View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Montana

University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, & Professional Papers

Graduate School

2003

Object-oriented design and implementation of the genetic ensemble feature selection method.

Lee Stuart Slater
The University of Montana

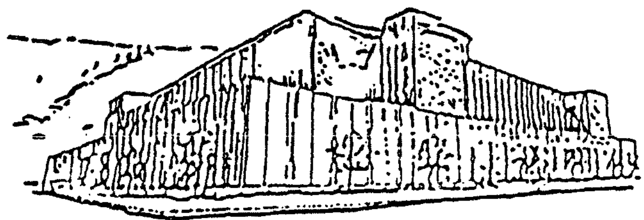
Follow this and additional works at: https://scholarworks.umt.edu/etd

Let us know how access to this document benefits you.

Recommended Citation

Slater, Lee Stuart, "Object-oriented design and implementation of the genetic ensemble feature selection method." (2003). *Graduate Student Theses, Dissertations, & Professional Papers*. 5098.
https://scholarworks.umt.edu/etd/5098

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



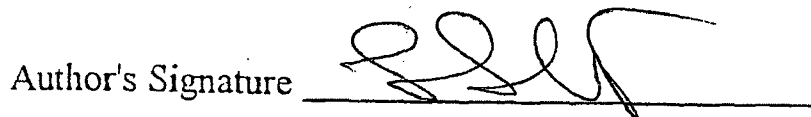
Maureen and Mike
MANSFIELD LIBRARY

The University of **MONTANA**

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

** *Please check "Yes" or "No" and provide signature* **

Yes, I grant permission ___X___
No, I do not grant permission ______

Author's Signature 

Date 5/29/03

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.

AN OBJECT-ORIENTED DESIGN AND IMPLEMENTATION OF THE

GENETIC ENSEMBLE FEATURE SELECTION METHOD

by

Lee Stuart Slater

B.S. Montana State University, 1990

M.S. Montana State University, 1994

presented in partial fulfillment of the requirements

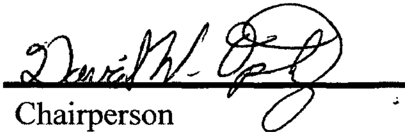
for the degree of

Master of Science

The University of Montana

April 2003

Approved by:



Chairperson

Dean, Graduate School

Date

UMI Number: EP40562

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI EP40562

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Slater, Lee S. M.S., April 2003 Computer Science

An Object-Oriented Design and Implementation of the Genetic Ensemble Feature Selection Method

Director: David Opitz DWO

A goal of modern machine learning research is to find computational classification and regression methods which generalize well. The Genetic Ensemble Feature Selection (GEFS) method uses a genetic algorithm to evolve feature sets that produce predictors that are accurate yet diverse which when combined, produce a concept which generalizes well. Herein an object-oriented design of the GEFS method is modeled in the Unified Modeling Language (UML) and implemented in C++. The system embedding the GEFS method, named the Machine Learning System, provides a platform to conduct various machine learning experiments. As implemented, the system performs the percentage train/test and n-fold cross-validation experiments using GEFS and single predictor learning methods with neural network predictors. The test set errors of the GEFS and single predictor learning methods with the 10-fold cross-validation experiment were computed over a sample of datasets with encouraging results.

Table of Contents

I. Introduction .... p. 1

II. The GEFS Method .... p. 3

III. Object-Oriented Design and Implementation .... p. 7

A. The *commands* Class .... p. 9

B. The *Patterns/patterns* Class .... p. 10

C. The *experiment* Class .... p. 11

D. The *learningMethod* Class .... p. 12

E. The *GEFS* Class .... p. 14

F. The *learner* Class .... p. 15

IV. Object Interaction and System Operation .... p. 17

A. System Operation of *main()* .... p. 18

B. System Object Instantiation and Parameter Assignment .... p. 19

C. System Operation When Performing Experiments .... p. 27

V. Results and Discussion .... p. 30

VI. Conclusions .... p. 35

VII. Bibliography .... p. 35

List of Tables

Table 1. Selected Datasets and the Associated Networks Parameters Used In Study .... p. 31

Table 2. Computed Test Set Errors (in %) for Machine Learning System on Selected UCI Datasets .... p. 33

List of Figures

Figure 1. A predictor ensemble .... p. 1

Figure 2. The GEFS algorithm .... p. 4

Figure 3. Class Diagram for Machine Learning System .... p. 8

Figure 4. Sequence Diagram for *main()* .... p. 19

Figure 5. Sequence Diagram for Object Instantiation and Parameter Assignment .... p. 21

Figure 6. Sequence Diagram for *experiment* Operation .... p. 28

I. Introduction

The main goal of modern machine learning research is to create learning methods with high generalization accuracy. In the early 1990's ensemble learning methods were developed that had better generalization than single predictors. These methods produced a collection of predictors each separately trained by an inductive learning algorithm whose output was combined to form the prediction (Figure 1). It was shown that for learned ensembles to generalize well, the predictors of the ensemble need to be both accurate and diverse (i.e. make their errors over different subspaces of the example space).

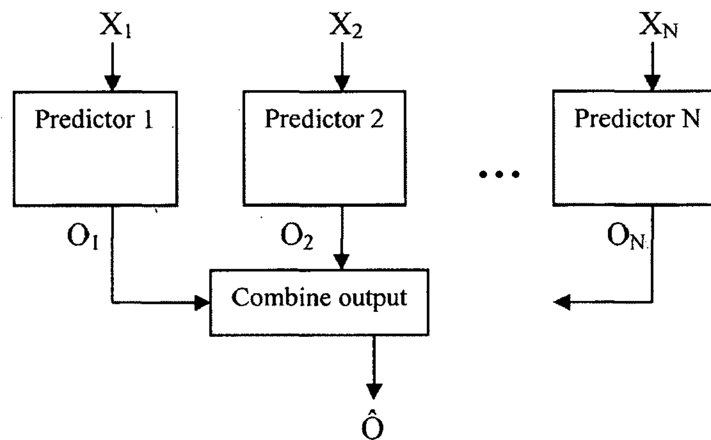


Figure 1. A predictor ensemble.

Also of interest to the machine learning community and statistical data modelers are feature selection methods. Given a set of examples, these methods determine a subset of the features that is sufficient to accurately predict a regression or classification of other instances having structure similar to the examples. These techniques can be successful in eliminating redundant or irrelevant attributes (features) or in finding transformations of the attributes of the data sets being modeled. Irrelevant features in a dataset can lead to

poor generalization and computational inefficiency when inducing a predictor. Feature selection methods have predominately been applied with single predictor models where the goal was to find the optimal feature set relevant to both the learning task and selected inductive learning algorithm. This type of selection method is termed a wrapper method. Another type of selection method, termed a filtering method, acts as a preprocessor of the dataset by determining a feature set without directly attempting to optimize a learning method. By producing a good feature set, improvements in predictor performance and a better understanding of the underlying concept generating the instances can be achieved.

The union of these two ideas forms the concept of *ensemble feature selection* - the selection of feature sets, each set representing the inputs for a predictor of the ensemble, that produce accurate yet diverse predictors over the example space (1). This task is more difficult than traditional feature selection in that the feature sets must in also promote diversity among the ensemble's predictors. The Genetic Ensemble Feature Selection (GEFS) method (1) is a machine learning technique that determines feature sets that produce accurate but diverse predictors within an ensemble. The unique approach of GEFS is that this is accomplished by evolving the feature sets to optimality with a genetic algorithm (GA). GAs are known to effectively search large search spaces and the search space of the required feature sets is large for non-trivial problems. The GEFS algorithm is detailed in the next section.

The GEFS method is showing great utility in many machine learning applications such as image analysis problems (2). The initial implementation of the GEFS method provided a successful prototype and much was learned about the behavior of the method in different machine learning contexts (3). However, the initial design was constructed

using a structured programming paradigm and resulted from the extension of a research code consisting of 5 separate machine learning systems that had evolved over 10 years. Thus, the maintenance of this code was difficult. To efficiently further develop the GEFS method in a stable, maintainable manner, the need to adopt the object-oriented paradigm was recognized. The objective of this project is to develop an object oriented design and implementation for the GEFS method which could be easily maintained yet largely retain the efficiency of the original implementation written in C.

