Spring 2015

Machine Learning And A Workflow Engine For An Agent-Based Structural Health Monitoring System

Jr. William Nick
North Carolina Agricultural and Technical State University

Machine Learning and a Workflow Engine for an Agent-based Structural Health Monitoring System

William Nick, Jr.

North Carolina A&T State University

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Department: Computer Science

Major: Computer Science

Major Professor: Dr. Albert Esterline

Greensboro, North Carolina

2015

The Graduate School
North Carolina Agricultural and Technical State University

This is to certify that the Master's Thesis of

William Nick, Jr.

has met the thesis requirements of
North Carolina Agricultural and Technical State University

Greensboro, North Carolina
2015

Approved by:

Dr. Albert Esterline Major Professor	Dr. Zhijun Zhan Committee Member
Dr. Kelvin Bryant Committee Member	Dr. Gerry Dozier Department Chair

Dr. Sanjiv Sarin
Dean, The Graduate School

Biographical Sketch

William Nick was born on May 23, 1990 in Morgantown, West Virginia. He received his Bachelor of Science degree in Computer Science from North Carolina Agricultural and Technical State University in December of 2012. From the summer of 2013 until his graduation date, William has been funded under the NASA Center for Aviation Safety and has conducted research related to this field of study. He is a candidate for the MS in Computer Science.

Dedication

First of all, I would like to thank God because, if not for Him, I would not have the strength to do this. I would like to dedicate this thesis to my mother and father for allowing me to live rent free while going through my masters and providing a shoulder to cry on. I would also like to dedicate this to my advisor, Dr. Albert Esterline, for putting up with me and my ADHD antics. I also dedicate this to the late great Dr. Denis MacDowell and Dr. Robert Nakon from the chemistry department at West Virginia University. I would also like to dedicate this to Mrs. Tina Mardis, for without her I would not have the skill and mechanisms to deal with my ADHD and probably would not have finished this thesis. To quote the immortal Joe Cocker, “I get by with a little help from my friends,” and that is especially true in the case of this thesis.

Acknowledgments

The author would like to thank Army Research Office funding for proposal number 60562-RT-REP and NASA Grant # NNX09AV08a for the financial support. Thanks are also due to members of ISM lab at North Carolina A&T State University for their assistance.

Table of Contents

List of Figures

List of Tables

Abstract

This thesis reports on work in machine learning and high-performance computing for structural heath monitoring. The data used are acoustic emission signals, and we classify these signals according to source mechanisms, those associated with crack growth being particularly significant. The work reported here is part of a larger project to develop an agent-based structural health monitoring system. The agents are proxies for communication- and computation-intensive techniques and respond to the situation at hand by determining an appropriate constellation of techniques. The techniques thus structured are executed by a workflow engine, which is part of the contribution reported here. It is critical that the system have a repertoire of classifiers with different characteristics so that a combination appropriate for the situation at hand can generally be found. The classifiers are trained using machine-learning techniques, and we report on investigations we conducted on three supervised and two unsupervised learning techniques to determine which techniques are the best to use in a particular situation.

CHAPTER 1

Introduction

Structural health monitoring (SHM) provides a way to capture structural responses and allows for structural conditions to be assessed for various purposes. The monitoring of an airplane, for example, minimizes the cost of repair and improves public safety. In monitoring the structure of an airplane, the data must be processed not only in an intelligent and flexible way but also in near real-time.

Our target architecture involves a multiagent system that sends a message to a workflow with the specification of the task and its structure. The workflow is set up in a pipeline (sequence) pattern where each stage (task) is its own process. Our setup involves agents that advocate for computational-intensive techniques. These agents are endowed with enough intelligence to be able to reason amongst themselves about which techniques are appropriate for the current situation and how those techniques are to be connected in a workflow. An agent submits a multitask request to the workflow system and the workflow system attaches input and output sources. So the agents are the "brains" and the workflow is the "brawn" [FJK2004].

The basic data that we work with consists of acoustic signals that we interpret regarding the events that produce them (e.g., crack growth). At the lowest level, for a given data stream, the agents in the target system will find techniques to extract features from the stream and classification techniques that take vectors of these features as input and produce classifications of the corresponding events. The classifiers are trained using machine-learning techniques. A major part of the work reported here relates to testing various machine-learning techniques for our intended application, using data provided by our mechanical-engineering collaborators.

The rest of this thesis is organized as follows: Chapter 2 provides a literature review of SHM, machine learning, and agents. Chapter 3 presents the software used. Chapter 4 presents results of machine learning in the context of SHM. Chapter 5 describes the overall design and implementation of our architecture. Chapter 6 concludes and suggests future work.

