JAM: Java Agents for Meta-Learning over Distributed Databases*

Salvatore Stolfo, Andreas L. Prodromidis†
Shelley Tselepis, Wenke Lee, Dave W. Fan
Department of Computer Science
Columbia University
New York, NY 10027
{sal, andreas, sat, wenke, wfan}@cs.columbia.edu

Philip K. Chan
Computer Science
Florida Institute of Technology
Melbourne, FL 32901
pkc@cs.fit.edu

Abstract

In this paper, we describe the JAM system, a distributed, scalable and portable agent-based data mining system that employs a general approach to scaling data mining applications that we call *meta-learning*. JAM provides a set of learning programs, implemented either as JAVA applets or applications, that compute models over data stored locally at a site. JAM also provides a set of *meta-learning* agents for combining multiple models that were learned (perhaps) at different sites. It employs a special distribution mechanism which allows the migration of the derived models or *classifier agents* to other remote sites. We describe the overall architecture of the JAM system and the specific implementation currently under development at Columbia University. One of JAM's target applications is fraud and intrusion detection in financial information systems. A brief description of this learning task and JAM's applicability are also described. Interested users may download JAM from http://www.cs.columbia.edu/~sal/JAM/PROJECT.

Introduction

One means of acquiring new knowledge from databases is to apply various machine learning algorithms that compute descriptive representations of the data as well as patterns that may be exhibited in the data. The field of machine learning has made substantial progress over the years and a number of algorithms have been popularized and applied to a host of applications in diverse fields. There are numerous algorithms ranging from those based upon stochastic models, to algorithms based upon purely symbolic descriptions like rules and decision trees. Thus, we may simply apply the current generation of learning algorithms to very large databases and wait for a response! However, the question is how long might we wait? Indeed, do the current generation of machine learning algorithms **scale** from tasks common today that include thousands of data items to new learning tasks encompassing as much as two orders of magnitude or more of data that is physically distributed? Furthermore, many existing learning algorithms require all the data to be resident in main memory, which is clearly untenable in many realistic databases. In certain cases, data is inherently distributed and cannot be localized on any one machine (even by a trusted third party) for a variety of practical reasons including physically dispersed mobile platforms like an armada of ships, security and fault tolerant distribution of data and services, competitive (business) reasons, as well as statutory constraints imposed by law. In such situations, it may not be possible, nor feasible, to inspect all of the data at one processing site to compute one primary "global" classifier. We call the problem of learning useful new knowledge from large inherently distributed databases the *scaling problem for machine learning.* We propose to solve the scaling problem by way of a technique we have come to call "meta-learning". Meta-learning seeks to compute a number of independent classifiers by applying learning programs to a collection of independent and inherently distributed databases in parallel. The "base classifiers" computed are then integrated by another learning process. Here meta-learning seeks to compute a "meta-classifier" that integrates in some principled fashion the separately learned classifiers to boost overall predictive accuracy.

In the following pages, we present an overview of meta-learning and we describe JAM (Java Agents for Meta-Learning), a system that employs meta-learning to address large scale distributed applications. The JAM architecture reveals a powerful, portable and extensible network agent-based system that computes meta-classifiers over distributed data. JAM is being engaged in experiments dealing with real-world learning tasks such as solving key problems in fraud and intrusion detection in financial information systems.

Meta-Learning

Meta-learning provides a unifying and scalable solution

*This research is supported by the Intrusion Detection Program (BAA9603) from DARPA (F30602-96-1-0311), NSF (IRI-96-32225 and CDA-96-25374) and NYSSTF (423115-445).

†Supported in part by IBM.

that improves the efficiency and accuracy of inductive learning when applied to large amounts of data in wide area computing networks for a range of different applications.

Our approach to improve efficiency is to execute a number of learning processes (each implemented as a distinct serial program) on a number of data subsets (a *data reduction* technique) in parallel (eg. over a network of separate processing sites) and then to combine the collective results through meta-learning. This approach has two advantages, first it uses the same serial code at multiple sites without the time-consuming process of writing parallel programs and second, the learning process uses small subsets of data that can fit in main memory. The accuracy of the learned concepts by the separate learning process might be lower than that of the serial version applied to the entire data set since a considerable amount of information may not be accessible to each of the independent and separate learning processes. On the other hand, combining these higher level concepts via meta-learning, may achieve accuracy levels, comparable to that reached by the aforementioned serial version applied to the entire data set. Furthermore, this approach may use a variety of different learning algorithms on different computing platforms. Because of the proliferation of networks of workstations and the growing number of new learning algorithms, our approach does not rely on any specific parallel or distributed architecture, nor on any particular algorithm, and thus distributed meta-learning may accommodate new systems and algorithms relatively easily. Our meta-learning approach is intended to be *scalable* as well as *portable* and *extensible.*