II. The GEFS Method

The main intent of the GEFS method is to select sets of features where each set of features are the inputs to a predictor of an ensemble such that the predictors, once trained, are accurate yet diverse. Since the search space of the feature sets satisfying this property is large, a GA, having been shown to be an effective global optimization technique, is a logical choice to accomplish this. A hill-climbing strategy is better suited to refine an already near optimal solution and thus is not as appropriate here since the search space should be sufficiently explored before being exploited. To this end, the main structure of the GEFS algorithm follows a GA formalism where the population to be optimized are sets of features having a 1:1 correspondence to the predictors of the ensemble. The GEFS algorithm is summarized in Figure 2.

In the initial implementation of GEFS, neural networks were used as predictors. In this object-oriented implementation, the type of predictors allowed are very general and an ensemble can comprise a mixture of different predictor types. The requirements of a predictor (learner) will be discussed in the implementation section.

GOAL: Find a set of features (inputs) to create an accurate yet diverse predictor ensemble.

1. Using varying inputs, create and train the initial population (ensemble) of N predictors.
2. Until a termination criterion is met
   a) Apply genetic operators to create new predictors, adding them to the population.
   b) Assess the accuracy of each predictor over the training examples.
   c) Assess the diversity of each predictor with respect to the current population.
   d) Normalize the accuracy and diversity scores of all predictors in the population.
   e) Calculate the fitness of each population member.
   f) Prune the population to the N fittest predictors forming the new ensemble.
   g) Adjust λ.- the weight of the diversity term within the fitness.

Figure 2. The GEFS Algorithm

The GEFS method creates an initial population (ensemble) of predictors such that the inputs of each predictor are generated by randomly selecting a subset of the features. First, the number of features (inputs), N_i, for each predictor is randomly selected. This is chosen, independently and uniformly for each predictor, to be between 1 and twice the number of the original features in the dataset. Then the N_i features are randomly chosen with replacement. Consequently, some features may have multiple occurrences while others may not exist within a feature set. This allows for certain features to better survive during evolution and may reinforce the influence of the feature within certain types of predictors. The initial ensemble is then trained using the learning method characteristic of the predictor type. A validation set can be used during training if desired but this is defined within the scope of the predictor learning method and is independent of GEFS. The accuracy, diversity, and fitness (defined below) are computed for each predictor of the initial population before the ensemble evolution begins.

During ensemble evolution, new predictors are continually added to the existing population. These are created by applying the crossover and mutation operators to feature sets of selected members of the population. These members can be selected at random or proportional to fitness. The type of the new predictor(s) created is determined by the type of their parent predictor(s) and are trained with their respective learning method. The crossover operator defined for GEFS uses dynamic-length, uniform crossover but a single-point crossover has also been implemented. Each feature in both parent's subset is randomly placed in the feature set of one of the two children. This give the offspring feature sets dynamic length and may be larger or smaller than the selected parent's feature set size. However, each child is required to have at least one feature which is randomly transferred from the other child if necessary. The GEFS mutation operator is defined traditionally as randomly selecting a small percentage of features to change to distinct, randomly chosen features.

After new predictors are trained and added to the population, GEFS is defined to compute the predictor's accuracy score over the training set. Accuracy could also be scored over a validation set but this has not been implemented. Additionally, GEFS computes the predictor's diversity score with respect to the current, expanded population. The accuracy and diversity scores are individually normalized and GEFS computes each predictor's fitness as:

$$\text{Fitness}_i = \text{Accuracy}_i + \lambda \text{ Diversity}_i$$

where the parameter λ defines the tradeoff between the accuracy and diversity. The accuracy of each predictor is implemented as one minus the predictor error where the predictor error is defined according to the predictor type. The diversity of each predictor

is defined to be the average difference between the predictor output and the expanded ensemble output corresponding to the current population. Normalizing the accuracy and diversity individually so that the predictor's scores range between 0 and 1 allows λ to have the same meaning across domains.

It is not clear to what value λ should be set and its value may vary during the course of ensemble evolution to achieve the goal of accurate yet diverse predictors. As implemented, λ is automatically adjusted based on approximate derivatives (finite differences) of the ensemble error $\hat{E}$, the average population error $\bar{E}$, and the average diversity D within the ensemble. The condition for adjusting λ is the following: while $\hat{E}$ is decreasing, λ remains unchanged; otherwise a) if $\bar{E}$ is not increasing and the population diversity D is decreasing then increase λ; or b) if $\bar{E}$ is increasing and D is not decreasing then decrease λ. The default initial value of λ is 1.0 and changes by 10% of its current value.

Although the population expands temporarily as new predictors are evolved and included in the population, at the end of each iteration of GEFS, the population is pruned to the N fittest population members. This pruned population then forms the current ensemble. At this point, the current ensemble can be evaluated on a set of test examples or output to inspect predictor structure. Because GEFS continually considers new predictors to include in the ensemble during its operation and at anytime during its operation (at the end of an iteration), the GEFS ensemble represents the "best" ensemble evolved so far, it is termed an "anytime" learning algorithm. Such a learning algorithm should produce a good concept quickly then continue to search concept space for better concepts. This behavior is observed in GEFS and will be discussed in the results section.

In common GA operation, the GA usually terminates when the population fitness does not change over time. As currently implemented, the termination condition of GEFS is whether the number of new predictors to be searched (a user-defined parameter) has been reached. This allows further searching for a global optimum even when the presence of a local optimum has been detected. It also gives the user more control as when to terminate the algorithm and test or output the current ensemble.

III. Object Oriented Design and Implementation

In addition to the GEFS method, a complete system was designed and implemented to provide a platform to perform various machine learning experiments. The implementation language for the system is C++, chosen for its object-oriented expressiveness and efficient executable code. The collection of developed C++ classes in this project is referred to as the *Machine Learning System*. To document the object structure and sequence of operation using the UML (Unified Modeling Language), the system was reversed-engineered with the software suite Rational Rose (4).

A class diagram (5) shown in Figure 3 provides the static view of the class structure of the system at a high level of abstraction. The five main classes of the system are *experiment*, *learningMethod*, *Patterns*, *commands*, and *learner*. The classes *learningMethod*, *experiment*, and *learner* are abstract base classes and form key interfaces for the system design. As illustrated for the GEFS class in Figure 3, derivation from these interfaces allows the development of many different classes giving the system inherent flexibility for modification.