CHAPTER 2

Literature Review

This chapter provides a review of some of the technologies and concepts that will be used in the design. It gives a reasonable idea of the system being implemented. The discussion will begin with a review of some key points regarding structural health monitoring. Some of the research from Farrar, Worden and their colleagues [FWo2007, FLi2007, WFM2007] will be discussed. In the second section, the discussion turns to workflows. A scientific workflow is utilized to process and classify data. In the proposed design, a scientific workflow is used to classify data as well as analyze structures.

The third section includes a discussion of multiagent systems. The multiagent system gives the system collaborative abilities. This section also discusses the use of the Contract Net Protocol (CNP) for control and collaboration among agents and the contracting and subcontracting of task in the network. In the fourth section, the discussion turns to machine learning. This section outlines techniques in dimensionality reduction and supervised and unsupervised learning.

2.1 Structural Health Monitoring

In general, damage is defined as change introduced into a system that will adversely affect its current or future performance [FWo2007]. This idea of damage is meaningless without a comparison between two states of the system, one assumed to be the unloaded and undamaged state. For mechanical structures, damage can be defined more narrowly as change to the material and/or geometric properties of a structural system. Structural health monitoring (SHM) is the process of implementing a strategy for damage identification for aerospace, civil and mechanical

engineering infrastructure. With regard to maintenance, SHM is used to screen the condition and provide real-time information on the integrity of the structure.

In characterizing the state of damage in a system, we can ask whether there is damage, where in the system it is, what kind of damage it is, and how severe it is. Damage prognosis is the estimation of the remaining useful life of an engineering system [FLi2007]. Such an estimation may be the output from models that predict behavior.

2.1.1 The Fundamental Axioms of Structural Health Monitoring

The field of SHM has matured to the point where several fundamental axioms or general principles have emerged. Worden and his colleagues [WFM2007] suggest the following seven axioms for SHM.

Axiom I: All materials have inherent flaws or defects.

Axiom II: The assessment of damage requires a comparison between two system states.

Axiom III: Identifying the existence and location of damage can be done in an unsupervised learning mode, but identifying the type of damage present and the damage severity can generally only be done in a supervised learning mode.

Axiom IVa: Sensors cannot measure damage. Feature extraction through signal processing and statistical classification is necessary to convert sensor data into damage information.

Axiom IVb: Without intelligent feature extraction, the more sensitive a measurement is to damage, the more sensitive it is to changing operational and environmental conditions.

Axiom V: The length- and time-scales associated with damage initiation and evolution dictates the required properties of the SHM sensing system.

Axiom VI: There is a trade-off between the sensitivity to damage of an algorithm and its noise rejection capability.

Axiom VII: The size of the damage that can be detected from changes in system dynamics is inversely proportional to the frequency range of excitation.

The axioms of importance to this research are axioms III, IV and V.

2.2 Workflows

Van der Aalst and his colleague defined a workflow as a collection of tasks organized to accomplish some business process [VVa2004]. A workflow system's primary objective is to deal with *cases*. Each case has its own unique identity and a limited lifetime. Each case has a particular state, which consists of three elements: 1) the values of the relevant case attributes, 2) the conditions which have been fulfilled and 3) the content of the case. Each case has a range of variables used to manage it. Case attributes cannot be used to see how far a case has progressed. Rather, conditions are used to determine what tasks have finished and what tasks have yet to be performed.

A *task* is a logical unit of work [VVa2004]. Tasks are often performed by one or more software systems, one or a team of humans, or a combination of these. If anything goes wrong during the performance of a task, then the task is restarted from the beginning, which is referred to as a rollback. That is, tasks are atomic and are like transactions. A task is not the performance of an activity for a specific case; rather, "task" refers to a generic piece of work. The terms "work item" and "activity" are used in order not to confuse the task itself and the performance of a task in a particular case. A work item is the combination of a case and the task that is about to be carried out. A work item is created as soon as it is allowed by the state of the case. The term "activity" refers to a work item being performed. A work item becomes an activity when work begins on it.

A process is the procedure for a particular type of case [VVa2004]. The process indicates which tasks need to be carried out and in what order. It is possible, based on the attributes of certain cases, to enable special treatment. The order in which tasks are performed is determined by the properties of the case. We use conditions to figure out what order is followed. Processes are constructed from tasks and conditions as well as subprocesses, which may themselves have subprocesses. Explicitly defining subprocesses allows for frequently occurring subprocesses to be used repeatedly. A process defines the lifecycle of a case in that it has a beginning and end, which mark the start and finish of a case, respectfully.

