typical object-oriented framework, component creation can be performed by directly invoking the corresponding constructor or by resorting to the reflection capabilities included within modern programming platforms.

For example, the following AspectJ snippet inserts the appropriate advice after every call to any constructor of any of the subclasses of the `Classifier` class:

```
after() returning (TMinerComponent component)
  : call( Classifier+.new(..) ) {
    addDynamicPort(component);
}
```

We can also deal with reflective object instantiation by intercepting calls to the Java `Class.newInstance` method:

```
after() returning (Object object)
  : call( Object Class.newInstance() ) {
    if (object instanceof Classifier) {
       addDynamicPort((TMinerComponent)object);
    }
}
```

In both cases, we use AspectJ `after() returning` advice in order to obtain a reference to the newly created component. Using this reference, we can employ the infrastructure provided by our data mining framework to attach a dynamic port to such new component; i.e. a hook where we will store the performance measurements our aspect will perform.

For instance, we might be interested in counting how many times our data mining algorithm has to read its training data. We could do it just by using a counter that is reset when we start the classifier training phase (when we call its `build` method) and is incremented each time we access a dataset while we are building the classifier (i.e. when we `open` it):

```
// Reset counter before classifier construction
before():
    call (void Classifier+.build()) {
    // ... reset counter ...
}

// Dataset scan: Increment counter
before(Dataset ds):
    call (void Dataset+.open()) && target(ds) {
    // ... counter++ ...
}
```

We can easily evaluate the I/O performance of any algorithm just by using aspects written as above. Our aspect-oriented performance evaluation approach provides three main benefits with respect to more intrusive techniques we could have used:

- First, using our approach, we do not need to modify the source code of the algorithm under test (in fact, we do not even need to have access to its source code).
- Second, since we do not touch the code of the underlying system, we do not inadvertently introduce subtle bugs in its implementation (nor in the measurement code itself, since it is automatically woven by the AspectJ compiler).
- Third, the experimenter can easily adjust the measurements she wants to obtain, just by tweaking the aspect code to add as many dynamic ports to her components. This would be extremely hard to do if she had to fine-tune the underlying system source code. Moreover, our data mining framework is designed so that measurements attached to a component via its dynamic ports are automatically analyzed by the framework reporting capabilities, requiring no additional effort on her part.

4 EXPERIMENTAL RESULTS

In Data Mining applications, CPU time is not the only relevant factor to be considered when evaluating competing alternatives. A more in-depth analysis of the performance of those alternatives is usually needed to evaluate their scalability, i.e. their ability to cope with ever increasing data volumes. We will now illustrate the kind of results we can easily obtain using our aspect-oriented performance evaluation approach.

We have used thirteen different datasets taken from the UCI Machine Learning Repository (Blake and Merz, 1998) for the construction of different kinds of classifiers:

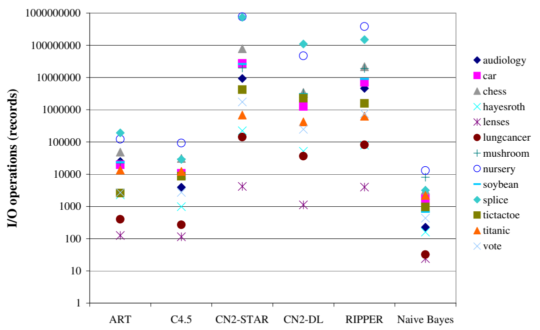

Figure 1: I/O cost for different algorithms in terms of the number of records fetched during the training process.

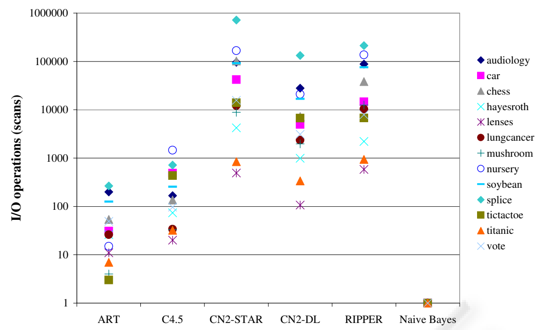

Figure 2: Number of times a dataset is sequentially scanned during classifier construction. It should be noted that the scanned dataset is only a fraction of the whole training dataset once the classification model has been partially built.

- An associative classifier, ART, whose acronym stands for *Association Rule Tree* (Berzal et al., 2004).
- A well-known algorithm for the construction of decision trees: Quinlan's C4.5 (Quinlan, 1993), a derivative from ID3 (Quinlan, 1986).
- Two variants of CN2, a rule learner (Clark and Boswell, 1991) (Fürnkranz and Widmer, 1994).
- A decision list learner called RIPPER (Cohen, 1995), and
- A simple Bayesian classifier, Naive Bayes, to be used as a point of reference since its construction requires just a single sequential scan over the whole training dataset (its I/O cost is optimal).

Figures 1 through 3 illustrate the I/O costs associated to each learning algorithm we have tested.

If we evaluated these different algorithms just by measuring the CPU time required to build the classifiers for the UCI datasets, we could draw the wrong conclusions with respect to which methods might be better suited for real-world databases. The actual number of I/O operations might be a better indicator of real-world performance (see Figure 1).