In prior publications we introduced a number of meta-learning techniques including arbitration, combining (Chan & Stolfo 1993) and hierarchical tree-structured meta-learning systems. Other publications have reported performance results on standard test problems and data sets with discussions of related techniques, Wolpert's stacking (Wolpert 1992), Breiman's bagging (Breiman *et al.* 1984) and Zhang's combining (Zhang *et al.* 1989) to name a few. Here we describe the JAM system architecture designed to support these and perhaps other approaches to distributed data mining.

The JAM architecture

JAM is architected as an agent based system, a distributed computing construct that is designed as an extension of OS environments. It is a distributed meta-learning system that supports the launching of learning and meta-learning agents to distributed database sites. JAM is implemented as a collection of distributed learning and classification programs linked together through a network of *Datasites.* Each JAM Datasite consists of:

- A local database,
- One or more learning agents, or in other words machine learning programs that may migrate to other sites as JAVA applets, or be locally stored as native applications callable by JAVA applets,
- One or more meta-learning agents,
- A local user configuration file,
- Graphical User Interface and Animation facilities.

The JAM Datasites have been designed to collaborate[1] with each other to exchange classifier agents that are computed by the learning agents.

First, *local learning agents* operate on the *local database* and compute the Datasite's local classifiers. Each Datasite may then import (remote) classifiers from its peer Datasites and combine these with its own local classifier using the *local meta-learning agent.* Finally, once the base and meta-classifiers are computed, the JAM system manages the execution of these modules to classify and label datasets of interest. These actions may take place at all Datasites simultaneously and independently.

The owner of a Datasite administers the local activities via the *local user configuration file.* Through this file, he/she can specify the required and optional local parameters to perform the learning and meta-learning tasks. Such parameters include the names of the databases to be used, the policy to partition these databases into training and testing subsets, the local learning agents to be dispatched, etc. Besides the static [2] specification of the local parameters, the owner of the Datasite can also employ JAM's *graphical user interface* and *animation facilities* to supervise agent exchanges and administer dynamically the meta-learning process. With this graphical interface, the owner may access more information such as accuracy, trends, statistics and logs and compare and analyze results in order to improve performance.

The configuration of the distributed system is maintained by the Configuration File Manager (CFM), a central and independent module responsible for keeping the state of the system up-to-date. The CFM is a server that provides information about the participating Datasites and logs events for future reference and evaluation.

The logical architecture of the JAM meta-learning system is presented in Figure 1. In this example, three JAM Datasites Marmalade, Mango and Strawberry exchange their base classifiers to share their local view of the learning task. The owner of the Datasite controls the learning task by setting the parameters of the user configuration file, i.e. the algorithms to be used, the images to be used by the animation facility, the folding parameters, etc. In this example, the CFM runs

[1] A Datasite may also operate independently without any changes.

[2] Before the beginning of the learning and meta-learning tasks.

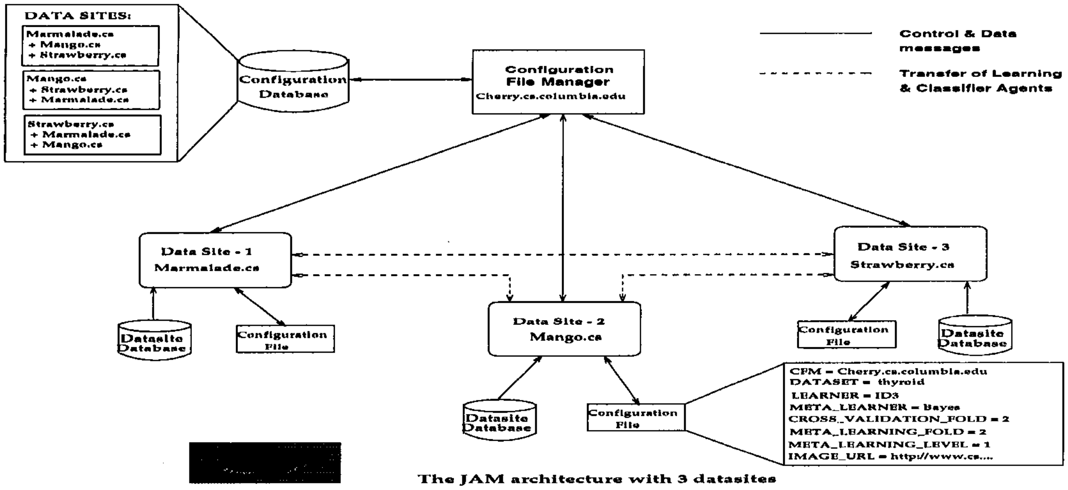

Figure 1: The architecture of the meta-learning system.

on Cherry and each Datasite ends up with three base classifiers (one local plus the two imported classifiers).