Routing determines what tasks need to be performed along a particular branch [VVa2004]. There are four basic routing patterns: 1) sequential routing, 2) parallel routing, 3) selective routing and 4) routing iteration. Sequential routing is where tasks are carried out one after another. In parallel routing, two tasks are performed without the results of one affecting the other. Selective routing occurs when there is a choice between two or more tasks. This choice is made based on the specific properties of the case. Routing iteration is where a task is performed several times.

2.2.1 Scientific Workflows

"Scientific workflow" is a term used to describe a set of structured activities and computations that come up when solving scientific problems [SVo1996]. A scientific workflow allows a scientist to design, execute and communicate analytical procedures repeatedly and with minimal effort. These scientific workflows operate on large amounts of heterogeneous data. The three workflow engines that we looked at are Kepler, Pegasus, GridNexus, and Karajan.

Kepler [ABJ2004] is a scientific workflow engine based on the Ptolemy II project, an open source project from the University of California at Berkley that has been used to study

heterogeneous modeling, simulation, and design of concurrent systems with a focus on embedded systems, particularly those that mixed technologies. The Ptolemy II framework is an actor framework for Java. Kepler makes use of a library of reusable processing steps: actors. Each actor can have zero or more typed input ports and zero or more typed output ports. The actors can be connected into a directed graph through their ports, which allows the data to flow between actors. Most of the actors in Kepler run on a local machine, but there are some actors that can call services on the web or on the grid. Those actors expose an operation to the user in a given Web Service Description Language (WSDL) or Grid Web Service Description Language (GWSDL). Each workflow is serialized using a valid instance of an XML schema called ModelML (MoML). Kepler has a GUI interface that allows for building and executing workflows.

Pegasus is a scientific workflow project that takes a different approach than Kepler. Pegasus [DBG2004] maps abstract workflows onto grid environments. These abstract workflows can be constructed with Chimera or written by the user. In the Chimera system, the inputs are partial workflow descriptions of the logical input files, the logical transformations and their parameters, as well as the names of the logical output files produced by these transformations. A logical input file is a file that may exist or be generated by other application components. These abstract workflows are expressed in the Chimera Virtual Data Language (VDL). Chimera chains the partial workflows together and produces a XML file that specifies the abstract workflow using the DAG XML description schema (DAX). Then Pegasus takes the DAX file produced by Chimera and turns it into a submit file that can be run by HTCondor's Directed Acyclic Graph Manager (DAGMan) [CKR2007]. In DAGMan, job dependencies are expressed as a directed acyclic graph.

GridNexus [BFH2005] is a scientific workflow project that is very similar to Kepler. Like Kepler, GridNexus is built upon the Ptolemy II framework. But GridNexus has three types of modules: Source, Sink and Transformer. A Source module is a module that provides data but does not need an input connection from any module. A Sink module is a module that does not produce outputs. A Transformer module is a module that takes an input, transforms it and produces a new output. GridNexus uses a LISP-like XML-based scripting language, called JXPL, which is implemented in Java to serialize workflows.

The Karajan workflow engine [Las2005] that comes with the Java Commodity Grid kit is a derivation of GridAnt. It has added support for flow control constructs such as conditionals. In the Karajan workflow engine, a project refers to a workflow project and named projects allow workflow projects to be referenced by name. Karajan also has the concept of elements. An element is named and has a number of parameters, and once defined, can be reused by name later in the code. The Karajan workflow engine supports conditions, loops and choices as well as hierarchal graphs.

Our workflow engine must be a system that can handle the vast amount of data being streamed over in a short period. To accomplish this, we used processes and pipes to provide the level of concurrency needed to solve the problem in near-real time. This other workflow engines do not allowed to be controlled by an agent or agent system. To solve this problem, we used web services to submit the pattern of the workflow to the workflow engine, and the workflow engine sets up the resources.

2.3 Agents

An agent as an autonomous, problem-solving, computational entity that is capable of effectively processing data and functioning singularly or in a community within dynamic and

open environments [LMP2003]. Agents interact and cooperate with other agents (including both people and software) in the environments to which they are deployed. Within the context of a given environment, agents contain beliefs, capabilities, choices and commitments. Human interaction with a system can be referred to as an agent and may be included in the design for a multiagent system, but, for the purposes of the architecture proposed here, the term "agent" refers to software agents that make up the control hierarchy for passing, storing and manipulation of data.

There are many benefits of utilizing an agent system for control as well as data distribution. In a well-defined agent system, agents can communicate with other agents and users as well as propose projects and bid on those proposed projects. Agents' ability to adapt and solve problems makes them ideal for environments where new problems tend to arise and that require flexibility and subcontracting ability. A multiagent system [Woo2009] consists of several interacting agents, autonomous in nature, that work together to achieve individual and group goals. An agent in a multiagent system represents or acts on behalf of users or owners with various goals and motivations. In order to successfully interact, agents must be able to cooperate, coordinate and negotiate in the same ways humans cooperate, coordinate and negotiate in their daily lives.