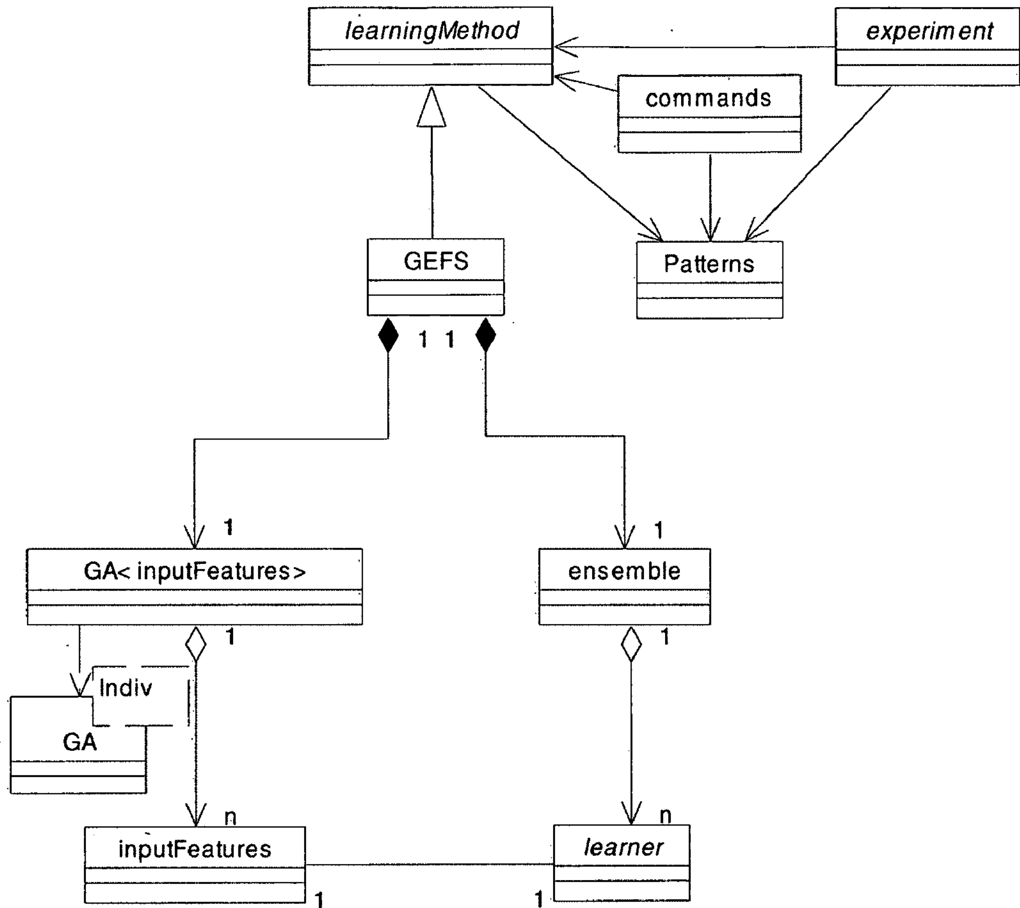


Figure 3. Class Diagram for Machine Learning System Highlighting the GEFS Class

A. The *commands* Class

The *commands* class is responsible for scanning the command file and initializing the objects of the system with the parameters set in the command file. The command file (given a .cmd extension) specifies the machine learning experiment to perform. The operation of the *commands* object processes the command file in one pass. A corresponding method of *commands* scans for and processes the following sections of the command file: seed, Patterns, experiment, learningMethod, and learners. The central feature of the implementation of these methods is the use of *map<string,fnptr>* STL (standard template library) containers to hold function pointers to external methods (nonmember functions). Each allowable class of the system has a corresponding external method defined. The initial string encountered in a section of the command file triggers the appropriate external method to execute. The execution of this external method initializes a function pointer to a constructor of the corresponding class. It also initializes another map containing function pointers to external methods that set parameters on an instantiated object of the class. The assigned constructor is executed with the specified object instantiated. Each subsequent parameter string read from the command file triggers the execution of the appropriate method within the parameter map to assign the parameter to the instantiated object. When one develops a new experiment, learning method, or learner for the system, the proper external methods and constructor must be coded. This is easily done by analogy to the existing external methods and constructors.

B. The *Patterns/patterns* Class

The *Patterns* class is a base class for a more functional *patterns* class that internally represents and manipulates the patterns or examples (stored in a file having a .pat extension) representing known instances of a classification or regression problem. The base class *Patterns*, contains the data structures which hold the complete set of X attributes and T attributes read from the .pat file, adopting the instance notation of Mitchell (6). The derived class *patterns* implements a scheme to designate which patterns, X attributes, and T attributes are "active." The active set of patterns, X attributes, or T attributes are a subset of the complete set of patterns, X attributes, or T attributes that can be accessed by a particular object. This convention allows the implementation of the learners and learning methods that don't have to perform their own partitioning of the patterns, X attributes or T attributes. With this convention, it is not necessary to create a separate object containing the subset either. In this system, active patterns and active X attributes are used. For example, when a learner receives a message to train itself, the *patterns* object is passed to the learner with the training set patterns previously set as active. The learner only "sees" these patterns during training and does not have to internally partition the patterns itself. Notice that this doesn't preclude the use of a validation set selected from the training patterns by the learner. Likewise, when a *GEFS* learning method object directs its ensemble learners to train, the X attributes of the patterns corresponding to input feature set of the learner are selected as active.

C. The *experiment* Class

The *experiment* class is an interface for the machine learning experiment to perform. This provides the developer the opportunity to use new types of machine learning experiments. The experiments that derive from the *experiment* abstract class must implement the following functionality through virtual methods:

void *initExpt*(learningMethod* lm, Patterns* p)

- create or read train/test sets from patterns, initialize output objects

void *conductExpt*(learningMethod* lm, Patterns* p)

- initialize, refine, and output the results from the computed hypothesis

void *finalizeExpt*(learningMethod* lm, Patterns* p)

- write the experiment parameters, train/test sets

istream& *read*(istream& is)

- code to reconstruct the *experiment* object reading from an input stream

ostream& *write*(ostream& os)

- code to write the *experiment* object to a output stream

There are two classes derived from the experiment class, the *CV* class and the *PTT* class. The *CV* class and the *PTT* class implement the n-fold cross-validation and percentage train and test experiments, respectively. In the *PTT* class, *initExpt()* is defined to randomly select the specified percentage of training patterns, the remaining being designated test patterns. These two sets of pattern indices are stored in two arrays of integers with the number of training patterns retained in an integer. The *conductExpt()* method of *PTT* is defined to perform a single hypothesis initialization, refinement, finalization, and results output. The hypothesis computation is dictated by the virtual

implementation of the dynamically bound *learningMethod* object discussed in the next section. Whereas in the *CV* class, *initExpt()* is defined to create a random partition of the patterns. Each subset of this partition is the test set of a fold of the cross-validation. The remaining subsets are combined and randomized to form the training set for the fold. The test set partition of pattern indices is stored in a single integer array and the training set pattern indices for all folds are stored in another integer array. The number of test set and training set indices for each fold are stored in two integer arrays. For the *CV* class, *conductExpt()* is defined to perform n iterations (one iteration for each fold) of the hypothesis initialization, refinement, finalization, and results output. The other methods of the interface are defined appropriately to write and read the differing train/test sets and associated parameters. These differences in operation are reflected in different implementations of the virtual methods of the *experiment* interface while the *main()* function, which calls some of these methods, remains unchanged. Additional subclasses that derive from *CV* and *PTT* have been defined. *CVevaluate* and *PTTevaluate* are classes which have slightly modified *conductExpt()* methods in that they do no hypothesis refinement, only initialization (i.e. read from file) and then output the results of the evaluation of the initial hypothesis on selected training/test sets of patterns.

D. The *learningMethod* Class

The *learningMethod* class is the interface for a learning method that a user may implement. In developing a learning method for this system, the developer must implement the following functionality through virtual methods:

void *initHypothesis*(Patterns* p)

- initialize the learning method, construct the initial ensemble

void *refineHypothesis*(Patterns* p)

- refine the hypothesis, evolve the ensemble

void *evaluateHypothesis*(Patterns* p, double2D& o)

- evaluate the hypothesis over the patterns, store output

void *finalizeHypothesis*(Patterns* p)

- write hypothesis to file

void *appendLearners*(int numL, learner** L)

- append instantiated *learner* object pointers to learning method

istream& *read*(istream& is)

- code to reconstruct the *learningMethod* object from an input stream

ostream& *write*(ostream& os)

- code to write the *learningMethod* object to a output stream

In this project, the *GEFS* class and the *singleLearner* class are derived from the *learningMethod* interface. The virtual method implementations of the *GEFS* class are more complicated than the *singleLearner* definitions. The *initHypothesis()* method of the *GEFS* class is defined to compute or *read()* the initial ensemble. This involves the generation of learners having random number of inputs (as described in the GEFS method section), training these learners, computing the outputs of the ensemble over the training patterns, and computing the fitness of each member of the ensemble. Within the *singleLearner* class, the *initHypothesis()* initializes or *read()*s the individual *learner* object. In *GEFS*, *refineHypothesis()* must evolve the ensemble which is simply handled by a method invocation but the outputs over the training patterns must be computed prior

to evolution in the event that the hypothesis has been *read()* from a file. In *singleLearner*, the learner invokes *train()* to refine the hypothesis (see below for interface definitions of the *learner* class). The *finalizeHypothesis()* method is similar in both *GEFS* and *singleLearner* but the actually *write()* of the *GEFS* object to file is more involved since both the *GA<inputFeatures>* object and the *ensemble* object must be written (see the *GEFS* structure described next). The former involves writing each *inputFeatures* object and the latter involves writing each *learner* object. A similar level of complexity is involved in reconstructing the GEFS object through *read()*. In *appendLearners()*, the GEFS method must append the learner to the ensemble but also instantiate a new *inputFeatures* object to be appended to the GA population while the *singleLearner* object only needs to assign a pointer.