We have used JAVA technology to build the infrastructure of the system, to develop the specific *agent operators* that compose and spawn new agents from existing classifier agents, to implement the GUI, the animation facilities and most of the machine learning algorithms. The platform-independence of JAVA technology makes it easy to port JAM and delegate its agents to any participating site. The only parts that were imported in their native (C++) form and are not yet platform independent were some of the machine learning programs; and this was done for faster prototype development and proof of concept.

Configuration File Manager

The CFM assumes a role equivalent to that of a name server of a network system. The CFM provides registration services to all Datasites that wish to become members and participate in the distributed meta-learning activity. When the CFM receives a JOIN request from a new Datasite, it verifies both the validity of the request and the identity of the Datasite. Upon success, it acknowledges the request and registers the Datasite as active. Similarly, the CFM can receive and verify the DEPARTURE request; it notes the requestor Datasite as inactive and removes it from its list of members. The CFM, maintains the list of active member Datasites to establish contact and cooperation between peer Datasites. Apart from that, the CFM keeps information regarding the groups that are formed (which Datasites collaborate with which Datasites), logs the events and displays the status of the system. Through the CFM, the JAM system administrator may screen the Datasites that participate.

Datasites

Unlike CFM which provides a passive configuration maintenance function, the Datasites are the active components of the meta-learning system. They manage the local databases, obtain remote classifiers, build the local base and meta classifiers and interact with the JAM user. Datasites are implemented as multi-threaded Java programs with a special GUI.

Upon initialization, a Datasite starts up the GUI, through which it can accept input and display status and results, registers with the CFM, instantiates the local learning engine/agent[3] and creates a server socket for listening for connections[4] from the peer Datasites. Then, it waits for the next event to occur, either a command issued by the owner via the GUI, or a message from a peer Datasite via the open socket.

In both cases, the Datasite verifies that the input is valid and can be serviced. Once this is established, the Datasite allocates a separate thread and performs the required task. This task can be any of JAM's functions: computing a local classifier, starting the meta-learning process, sending local classifiers to peer Datasites or requesting remote classifiers from them, reporting the current status, or presenting computed results.

Figure 2 presents a snapshot of the JAM system during the meta-learning phase. In this example three Datasites, Marmalade, Strawberry and Mango (see the group panel of the figure) collaborate in order to share and improve their knowledge in diagnosing hypothyroidism. The snapshot taken is from "Marmalade's point of view". Initially, Marmalade consults the Dat-

[3]The Datasite consults the local Datasite configuration file (maintained by the owner of the Datasite) to obtain information regarding the central CFM and the types of the available machine learning agents.

[4]For each connection, the Datasite spawns a separate thread.

asite configuration file where the owner of the Datasite sets the parameters. In this case, the dataset is a medical database with records (Merz & Murphy 1996), noted by thyroid in the Data Set panel. Other parameters include the host of the CFM, the Cross-Validation Fold, the Meta-Learning Fold, the Meta-Learning Level, the names of the local learning agent and the local meta-learning agent, etc. (Refer to (Chan 1996) for more information on the meaning and use of these parameters.) Notice that Marmalade has established that Strawberry and Mango are its peer Datasites, having acquired this information from the CFM.

Then, Marmalade partitions the thyroid database (noted as thyroid.1.bld and thyroid.2.bld in the Data Set panel) for the 2-Cross-Validation Fold and computes the local classifier, noted by Marmalade.1 (here by calling the ID3 (Quinlan 1986) learning agent). Next, Marmalade imports the remote classifiers, noted by Strawberry.1 and Mango.1 and begins the meta-learning process. Marmalade employs this meta-classifier to predict the classes of input data items (in this case unlabelled medical records). Figure 2 displays a snapshot of the system during the animated meta-learning process where JAM's GUI moves icons within the panel displaying the construction of a new meta-classifier.