The Foundation of Intelligent Physical Agents (FIPA) was an international non-profit organization that produced specifications of generic agent technologies [BCG07]; it is now an IEEE standards committee. FIPA defines an agent platform (AP) as the physical infrastructure on which an agent is deployed. The agent management system (AMS) is a mandatory part of an AP that is responsible for the operation management of an AP. The directory facilitator (DF) is an

optional part of an AP and provides a yellow page service to the AP. A message transport service is provided by the AP to transport FIPA-ACL messages to agents on/ between APs.

The rest of this section will cover agent communication languages, the contract net protocol and agent controlled compute resources.

2.3.1 Agent Communication Language

An agent communication language (ACL) [LFi1998] allows for effective communication and knowledge exchange between agents despite differences in hardware, operating system, architecture, programming language or representation and reasoning systems. In the 1990s, a knowledge sharing effort was initiated by the Defense Advanced Research Projects Agency (DARPA) to develop protocols for exchanging knowledge between autonomous information systems. The two most used ACLs are the knowledge query and manipulation language (KQML, which arose from the DARPA knowledge-sharing effort) and the more recent FIPA ACL.

ACLs are based on *speech act theory*, which treats communication as a sequence of actions [Woo2009]. *Performatives* are verbs that describe what they do while performing that intended act. An example of a performative is confirm. There are three aspects to a speech act: 1) a *locutionary* act, 2) an *illocutionary* act, and 3) a *perlocutionary* act. The *locutionary* act is the act of making an utterance (the act of saying something out loud) for example “I warn you that smoking cigarettes are bad for your health”. The *illocutionary* act is the action performed in saying something for example “He requested that I stop smoking cigarettes”. The *perlocutionary* act is the act of bringing about an effect for example “He got me to quit smoking cigarettes”. The *felicity condition* of a performative is the condition required for successfully carrying it out. Communication between agents is based on message passing, where agents formulate and send individual messages to each other.

The KQML agent communication language [Woo2009] is a message-based language. KQML messages can be crudely thought of as objects (in the sense of object-oriented programming). In KQML, every message has a *performative* (which can be thought of as the class of the message) and a number of *parameters* (which can be thought of as instance variables) as shown in table 1.

Table 1: KQML Parameters

Parameter	**Meaning**
`:content`	Content of the message
`:force`	If the content of the message will ever be denied by the sender
`:reply-with`	Specifies whether the sender is expecting a reply and if so, the identifier for reply
`:in-reply-to`	Refers to the : `reply-with` parameter including the dictated identifier.
`:sender`	Identifier for the sender of the message
`:receiver`	Identifier for the intended recipient of a message

The FIPA standard specifies the Agent Communication Language (ACL) [BCG07]. The FIPA ACL specifies a standard message language by setting out the encoding, semantics and pragmatics of the messages. The FIPA ACL is superficially similar KQML: an outer language for messages is defined and twenty *performative* are also defined. Some of the *performatives* available are shown in table 2. Table 3 shows some of the possible parameters for FIPA ACL.

Table 2: FIPA ACL Performatives

Performative	Description
`:accept-proposal`	The action of accepting a previously submitted proposes to perform an action.
`:cfp (call for proposal)`	Contains the action to be carried out and any other terms of the agreement
`:confirm`	The sender confirms to the receiver the truth of the content. The sender initially believed that the receiver was unsure about it.
`:inform`	Tell another agent something. The sender must believe in the truth of the statement.
`:propose`	Agent proposes a deal.
`:reject-proposal`	The action of rejecting a proposal to perform some action during a negotiation.
`:request`	The sender requests the receiver to perform some action.

Table 3: FIPA ACL Parameters

Parameter	Meaning
`sender`	Identifies the sender of the message
`receiver`	Identifies the receiver of the message
`reply-to`	Identifies the name of the agent who the subsequent meassages are send to
`content`	Content of the message
`ontology`	The name of the ontology that is used to give meaning to a symbol
`encoding`	Specifies the encoding of the content language expression

2.3.2 The Contract Net Protocol

The contract net protocol (CNP) is a high level protocol for achieving efficient cooperation through task sharing in networks of communicating problem solvers [Smi1980, DSm1983]. As the name suggests, the basic metaphor used in the CNP is contracting. This technique was inspired by the process that companies use to contract out tenders.

Each node in the network can, at different times or for different tasks, be a manager or a contractor. The CNP controls the flow of collaboration, contracting and subcontracting, throughout the network of agents. Agents break down problems or tasks into sub-problems or sub-tasks, allowing available agents to bid on the sub-tasks. This coordination and cooperation allows the system to monitor more responsively and efficiently.