E. The *GEFS* Class

The high-level architecture of the *GEFS* class is also shown in Figure 3. The composite objects *inputGA* and *learners* are the two main components. These objects are instantiated by value from the classes *GA<inputFeatures>* and *ensemble*. The *GA<inputFeatures>* class binds the *GA<Indiv>* class and the *inputFeatures* class. The *GA<Indiv>* template class is a genetic algorithm class that implements a variation of the steady-state (incremental) algorithm for population evolution. The minimum number of evolved individuals it adds to the population at each iteration of the GEFS algorithm is parameter controlled. The default is to add a minimum of one individual ensuring that only a mutated individual or two crossover progeny are added. Once the minimum number of individuals are added, the iteration terminates. This algorithm probabilistically

chooses to perform a mutation or crossover operation based on a mutation probability parameter. Population individuals can be selected for genetic operations randomly or by fitness proportionality. A generational algorithm has also been implemented but not tested. The instantiation of *GA<inputFeatures>* then contains a population of *inputFeatures* objects (by reference). The *inputFeatures* class (or any class that binds to *GA<Indiv>*) must implement the genetic operators - *mutate()*, *crossover()*, and the input/output operators - *operator<<()*, and *operator>>()*. The mutate operator must mutate the calling object and the crossover operator takes an individual object as an argument and produces the progeny within the calling and argument objects. The *inputFeatures* class implements the feature set as an array of integers where the integers correspond to the numbering of the X attributes within the original pattern file read. The other main, composite object in the *GEFS* class, *learners*, is an instantiation of the *ensemble* class. The *ensemble* object contains (by reference) the ensemble component *learner* objects. In this implementation, pointers to *learner* base class objects can address any object of a class that derives from *learner* and thus various learners, in addition to *stdNN*, can be implemented for use in the system. The *learner* class is discussed below. Note from the class diagram association that the *inputFeatures* objects are in a 1:1 correspondence with the *learner* objects. In the *singleLearner* class only a single pointer to a *learner* object is contained.

F. The *learner* Class

The *learner* class is an abstract base class acting as an interface for various learners or predictors. This class gives the *GEFS* class the flexibility to use other learners

(or mixtures of learners) within its ensemble though dynamic binding of the derived objects. For a learner to be developed for the system, it must implement the following functionality through virtual functions:

learner createNewLearner*()

- create an new instantiation of the object with default constructor

void *initLearner*(int numInputs, int numOutputs)

- initialize a learner to have the specified number of inputs and outputs

void *train*(Patterns* p)

- train the learner using the corresponding learning procedure

void *test*(Patterns* p, double2D& o)

- compute and store output of the learner on the patterns

double *error*(Patterns* p, double2D& o)

- compute appropriate error of the learner given outputs and T attributes

istream& *read*(istream& is)

- code to reconstruct the *learner* object from an input stream

ostream& *write*(ostream& os)

- code to write the *learner* object to a output stream

~learner()

- virtual destructor for the learner

The *stdNN* class derives from the *learner* interface and implements a standard 1 or 2 layer neural network. In this implementation, the network is represented by two arrays – one of network weights and one for network bias. The weights can be indexed in the array by knowing the number of input units, hidden units, and output units. The

createNewLearner() method instantiates a new *stdNN* object returning a *learner* pointer to the object. The *initLearner()* method allocates the required array storage for the network and initializes the number of input units and output units – quantities common to all learner types. The number of hidden units can be specified in the command file or a default value used. The *train()* method performs the backpropagation algorithm given in Mitchell (6) and the *test()* method evaluates the network in a feedforward manner. The *error()* method computes the sum of squared errors of the computed outputs and targets given as the T attributes in the *patterns* object.

IV. Object Interaction and System Operation

While class diagrams provide a static, structural view of a system, interaction diagrams provide a dynamic view of the object interaction during system execution (5). There are two types of interaction diagrams commonly used to model object-oriented behavior – sequence diagrams and collaboration diagrams. Sequence diagrams highlight the time ordering of message passing between objects during execution while collaboration diagrams better exhibit the object organization in message passing during execution. Multiple interaction diagrams can be used to model and document different parts of the system execution and at differing levels of abstraction. To demonstrate the execution of the *Machine Learning System*, a sequence diagram to describe the high-level behavior of *main()* and separate sequence diagrams to further detail the actions of *main()* are employed.

A. System Operation of *main()*

Figure 4 shows the sequence diagram for the *main()* function at the highest level of abstraction. Two actions are performed by *main()*: 1) processing the command file wherein the objects of the system are instantiated and 2) performing the machine learning experiment using the instantiated system objects. The first seven actions performed within *main()* are responsible for processing the command file and instantiating the system objects while the last three actions direct the *experiment* object to perform the specified machine learning experiment. Initially, *main()* instantiates a *commands* object, *c*. Then, *main()* directs *c*, through passed messages to *c*, to process each section of the command file. Some sections of the command file are necessary while others are optional. The methods of *commands* can detect the presence of the necessary sections and error messages followed by system termination result if absent. The actions performed by the methods are evident from their names. Each method generally scans the command file for the appropriate section heading, instantiates the corresponding object(s) of the section being processed and assigns any parameters specified in the section to the instantiated object. The implementation of these methods was discussed in the previous section. The operation of these methods is further detailed in the sequence diagram in Figure 5. While all messages received by *c* instantiate objects of the system (except *assignRandNumSeed()*), only the *experiment* object instantiation is depicted in Figure 4 since its methods are directly invoked within *main()*. The *experiment* virtual methods perform the machine learning experiment. Their operation is dictated by the implementation of dynamically bound object but generally consists of initializing the experiment, conducting the experiment, and finalizing the experiment. The specific

implementation of these methods for the CV and PTT classes were previously described and their general operation is further detailed in the sequence diagram given in Figure 6.