Classifier Visualization

JAM provides graph drawing tools to help users understand the learned knowledge (Fayyad, Piatetsky-Shapiro, & Smyth 1996). There are many kinds of classifiers, e.g., a decision tree by ID3, that can be represented as graphs. In JAM we have employed major components of *JavaDot* (Lee & Barghouti 1997), an extensible visualization system, to display the classifier and allows the user to analyze the graph. Since each machine learning algorithm has its own format to represent the learned classifier, JAM uses an algorithm-specific translator to read the classifier and generate a *JavaDot* graph representation.

Figure 2 shows the JAM classifier visualization panel with a decision tree, where the leaf nodes represent classes (decisions), the non-leaf nodes represent the attributes under test, and the edges represent the attribute values. The user can select the **Attributes** command from the **Object** pull-down menu to see any additional information about a node or an edge. In the figure, the **Attributes** window shows the classifying information of the highlighted leaf node[5]. It is difficult to view clearly a very large graph (that has a large number of nodes and edges) due to the limited window size. The classifier visualization panel provides commands for the user to traverse and analyze parts of the graph: the user can select a node and use the **Top** command from the **Graph** menu to make the subgraph starting from the selected node be the entire graph in display; use the **Parent** command to view the enclosing graph; and use the **Root** command to see the entire original graph.

Some machine learning algorithms generate concise and very readable textual outputs, e.g., the rule sets from Ripper (Cohen 1995). It is thus counter-intuitive to translate the text to graph form for display purposes. In such cases, JAM simply pretty formats the text output and displays it in the classifier visualization panel.

Animation

For demonstration and didactic purposes, the meta-learning component of the JAM graphical user interface contains a collection of animation panels which visually illustrate the stages of meta-learning in parallel with execution. When animation is enabled, a transition into a new stage of computation or analysis triggers the start of the animation sequence corresponding to the underlying activity. The animation loops continuously until the given activity ceases.

The JAM program gives the user the option of manually initiating each distinct meta-learning stage (by clicking a **Next** button), or sending the process into automatic execution (by clicking a **Continue** button). The manual run option provides a temporary program halt. For "hands free" operation of JAM, the user can start the program with animation disabled and execution set to automatic transition to the next stage in the process.

Agents

JAM's extensible plug-and-play architecture allows snap-in learning agents. The learning and meta-learning agents are designed as objects. JAM provides the definition of the parent agent class and every instance agent (i.e. a program that implements any of your favorite learning algorithms ID3, Ripper, Cart (Breiman *et al.* 1984), Bayes (Duda & Hart 1973), Wpebls (Cost & Salzberg 1993), CN2 (Clark & Niblett 1989), etc.) are then defined as a subclass of this parent class. Among other definitions which are inherited by all agent subclasses, the parent agent class provides a very simple and minimal interface that all subclasses have to comply to. As long as a learning or meta-learning agent conforms to this interface, it can be introduced and used immediately in the JAM system even during execution. To be more specific, a JAM agent needs to have the following methods implemented:

1. A *constructor method* with no arguments. JAM can then instantiate the agent, provided it knows its name (which can be supplied by the owner of the Datasite through either the local user configuration file or the GUI).
2. An *initialize() method.* In most of the cases, if not all, the agent subclasses inherit this method from

[5]Thus visually, we see that for a test data item, if its "p-2" value is 3 and its "p-14" value is 2, then it belongs to class "0" with .889 probability.

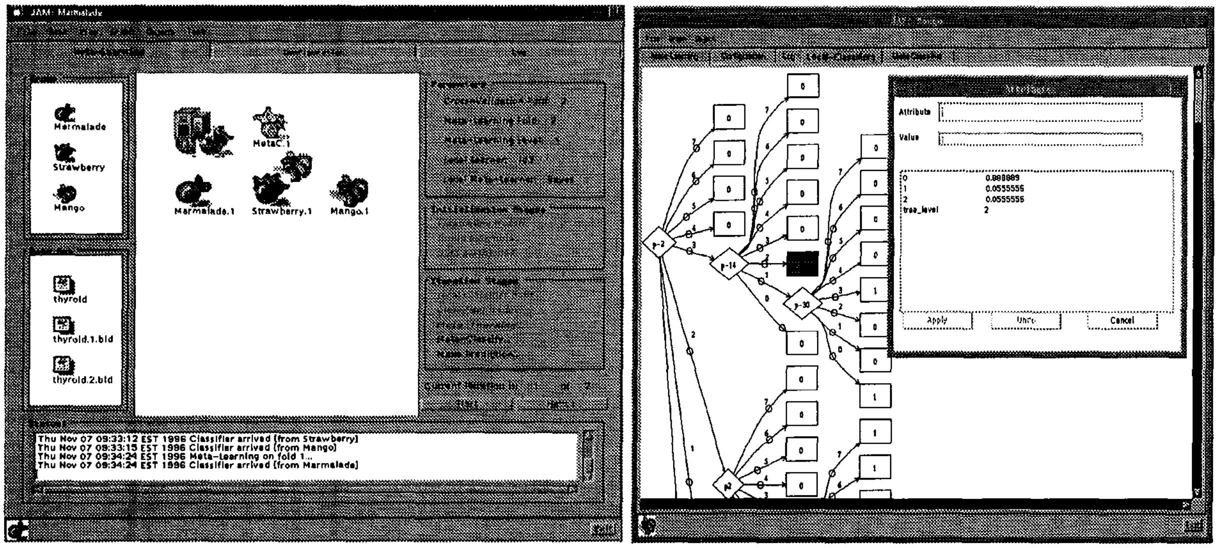

Figure 2: Two different snapshots of the JAM system in action. Left: Marmalade is building the meta classifier (meta learning stage). Right: An ID3 tree-structured classifier is being displayed in the Classifier Visualization Panel.

the parent agent class. Through this method, JAM can supply the necessary arguments to the agent. Arguments include the names of the training and test datasets, the name of the dictionary file, and the filename of the output classifier.

3. A *buildClassifier() method.* JAM calls this method to trigger the agent to learn (or meta-learn) from the training dataset.

4. A *getClassifier()* and *getCopyOfClassifier() methods.* These methods are used by JAM to obtain the newly built classifiers which are then encapsulated and can be "snapped-in" at any participating Datasite! Hence, remote agent dispatch is easily accomplished.

The class hierarchy (only methods are shown) for five different learning agents is presented in Figure 3. ID3, Bayes, Wpebls, CART and Ripper inherit the methods initialize() and getClassifier() from their parent learning agent class. The Meta-Learning, Classifier and Meta-Classifier classes are defined in similar hierarchies.

JAM is designed and implemented independently of the machine learning programs of interest. As long as a machine learning program is defined and encapsulated as an object conforming to the minimal interface requirements (most existing algorithms have similar interfaces already) it can be imported and used directly. In the latest version of JAM for example, ID3 and CART are full JAVA agents, whereas Bayes, Wpebls and Ripper are stored locally as native applications. This plug-and-play characteristic makes JAM truly powerful and extensible data mining facility.

Fraud and Intrusion Detection

A secured and trusted interbanking network for electronic commerce requires high speed verification and authentication mechanisms that allow legitimate users easy access to conduct their business, while thwarting fraudulent transaction attempts by others. Fraudulent electronic transactions are a significant problem, one that will grow in importance as the number of access points in the nation's financial information system grows.

Financial institutions today typically develop custom fraud detection systems targeted to their own asset bases. Recently though, banks have come to realize that a unified, global approach is required, involving the periodic sharing with each other of information about attacks.

We have proposed another wall to protect the nation's financial systems from threats. This new wall of protection consists of pattern-directed inference systems using models of anomalous or errant transaction behaviors to forewarn of impending threats. This approach requires analysis of large and inherently distributed databases of information about transaction behaviors to produce models of "probably fraudulent" transactions. We use JAM to compute these models.

The key difficulties in this approach are: financial companies don't share their data for a number of (competitive and legal) reasons; the databases that companies maintain on transaction behavior are huge and growing rapidly; real-time analysis is highly desirable to update models when new events are detected and easy distribution of models in a networked environment is essential to maintain up to date detection capability.

JAM is used to compute local fraud detection agents

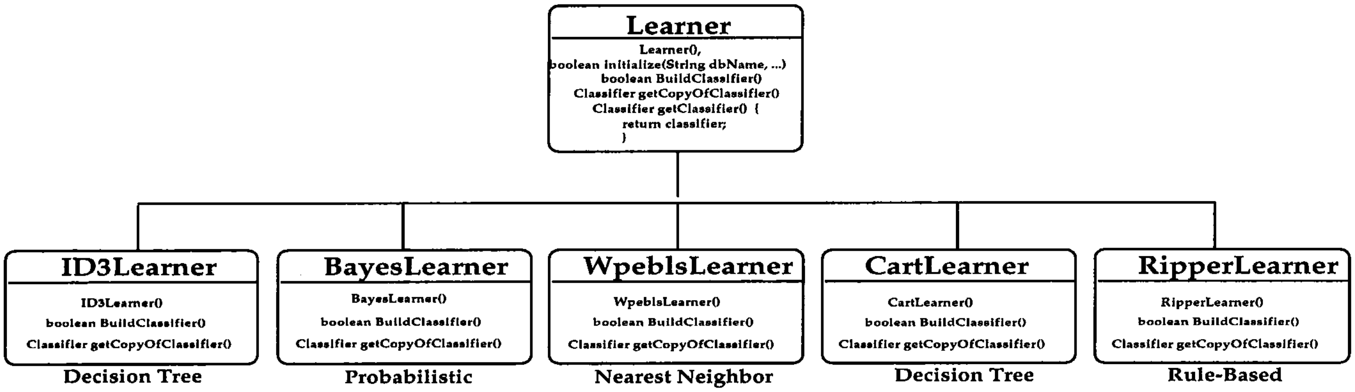

Figure 3: The class hierarchy of learning agents.

that learn how to detect fraud and provide intrusion detection services within a single corporate information system, and an integrated meta-learning system that combines the collective knowledge acquired by individual local agents. Once derived local classifier agents or models are produced at some Datasite(s), two or more such agents may be composed into a new classifier agent by JAM's meta-learning agents.

JAM allows financial institutions to share their models of fraudulent transactions by exchanging classifier agents in a secured agent infrastructure. But they will not need to disclose their proprietary data. In this way their competitive and legal restrictions can be met, but they can still share information. The meta-learned system can be constructed both globally and locally. In the latter guise, each corporate entity benefits from the collective knowledge by using its privately available data to locally learn a meta-classifier agent from the shared models. The meta-classifiers then act as sentries forewarning of possibly fraudulent transactions and threats by inspecting, classifying and labeling each incoming transaction.

How Local Detection Components Work

Consider the generic problem of detecting fraudulent transactions, in which we are not concerned with global coordinated attacks. We posit there are two good candidate approaches.

Approach 1:

i) Each bank $i, 1 < i <= N$, uses some learning algorithm or other on one or more of their databases, DB_i, to produce a classifier f_i. In the simplest version, all the f_i have the same input feature space. Note that each f_i is just a mapping from the features space of transaction, x, to a bimodal fraud label.

ii) All the f_i are sent to a central repository, where there is a new "global" training database, call it $DB*$.

iii) $DB*$ is used in conjunction with some learning algorithm in order to learn the mapping from $< f_1(x), f_2(x), ...f_N(x), x >$ to a fraud label (or probability of fraud). That mapping, or meta-classifier, is $f*$.

iv) $f*$ is sent to all the individual banks to use as a data filter to mark and label incoming transactions with a fraud label.

Approach 2:

i) Same as approach 1.

ii) Every bank i sends to every other bank its f_i. So at the end of this stage, each bank has a copy of all N classifiers, $f_1, ...f_N$.

iii) Each bank i had held separate some data, call it T_i, from the DB_i used to create f_i. Each bank then uses T_i and the set of all the f's to learn the mapping from $< f_1(x), f_2(x), ...f_N(x), x >$ to a fraud label (or probability of fraud, as the case may be). (This is exactly as in step (iii) of approach 1, except the data set used for combining is T_i (a different one for each bank) rather than $DB*$.) Each such mapping is F_i.

iv) Each bank uses its F_i as in approach 1.

Credit Card Fraud Transaction Data

Here we provide a general view of the data schema for the labelled transaction data sets compiled by a bank and used by our system. For purposes of our research and development activity, several data sets are being acquired from several banks, each providing .5 million records spanning one year, sampling on average 42,000 per month, from Nov. 1995 to Oct. 1996.

The schema of the database was developed over years of experience and continuous analysis by bank personnel to capture important information for fraud detection. The general schema of this data is provided in such a way that important confidential and proprietary information is not disclosed here. (After all we seek not to teach "wanabe thieves" important lessons on how to hone their skills.) The records have a fixed length of 137 bytes each and about 30 numeric attributes including the binary classification (fraud/legitimate transaction). Some of the fields are arithmetic and the rest categorical, i.e. numbers were used to represent a few discrete categories.

In this section, we describe the setting of our experiments. In particular, we split the original data set provided by one bank into random partitions and we distributed them across the different sites of the JAM network. Then we computed the accuracy from each model obtained at each such partition.

To be more specific, we sampled 84,000 records from the total of 500,000 records of the data set we used in our experiments, and kept them for the *Validation* and *Test* sets to evaluate the accuracy of the resultant distributed models. The learning task is to identify patterns in the 30 attribute fields that can characterize the fraudulent class label.

Let's assume, without loss of generality, that we apply the ID3 learning process to two sites of data (say sites 1 and 2), while two instances of Ripper are applied elsewhere (say at sites 3 and 4), all being initiated as *Java agents*. The result of these four local computations are four separate classifiers, $C_{ID3-i}(), i = 1, 2,$ and $C_{Ripper-j}(), j = 3, 4$ that are each invocable as agents at arbitrary sites of credit card transaction data.

A sample (and sanitized) Ripper rule-based classifier learned from the credit card data set is depicted in Figure 4, a relatively small set of rules that is easily communicated among distributed sites as needed.[6] To extract fraud data from a distinct fifth site of data, or any other site, using say, $C_{Ripper-3}()$ the code implementing this classifier would be transmitted to the fifth site and invoked remotely to extract data. This can be accomplished for example using a query of the form:

Select X. From Credit-card-data*
Where $C_{Ripper-3}(X.fraud - label) = 1.$

Naturally, the select expression rendered here in SQL in this example can instead be implemented directly as a data filter applied against incoming transactions at a server site in a fraud detection system.

The end result of this query is a stream of data accessed from some remote source based entirely upon the classifications learned at site 3. Notice that requesting transactions classified as "not fraud" would result in no information being returned at all (rather than streaming all data back to the end-user for their own sifting or filtering operation). Likewise, in a fraud detection system, alarms would be initiated only for those incoming transactions that have been selectively labelled as potentially fraudulent.

Next, a new classifier, say M can be computed by combining the collective knowledge of the 4 classifiers using for example the ID3 meta-learning algorithm. M is trained over *meta-level training data*, i.e. the class predictions from four base classifiers, as well as the raw training data that generated those predictions[7]. The meta-training data is a small fraction of the total amount of distributed training data[8]. In order for M to generate its final class predictions, it requires the classifications generated by $C_{ID3-1}()$, $C_{ID3-2}()$, $C_{Ripper-3}()$ and $C_{Ripper-4}()$. The result is a tree structured meta-classifier depicted in Figure 4.

In this figure, the descendant nodes of the decision tree are indented while the leaves specify the final classifications (fraud label 0 or 1). A (logic-based) rule equivalent of the first branch at the top of the ID3 Decision tree is:

"If (X.Prediction{site − 1} = 1) and
(X.Prediction{site − 2} = 1)
then the transaction is fraudulent i.e.
X.Fraud-label = 1."

Similarly, with M, we may access credit card fraud data at any site in the same fashion as $C_{Ripper-3}$ used over site 5.

Our experiments for fraud and intrusion detection are ongoing. In one series of measurements, we trained the base classifiers and meta-classifiers over a sample data set with 50% fraudulent and 50% non-fraudulent transactions. Then we tested their accuracy against a different and unseen sample set of data with 20%/80% distribution. In summary, Ripper and Cart were the best base classifiers, and Bayes, the best and most stable meta-classifier. Ripper and Cart were each able to catch 80% of the fraudulent transactions (True Positive or TP) but also misclassify 16% of the legitimate transactions (False Positive or FP) while Bayes exhibited 80% TP and 13% FP in one setting and 80% TP and 19% FP in another. In the first setting, Bayes combined the three base classifiers with the least correlated error and in the second it combined the four most accurate base classifiers. The experiments, settings, rationale and results have been reported in detail in a companion paper (Stolfo *et al.* 1997) also available from http://www.cs.columbia.edu/~sal/JAM/PROJECT.

Conclusions

We believe the concepts embodied by the term meta-learning provide an important step in developing systems that learn from massive databases and that scale. A deployed and secured meta-learning system will provide the means of using large numbers of low-cost networked computers who collectively learn from massive databases useful and new knowledge, that would otherwise be prohibitively expensive to achieve. We believe meta-learning systems deployed as intelligent agents will be an important contributing technology to deploy intrusion detection facilities in global-scale, integrated information systems.

[6]The specific confidential attribute names are not revealed here.

[7]This meta-learning strategy is denoted *class-combiner* as defined in (Chan & Stolfo 1995a; 1995b).