The manager agent decomposes the problem into sub-problems and announces each sub-problem to all agents (a call for proposals). All potential contractors (those willing to perform the announced task) may reply with a bid that includes, for example, information about the resources and time needed to solve the sub-problem. The manager reviews the bids and selects the best bid, awarding the contract to the agent making that bid. The contractor reports the solution back to the manger, which compiles a solution to the overall problem from the solutions to the sub-problem. Figure 1 is the agent UML of the messages sent.

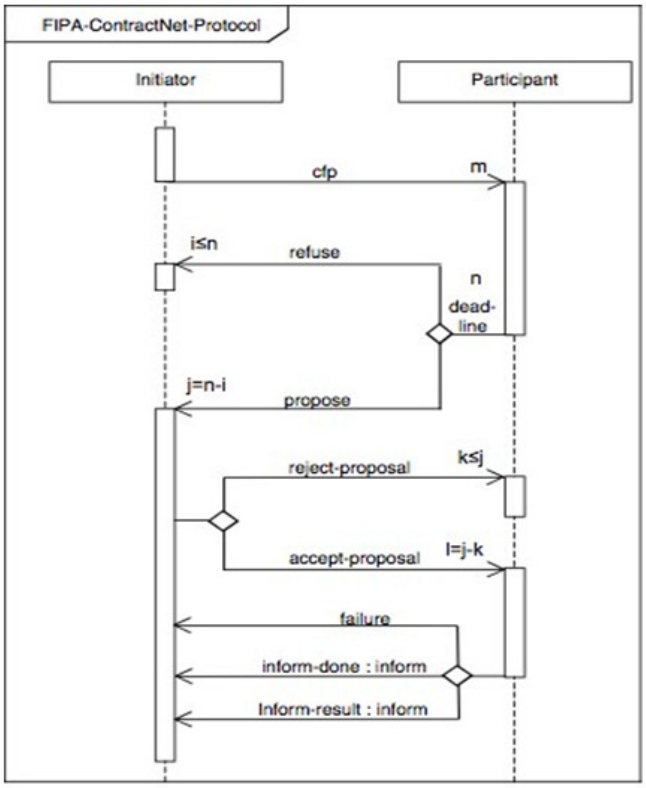

Figure 1: FIPA Contract Net Interaction Protocol [IEEE2009]

2.3.3 Agent-Controlled Computing Resources

This section takes a look at computational resources that are controlled by a multiagent system. We take a look at several systems from various researchers in the field. The three

systems that we look at are The CONOISE-G architecture [PJL2005], scientific workflow system by Buhler and his colleagues [BVi2005] and Poggi and his colleagues [PTT2004].

Vidal and his colleagues [BVi2005] proposed an architecture that uses a BPEL4WS process description language to impose an initial social order upon a collection of agents. In BPEL4WS, a partner is a workflow participant. Each partner has a service type link and the role that it will perform as part of the service link. The strategy used is to construct a Petri net for the workflow. The Petri net is partitioned based upon partner information. Agents represent the partners and execute the workflow in a distributed way. Agents communicate to each other using the FIPA ACL. Communication between agents and web services is accomplished using SOAP messages.

The problem that arises between agents and grid computing [PTT2004] that needs to be solved concerns the execution of applications/services that are composed of large sets of independent or loosely coupled tasks. This kind of problem requires intelligent movement of tasks between nodes to reduce the cost of communication for managing the interaction between remote agents. This also requires intelligent task composition and generation to manage new interactions and tasks defined during execution. To adapt JADE for grid applications, a new type of agent are needed, one that supports 1) rule-based creation and composition of tasks and 2) mobility of code at the task level. Two new agents were created to handle the adaptation: 1) BeanShell Agents and 2) Drools Agents. A BeanShell agent integrates with the BeanShell scripting engine inside of a JADE agent and provides an API for interacting with the BeanShell agent through FIPA ACL. The BeanShell agent ontology describes the AgentAction objects used to submit tasks and manipulate BeanShell variables. A drools agent integrates with the drools rule engine. A Drools agent exposes an API that allows the manipulation of the drool agent

internal working memory. The drool agent's ontology describes AgentAction objects that allow one to add rules and to assert, modify and retract facts.