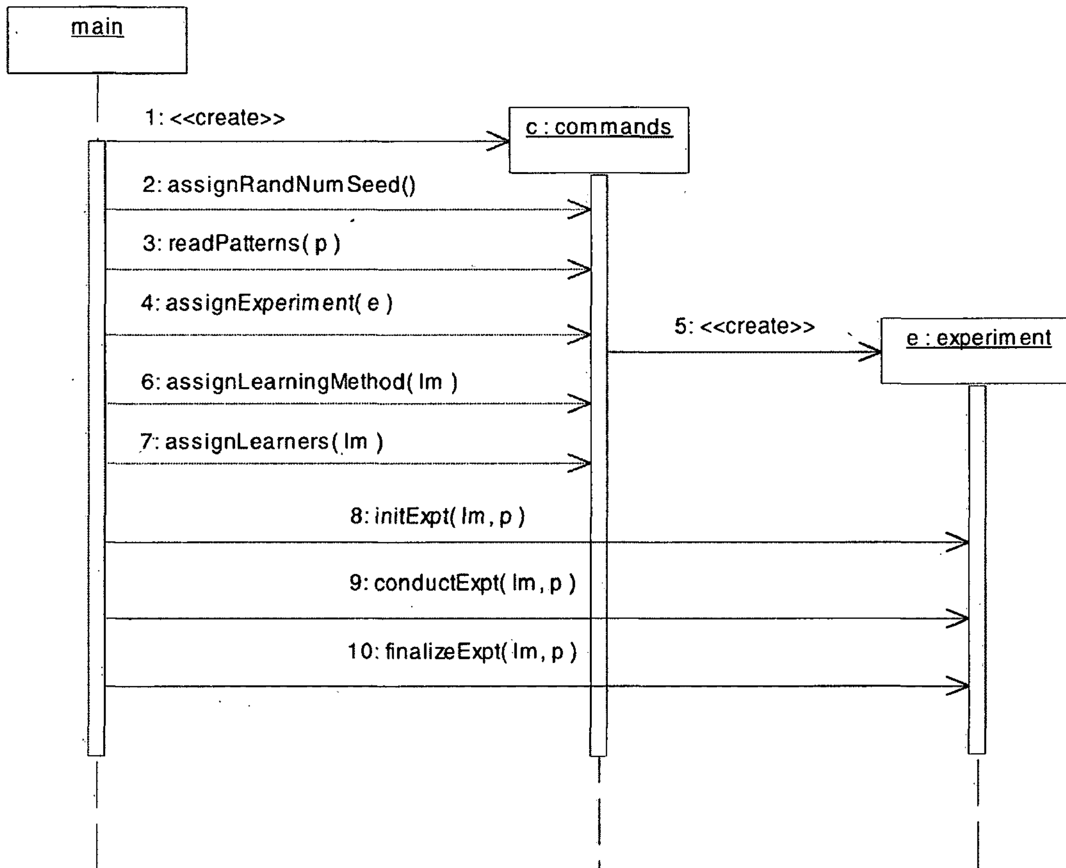


Figure 4. Sequence Diagram for *main()*

B. System Object Instantiation and Parameter Assignment

Another sequence diagram that details command file processing is shown in Figure 5. After the commands object *c*, has been created and the command file scanned for an optional random number generator seed, *main()* sends a message to the *commands* object to *readPatterns()*. This method then creates a *Patterns* object, *p*. As implemented,

a *patterns* object (instantiated from a subclass of Patterns) is always specified in the command file but other classes can be derived from Patterns and used in the system. After *p* is instantiated, any parameters are read from the command file and assigned to the object. A list of the permissible *Patterns* parameters are:

patternsFile <string> : required

- the name of the patterns file, must include the .pat extension

numPatterns <int> : required

- total number of patterns in .pat file to read, assumed positive

numXAttribs <int> : required

- total number of X attributes in .pat file, assumed positive

numTAttribs <int> : required

- total number of T attributes in .pat file, assumed positive

continuousOutput <bool> : default 0

- are outputs (scalar or vector) continuous? (1 = yes, 0 = no)

There are no parameters specific to the *patterns* class. With the parameters assigned, *c* directs *p* to *allocPatterns()* and *readPatternFile()*. The first action allocates the storage needed to store the patterns having the specified dimensions. The second action reads the patterns from the specified .pat file. This fully initializes the *patterns* object for the machine learning experiment.

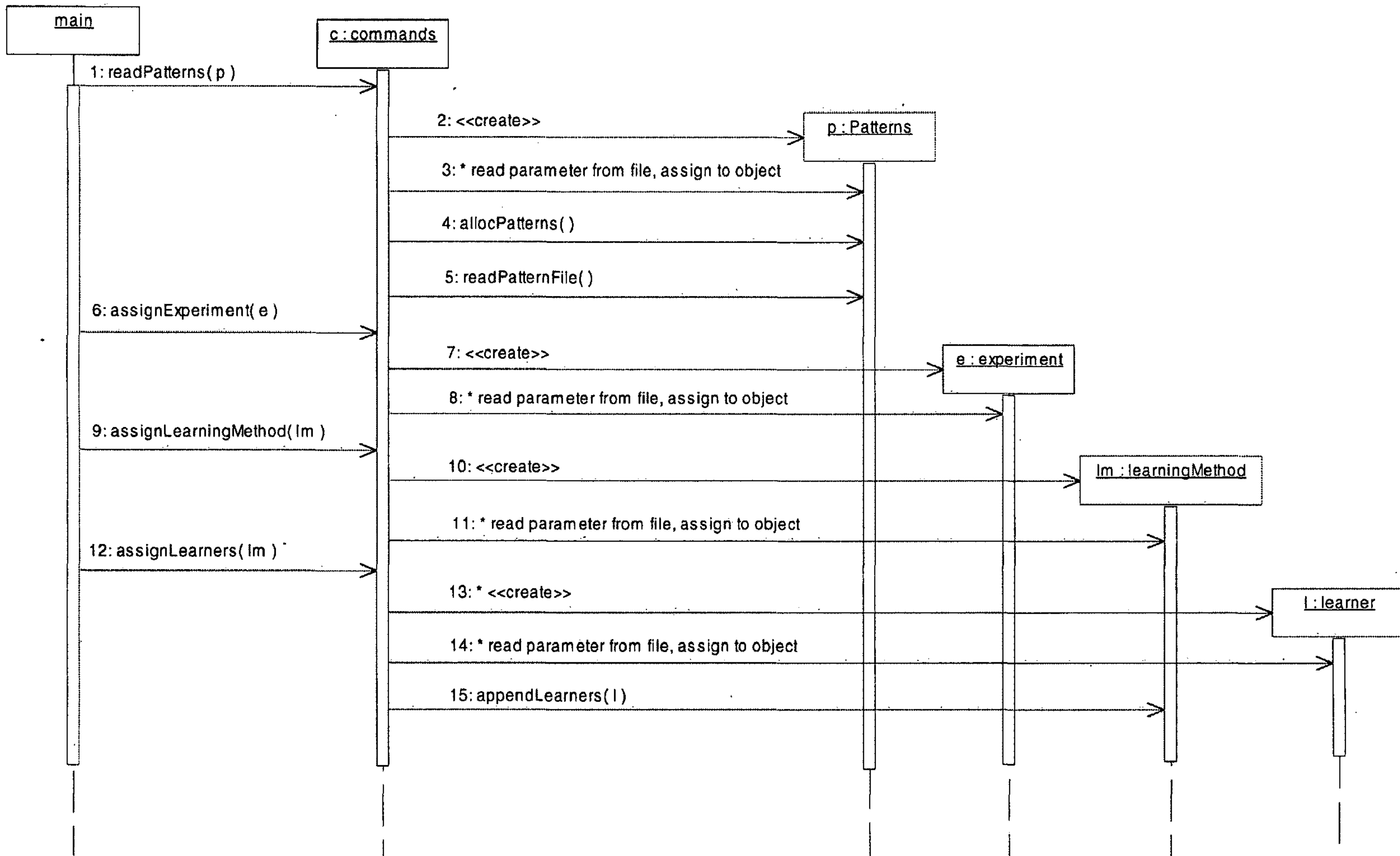


Figure 5. Sequence Diagram for Object Instantiation and Parameter Assignment

Next, *main()* directs *c* to *assignExperiment()*. The operation of this method instantiates the *experiment* object and assigns any parameters specified in the command file. The type of experiments currently defined in the system are:

PTT – performs a percentage train and test experiment.

PTTevaluate – performs a PTT experiment but no hypothesis refinement.

CV – performs an n-fold cross-validation.

CVevaluate – performs a CV experiment but no hypothesis refinement.