[8]The section detailing the meta-learning strategies in (Chan & Stolfo 1995b) describes the various empirically determined bounds placed on the meta-training data sets while still producing accurate meta-classifiers.

```
Fraud :-                 Fraud :-
      a >= 148,                a >= 774.
      b >= 695.

Fraud :-                 Fraud :-
      c <= 12,                 e = '8',
      d <= 4814.               f = '3',
                               c <= 200,
                               a >= 4.
NonFraud :- true
```

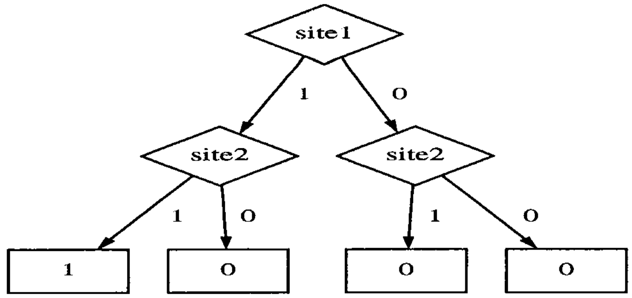

Figure 4: Left: This sample rule-based model, covers 1365 non-fraudulent and 290 fraudulent credit card transactions. Right: A portion of the ID3 decision tree meta-classifier, learned from the predictions of the four base classifiers. In this portion, only the classifiers from site-1 and site-2 are displayed

In this paper we described the JAM architecture, a distributed, scalable, extensible and portable agent-based system that supports the launching of learning and meta-learning agents to distributed database sites and build upon existing agent infrastructure available over the internet today. JAM can integrate distributed knowledge and boost overall predictive accuracy of a number of independently learned classifiers through meta-learning agents. We have engaged JAM in a real, practical and important problem. In collaboration with the FSTC we have populated these database sites with records of credit card transactions, provided by different banks, in an attempt to detect and prevent fraud by combining learned patterns and behaviors from independent sources.

Acknowledgements

We wish to thank David Wolpert, formerly of TXN and presently at IBM Almaden, Hank Vacarro of TXN, Shaula Yemini of Smarts, Inc. and Yechiam Yemini for many useful and insightful discussions. We also wish to thank Dan Schutzer of Citicorp, Adam Banckenroth of Chase Bank, Tom French of First Union Bank and John Doggett of Bank of Boston, all executive members of the FSTC, for their support of this work.

References

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification and Regression Trees.* Belmont, CA: Wadsworth.

Chan, P., and Stolfo, S. 1993. Toward parallel and distributed learning by meta-learning. In *Working Notes AAAI Work. Knowledge Discovery in Databases*, 227–240.

Chan, P., and Stolfo, S. 1995a. A comparative evaluation of voting and meta-learning on partitioned data. In *Proc. Twelfth Intl. Conf. Machine Learning*, 90–98.

Chan, P., and Stolfo, S. 1995b. Learning arbiter and combiner trees from partitioned data for scaling machine learning. In *Proc. Intl. Conf. Knowledge Discovery and Data Mining*, 39–44.

Chan, P. 1996. *An Extensible Meta-Learning Approach for Scalable and Accurate Inductive Learning.* Ph.D. Dissertation, Department of Computer Science, Columbia University, New York, NY.

Clark, P., and Niblett, T. 1989. The CN2 induction algorithm. *Machine Learning* 3:261–285.

Cohen, W. W. 1995. Fast effective rule induction. In *Proc. Twelfth International Conference.* Morgan Kaufmann.

Cost, S., and Salzberg, S. 1993. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning* 10:57–78.

Duda, R., and Hart, P. 1973. *Pattern classification and scene analysis.* New York, NY: Wiley.

Fayyad, U.; Piatetsky-Shapiro, G.; and Smyth, P. 1996. The kdd process for extracting useful knowledge from data. *Communications of the ACM* 39(11):27–34.

Lee, W., and Barghouti, N. S. 1997. Javadot: An extensible visualization environment. Technical Report CUCS-02-97, Department of Computer Science, Columbia University, New York, NY.

Merz, C., and Murphy, P. 1996. UCI repository of machine learning databases [http://www.ics.uci.edu/~mlearn/mlrepository.html]. Dept. of Info. and Computer Sci., Univ. of California, Irvine, CA.

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

Stolfo, S.; Fan, W.; Lee, W.; Prodromidis, A.; and Chan, P. 1997. Credit card fraud detection using meta-learning: Issues and initial results. AAAI Workshop on AI Approaches to Fraud Detection and Risk Management.

Wolpert, D. 1992. Stacked generalization. *Neural Networks* 5:241–259.

Zhang, X.; Mckenna, M.; Mesirov, J.; and Waltz, D. 1989. An efficient implementation of the backpropagation algorithm on the connection machine CM-2. Technical Report RL89-1, Thinking Machines Corp.