The CONOISE-G architecture [PJL2005] has several different agents, including system agents and service providers (SPs). Service providers advertise their services to a yellow pages (YP). The process of forming a virtual organization (VO) starts with a Requester Agent (RA). The RA, a SP acting on behalf of the user, analyzes the requester's service requirements. The RA searches the YP for relevant providers, and then invites the providers found to bid for the requested service. The Quality Agent (QA) and the trust component, respectively, assess the quality and trustworthiness of the bids. The Clearing Agent (CA) combines the price structure along with the outcome of the QA to determine what service providers will come together to form an optimal VO for the requester. The VO is formed at this point, and the role of the VO Manager (VOM) is taken on by the RA. The VOM is responsible for making sure that each member of the VO provides its services according to the contract. During the VO's operation phase, the VOM can request a Quality of Service (QoS) consultant to monitor services provided by a member of the VO. Members of the VO may call upon the Policy Agent to investigate disputes over service provisioning.

2.3.4 JADE

The Java Agent Development Environment (JADE) [BCG07] is a framework for developing agent applications in compliance with FIPA standards. JADE makes FIPA standard assets available to the programmer through object-oriented abstraction. The JADE communication architecture offers flexible messaging. JADE chooses the most appropriate method of transportation and uses state-of-the-art object distribution technology embedded inside the Java Runtime Environment (JRE). The JADE system has one or more agent containers, each

living in a separate Java Virtual Machine (JVM) and delivering runtime environment support to some JADE agents. The JADE execution model is based on behavior abstraction and tries to limit the number of threads. The Behaviour (spelled in the British fashion as JADE comes from Italy, an EU partner of the UK) abstraction models agent tasks as a collection of behaviours that are scheduled and executed to carry out the agent's duties.

A JADE system has three distinctive properties. First of all, JADE agents are autonomous and proactive. To achieve this, each agent has its own thread of execution but cannot provide callbacks. Each agent runs on its own thread to control its lifecycle and decide which actions to perform and when to perform them. Secondly, JADE agents have the right to refuse, and agents are loosely coupled. The receiving agent decides which messages to refuse and what actions are performed at which priority. Thirfly, the system is peer-2-peer (P2P). Each agent has a unique name or agent identifier (AID). Agents can join or leave the system by registering or deregistering with the AMS. Agents can look up other agents by AID using the services provided by the DF.

2.4 Machine Learning

The results of machine learning allow us to look at the problem of structural damage identification at a more sophisticated level. We may then address a multitude of issues and provide diagnoses of the problems. Of particular interest are Axiom III of Worden and his colleagues, which that unsupervised learning can be used for identifying the existence and location of damage but identifying the type and severity of damage can only be done with supervised learning. Supervised learning tries to generalize responses based on a training set with the correct responses indicated. Unsupervised learning tries to categorize the inputs based on

their similarities and not from labeled data. This section covers PCA as well as unsupervised and supervised learning.

2.4.1 Principal Component Analysis

Machine learning is generally facilitated by reducing the dimensionality of the data. For this, we use principal component analysis (PCA) [Bis2006]. PCA is an algorithm that centers the data by subtracting off the mean then choosing the eigenvector of the data covariance matrix with the largest eigenvalue. It places an axis in that direction, and then incrementally and similarly places the other axes orthogonally to their predecessors in a way maximizing the possible variation. The number of axes is chosen to be fewer than the number of axes (dimensionality) of the original data set. The data is thus reduced in dimensionality while most of the variation is retained.

2.4.2 Unsupervised Learning

Recall that unsupervised learning allows for data to be classified based on similarity and does not rely on labeled data. Worden and his colleagues suggest that unsupervised learning can be used to determine the existence and location of damage. The k-means algorithm [Bis2006] classifies n observations into k clusters. The value of k is set by the user. The cluster centers are distributed randomly at first, and a data point is assigned to the cluster nearest it in terms of Euclidean distance. Each cluster center is then updated to be the average of the points assigned to it. The data points are reassigned to clusters and the cluster means are recomputed. This process is repeated until the distance of the data points and the centers are within some threshold or some maximum number of iterations is reached.

A self-organizing map (SOM) [Bis2006] is a type of neural network used to produce a low-dimensional, discretized representation of the space of the training data. A SOM identifies

features across the range of input patterns. Neurons compete to be activated, and only one is activated at any one time. A SOM has only two layers: an input and output layer. The neurons in the output layer are arranged into a lattice, that is, a partial-ordered set where every two elements has a unique least upper bound and a unique greatest lower bound. A SOM needs very little to no preliminary data cleansing [DHS1999].

2.4.3 Supervised Learning

Recall that supervised learning allows for generalization based on a training set. Worden and his colleagues suggest that damage type and severity can only be done using supervised learning. An artificial neural network (ANN) is a computational model based on the structure and functions of a biological neural network [Bis2006].

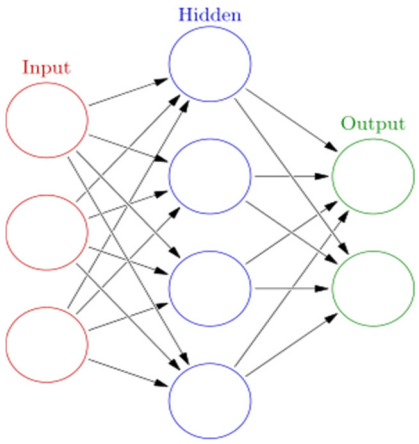

Figure 2: Artificial Neural Network [Wik2014a]

In a feed-forward neural network, or multilayer perceptron, input vectors are put into input nodes and fed forward in the network--see Figure 2. The inputs and first-layer weights determine whether the hidden nodes will fire. The output of the neurons in the hidden layer and the second-layer weights are used to determine which of the output layer neurons fire. The error between the network output and targets is computed using the sum-of-squares difference. This

error is fed backward through the network to update the edge weights in a process known as back propagation.

Support vector machines (SVMs) [DHS1999] rely on preprocessing to represent patterns in the data in a high dimension, usually higher than the original feature space, so that classes that are entangled in the original space are separated by hyper-planes at higher dimension. In a bit more detail, if the training data are linearly separable, we select from all pairs of hyperplanes that separate the data points (with no data points between them) those two that have the maximum distance between them—see Figure 3. If we replace the dot product in distance computations with a nonlinear "kernel function," then we can fit the maximum-margin hyperplane in a transformed feature space. If the transformation is nonlinear and the transformed space high dimensional, then the classifier is a hyperplane in the high-dimensional feature space but nonlinear in the original input space. Training a SVM involves choosing a nonlinear function that maps the data to a higher-dimensional space. Choices are generally decided by the user's knowledge of the problem domain. SVMs can reduce the need for labeled training instances.

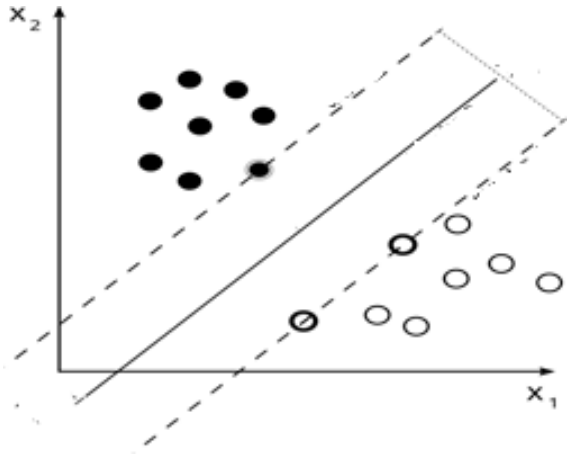

Figure 3: Support Vector Machine [Wiki2014b]

Naïve Bayes classifiers form a supervised learning technique that belongs to a family of classifiers based on Bayes' theorem with a strong assumption about the independence of the

feature values [DHS1999]. Bayes' theorem states that $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$. For example, we would like to know whether or not we should play tennis given that it is sunny outside. Using Bayes' theorem, we can express this as $P(outside = sunny|playstennis = yes) = \frac{P(playstennis = yes|outside = sunny)P(outside=sunny)}{P(playstennis=yes)}$. Two events are independent if the occurrence of one does not affect the probability of the other.

Assumptions and the underlying probabilistic model allow us to capture any uncertainty about the model. This is generally done in a principled way by determining the probabilities of the outcomes. Bayes classifiers were introduced to solve diagnostic and predictive problems. Bayesian classification provides practical learning through the use of algorithms, prior knowledge, and observation of the data in combination. A Gaussian naïve Bayes classifier assumes that the conditional probabilities follow a Gaussian or normal distribution.

2.4.4 Machine Learning for SHM

Most previous work on machine learning for SHM has targeted bridges; we consider mature, representative work in this area and then turn to research that has targeted aircraft, which is our domain. Figueiredo and his colleagues performed an experiment on a three-story frame aluminum structure that used a load cell and four accelerometers [3]. For each test of state conditions, the parameters were estimated by using a least squares technique applied to time-series from all four accelerometers and stored into feature vectors. They used four machine learning techniques in an unsupervised learning mode: 1) auto-associative neural network (AANN), 2) factor analysis (FA), 3) singular value decomposition (SVD), and 4) Mahalanobis squared distance (MSD).