Some parameters are common to all experiments while other parameters are dependent on the experiment specified. The permissible parameters common to all experiments are:

evalInitHypothOnTrainPatts <bool> : default 0

- evaluates initial hypothesis on the training patterns, outputs error

evalInitHypothOnTestPatts <bool> : default 0

- evaluates initial hypothesis on the test patterns, outputs error

*evalRefinedHypothOnTrainPatts** <bool> : default 0

- evaluate refined hypothesis on training patterns, outputs error

*evalRefinedHypothOnTestPatts** <bool> : default 0

- evaluate refined hypothesis on test patterns, outputs error

outputInitHypothOnTrainPatts <bool> : default 0

- performed if *evalInitHypothOnTrainPatts* = 1, outputs predictions

outputInitHypothOnTestPatts <bool> : default 0

- performed if *evalInitHypothOnTestPatts* = 1, outputs predictions

*outputRefinedHypothOnTrainPatts** <bool> : default 0

- performed if *evalRefinedHypothOnTrainPatts* = 1, outputs predictions

*outputRefinedHypothOnTestPatts** <bool> : default 0

- performed if *evalRefinedHypothOnTestPatts* = 1, outputs predictions

readExptFromFile <bool> : default 0

- read *experiment* object from file

writeExptToFile <bool> : default 0

- write *experiment* object to file

inputFilename <string> : default "defaultInput"

- input filename to use if *readExptFromFile* = 1, appends .exp extension

outputFilename <string>: default "defaultOutput"

- output filename to use if *writeExptToFile* = 1, appends .exp extension

Note that "RefinedHypoth" parameters (denoted by an asterisk) are not defined for the *PTTevaluate* and *CVevaluate* experiments. The additional permissible parameter for the *CV* and *CVevaluate* experiment is:

numFolds <int> : default 10

- number of folds to perform in the n-fold cross validation experiment

The additional permissible parameter for the *PTT* and *PTTevaluate* experiment is:

percentTrain <double> : default 0.5

- percentage of patterns to use in training set

Next, *c* receives the message to *assignLearningMethod()*. This method of the *commands* object instantiates a *learningMethod* object, *lm*, and assigns any specified parameters to *lm*. The learning methods defined in the system are:

GEFS – Genetic Ensemble Feature Selection

singleLearner – a single learner of specified type

As with the experiment class, some parameters are common to all learning methods while others are applicable to the learning method specified. The parameters common to all learning methods are:

readLearnMethFromFile <bool> : default 0

- reads *learningMethod* object from file

writeLearnMethToFile <bool> : default 0

- writes *learningMethod* object to file

readParamsFromFile <bool> : default 1

- reads *learningMethod* parameters into object previously written to file
- will overwrite any new parameters specified in command file

multipleInputFiles <bool> : default 0

- specifies that multiple *learningMethod* input files will be read
- used if performing *CV* experiment with a *learningMethod* read
- appends integer file extension

multipleOutputFiles <bool> : default 0

- specifies that multiple *learningMethod* output files will be written
- used if performing *CV* experiment with a *learningMethod* write
- appends integer file extension

inputFilename <string> : default "defaultInput"

- filename to use if *readLearnMethFromFile* = 1, appends .lm extension

outputFilename <string> : default "defaultOutput"

- filename to use if *writeLearnMethToFile* = 1, appends .lm extension

There are no parameters exclusive to the *singleLearner* learning method. The parameters specific to the *GEFS* learning method are:

minSearchLength <int> : default 100

- minimum number of learners to add to initial ensemble during evolution.

minLearnersEvolved <int> : default 1

- minimum number of learners to add to ensemble each GEFS iteration

lambda <double> : default 1.0

- weight of diversity in fitness computation

dLambda <double> : default 0.1

- fraction of current lambda to add to lambda during lambda change

dLambdaFreq <int> : default 1

- number of GEFS iterations before test for lambda change

inFt_MaxNumRandomInputs <int> : default 2 * number of X attributes

- maximum number of features (learner inputs) to include in set

inFt_PercentMutatedInputs <double> : default 0.075

- percent of mutated features in set if mutation performed

inFt_CrossoverProb <double> : default 0.5

- probability that a feature will be crossed over in progeny

GA_MutationProb <double> : default 0.5

- probability evolution will perform a mutation instead of crossover

GA_SelectFitnessProp <bool> : default 0

- learner(s) to evolve is selected by proportionality to fitness

Finally, the *c* object is directed to *assignLearners()*. As denoted on the sequence diagram, multiple *learner* object instantiations are possible and this occurs if *GEFS* is the specified *learningMethod* object. Parameters also need to be read from the file and assigned to the multiple learners. The most efficient way of assigning parameters to multiple learners of a single type is to use a template learner. A template *learner* object of the corresponding type is instantiated and the parameters are read from the file and assigned to the template object. Then, copies of this template object are constructed and appended to a transient array of *learner* objects. Each learner type is processed in this manner (if a mixed ensemble of different learner types is specified) and the complete array is constructed. Then the *commands* object directs the *learningMethod* object to *appendLearners()* to the learning method using the complete array of learners as an argument. The *stdNN* learner class has been derived from *learner* for use with the system. In the class definition of *learner*, there are no parameters common to all learners, only parameters specific to the derived learner *stdNN*. The parameters specific to the *stdNN* learner are:

numEpochs <int> : default 100

- number of epochs to perform in backpropagation

learnRate <double> : default 0.1

- learning rate, η = step size in gradient descent optimization

momentum <double> : default 0.9

- momentum constant, α

randWtMag <double> : default 0.5

- magnitude of the random weight initialization

numEpochRand <int> : default = numEpochs, no randomization

- number of epochs to perform before randomizing the patterns

numInputUnits <int> : default = number of X attributes in *patterns* object

- number of input units in network

numHiddenUnits <int> : default max(5,max(numOutputUnits,numInputUnits/10))

- number of hidden units in network

numOutputUnits <int> : default = number of T attributes in *patterns* object

- number of output units in network

C. System Operation When Performing Experiments

Figure 6 provides a view into the operation of how the system performs a machine learning experiment. When *main()* passes a message to the *experiment* object *e* to *initExpt()*, *e* computes the train/test sets appropriate to the experiment. This can be done randomly or with the use of a random number generator seed to ensure a partition of the patterns into invariant train/test sets. Alternatively, the train/test sets may be read when the experiment is read from a file if the parameter *readExptFromFile* is set. This allows the saving and reusing of a particular partition of train/test set patterns within other machine learning computations. This also allows one to examine the behavior of a learning method on a particular partition of the patterns by creating the experiment file manually and reading it into the system. Also depicted on the sequence diagram, *e* instantiates some minor objects used to hold output predictions.