Associative classifiers, such as ART, internally use efficient association rule mining techniques to build classification models. When working with relatively small datasets, such as those from the UCI, this introduces a significant overhead that could make us think they are not suitable for real-world scenarios.

In fact, ART requires more CPU time than the traditional C4.5 decision tree learner. This is due, among other things, to the fact that ART searches in a larger solution space than C4.5: it looks for multi-variate splits while C4.5 is just a greedy algorithm that looks for the best single variable that can be used to split the training set at each node of the decision tree.

As other decision list and rule inducers, ART constraints the rule size to efficiently bound its search space. However, ART can be an order of magnitude faster than CN2 or RIPPER just because of its search strategy. Where previous rule inducers discover one rule at a time, ART directly looks for sets of rules, thus reducing the number of database scans it must perform to evaluate candidate solutions (see Figure 2). These differences could be dramatically exacerbated when the training dataset does not fit into main memory, a typical situation in data mining scenarios.

ART I/O performance is bound by the resulting classifier complexity, as decision tree learners. Since ART search strategy is designed to lead to compact classifiers, the final number of dataset scans required by ART is even smaller that the number of scans required by our efficient RainForest-like implementation of C4.5 (Gehrke et al., 2000). Our decision tree learner performs two dataset scans at each internal node of the decision tree: one to collect the statistics which are necessary to evaluate alternative splits, another to branch the tree. Decision list and rule inducers, on their hand, perform one dataset scan for each formulated hypothesis, which creates a large I/O bottleneck when datasets do not to fit into main memory.

We have also measured the number of disk pages read by each algorithm for different page sizes, as shown in Figure 3. This quantity can serve as a strong indicator of the algorithms scalability. C4.5 follows a recursive top-down strategy which fragments the training dataset into disjunct subsets, hence the non-linearity is shown in Figure 3. On the other hand, since ART, CN2, and RIPPER are iterative algorithms, the number of disk pages read by any of those algorithms proportionally decrease with the page size.

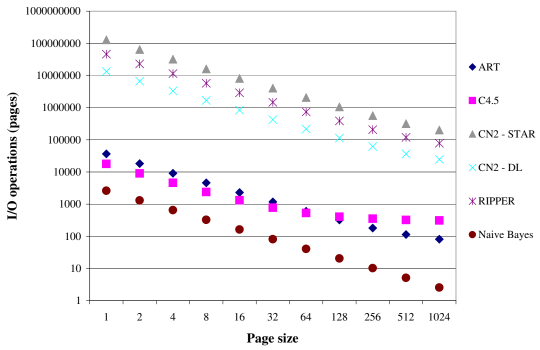

Figure 3: Number of disk pages read by each algorithm for different page sizes. The page size indicates the number of training examples each page contains.

However, while ART I/O cost is bound by the classifier complexity, CN2 and RIPPER performance is determined by the search space they explore.

In summary, ART classifiers exhibit excellent scalability properties, which make them suitable for data mining problems. They provide a well-behaved alternative to decision tree learners where rule and decision list inducers do not work in practice.

5 CONCLUSIONS

In this paper, we have shown how AspectJ, an aspect-oriented extension for the Java programming language, can be used in real-world applications to provide fine-grained performance evaluation and monitoring capabilities. This unintrusive technique avoids the inadvertent insertion of bugs into the system under evaluation. It also frees developers from the burden of introducing scattered code to do their performance evaluation and monitoring work.

Finally, we have described how our proposed approach can be employed for evaluating the I/O cost associated to some data mining techniques. In our experiments, we have witnessed how associative classifiers such as ART possess good scalability properties. In fact, the efficient association rule mining algorithms underlying ART make it orders of magnitude more efficient than alternative rule and decision list inducers, whose I/O requirements heavily constrain their use in real-world situations unless sampling is employed. Moreover, we have confirmed that the additional cost required by ART, when compared to decision tree learners such as C4.5, is reasonable if we take into account the desirable properties of the classification models it helps us obtain, thus making of associative classifiers a viable alternative to standard decision tree learners, the most common classifiers in data mining tools nowadays.

ACKNOWLEDGEMENTS

Work partially supported by research project TIN2006-07262.

REFERENCES

Berzal, F., Cubero, J. C., Sánchez, D., and Serrano, J. M. (2004). ART: A hybrid classification model. *Machine Learning*, 54(1):67–92.

Blake, C. and Merz, C. (1998). UCI repository of machine learning databases. Available at http://www.ics.uci.edu/∼mlearn/MLRepository.html.

Clark, P. and Boswell, R. (1991). Rule induction with CN2: Some recent improvements. In *EWSL*, pages 151–163.

Cohen, W. W. (1995). Fast effective rule induction. In Prieditis, A. and Russell, S., editors, *Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA. Morgan Kaufmann.

Fürnkranz, J. and Widmer, G. (1994). Incremental reduced error pruning. In *ICML*, pages 70–77.

Gehrke, J., Ramakrishnan, R., and Ganti, V. (2000). Rainforest - a framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/3):127–162.

Gradecki, J. D. and Lesiecki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). Getting started with aspectj. *Communications of the ACM*, 44(10):59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP'97: 11th European Conference on Object-Oriented Programming, LNCS 1241*, pages 220–242.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann.