Distributed Data Mining: The JAM System Architecture*

Andreas L. Prodromidis Salvatore J. Stolfo Shelley Tselepis
Terrance Truta Jeffrey Sherwin David Kalina

Columbia University, Department of Computer Science
1214 Amsterdam Ave., New York, NY 10027
{andreas,sal,sat,ttruta,jcs52,dak40}@cs.columbia.edu

Abstract

This paper describes the system architecture of JAM (Java Agents for Meta-learning), a distributed data mining system that scales up to large and physically separated data sets. An early version of the JAM system was described in [53]. Since then, JAM has evolved both architecturally and functionally and here we present the final design and implementation details of this system architecture.

JAM is an extensible agent-based distributed data mining system that supports the remote dispatch and exchange of agents among participating data sites and employs meta-learning techniques to combine the multiple models that are learned. 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 and summary results are also discussed.

Keywords: Distributed Data Mining, System Architecture, Intelligent Agents, Meta-Learning, Fraud Detection.

Email address of contact author: andreas@cs.columbia.edu

*This research was 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).

1 Introduction

The main objective of this work was the design and implementation of a system that supports the mining of information from distributed data sets. In a relational database context a *data mining task* is to explain and predict the value of some attribute given a collection of tuples with known attribute values. One means of performing such a task is to employ various machine learning algorithms. In the centralized approach, an existing relation, drawn from some domain, is thus treated as training data for a learning algorithm that computes a descriptive model, or a *classifier*. This classifier can later be used to predict (for a variety of strategic and tactical purposes) the value of a desired or target attribute for some record whose desired attribute value is unknown.

One of the main challenges in machine learning and data mining, however, is the development of inductive learning techniques that scale up to large and possibly physically distributed data sets. Some approaches that have already been described in the literature include IBM's SLIQ [28] and SPRINT [52] decision tree algorithms and Provost and Hennessy's rule-based DRL algorithm [47] for multi-processor learning. Our approach to this problem is to employ the algorithm-independent *meta-learning* technique.

Meta-learning seeks to compute a number of independent classifiers by applying learning programs to a collection of 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.

Several methods for integrating ensembles of models have been studied, including techniques that combine the set of models in some linear fashion [1, 3, 4, 17, 24, 25, 27, 33, 35, 51, 54], e.g., majority or weighted voting, bagging, etc., techniques that employ referee functions to arbitrate among the predictions generated by the classifiers [7, 20, 22, 50, 21, 23, 34], e.g., arbiters, mixture of experts, etc., methods that rely on principal components analysis [29, 31], e.g., SCANN, or methods that apply inductive learning techniques to learn the behavior and properties of the candidate classifiers [55, 7], e.g., stacking. Our distributed system is designed to support any of these meta-learning methods. However, in this study we report results obtained using three representative techniques: voting, stacking and SCANN.

With meta-learning to provide the means for combining information across separate data sources (by integrating individually computed classifiers), we developed a system called JAM.

JAM facilitates the sharing of information among multiple sites without the need of exchanging or directly accessing remote data. The name JAM stands for Java Agents for Meta-learning; Agents implemented in Java [2] generate and transport the trained classifiers while Meta-learning underlines the key component of the system for combining these classifiers. The system improves the efficiency of inductive learning when applied to large amounts of data in wide area computing networks for a range of different applications.

We applied JAM to the real-world data mining task of modeling and detecting credit card fraud with notable success.[1] Inductive learning agents are used to compute detectors of anomalous or errant behavior over inherently distributed data sets and meta-learning methods integrate their collective knowledge into higher level classification models or meta-classifiers. By supporting the exchange of models or classifier agents among data sites, our approach facilitates the cooperation between financial organizations and provides unified and cross-institution protection mechanisms against fraudulent transactions.

The remainder of this paper is organized as follows. In Section 2 we describe the architecture of JAM and the implementation aspects of the system. The description includes details on the distributed protocols adopted and the scalability, portability, and extensibility properties of the system. Sections 3 and 4 focus on the necessary changes of JAM as we add support for two techniques, pruning and bridging. We developed pruning and bridging to address two shortcomings of meta-learning, namely, the increased demand for run-time system resources, and the inability to combine multiple models computed over distributed data sets with different schemas. Section 5 outlines the data mining task of detecting fraudulent use of credit cards and summarizes the experiments and performance results. Finally, Section 6 concludes this paper and discusses future research directions.

2 JAM System Architecture

The JAM system is designed around the idea of meta-learning to benefit from its inherent parallelism and distributed nature. Recall that meta-learning improves efficiency by executing in parallel the same or different serial learning algorithms over different subsets of the training data set. An early version of the architecture of JAM appeared in [53]. Here we describe the final design and implementation details of this system architecture.

[1]The main purpose of this paper is to describe the system architecture of JAM. Additional multiple-model experiments with results on other tasks and data sets can be found in [15, 14].

JAM is architected as a distributed computing construct developed on top of OS environments. It can be viewed as a coarse-grain parallel application, with each constituent process running on a separate database site. JAM is an agent based system that supports the launching of learning, classifier and meta-learning agents to distributed database sites. Under normal operation, each JAM site (i.e., the database site) functions autonomously and (occasionally) exchanges classifiers with the rest. JAM is implemented as a collection of distributed learning and classification programs linked together through a network of JAM sites. Each JAM site consists of:

- one or more local databases,
- one or more learning agents, or in other words machine learning programs that may migrate to other sites as Java objects, or be locally stored as native programs callable by Java agents,
- one or more meta-learning agents, or programs capable of combining a collection of classifier agents,
- a repository of locally computed and imported base- and meta-classifier agents,
- a local configuration file and,
- a Graphical User Interface and Animation facilities or a Text-based User Interface.

The JAM sites have been designed to collaborate[2] with each other to exchange classifier agents computed by learning agents. When JAM is initiated, *local or imported learning agents* execute on the *local database* to compute the local classifiers. Each JAM site may then import (remote) classifiers from its peer JAM sites and combine these with its own local classifiers using the *local meta-learning agent*. Finally, once the base and meta-classifiers are generated, the JAM system manages the execution of these modules to classify new unlabeled data sets. Each JAM site stores its base- and meta-classifiers in its *classifier repository*, a special database for classifiers. These actions take place at all JAM sites simultaneously and independently.

The owner (user) of a JAM site administers the local activities via the *local configuration file*. Through this file, he/she can specify the required and optional local parameters to

[2]A JAM site may also operate independently without any changes.

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[3] specification of the local parameters, the owner of a JAM site can also use JAM's *graphical user interface* and *animation facilities* to supervise agent exchanges and administer dynamically the meta-learning process. Through this graphical interface, the owner can access more information such as accuracy, trends, statistics and logs and compare and analyze results in order to improve performance. Alternatively, the owner has the option of using a command-driven (text-based) interface to manage the JAM site.

The configuration of the distributed system is maintained by a logically independent module, the Configuration Manager (hereinafter CM). The CM can be regarded as the equivalent of a domain name server of a system. It is responsible for providing information about the participating JAM sites and for keeping the state of the system up-to-date. The logical architecture of the JAM system is presented in Figure 1. Notice, the CM runs on Marmalade and three JAM sites Mango Bank, Orange Bank and Cherry Bank exchange their base classifiers to share their local view of the learning task. Mango, for example, has acquired four base classifiers (two are computed locally, one was imported from Orange and one from Cherry) that may be combined in a meta-classifier. The owner of the JAM site controls the learning task by setting the parameters of the configuration file, i.e., the algorithms to be used, the images to be used by the animation facility, the cross validation and folding parameters, etc.

2.1 Configuration Manager

The CM provides registration services to all JAM sites that wish to become members and participate in the distributed meta-learning activity. When the CM receives an ACTIVE request from a new JAM site, it verifies both the validity of the request and the identity of the JAM site. Upon success, it acknowledges the request and registers the JAM site as active. Similarly, the CM can receive and verify an INACTIVE request; it notes the requestor JAM site as inactive and removes it from its list of members. The CM, maintains the list of active member JAM sites that seek to establish contact and collaborate with peer JAM sites. By issuing a special QUERY request to the CM, registered JAM sites can obtain this list

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

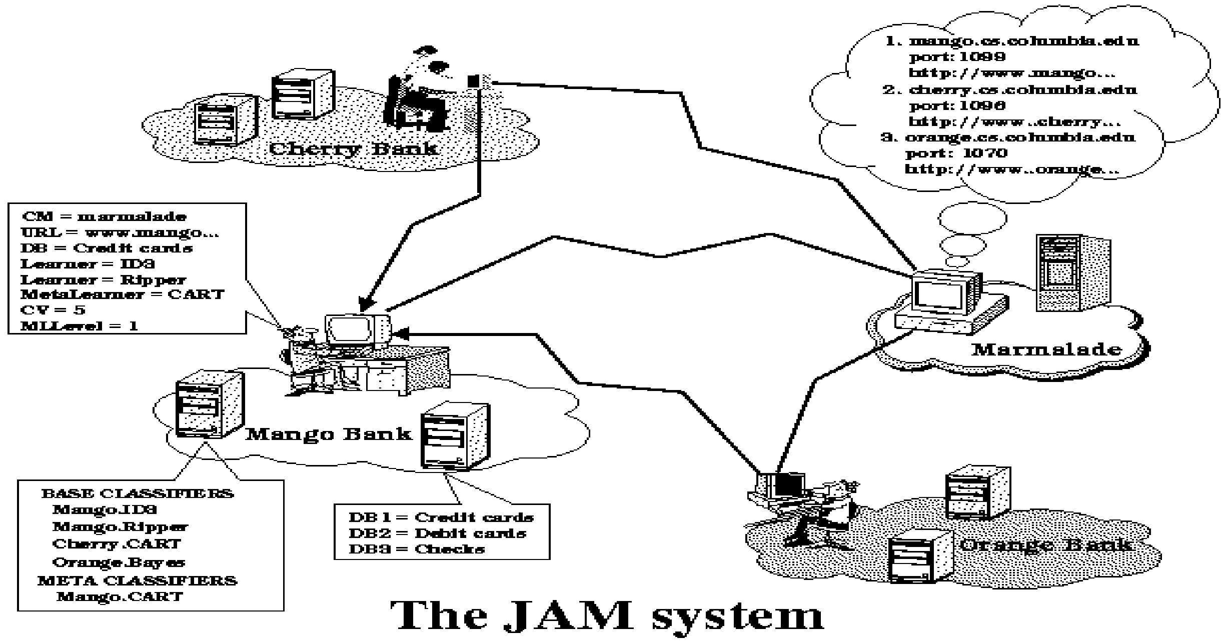

Figure 1: The architecture of the meta-learning system.

of active members. Apart from ACTIVE, INACTIVE and QUERY, the CM also supports UPDATE requests that allow JAM sites to change their entries in the list of active members. The complete set of the different type of messages supported by the CM are described in Table 1. In addition, the table includes the acknowledgment messages from the CM to the client JAM site requests.

Table 1: Types of messages supported by the CM

Message Header	Message body	Direction	Description
JAM_ACTIVE	Identity, contact information	incoming	Join the group
JAM_ACK_ACTIVE		outgoing	Join acknowledged
JAM_INACTIVE	Identity	incoming	Departure notification
JAM_ACK_INACTIVE		outgoing	Departure acknowledged
JAM_QUERY	Identity	incoming	Request list of sites
JAM_ACK_QUERY	List of JAM sites	outgoing	Return list
JAM_UPDATE	Identity, new information	incoming	Change JAM sits entry
JAM_ACK_UPDATE		outgoing	Update successful

Using a single CM within JAM is not a limiting factor to the scalability of the system. The bulk of the communication between the CM and the JAM sites occurs during the initialization stage of each site. On average, a JAM site is expected to issue UPDATE and QUERY requests fairly infrequently. Moreover, the overhead incurred due to the transfer of information between the sites and the CM is minimal. (Each entry in the list of active JAM sites accounts for only a few bytes.)

The CM is a logical unit. Hence, even if the number of participating data sites increases, the CM can be decomposed and distributed across several hosts in a straightforward manner. The architecture follows that of the name servers in a network environment. A single server is responsible for a limited number of network devices; if the address of a device is unknown to a name server, that server will contact another server in an attempt to resolve the name.

2.2 JAM Site Architecture

Unlike the CM, which provides a passive configuration maintenance function, the JAM sites 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. JAM sites are implemented as multi-threaded Java programs with a special GUI.

Each JAM site is organized as a layered collection of software components shown in Figure 2. In general, the system can be decomposed into four separate subsystems, the User Interface, the JAM Engine and the Client and Server subsystems. The User Interface (upper tier) materializes the front end of the system, through which the owner can define the data mining task and drive the JAM Engine. The JAM Engine constitutes the heart of each JAM site by managing and evaluating the local agents, by preparing/processing the local data sets and by interacting with the Database Management System (DBMS), if one exists. Finally, the Client and Server subsystems compose the network component of JAM and are responsible for interfacing with other JAM sites to coordinate the transport of their agents. Each site is developed on top of the JVM (Java Virtual Machine), with the possible exception of some agents that may be used in a native form and/or depend on an underlying DBMS. A Java agent, for instance, may be able to access a DBMS through JDBC (Java Database Connectivity). The RMI registry component displayed in Figure 2 corresponds to an independent Java process that is used indirectly by the JAM server component and is described later.

2.2.1 User Interface and JAM Engine Components

Upon initialization, a JAM site undertakes a series of tasks; it starts up the GUI on a separate thread; it registers with the CM; it instantiates the JAM Client and finally spawns the JAM Server thread for listening for requests/connections from the outside. The necessary

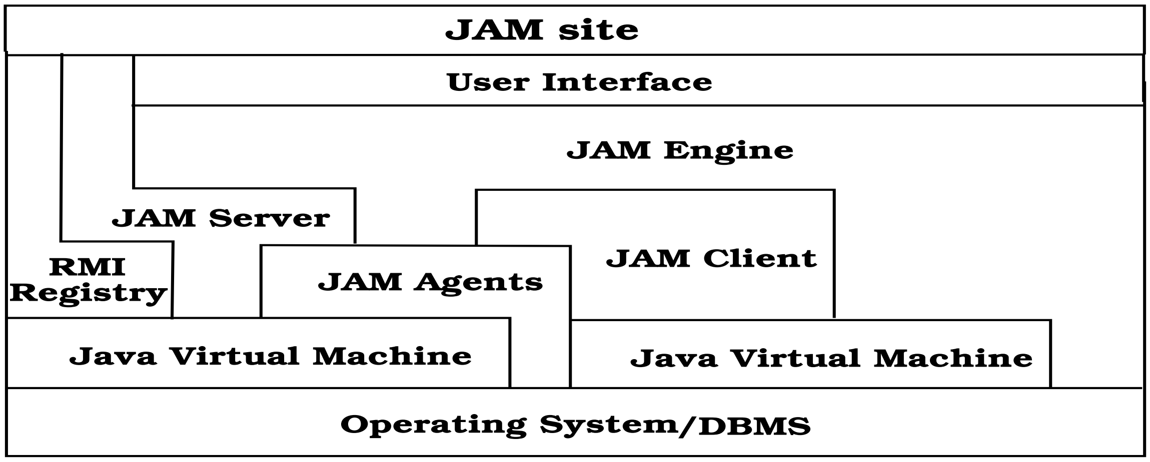

Figure 2: JAM site layered model.

information to carry out these tasks (e.g., the host name and the port number of the server socket of the CM, required URLs, the path names to local agents and data sets, etc.) is maintained in the local configuration file and is administered by the owner of the JAM site.

JAM sites are event-driven systems; they wait for the next event to occur, either a command issued by the owner via the GUI, or a request from a peer JAM site via the JAM Server. Such tasks can be any of JAM's functions, from computing a local classifier and starting the meta-learning process to sending the local classifiers to peer JAM sites, to requesting remote classifiers from other sites or to reporting the current status and presenting computed results. GUI commands can either be single-action instructions (e.g., partition the data set into training and test sets under specific constraints) or batch-mode instructions (e.g., perform a 10-fold cross validation meta-learning experiment).[4] A GUI command activates the JAM Engine, which will subsequently translate it, verify its validity and execute it. Depending on the nature of this command, the JAM Engine may, in turn, call the JAM Client. For example, on an "import and meta-learn remote classifier agents" command, the JAM Engine would rely on the JAM Client component to obtain the remote classifier agents. The status of the system and the final outcome of the actions of the JAM Engine are returned and reported to the owner through the GUI.

Figure 3 presents a snapshot of the JAM system. In this example, three JAM sites, Marmalade, Strawberry and Mango collaborate in order to share and improve their performance

[4]The Text-based user interface provides a similar, albeit more limited set of commands. A fine control of the JAM site, however, is still possible through the local configuration file.

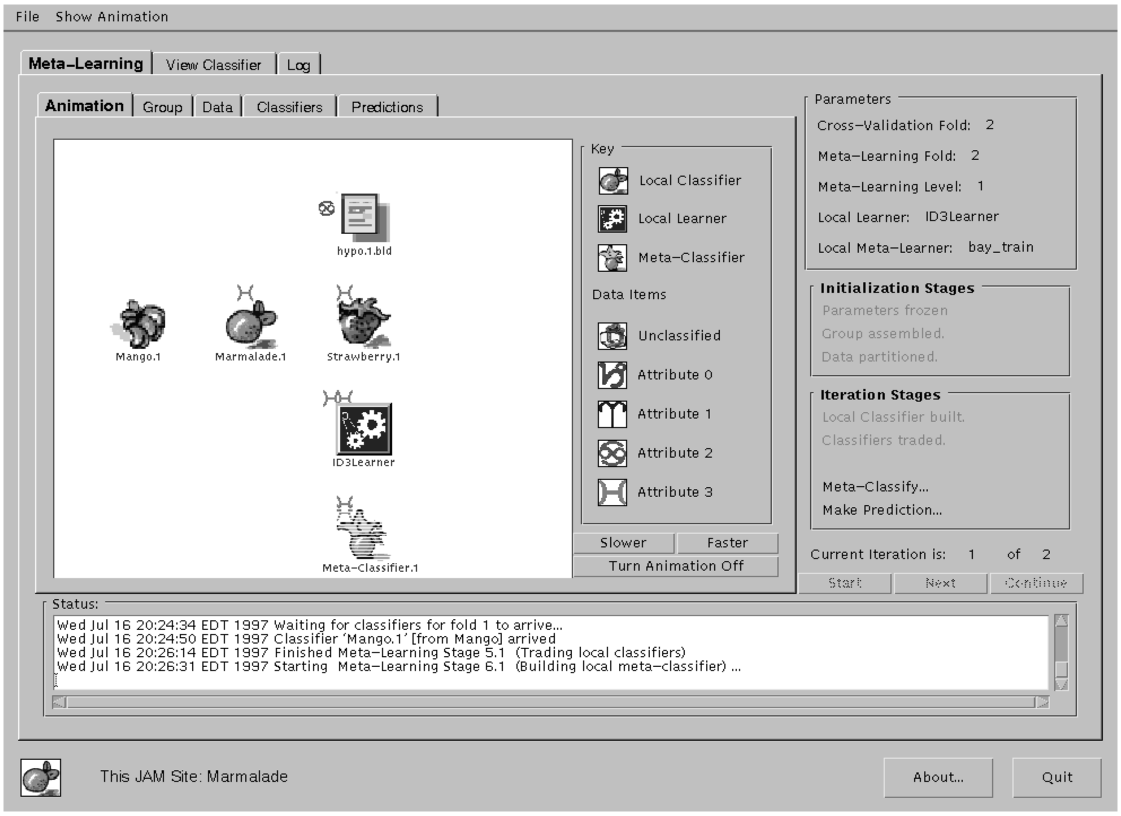

Figure 3: Snapshot of the JAM system in action: Marmalade is building the meta-classifier.

in diagnosing hypothyroid-related problems [30]. The snapshot is taken from "Marmalade's point of view". It displays the system during the meta-learning phase. Notice that Marmalade has established that Strawberry and Mango are its potential peer JAM sites by acquiring information through a QUERY request to the CM.

The right side of panel of the GUI keeps information about the current stage of the system and displays the settings of several key parameters, including the Cross-Validation fold, the Meta-Learning fold (i.e., the data partitioning scheme used in the meta-learning stage), the Meta-Learning level, the names of the local learning and meta-learning agents, etc. The bottom part of the panel logs the various events, and records the current status of the system. In this instance, the Marmalade JAM site partitions the hypothyroid database into the hypo.1.bld and hypo.2.bld data subsets according to the 2-fold Cross Validation Scheme. During the learning phase of the first fold, Marmalade computes the local classifier Marmalade.1 by applying the ID3Learner agent on hypo.1.bld. Next, it imports the remote classifiers, noted by Strawberry.1 and Mango.1 and begins the meta-learning process. In this experiment, each site contributes a single classifier agent. During the meta-learning phase of the first fold, Marmalade applies the three base classifier agents Mango.1, Marmalade.1 and Strawberry.1 on the hypo.1.bld data subset using the 2-fold meta-learning scheme([6]), to generate the meta-level training set. The final ensemble meta-classifier, noted as Meta-Classifier.1 is computed via the stacking method using the "native" bay_train Bayesian learning algorithm over this meta-level training set.

Marmalade will employ the Meta-Classifier.1 to predict the classes of the hypo.2.bld test set as dictated by the 2-fold Cross Validation evaluation scheme. If Cross Validation was set to one, the JAM site would use Meta-Classifier.1 to classify single data instances (in this case unlabeled medical records), or optionally label a separate test set provided by the owner.

The snapshot of Figure 3 displays the system during the animated meta-learning processes, where JAM's GUI moves icons within the Animation Tabbed folder of the JAM site displaying the construction of the new meta-classifier. Detailed information (not shown here) about the participating JAM sites and the local hypothyroid data sets are found inside the Group tabbed folder and the Data tabbed folder respectively. In addition, the User Interface provides a Classifier Tabbed folder and a Predictions Tabbed folder. The Classifier Tabbed folder allows the owner of the JAM site to study the base- and meta-classifiers more closely, while the Predictions Tabbed folder lets him/her administer the test phase, e.g., subject the various models in batch testing (generate predictions on multiple test instances of a test file) or single testing (classify one example at a time) to evaluate the performance of the derived classifiers and meta-classifiers.

2.2.2 JAM Client and JAM Server Components

The JAM sites are designed to work in parallel and autonomously. In particular, the JAM system is architected as a collection of loosely coupled processes (the JAM sites), each performing its own local data mining (in this case, learning/classification) and occasionally collaborating with its peer processes to import or export local classifiers. The design follows that of a client-server architecture. Specifically, each JAM site can operate simultaneously as a Client site requesting learning or classifier agents from remote servers and as a Server site responding to similar requests from other sites.

To avoid synchronization barriers and minimize busy-wait scenarios, both the Client and the Server components are implemented in a multi-threaded fashion. Figure 4 shows JAM

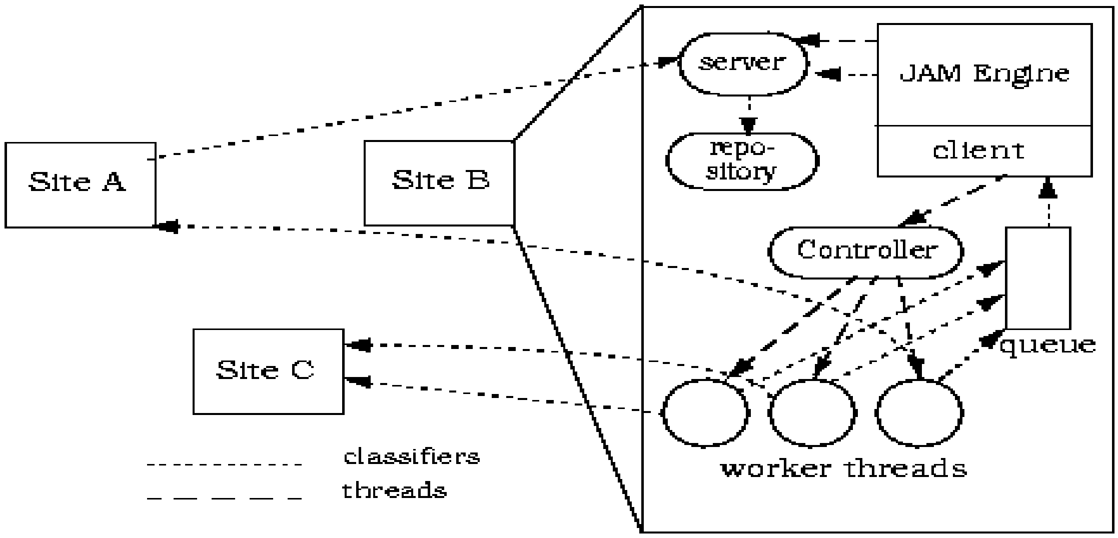

Figure 4: JAM as a Client-Server architecture.

site B acting as a Client to sites A and C and as a Server for site A. In this example, the JAM Engine of site B instructs the JAM client to obtain three remote classifier agents, one from JAM Site A and two from JAM Site C. To service the Engine's request, the Client spawns a main *Controller* thread that creates a local *Queue* (i.e., a buffer) for storing the results (e.g., the returned classifiers) and spawns, in turn, three *Worker* threads, one for each classifier agent. The benefit of this design is that the JAM sites are capable of issuing multiple requests to their peer JAM sites in parallel. Upon completion, each *Worker* thread obtains the lock of the *Queue*, inserts the result into the *Queue* and releases the lock. Every t seconds, currently set at 5 seconds, the *Controller* thread obtains the lock of the *Queue* and conveys any returned results to the JAM Client. When the complete set of results is available, the JAM Client returns it to the JAM Engine, which continues with normal execution.

Besides collecting these results, the *Controller* thread is responsible for monitoring its *Worker* threads' progress. To account for the probability of site failures and network outages, for example, the *Controller* thread imposes a hard limit as to how long it may wait for a response from its *Worker* threads. Any *Worker* thread violating this limit is deemed blocked and is killed. In such a case, the *Controller* thread and, subsequently, the JAM Client provide to the calling JAM Engine an appropriate error code along with the partial set of results.

At the opposite end, JAM Servers are responsible for satisfying requests. As with the

Table 2: Interface published by the JAMServer

Method call	Method parameters	Return result
JAMGetDBDirectory		Vector of local database names
JAMGetDBProperties	DBName	Schema description
JAMGetAgentDirectory	DBName	Directory of local agents
JAMGetAgent	TimeStamp	Single (Learner/Classifier) agent
JAMGetBaseLearnersNames		Vector of BaseLearners' names
JAMGetBaseLearners	LearnerNames, TimeStamp	Vector of BaseLearners
JAMGetMetaLearnersNames		Vector of MetaLearners' names
JAMGetMetaLearners	MetaLearnerNames, TimeStamp	Vector of MetaLearners
JAMGetClassifiers	DBName, AlgorithmNames, IsMeta, FoldNumber, TimeStamp	Vector of Classifiers

JAM Client, the JAM Server is also multi-threaded to support multiple calls simultaneously, both local (e.g., from the JAM Engine), and remote (from other JAM Sites). This version of the JAM Server is built upon the existing Remote Method Invocation (RMI) technology offered as part of Sun's Java package. As the name implies, RMI enables the invocation of methods of remote Java objects from other virtual machines, possibly on different hosts. JAM Clients invoke remote object methods through references provided by the *RMI registry* [2].

An *RMI Registry* corresponds to a name server at the server side that allows remote clients to get a reference to server objects. Typically, there is one *RMI Registry* for every JAM site. The *RMI Registry* and the JAM site run as separate processes sharing the same host machine. Upon initialization, the JAM server uses the *RMI Registry* to bind its list of available objects to names. Subsequently, a JAM Client can access and lookup up the server objects at the *RMI Registry* based on the Uniform Resource Locator (URL), and invoke the server methods as needed. By integrating RMI into JAM and by defining the set of object methods exported by the JAM Servers, we specified the communication protocol among sites. Then we materialize it via remote object method calls from the JAM Clients to the JAM servers. In addition to being a clean and straightforward approach, RMI provides the additional benefit of being extensible; by allowing the JAM Servers to define and export additional methods through the *RMI Registry*, the communication protocol can be extended to support new functionality. The interface (the server object methods) provided as part of the current design of the JAM server is presented in Table 2.

The first four rows of the table contain the necessary and sufficient methods that need to be defined by a JAM Server. The first two methods provide the means for a JAM Client to access remote database information, whereas the next two rows describe the methods for requesting the list of available agents and obtaining the desirable remote learning or classifier agents. The design of the interface of the JAM Server, however, is extended with additional methods to allow alternative, more flexible and easier use, i.e., it provides methods for requesting the names of the base-learning and meta-learning algorithms, for acquiring the base-learning and meta-learning agents and for obtaining the needed classifier agents.

The learning and classifier agents are uniquely identified by the *TimeStamp* index, i.e., an index created at the instant each agent is inserted in the JAM Site repository (discussed later in more detail). Besides the *TimeStamp* index other information associated with each agent include:

1. the name of learning algorithm,
2. the cross validation fold number (zero for learning agents),
3. a boolean parameter distinguishing whether it is a base-level of meta-level agent and
4. the name of the database over which it is computed (only for classifier agents).

The JAM Server is designed to provide to a JAM Client all of its agents that match the parameters of the calling methods. For instance, if *DBName* is set to hypo, *IsMeta* is set to false, and *FoldNumber* is set to one, and both *AlgorithmNames* and *TimeStamp* are set to null, the *JAMGetClassifiers* method will return all base classifiers computed over the hypothyroid database under the first cross-validation fold, independently of the learning algorithm or the time they were created. An error code and a null vector are returned in case the input parameters of a remote method call are conflicting.

To avoid exposing the wrong agents when confidentiality issues and distribution rights are of matter, we followed the conservative approach and designed the JAM Server to export only its local learning and classifier agents and not any agents acquired from other sites. Nevertheless, it is easy to relax these constraints, if required, by extending the *TimeStamp* index to include the name of the remote site from where an agent originates. This change would enable JAM Clients to index and obtain any agent that resides at a particular JAM Server, regardless of it being remote (obtained from a peer JAM Server) or local to that Server.

Each JAM Server interacts with the local *Repositories* that maintain the agents and make them available as required. The JAM Engine instantiates a separate *Repository* object for each local data set, i.e., for each *DBName*. The *Repository* consists of a database of local (introduced/installed by the owner) and remote (transferred from another site) learning agents, and local and remote classifier agents. By local classifier agents we mean the classifiers computed over a local data set by local or remote learning agents; by remote classifier agents we denote the classifiers derived over remote data either by remote learning agents or by local learning agents that migrated at the remote site. A learning agent can represent either a base-learning algorithm or a meta-learning technique. Similarly, a classifier agent can either be a single base-classifier or a meta-classifier that combines multiple classifier agents.

Table 3: JAM site Repository Interface

Method call	Method parameters	Description
JAMInsert	JAMSite, AlgorithmName, isMeta, FoldNumber, TimeStamp	Add an agent to the Repository
JAMDelete	JAMSite, TimeStamp	Remove an agent from the Repository
JAMGetAgent	JAMSite, TimeStamp	Return a specific agent (Learner/ Classifier) (Learner or Classifier)
JAMLoad	URL to storage location	Populate the Repository with existing agents from previous runs
JAMGetLearner	JAMSite, TimeStamp	Return a specific Learner agent
JAMGetClassifier	JAMSite, TimeStamp	Return a specific Classifier agent
JAMSelect	JAMSite, AlgorithmName, FoldNumber, isMeta	Return a vector of agents that match the input parameters

The *Repository* supports a small number of primitives for accessing and updating the available agents, as described in Table 3. Each entry in the database, i.e., a learning or a classifier agent, is indexed by the name of its originating JAM site and a time stamp created upon entrance into that database. Other attributes defined for each entry in the *Repository* include the name of the learning algorithm, the fold number that generated a specific classifier agent in a k-fold Cross-Validation experiment (this number is set to zero for learning agents) and the boolean attribute *isMeta* that distinguishes base-level from meta-level agents.

JAMInsert, JAMDelete and *JAMGetAgent* are the main primitives for adding, removing and retrieving agents. Similar to the JAM Server interface, however, the *Repository* provides a second set of primitives to support additional functionality; *JAMLoad* provides the means to populate the *Repository* with existing agents that were computed and stored during past executions of the JAM system; *JAMGetLearner* and *JAMGetClassifier* define alternative methods to access the learning and classifier agents respectively; and finally, *JAMSelect* returns all agents that match the parameters of the calling method.

2.3 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 an abstract parent agent class and every instance agent object (i.e., a program that implements a learning algorithm ID3, Ripper, CART [5], Bayes [13], Wpebls [11], CN2 [9], etc.) is then defined as a subclass of this parent class. Through the variables and methods inherited by all agent subclasses, the parent agent class describes a 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 as part of the JAM system. To be more specific, a JAM learning agent needs to implement the following methods:

1. A *constructor* method with no arguments. The JAM Engine calls this method to instantiate the agent, provided it knows its name (it is supplied by the owner of the JAM site through the local configuration file or the GUI).

2. An *initialize* method. In most of the cases, the sub-classed agents inherit this method from the parent agent class. Through this method, the JAM Engine supplies the necessary arguments to the agent including the name of the training data set, the name of the dictionary file (also known as attribute file), and the file name of the output classifier, if required.

3. A *buildClassifier* method. The JAM Engine calls this method to trigger the agent to learn (or meta-learn) a classifier from the training data set.

4. A *getCopyOfClassifier* method. This method is used by the JAM Engine to obtain the newly built classifier. The derived Classifier, a Java object itself, can be subsequently transferred and "snapped-in" at any participating JAM site. Hence, remote agent dispatch is easily accomplished.

5. Additional methods, such as *getAlgorithmName*, *getDBName*, *getDictionaryExtension*, etc. that facilitate the access of agent-specific and task-specific information. These methods are defined at the Learner class level and inherited by the sub-classed agents.

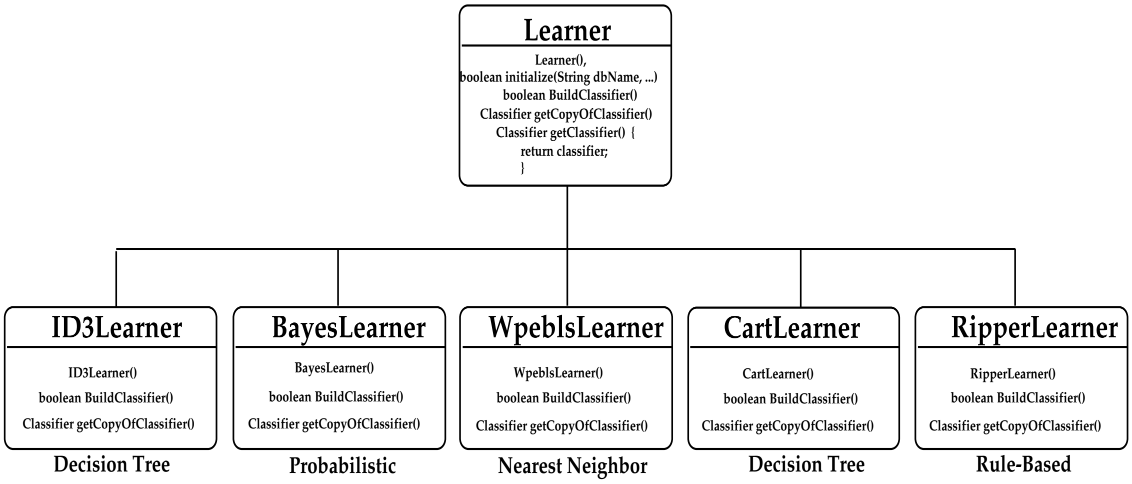

Figure 5: The class hierarchy of Learning agents.

The class hierarchy (only methods are shown) for five different learning agents is presented in Figure 5. ID3, Bayes, Wpebls, CART and Ripper re-define the *buildClassifier* and *getCopyOfClassifier* methods but inherit the *initialize* method from their parent learning agent class as well as the methods for acquiring specific information (e.g., *getAlgorithmName*).

Due to this design, no task- or algorithm-specific information (such as the name of the algorithm, program options, input and output parameters, etc) is present in the source code of the JAM Engine. As a result, the system need not be re-compiled if a new algorithm is introduced. Instead, the JAM Engine can access an agent by calling the redefined methods of the instantiated sub-classed objects of the abstract parent Learner class. The additional methods (e.g., *getAlgorithmName*, etc) described earlier, are defined as a means to expose information that is specific to each sub-classed object (e.g., the name of the learning algorithm).

The abstract MetaLearner class follows the Learner class design, but with the addition of an extra *baseClassifiers* data member and a *prepareMLSet* method. The *baseClassifiers* data member corresponds to the vector of classifiers combined by the meta-learning algorithm and the *prepareMLSet* method implements the generic function of composing the meta-level training set based on the predictions of the base classifiers on the validation set.Different meta-learning schemes, such as Stacking, Voting, SCANN, etc., can be introduced in JAM by sub-classing the MetaLearner class and by defining the *buildMetaClassifier* method (instead of the *buildClassifier* method of the Learner class) and inheriting or redefining (if needed) the *prepareMLSet* method.

Base- and meta-classifiers are defined as Java objects as well. JAM provides the definition of the abstract parent Classifier agent class and every instance agent object (base-classifier of meta-classifier) is defined as a subclass of this parent class. A Classifier agent is the product of a Learner or MetaLearner agent when applied to a data set. As with the Learner and MetaLearner classes, as long as a Classifier agent conforms to the specific interface, it can be introduced and used immediately as part of the JAM system. Specifically, a JAM Classifier agent needs to implement the following methods:

1. A *constructor* method. A sub-classed object of the Learner class calls this method to instantiate an object of the corresponding Classifier subclass.

2. A *getClassifierEngine* method. It returns an object of the ClassifierEngine class, that is subsequently used to classify new examples. More specifically, the ClassifierEngine class provides the *classifyFile* method for generating batch predictions on a test set and the *classifyItem* method to classify a single instance.

   The ClassifierEngine object is made part of Classifier to accommodate a number of existing learning programs of the public domain that require that a data dictionary accompanies each training or test set. This requirement compels Classifier agents to read the data dictionary multiple times when classifying multiple single instances. The ClassifierEngine object, allows the decoupling of the parsing of the data dictionary information and classification process, thus making it possible to read data dictionaries only once.

3. A *displayClassifier* method. It is defined by each sub-classed Classifier agent and is tailored to the specific representation of the learning algorithm and the particular implementation. The method is called from within the Classifier Tabbed folder when the owner seeks to study the internal of the Classifier agent.

4. An *isMetaClassifier* method. It is used to distinguish between base-classifiers from meta-classifier agents.

5. A *setBaseClassifiers* and a *getBaseClassifiers* methods for populating and retrieving the base Classifier agents from the baseClassifiers vector of the meta-classifiers. For base classifiers, both methods return null values.

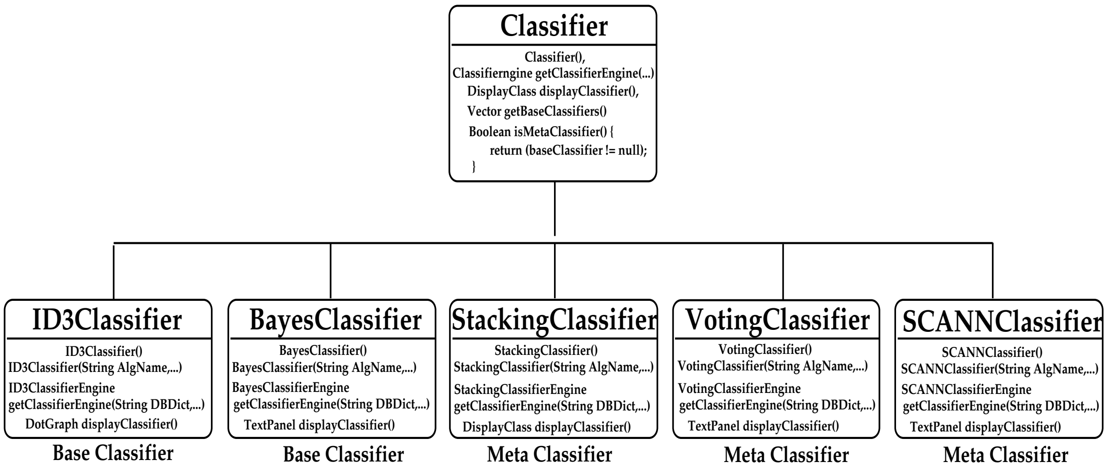

Figure 6: The class hierarchy of (base- and meta-) Classifier agents.

6. Additional methods such as *getOriginatingJAMSite*, *getDBName*, *getCVFold*, etc. that provide detailed information regarding the origin and the conditions a classifier was computed.

The class hierarchy (only methods are shown) for five different classifier agents (base- and meta-classifiers) is presented in Figure 6. ID3 and Bayes, represent base-classifier objects while Stacking, Voting and SCANN correspond to meta-classifier objects. All subclasses redefine their constructors and the algorithm-specific *getClassifyEngine* and *displayClassifier* methods, but inherit other methods such as *isMetaClassifier*, *getBaseClassifiers*, *getOriginatingJAMSite*, etc. The definition of the ClassifierEngine class used by the Classifier class follows a similar approach. For each Classifier subclass, a ClassifierEngine subclass tailors its *classifyFile* and *classifyItem* methods to execute its own base- or meta-classification scheme.

The Learning and Classifier agents are transferred among the various data sites using Java's Object Serialization capabilities [2]. Object Serialization extends Java's Input and Output classes with support for objects by marshaling and unmarshaling them to and from a stream of bytes, respectively. To efficiently transport Classifier agents across JAM sites, we overrode the default object serialization mechanism by customizing the *writeObject* and *readObject* methods for each agent subclass. Methods *writeObject* and *readObject* are part of Java's ObjectOutputStream and ObjectInputStream class definitions respectively for serializing and de-serializing a given object through an RMI or socket connection.

2.4 Portability

We used the Java technology to build the infrastructure and the various components of the JAM system, including the specific *agent operators* that compose and spawn new agents from existing learning agents, the implementation of the User Interface (Graphical and Text-based), the animation facilities and most of the machine learning algorithms and the classifier and meta-learning agents.

Java provides the means to develop a system that is capable of operating under different hardware and software configurations (e.g., across the Internet), as long as the Java Virtual Machine (JVM) [26] is installed on these environments. Moreover, by adopting the meta-learning framework as the unifying machine learning approach, JAM constitutes an algorithm independent data mining system. Meta-learning has the advantage of not being constrained to any specific representation, internal structures or strategies of the learning algorithms, but only to the output (predictions) of the individual classifiers. The platform independence of Java and the algorithm independence of meta-learning make it easy to port JAM and delegate agents to participating sites. As a result, JAM has been successfully tested on the most popular platforms including Solaris, Windows and Linux simultaneously, i.e., JAM sites can import and utilize classifiers that are computed over different platforms.

In cases where Java's computational speed is of concern, JAM is designed to also support the use of native (e.g., C or C++) learning algorithms to substitute slower Java implementations, a benefit stemming from JAM's extensible design. Native learning programs can be embedded within appropriate Java wrappers to interface with the JAM system and can subsequently be transfered and executed at a different site, provided, of course, that both the receiving site and the native program are compatible.

2.5 Extensibility

The independence of JAM from any particular learning or meta-learning method, in conjunction with the object oriented design ensure the system's capability to incorporate and use new algorithms and tools. As discussed in Section 2.3 introducing a new technique requires the sub-classing of the appropriate abstract class and the encapsulation of the tool within an object that adheres to the minimal interface. In fact, most of the existing implemented algorithms have similar interfaces already.

This plug-and-play characteristic makes JAM a powerful and extensible data mining facility. It is exactly this feature that allows users to employ native programs within Java agents if computational speed is of concern. For faster prototype development and proof of concept, for example, we implemented the ID3 and CART learning algorithms as full Java agents and imported and used the Bayes, Wpebls, Ripper and CN2 learning programs in their native (C++) form. For the latter cases, we developed program-specific Java wrappers that define the abstract methods of the parent classes and are responsible for invoking the executables of these algorithms. Furthermore, to support the transfer of native classifiers across multiple sites, we overrode the default *writeObject* and *readObject* methods to transport files instead of objects. Contrary to the Java classifiers that are represented as objects with the ability to execute, native classifiers are, in their majority, passive constructs. By storing these native classifiers into conventional files and by re-defining the *writeObject* and *readObject* methods to transport files we achieve transparency between Java and native programs.

3 Pruning meta-classifiers

The benefits of meta-learning in distributed data mining come at the expense of an increased demand for run-time system resources. Meta-classifiers can be defined recursively as collections of classifiers structured in multi-level trees [8], which suggests that the final ensemble meta-classifier may consist of a large collection of base classifiers. Hence, to classify unlabeled instances, predictions need to be generated from all base classifiers before the meta-classifier can produce its final classification. This results in significant decrease in classification throughput (the rate at which a stream of data items can be piped through and labeled by a meta-classifier) and increased demand for system resources (including memory to store base classifiers).

To alleviate the problem, we investigated the effects of *pruning*, i.e. discarding certain base classifiers. The objective of pruning was to compute partially grown meta-classifiers (meta-classifiers with pruned sub-trees) that are more efficient and scalable and at the same time achieve comparable or better predictive performance results than fully grown (unpruned) meta-classifiers. We introduced two stages for pruning meta-classifiers, the *a-priori pruning* or *pre-training pruning* and the *a-posteriori pruning* or *post-training pruning* stages. Both levels are essential and complementary to each other with respect to the improvement of the accuracy and efficiency of the system.

A-priori pruning or *pre-training pruning* refers to the filtering of the classifiers before they are combined. Instead of combining classifiers in a brute force manner, with pre-training pruning we introduce a preliminary stage for analyzing the available classifiers and qualifying them for inclusion in a combined meta-classifier. Only those classifiers that appear (according to one or more pre-defined metrics, e.g. accuracy, true positive, cost, diversity etc.) to be most "promising" participate in the final meta-classifier. Conversely, *a-posteriori pruning* or *post-training pruning*, denotes the evaluation and pruning of constituent base classifiers after a complete meta-classifier has been constructed.

3.1 Incorporating pruning in JAM

The algorithmic details of pruning is outside the scope of this paper. The details and an extensive empirical evaluation of three pre-training and two post-training pruning algorithm have appeared elsewhere [38, 40, 41, 43, 44]. Instead, we focus on the integration of pruning with JAM and the resulting architecture design.

To integrate the various techniques within JAM and at the same time be consistent with the system's objectives, we followed an object-oriented design for pruning as well. As with the Learner and Classifier classes (Section 2.3), JAM provides the abstract parent Prune class and defines several class members including two data members, namely the vector of base classifiers objects and the meta-learning agent object, and one method member, i.e. the *selectClassifiers* abstract method. Then every pruning technique can be subsequently defined by subclassing this parent class and by implementing the *selectClassifiers* method. This method is responsible for evaluating the candidate classifiers and for returning the new vector of the selected classifiers; different implementations of this method, materialize different pruning algorithms. To deploy one of the pruning methods JAM simply needs to instantiate and initialize the corresponding subclass with the appropriate arguments (the vector of candidate base classifiers agents, the meta-learning agent, the stopping criteria, etc.) prior to meta-learning, and invoke the redefined *selectClassifiers* method. As long as a pruning object conforms to the interface defined by the abstract parent Prune class, it can be introduced and used immediately as part of the JAM system.

4 Combining incompatible classifiers

In meta-learning and distributed data mining is assumed that all base classifiers are trained over databases with identical schema [12]. This however, is not always the case. Differences in the type and number of attributes among different data sets are not uncommon. Even minor differences in the schema between databases derive incompatible classifiers, i.e., a classifier trained on one database cannot be applied on the another database with different formats. Yet, these classifiers may target the same concept.

In the credit card fraud detection problem, for instance, two institutions seeking to incorporate in their system useful information that would otherwise be inaccessible, may decide to exchange their classifiers. Indeed, for each credit card transaction the two institutions record similar information. However, they also include specific fields containing important information that each has acquired separately and which provides predictive value in determining fraudulent transaction patterns. As a result, a classifier from one institution cannot be applied to the data of the other institution. In a different scenario where databases and schemas evolve over time, it may be desirable for a single institution to combine classifiers from both past accumulated data with newly acquired data. To facilitate the exchange of knowledge and take advantage of incompatible and otherwise useless classifiers, we devised methods that *bridge* the differences imposed by the different schemas. The reader is advised not to confuse this with schema integration over federated/mediated databases where the effort is towards the definition of a common schema across multiple data sets.

4.1 Incorporating Bridging Agents in JAM

The basic idea of our approach, is to use special *bridging* agents that can be trained at one database to predict the values of the missing information of the other database. In this case, the target attribute is not the class attribute of that database, but one of the missing (uncommon) attributes. To approximate the values of that attribute, the predictive model relies either upon the values of the common attributes (e.g., it can be a classification or regression model), or upon a user-defined rule (when resolving semantic differences). In this manner, these bridging agents compose an intermediate layer that alleviates the differences among database with different schemas.The details of our bridging technique and a extensive empirical study is reported in [37]. Here, we focus on the architectural design of JAM and the manner the bridging agents integrate with the system.

JAM provides the definition of the parent Bridge class. A bridging agent object is an instance of this class. Central to the definition of the bridging agent object is the Classifier object (Section 2.3). After all, a bridging agent is, itself, a predictive model that is trained to estimate the value of a target attribute. The only difference is that the target attribute is a missing (uncommon) attribute of the database.

In addition to the Classifier object, a bridging agent includes other components as well. The parent Bridge class defines a method for pre-processing the data sets to adhere to the specific format expected by its Classifier object. For instance, the Classifiers object may expect to read the data sets as flat files with the last column allocated for the target attribute, while the underlying data set has positioned the target class in the first column. The Bridge class also defines a method for populating the target (missing) attribute with the predicted values and a method for post-processing the resulting data sets to fit the format expected by the classifier agent.

To integrate the notion of bridging agents within JAM in a manner that is consistent with the design of the system, we altered the definition of the Classifier class (Section 2.3) to also include a vector of Bridge objects. The vector allocates one Bridge object for each attribute of the originating JAM site that is not present at the destination JAM site. When a JAM Client requests a Classifier object from another JAM site, the JAM Server serializes and sends each entry of the vector of Bridge objects as part of serializing and sending the requested Classifier agent. By de-serializing the receiving data stream, the JAM Client populates the vector of the Bridge objects and re-composes the Classifier agent. The Bridge agents are created upon request of Classifier objects. Specifically, the current version of the JAM system implements the following protocol:

- The JAM Client of a JAM site A issues a *JAMGetClassifiers* call to the JAM Server of another JAM site B to request a classifier C.

- The JAM Server of B requests the database schema description of JAM site A via its JAM Client and a *JAMGetDBProperties* call.

- A's JAM Server responds with the schema description.

- B's JAM Server sorts alphabetically the attribute names of A's database and compares them to the attribute names of its local database. For each attribute that is not present in A, the JAM Server computes a bridging agent and inserts it in the vector of Bridge Objects of classifier C. The particular method used for generating a bridging agent (learning algorithm, regression technique, interpolation, etc.) is decided by the owner of JAM site B.

- B's JAM server returns classifier C and its bridging agents to A's JAM client.

The protocol is designed to comply to the interface published by JAM servers (see Table 2). It is possible to suppress or eliminate the second and third steps of the protocol in a future release of the JAM system, by allowing JAM Servers to cache the schema description, and/or by overloading the JAM Server interface (see Table 2) to support a *JAMGetAgent* and a *JAMGetClassifiers* methods that accept schema descriptions as input parameters (provided by the requesting JAM Client). Identifying attributes with syntactic or semantic differences when attribute names are identical, or distinguishing situations where names are different when in fact the attributes are the same, has not been addressed in this work. It is a matter of future research that entails the study and development of methods and languages for declaring and formally defining the schema of each database.

5 Applying JAM in Fraud Detection

The traditional way to defend financial information system has been to protect the routers and network infrastructure. Furthermore, to intercept intrusions and fraudulent transactions that inevitably leak through, financial institutions have developed custom fraud detection systems targeted to their own asset bases. Recently however, banks have come to realize that a unified, global approach that involves the periodic sharing of information regarding fraudulent practices is required. Here, we employ the JAM system as an alternative approach that supports the cooperation among different institutions and consists of *pattern-directed inference systems* that use models of anomalous or errant transaction behaviors to forewarn of fraudulent practices. This approach requires the analysis of large and inherently distributed databases of information about transaction behaviors to produce models of "probably fraudulent" transactions. An orthogonal approach to modeling transactions would be to model user behavior. An application of this method, but in cellular phone fraud detection has been examined in [16].

The key difficulties in our strategy are: financial companies do not 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 for up-to-date detection.

To address these difficulties and thereby protect against electronic fraud our approach has two key component technologies, both provided by JAM: *local fraud detection agents* that learn how to detect fraud within a single information system, and an integrated *meta-learning mechanism* that combines the collective knowledge acquired by the individual local agents. Thus, meta-learning allows financial institutions to share their models of fraudulent transactions without disclosing their proprietary data. This way their competitive and legal restrictions can be met, but they can still share information. Furthermore, by supporting the training of classifiers over distributed databases, JAM can substantially reduce the total learning time (parallel learning of classifiers over (smaller) subsets of data). The final meta-classifiers can be used as sentries forewarning of possible fraud by inspecting and classifying each incoming transaction.

To validate the applicability of this approach in the security of financial information systems we experimented with two data sets of credit card transactions supplied by two different financial institutions. By way of summary, we found that JAM, as a pattern-directed inference system constitutes a protective shield against fraud with the potential to exceed the performance of existing fraud detection techniques. The full details of the experiments are discussed in [36]. Next we present a summary of that evaluation.

5.1 Experimental Setting

We employed five inductive learning algorithms in our experiments, Bayes, C4.5, ID3, CART and Ripper. Bayes implements a naive Bayesian learning algorithm described in [32], CART [5], ID3 [48] and its successor C4.5 [49] are decision tree based algorithms, and Ripper [10] is a rule induction algorithm. We used multiple versions of decision tree algorithms for their property to generate diverse classifiers.

Then we employed eight different meta-learning techniques, based on the Voting, Stacking and SCANN methods.Specifically, we applied the two variations of voting, majority and weighted, the five learning algorithms (Bayes, C4.5, ID3, CART, Ripper) as meta-learning algorithms for stacking and the SCANN meta-learning method.

We obtained two databases (70MB approximately) from Chase and First Union banks, both members of FSTC (Financial Services Technology Consortium), each with 500,000 records of credit card transaction data spanning one year (form October 1995 to September 1996). Chase bank data consisted, on average, of 42,000 sampled credit card transactions records per month with a 20% fraud and 80% legitimate distribution, whereas First Union data were sampled in a non-uniform (many records from some months, very few from others, very skewed fraud distributions for some months) manner with a total of 15% fraud versus 85% legitimate distribution. The database schemas were developed over years of experience and continuous analysis by bank personnel to capture important information for fraud detection. The records had a fixed length of 137 bytes each and about 30 numeric and categorical attributes including the binary class label (fraud/legitimate transaction).

The first step in this data mining process involved the arduous process of cleaning and preprocessing the given data sets. In this case, dealing with real-world data entailed missing data fields (records with fewer attributes), invalid entries (e.g., real values out of bounds), legacy systems remains (e.g., in some cases, letters were used instead of signed numbers for compactness), undefined classes for certain categorical attributes, conflicting semantics (e.g., in some cases for the same attribute, a zero denoted both a missing value, and the value 0), etc. Furthermore, we simplified the learning task by removing insignificant data (e.g., the last four digits of the nine digit zip codes), by discretizing some real values (e.g., the time a transaction took place) and by transforming attributes to more informative representations (e.g., we replaced the date of the last payment with the number of days passed since the transaction date).

Although preprocessing is an early task in the data mining process, we had to backtrack (sometimes even after learning and meta-learning) and repeat it several times until we settled on the final format for each data set.

5.2 Learning Tasks

Our task was to compute effective classifiers that correctly discern fraudulent from legitimate transactions. To evaluate and compare the base- and meta-classifiers constructed, we adopted three metrics: the accuracy, the $(TP - FP)$ spread[5] and a cost model fitted to the credit card detection problem. Accuracy expresses the ability of a classifier to give correct predictions, $(TP - FP)$ denotes the ability of a classifier to catch fraudulent transactions while minimizing false alarms, and finally, the cost model captures the performance of a classifier with respect to the goal of this application (stop loss due to fraud).

Credit card companies have a fixed overhead that serves as a threshold value for challenging the legitimacy of a credit card transaction. If the transaction amount *amt*, is below this threshold, the transaction is authorized automatically. Each transaction predicted as fraudulent require an "overhead" referral fee for authorization personnel to decide the final disposition. This "overhead" cost is typically a "fixed fee" that we call $\$X$. Therefore, even if we could accurately predict and identify all fraudulent transactions, those whose *amt* is less than $\$X$ would produce $(X - amt)$ in losses anyway. In these experiments, we incorporated the threshold values and referral fees in the detection process and we sought to produce classifiers and meta-classifiers that maximize the total savings.

5.3 Summary Results

To generate our classification models we distributed each data set across six different data sites (each site storing two months of data) and we applied the five learning algorithms on each month of data, therefore creating 60 classifiers (10 classifiers per data site).[6] This "month-dependent" data partitioning scheme was used only on the Chase bank data set. The very skewed nature of the First Union data forced us to equi-partition the entire data set randomly into 12 subsets and assign two subsets in each data site.

Next, we had each data site import the "remote" base classifiers (50 in total) and apply them on its own data. Hence, each classifier was not tested unfairly on known data. Specifically, we had each site use half of its local data (one month) to test, prune and meta-learn the remote base-classifiers and the other half to evaluate the overall performance of the pruned or unpruned meta-classifier (for extensive details see [39, 41]). In essence, the setting of this experiment corresponds to a parallel six-fold cross validation.

Finally, we had the two banks exchange their classifier agents as well. In addition to its

[5] In comparing the classifiers, one can replace the TP-FP spread, which defines a certain family of curves in the ROC plot, with a different metric or even with a complete analysis [45, 46] in the ROC space.

[6] Extensive experiments evaluating different data distributions are presented in [42].

10 local and 50 "internal" classifiers (those imported from their peer data sites), each site also imported 60 external classifiers (from the other bank). Thus, each Chase data site was populated with 60 (10+50) Chase classifiers and 60 First Union classifiers and each First Union site was populated with 60 (10+50) First Union classifiers and 60 Chase classifiers. Again, the sites used half of their local data (one month) to test, prune and meta-learn the base-classifiers and the other half to evaluate the overall performance of the pruned or unpruned meta-classifier. To ensure fairness, each site meta-learned 110 base-classifiers. The 10 local base-classifiers of each site were not used in meta-learning.

The two databases, however, had the following schema differences:

1. Chase and First Union defined a (nearly identical) feature with different semantics (i.e., they used different time intervals to measure the number of times an event occurs),

2. Chase included two (continuous) features not present in the First Union data

For the first incompatibility, we had the values of the First Union data mapped to the semantics of the Chase data. For the second incompatibility, we deployed bridging agents to compute the missing values (a detailed discussion appears in [37]). When predicting, the First Union classifiers simply disregarded the real values provided at the Chase data sites, while the Chase classifiers relied on both the common attributes and the predictions of the bridging agents to deliver a prediction at the First Union data sites.

Tables 4 and 5 summarize our results for the Chase and First Union banks respectively. Table 4 reports the performance results of the best classification models on Chase data, while Table 5 presents the performance results of the best performers on the First Union data. Both tables display the accuracy, the TP-FP spread and savings for each of the fraud predictors examined and the best result in every category is depicted in bold. The maximum achievable savings for the "ideal" classifier, with respect to our cost model, is \$1,470K for the Chase and \$1,085K for the First Union data sets. The column denoted as "size" indicates the number of base-classifiers used in the classification system.

The first row of Table 4 shows the best possible performance of Chase's own COTS (Commercial Off The Shelf) authorization/detection system on this data set, while the second row presents the performance of the best base classifiers over a single subset. The next four meta-classifiers combine only "internal" (from Chase) base classifiers, while the last four combine both internal and external (from Chase and First Union) base classifiers. Bridging

Table 4: Performance results for the Chase credit card data set.

Type of Classification Model	Size	Accuracy	TP - FP	Savings
COTS scoring system from Chase	-	85.7%	0.523	$ 682K
Best base classifier over single subset	1	88.7%	0.557	$ 843K
Meta-classifier over Chase base classifiers	50	89.74%	0.621	$ 818K
Meta-classifier over Chase base classifiers	46	**89.76%**	0.574	$ 604K
Meta-classifier over Chase base classifiers	27	88.93%	0.632	$ 832K
Meta-classifier over Chase base classifiers	4	88.89%	0.551	**$ 905K**
Meta-classifier over Chase and First Union base classifiers (without bridging)	110	89.7%	0.621	$ 797K
Meta-classifier over Chase and First Union base classifiers (without bridging)	65	89.75%	0.571	$ 621K
Meta-classifier over Chase and First Union base classifiers (without bridging)	43	88.34%	**0.633**	$ 810K
Meta-classifier over Chase and First Union base classifiers (without bridging)	52	87.71%	0.625	$ 877K

agents were not used in these experiments, since all attributes needed by First Union agents, were already defined in the Chase data. The former four rows detail the performance of the unpruned (size of 50) and best pruned meta-classifiers for each of the evaluation metrics (size of 46 for accuracy, 27 for the TP-FP spread, and 4 for the cost model). Finally, the latter four rows report on the performance of the unpruned (size of 110) and best pruned meta-classifiers (sizes of 65, 43, 52) according to accuracy, the TP-FP spread and the cost model respectively.

Similar data is recorded in Table 5 for the First Union set, with the exception of First Union's COTS authorization/detection performance (it was not made available to us), and the additional results obtained when employing special bridging agents from Chase to compute the values of First Union's missing attributes.

The most apparent outcome of these experiments is the superior performance of meta-learning over the single model approaches and over the traditional authorization/detection systems (at least for the given data sets). The meta-classifiers outperformed the single base classifiers (local or global) in every category. Moreover, by bridging the two databases, we managed to further improve the performance of the meta-learning system. Notice, however, that combining classifiers agents from the two banks directly (without bridging) is not very effective. This phenomenon can be easily explained from the fact that the attribute missing from the First Union data set is significant in modeling the Chase data set. Hence, the

Table 5: Performance results for the First Union credit card data set.

Type of Classification Model	Size	Accuracy	TP - FP	Savings
Best base classifier over single subset	1	95.2%	0.749	$ 800K
Meta-classifier over First Union base classifiers	50	96.53%	0.831	$ 935K
Meta-classifier over First Union base classifiers	14	96.59%	0.797	$ 891K
Meta-classifier over First Union base classifiers	12	96.53%	0.848	$ 944K
Meta-classifier over First Union base classifiers	26	96.50%	0.838	$ 945K
Meta-classifier over Chase and First Union base classifiers (without bridging)	110	96.6%	0.843	$ 942K
Meta-classifier over Chase and First Union base classifiers (with bridging)	110	**98.05%**	0.897	**$ 963K**
Meta-classifier over Chase and First Union base classifiers (with bridging)	56	98.02%	0.890	$ 953K
Meta-classifier over Chase and First Union base classifiers (with bridging)	61	98.01%	**0.899**	$ 950K
Meta-classifier over Chase and First Union base classifiers (with bridging)	53	98.00%	0.894	$ 962K

First Union classifiers are not as effective as the Chase classifiers on the Chase data, and the Chase classifiers cannot perform at full strength at the First Union sites without the bridging agents.

An additional result, evident from these tables, is the invaluable contribution of pruning. In all cases, pruning succeeded in computing meta-classifiers with similar or better fraud detection capabilities, while reducing their size and thus improving their efficiency. A detailed description on the pruning methods and a comparative study between predictive performance and meta-classifier throughput can be found in [41, 36].

6 Conclusions and future research directions

In this paper we described the architecture of the JAM system, a distributed, scalable, portable and extensible agent-based system that supports the launching of learning and meta-learning agents to distributed database sites. JAM consists of a set of similar and collaborating JAM sites in a network configuration maintained by the Configuration Manager.

JAM is scalable in that it is designed with asynchronous, distributed communication protocols that enable the participating database sites to operate independently and collaborate with other peer sites as necessary, thus eliminating centralized control and synchronization points. JAM is portable because it is built upon existing agent infrastructure available over the Internet using Java technology and algorithm-independent meta-learning techniques. Extensibility is ensured by decoupling JAM from the learning algorithms and by introducing modular plug-and-play capabilities though a well-developed object-oriented design. At the same time, JAM is designed to support pruning and bridging, two techniques that address two drawbacks of meta-learning, the increased demand for run-time system resources, and the inability to combine multiple models computed over data sets with different schemas.

The JAM system can be further enhanced with *additional functionality*. For example, the current implementation of JAM defines a Configuration Manager that provides registration and membership services to each JAM site. Future extensions of the CM can support multiple groups of sites, varying levels of "visibility" (e.g., some sites may not be allowed to get access information about every other JAM site - a similar approach to access/capability lists between users and resources), authentication capabilities, directory services of databases and learning and classifier agents, fault tolerance, etc.

Furthermore, JAM sites can be extended with tools facilitating the *data selection problem*. The data selection problem refers to the preprocessing, transformation and projection of the available data to expressive and informative features, and is probably one of the hardest, but very important stages in the knowledge discovery process. The process depends on the particular data mining task and requires application domain knowledge. The current version of JAM, assumes well-defined schemas and data sets. The credit card data sets that were used as an application, for example, were first developed by experienced FSTC (Financial Services Technology Consortium) personnel and then cleaned and pre-processed by us in a separate off-line process before being used in JAM.

Introducing data selection tools and defining the JAM databases can be linked to the incompatible schema problem. Recall that comparing databases and identifying attributes with syntactic or semantic differences has not been addressed here. The study and development of *formal methods and languages* for declaring and defining schemas is a crucial and hard problem, suitable for extensive research (early work in this field can be found in [18, 19]). Resolving the incompatible schema problem can instigate the expansion of present data mining systems. The "visibility" of meta-learning systems will be extended to data sources that would otherwise remain unutilized, information will be shared more readily and meta-level classification models will improve their performance by automatically incorporating more diverse models.

7 Acknowledgments

Wenke Lee, Wei Fan and Matthew Schultz contributed to an early version of the JAM system. We are in debt to Chris Merz for sharing wth us his implementation of the SCANN algorithm.

References

[1] K. Ali and M. Pazzani. Error reduction through learning multiple descriptions. *Machine Learning*, 24:173–202, 1996.

[2] K. Arnold and J. Gosling. *The Java Programming Language, second edition*. Addison-Wesley, Reading, MA, 1998.

[3] L. Breiman. Heuristics of instability in model selection. Technical report, Department of Statistics, University of California at Berkeley, 1994.

[4] L. Breiman. Stacked regressions. *Machine Learning*, 24:41–48, 1996.

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

[6] P. Chan. *An Extensible Meta-Learning Approach for Scalable and Accurate Inductive Learning*. PhD thesis, Department of Computer Science, Columbia University, New York, NY, 1996.

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

[8] P. Chan and S. Stolfo. Sharing learned models among remote database partitions by local meta-learning. In *Proc. Second Intl. Conf. Knowledge Discovery and Data Mining*, pages 2–7, 1996.

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

[10] W. Cohen. Fast effective rule induction. In *Proc. 12th Intl. Conf. Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

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

[12] T.G. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4):97–136, 1997.

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

[14] W. Fan. *On the effective use of stacking*. PhD thesis, Department of Computer Science, Columbia University, New York, NY, 2000.

[15] W. Fan, W. Lee, S. Stolfo, and M. Miller. A multiple model cost-sensitive approach for intrusion detection. In *Proc. Eleventh European Conference of Machine Learning*, pages 148–156, Barcelona Spain, May 2000.

[16] T. Fawcett and F. Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1(3):291–316, 1997.

[17] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, pages 23–37. Springer-Verlag, 1995.

[18] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Integrating and accessing heterogeneous information sources in tsimis. In *Proc of the AAAI Symposium on Information Gathering*, pages 61–64, March 1995.

[19] L. M. Haas, R. J. Miller, B. Niswonger, M. Tork Roth, P. M. Schwarz, and E. L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *Data Engineering Bulletin*, 1999.

[20] R.A. Jacobs, M.I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixture of local experts. *Neural Computation*, 3(1):79–87, 1991.

[21] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214, 1994.

[22] M. I. Jordan and L. Xu. Convergence results for the em approach to mixtures of experts architectures. In *AI memo 1458*, 1993.

[23] E. B. Kong and T. Dietterich. Error-correcting output coding corrects bias and variance. In *Proc. Twelfth Intl. Conf. Machine Learning*, pages 313–321, 1995.

[24] A. Krogh and J. Vedelsby. Neural network ensembles, cross validation, and active learning. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Info. Proc. Sys. 7*, pages 231–238. MIT Press, 1995.

[25] M. LeBlanc and R. Tibshirani. Combining estimates in regression and classification. Technical Report 9318, Department of Statistics, University of Toronto, Toronto, ON, 1993.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, second edition*. Addison-Wesley, Reading, MA, 1999.

[27] N. Littlestone and M. Warmuth. The weighted majority algorithm. Technical Report UCSC-CRL-89-16, Computer Research Lab., Univ. of California, Santa Cruz, CA, 1989.

[28] M Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the fifth Int'l Conf. on Extending Database Technology*, Avignon, France, March 1996.

[29] C. Merz. Using correspondence analysis to combine classifiers. *Machine Learning*, 36:33–58, July 1999.

[30] C. Merz and P. Murphy. 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, 1996.

[31] C. Merz and M. Pazzani. A principal components approach to combining regression estimates. *Machine Learning*, 36:9–32, 1999.

[32] M. Minksy and S. Papert. *Perceptrons: An Introduction to Computation Geometry*. MIT Press, Cambridge, MA, 1969. (Expanded edition, 1988).

[33] D. W. Opitz and J. J. W. Shavlik. Generating accurate and diverse members of a neural-network ensemble. *Advances in Neural Information Processing Systems*, 8:535–541, 1996.

[34] J. Ortega, M. Koppel, and S. Argamon-Engelson. Arbitrating among competing classifiers using learned referees. *Machine Learning*, 1999. in press.

[35] M. P. Perrone and L. N. Cooper. When networks disagree: Ensemble methods for hybrid neural networks. *Artificial Neural Networks for Speech and Vision*, pages 126–142, 1993.

[36] A. Prodromidis. *Management of Intelligent Learning Agents in Distributed Data Mining Systems*. PhD thesis, Department of Computer Science, Columbia University, New York, NY, 1999.

[37] A. L. Prodromidis and S. J. Stolfo. Mining databases with different schemas: Integrating incompatible classifiers. In G. Piatetsky-Shapiro R Agrawal, P. Stolorz, editor, *Proc. 4th Intl. Conf. Knowledge Discovery and Data Mining*, pages 314–318. AAAI Press, 1998.

[38] A. L. Prodromidis and S. J. Stolfo. Pruning meta-classifiers in a distributed data mining system. In *Proc of the KDD'98 workshop in Distributed Data Mining*, pages 22–30, New York, NY, August 1998.

[39] A. L. Prodromidis and S. J. Stolfo. Pruning meta-classifiers in a distributed data mining system. In *Proc of the First National Conference on New Information Technologies*, pages 151–160, Athens, Greece, October 1998. Extended version.

[40] A. L. Prodromidis and S. J. Stolfo. Cost complexity-based pruning of ensemble classifiers. Technical Report, CUCS-028-99, 1999.

[41] A. L. Prodromidis, S. J. Stolfo, and P. K. Chan. Effective and efficient pruning of meta-classifiers in a distributed data mining system. Technical report, Columbia Univ., 1999. CUCS-017-99.

[42] A.L. Prodromidis and S.J. Stolfo. A comparative evaluation of meta-learning strategies over large and distributed data sets. In *Workshop on Meta-learning, Sixteenth Intl. Conf. Machine Learning*, pages 18–27, Bled, Slovenia, August 1999.

[43] A.L. Prodromidis and S.J. Stolfo. Minimal cost complexity pruning of meta-classifiers. In *Proc. Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, Orlando, FL, July 1999.

[44] A.L. Prodromidis and S.J. Stolfo. Cost complexity-based pruning of ensemble classifier. *Knowledge and Information Systems*, 2001. In press.

[45] F. Provost and T. Fawcett. Analysis and visualization of classifier performance: Comparison under imprecise class and cost distributions. In *Proc. Third Intl. Conf. Knowledge Discovery and Data Mining*, pages 43–48, 1997.

[46] F. Provost and T. Fawcett. Robust classification systems for imprecise environments. In *Proc. AAAI-98*. AAAI Press, 1998.

[47] F. Provost and D. Hennessy. Scaling up: Distributed machine learning with cooperation. In *Proc. AAAI-96*. AAAI Press, 1996. 74-79.

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

[49] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[50] Waterhouse S. R. and Robinson A. J. Classification using hierarchical mixtures of experts. In *IEEE Workshop on Neural Networks for Signal Processing IV*, pages 177–186, 1994.

[51] R. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–226, 1990.

[52] J. C. Shafer, R. Agrawal, and M. Metha. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd Int'l Conf. on Very Large Databases*, Bombay, India, September 1996.

[53] S. Stolfo, A. Prodromidis, S. Tselepis, W. Lee, W. Fan, and P. Chan. JAM: Java agents for meta-learning over distributed databases. In *Proc. 3rd Intl. Conf. Knowledge Discovery and Data Mining*, pages 74–81, 1997.

[54] Volker Tresp and Michiaki Taniguchi. Combining estimators using non-constant weighting functions. *Advances in Neural Information Processing Systems*, 7:419–426, 1995.

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