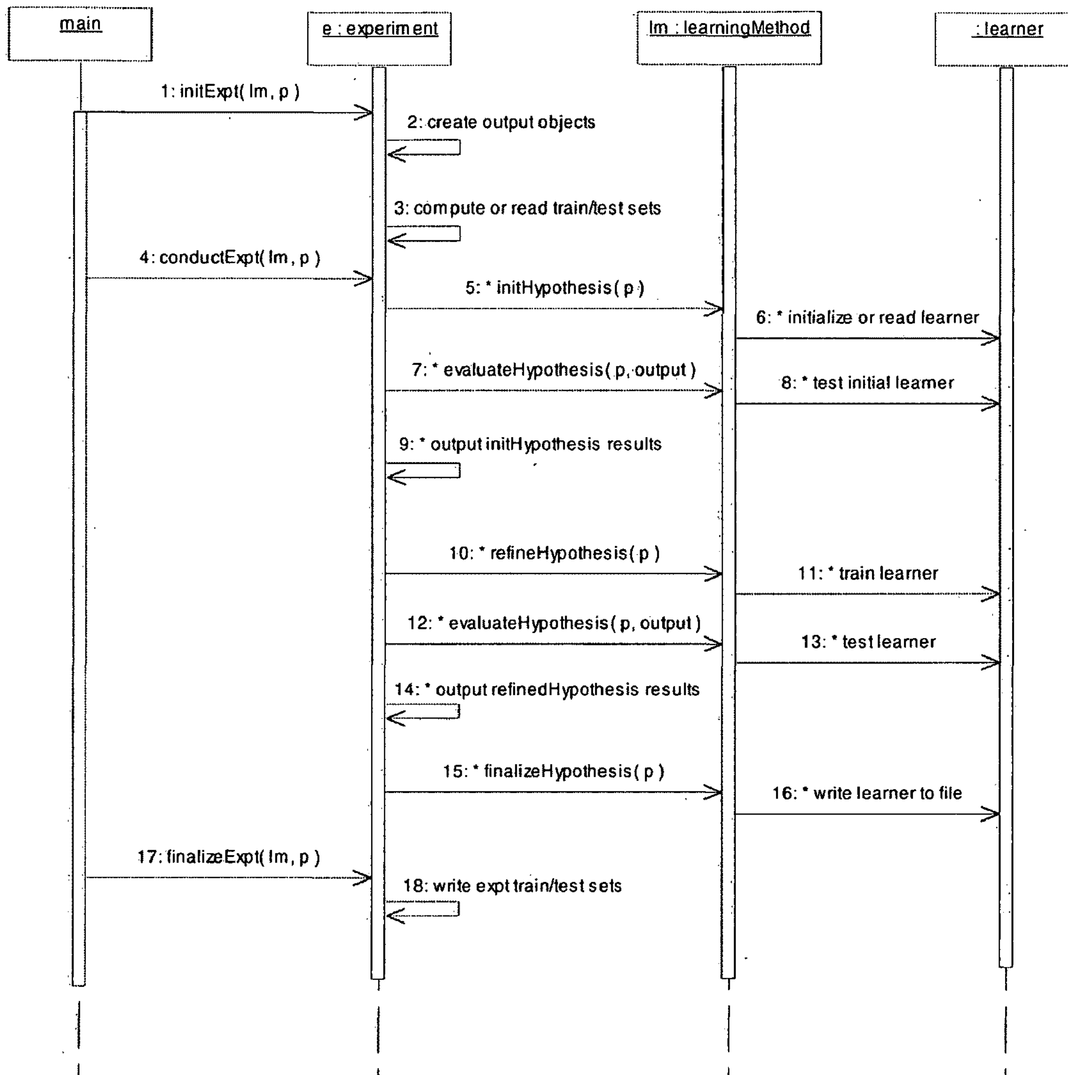


Figure 6. Sequence Diagram for *experiment* Operation

With the experiment initialized, *main()* directs *e* to conduct the experiment with the message *conductExpt()*. The operations performed within *conductExpt()* may be performed multiple times if a *CV* experiment is being conducted (one iteration per fold) or just a single time with a *PTT* experiment. The initial action is for *e* to pass a message to the learningMethod, *lm*, directing *lm* to *initHypothesis()*. Upon receiving this message,

lm initializes its hypothesis by preparing the initial *learner* objects as follows. If the *GEFS* learning method has been specified, multiple *learner* objects may be *read()* or *train()*ed. In this case, it is the *GEFS* ensemble that is initialized (but not evolved) and this requires that the *learner* objects of the ensemble be trained or read from a file. If a *singleLearner* learning method is specified, the referenced learner object is only initialized, but not trained. With an initial hypothesis constructed, it may be evaluated on the training and/or test set patterns. This action is performed through a message, *evaluateHypothesis()*, from the experiment *e* to learning method *lm*. Which patterns the hypothesis is evaluated on is specified during the command file processing by assigning the appropriate parameters of *experiment* (see above listing of parameters). To accomplish the hypothesis evaluation, the learning method *lm* instructs the composite initial learner(s) to *test()* on the selected patterns. The correctness of the initial hypothesis on the selected patterns is output and the actual predictions of the hypothesis may be output if specified. During the conductExpt() phase, the hypothesis is refined by *e* directing *lm* to *refineHypothesis()*. Note that this action is only performed during the *CV* and *PTT* experiments but not the *CVevaluate* and *PTTevaluate* experiments. This is accomplished in both *singleLearner* and *GEFS* by *lm* directing its composite learner(s) to *train()*. Previously, when the initial hypothesis was being computed, *GEFS* would train the learners but the not evolve the ensemble. During refinement, however, the *GEFS* ensemble evolves and during the course of evolution, new learners are continually being trained to refine the hypothesis. As previously discussed, the refined hypothesis is evaluated on the selected patterns by lm directing each of its learners to *test()*. The hypothesis correctness can be output as well as the predictions if specified. During the

last phase of the *conductExpt()*, the hypothesis is finalized by *e* directing *lm* to *finalizeHypothesis()*. This involves writing the hypothesis to file which is accomplished by *lm* writing each *learner* object to file. This is useful if one wishes to read in a refined hypothesis and evaluate it on another set of patterns or perform further refinement.

In finalizing the experiment, main directs *e* to *finalizeExpt()*. This involves writing some of *experiment* parameters and train/test sets to a separate file if the *writeExptToFile* parameter has been set. The experiment file can be read into another machine learning computation if one wants to use identical parameters and train/test sets.

V. Results and Discussion

The Machine Learning System was tested on the datasets listed in Table 1. These were originally obtained from the UCI Machine Learning Data Repository at the University of California at Irvine (7). The Repository represents a diverse set of machine learning problems and is commonly used to evaluate the efficacy of newly developed machine learning algorithms. The datasets used in this study represent a sampling of the Repository and were selected because the original implementation of GEFS was tested with them. All datasets used in this study represent only classification problems although the Machine Learning System is flexible enough to be easily modified to solve regression problems. There is a wide range in the number of cases or instances in the datasets and the number of inputs and outputs varies substantially among the datasets. Every dataset in Table 1 has no missing attributes and several datasets have vector-valued features composed of multiple continuous or discrete attributes. A discrete vector feature of N classes is represented by N binary attributes. When mapping to a neural network, each

attribute corresponds to an input of the network. The details of the feature structure can be obtained at the UCI Repository. The number of network units and the number of training epochs used in the study are also given in Table 1 and are identical to those used in testing the original implementation of GEFS.

Table 1. Selected Datasets and the Associated Network Parameters Used In the Study

Dataset			Features		Network			
	Cases	Classes	Continuous	Discrete	Inputs	Outputs	Hidden	Epochs
credit-a	690	2	6	9	47	1	10	35
credit-g	1000	2	7	13	63	1	10	30
Diabetes	768	2	9	-	8	1	5	30
Glass	214	6	9	-	9	6	10	80
heart-Cleveland	303	2	8	5	13	1	5	40
Hepatitis	155	2	6	13	32	1	10	60
house-votes-84	435	2	-	16	16	1	5	40
Hypo	3772	5	7	22	55	5	15	40
Ionosphere	351	2	34	-	34	1	10	40
Iris	159	3	4	-	4	3	5	80
kr-vs-kp	3196	2	-	36	74	1	15	20
Labor	57	2	8	8	29	1	10	80
Letters	20000	26	16	-	16	26	40	30
promoters-936	936	2	-	57	228	1	20	30
ribosome-bind	1877	2	-	49	196	1	20	35
Satellite	6435	6	36	-	36	6	15	30
Segmentation	2310	7	19	-	19	7	15	20
Sick	3772	2	7	22	55	1	10	40
Sonar	208	2	60	-	60	1	10	60
Soybean	683	19	-	35	134	19	25	40
Vehicle	846	4	18	-	18	4	10	40

The computational experiments consisted of 10 trials where each trial consisted of a randomly determined 10-fold cross-validation experiment. With cross-validation, the patterns are randomly partitioned into 10 nearly equal sized sets. Each set forms a test set for a fold of the experiment. Given a test set, the remaining sets are combined and randomized to form the training set for the fold. Each training set of patterns is used to induce a newly created learningMethod object at each fold of the experiment. The

correctness of the trial is the sum of the correct predictions of the test set patterns over all folds divided by the number of patterns. The reported mean error is 1 minus the sample mean of the correctness of the 10 trials obtained with *singleLearner* and *GEFS* learning methods using *stdNN* neural network learner(s). The *GEFS* learning method correctness was computed with the initial and refined hypotheses and *singleLearner* correctness using only the refined hypothesis. This results in the initial and refined (evolved) GEFS hypotheses evaluated on the same test set for each trial but a different test set for the singleLearner hypothesis. The neural network parameters used in the experiments are: learning rate = 0.15, momentum = 0.9, and a magnitude of 0.5 for randomly assigned weights having positive or negative sign. The same network parameters were used for both the *singleLearner* and *GEFS* ensemble experiments. The parameters for the GEFS learning method are: initial lambda = 1.0, a change of lambda of 0.6 when a change is warranted and a determination of whether to change lambda occurring at a frequency of every 5 iterations of the GEFS algorithm (see Figure 2). The GEFS parameters were fixed for all datasets considered.

The test set errors of the *singleLearner* and *GEFS* learning methods on the datasets are provided in Table 2. The average test set error and 95% confidence interval over 10 trials is reported for the single neural network predictor, the initial GEFS ensemble of 20 neural networks, and after searching 230 additional networks (for a total of 250 networks). In addition, the p-values from two different t-tests were reported. Of interest in this study is whether there is a statistically significant difference in mean test set error between the single learner and the GEFS method. The GEFS mean error used in the t-test is chosen to be the minimum error of the initial ensemble or after 250 networks

have been considered. This convention is adopted since it is observed that further evolution may not always produce a hypothesis with a better mean test set error. The t-test used in this case is a standard two-population test for a difference in population means (2-tailed) assuming unequal variance. This significance test is appropriate because the *GEFS* and *singleLearner* trials are evaluated on different test sets. The second question of interest is whether successful evolution has occurred to produce a hypothesis giving a different test set error than the initial ensemble. In this case, a paired difference test (two-tailed) is used and is appropriate since identical test sets are used for the initial and evolved hypotheses.

Table 2. Computed Test Set Errors (in %) for the Machine Learning System on Selected UCI Datasets

Dataset	Single Learner		GEFS Ensemble					
			Initial Pop		250 Networks		SL Diff	Evolution
	Mean	95% CI	Mean	95% CI	mean	95% CI	p	p
Credit-a	15.3	(14.7,15.9)	13.9	(13.6,14.2)	14.1	(13.9,14.2)	7.90e-04	.257
Credit-g	28.8	(28.2,29.4)	24.9	(24.3,25.5)	24.3	(23.9,24.8)	1.36e-09	.041
Diabetes	24.3	(23.3,25.3)	24.8	(23.9,25.8)	24.4	(23.6,25.0)	.896	.426
Glass	39.8	(38.0,41.6)	41.0	(38.9,43.1)	34.9	(33.8,35.9)	3.65e-04	1.14e-04
Heart-Cleveland	19.5	(19.1,19.9)	17.5	(16.7,18.4)	18.5	(17.6,19.2)	.001	.019
Hepatitis	18.3	(17.0,19.5)	17.2	(16.3,18.0)	19.0	(18.0,19.9)	.174	6.84e-04
House-votes-84	4.8	(4.6,5.1)	4.4	(4.1,5.0)	4.6	(4.1,5.0)	.048	.509
Hypo	6.7	(6.4,6.9)	6.9	(6.8,6.9)	5.1	(4.9,5.4)	1.11e-07	1.36e-06
Ionosphere	11.2	(10.3,12.1)	7.0	(6.5,7.4)	6.6	(6.4,6.9)	2.84e-06	.081
Iris	4.1	(3.0,5.2)	4.9	(4.3,5.4)	4.4	(3.6,5.2)	.640	.354
kr-vs-kp	2.6	(2.4,2.8)	3.0	(2.9,3.2)	0.99	(.62,1.4)	3.59e-06	4.55e-06
Labor	2.6	(1.9,3.4)	3.5	(2.6,4.4)	3.2	(2.5,3.8)	.331	.343
Letters	18.3	(18.0,18.5)	14.5	(14.4,14.7)	12.5	(12.4,12.7)	2.39e-16	3.74e-08
Promoters-936	5.3	(4.9,5.6)	4.1	(4.0,4.3)	5.1	(4.8,5.3)	5.75e-05	4.16e-04
ribosome-bind	9.4	(9.1,9.6)	8.1	(7.9,8.3)	8.3	(8.1,8.4)	5.25e-07	.306
Satellite	13.4	(13.2,13.5)	13.1	(12.8,13.4)	11.4	(11.2,11.6)	6.68e-12	6.13e-08
Segmentation	6.7	(6.5,6.9)	6.4	(6.0,6.7)	4.2	(4.0,4.3)	5.61e-13	6.83e-08
Sick	5.9	(5.7,6.1)	6.0	(5.9,6.1)	3.4	(3.1,3.8)	7.92e-09	1.40e-07
Sonar	18.7	(17.3,20.0)	16.7	(15.7,17.8)	16.0	(15.2,16.9)	.006	.304
Soybean	8.7	(8.2,9.1)	6.5	(6.2,6.8)	6.7	(6.4,7.1)	8.46e-07	.349
Vehicle	24.9	(24.1,25.6)	25.2	(24.8,25.6)	21.3	(20.9,21.8)	6.17e-07	1.57e-06

Examining the test set errors in Table 2, demonstrates a few trends. Using a neural network predictor with the *GEFS* learning method produced a lower mean test set error than the *singleLearner* in a majority of the datasets. In considering the p-values for the difference between the GEFS and single learner, 17 of 21 datasets show statistically significant differences having $p < \alpha = 0.05$. In all 17 cases, either the initial GEFS mean test error or the refined mean test error was less than the single learner mean test error.

A statistically significant difference between the initial ensemble and the refined ensemble having considered 250 networks over the same test set is demonstrated in 12 of the 21 datasets. The case of the refined mean test set error being less than the initial mean test set error provides strong evidence that evolution was successful in generating a more accurate hypothesis, capable of better generalization. This occurs in 9 of the 12 datasets and demonstrates successful anytime learning – the method produces a good initial concept and through evolution determines a concept possessing better generalization. In all nine cases, the refined mean test set error is less than the single learner mean test set error. In six of the nine cases, successful evolution occurred in the seven multi-class datasets. Although further experimentation is necessary, this suggests that successful GEFS evolution with these parameter settings may work better for multi-class datasets. The three cases for which the mean test set error of the evolved GEFS ensemble is greater than the initial ensemble suggests that further evolution didn't improve upon the initial hypothesis and may have found a less accurate hypothesis. This minor finding is most likely caused by inappropriately selected GEFS parameters for these datasets. It may indicate that the presumably high diversity of the random initial feature sets is important and the inappropriate GEFS parameters may decrease diversity

during evolution. It is probably not an initial ensemble dependent phenomenon for it occurs in most of the trials having random initial ensemble feature sets. Even in these cases, the initial mean test set error is still less than the single learner mean test set error.

VI. Conclusions

The development of the Machine Learning System provided a rewarding and educational experience in object-oriented design and implementation. In making modifications to the system during development, the true beauty of object-oriented systems emerged – the ability to modify and maintain a large software system by the reduction of complexity through encapsulation, and the increased code abstraction and reuse through inheritance and dynamic binding. Although the results over the datasets tested suggest that further exploration of the GEFS parameters is warranted, the system shows much potential for further development and now exists in a form that is flexible and maintainable.

VII. Bibliography

1) Opitz, D.,"Feature Selection for Ensembles", Sixteenth National Conference on Artificial Intelligence, (p.379-384), Orlando, FL.

2) Visual Learning Systems, www.vls-inc.com

3) Prabu, Ganesh, "Genetic Ensemble Feature Selection", M.S. Thesis, Dept. of Computer Science, University of Montana (1999).

4) Rational Rose, Copyright 1995-2002, www.rational.com/products/rose/index.jsp

5) Scott, Kendall, *UML Explained*, Addison-Wesley (2001); Booch, Grady; Rumbaugh, James; and Jacobson, Ivar; *The Unified Modeling Language User Guide*, Addison-Wesley (1999).

6) Mitchell, Tom M., *Machine Learning*, McGraw-Hill (1997).

7) University of California, Irvine Machine Learning Repository, www.ics.uci.edu/~mlearn/MLRepository.html