An auto-associative neural network [Kra1992] is a type of feedforward neural network that is trained to produce an approximation of an identity mapping between network inputs and

outputs using backpropagation or similar learning procedures. Factor analysis (FA) [Bis2006] is a statistical method that is used to describe variability among observations and correlates variables in terms of a potentially lower number of unobserved variables called factors. Singular value decomposition (SVD) [KLa1980] takes a high dimensional and highly variable set of data points and reduces this set to a lower dimensional space clearly exposing the substructure of the original data and ordering it from the most variation to the least. The difference between PCA and SVD is that SVD calculates variance by squaring the singular values[1] and PCA calculates variance by the eigenvalues of the covariance matrix. Mahalanobis distances provide a powerful method of measuring how similar some set of conditions is to a reference set of conditions. Mahalanobis squared distance (MSD) is a measure of the number of standard deviations a point is from the mean of a distribution. Specifically, where D is the Mahalanobis distance of point $\mathbf{x}$ from the mean $\mathbf{m}$ of the independent variables and C is the covariance matrix of the independent variables, we have $D^2 = (\mathbf{x} - \mathbf{m})^{\mathrm{T}} \mathrm{C}^{-1} (\mathbf{x} - \mathbf{m})$

In the work by Figueiredo et al. reviewed here, first the features from all undamaged states were taken into account. Then those feature vectors were split into training and testing sets. In this case, a feed-forward neural network was used to build-up the AANN-based algorithm to perform mapping and de-mapping. The network had ten nodes in each of the mapping and de-mapping layers and two nodes in the bottleneck layer. The network was trained using back-propagation. The type I error is the false positive indication of damage and type II error is the false negative indication of damage. The AANN- and MSD- based algorithms

[1] For a simple definition of *singular value*, consider the special (yet common) case of $\mathbf{M}$ an $m \times m$ real square matrix with positive determinant. We have in general (not just this case) $\mathbf{M} = \mathbf{U\Sigma V}^*$. In our special case, $\mathbf{U}$, $\mathbf{V}^*$, and $\mathbf{\Sigma}$ are real $m \times m$ matrices as well. $\mathbf{\Sigma}$ is a diagonal matrix and can be regarded as a scaling matrix. U and V are unitary $m \times m$ matrices (with V* in general the complex conjugate of V); $\mathbf{U}$ and $\mathbf{V}^*$ can be viewed as rotation matrices. So $\mathbf{U\Sigma V}^*$ can be intuitively interpreted as a composition of three geometrical transformations: a rotation, a scaling, and another rotation. The diagonal elements of $\mathbf{\Sigma}$ are non-negative and are the singular values of M. In 2D, the singular values can be interpreted as the semiaxes of an ellipse; this generalizes to n dimensions. If some of the singular values of M are zero, then the singular value decomposition reduces the dimension.

performed better at detecting damage, with a type II error of 1.3% (for the AANN) and of 1.0% (for the MSD). The SVD- and FA- based algorithms performed better at avoiding false indications of damage, with a type I error of 2.2% (for the FA) and of 6.2% (for the SVD).

Esterline and his colleagues [EKS2010] (also targeting aircraft) ran an experiment with two approaches. Their first approach used as training instances experimental data with eighteen traditional acoustic emission features, such as reverberation frequency, amplitude and duration, to train a SVM, while their second approach used six correlation coefficients between basic modes and waveforms from simulation data also to train a SVM. A correlation coefficient is computed between an observed waveform and each of the seven base modes. The vector of all six correlation coefficients characterizes the waveform. This signature can be compared against similar signatures produced by known sources (such as various stages of crack growth or forms of fretting), perhaps ones produced in simulation, in an attempt to identify the source from the characteristics of the acoustic emission. The SVM with the second approach performed as well or better than the SVM using the first approach, suggesting the superiority of a set of correlation coefficients over a substantial set of traditional acoustic emission features for learning to identify the source of acoustic emissions.

CHAPTER 3

Software

This chapter provides a discussion of the different software packages that were used for this work. The software at the center of this work is the Python programming language. This chapter covers software for numerical and scientific computing. It also covers software for machine learning as well as the hierarchical data format 5 used for this work.

3.1 Python Software

The Python programming language is an interactive, interpreted object-oriented language [San1999]. Python provides high level structures like lists, associative arrays (called dictionaries) and other construct similar to the Perl programming language. It has an extremely simple, elegant and easy to read syntax yet is a powerful general purpose programming language. Python is extended through the use of modules (most written in Python, others in C or C++) to provide operations such as string manipulation and Perl-style regular expressions. C or C++ programmers can embed a Python interpreter into larger applications. The Python interpreter can be extended or embedded into a C program by including `Python.h` in your `.c` or `.h` file.

NumPy [Oli07] is a Python package for numerical computing, which adds powerful mathematical functions into the Python environment. The key features of NumPy are:

- A powerful N-dimensional array object
- A list of sophisticated techniques such as broadcasting [Num2012]
    - Broadcasting in this context is the ability for smaller dimensional arrays to be either extended or replicated in order to work with arrays with larger dimensions.
- A host of tools specialized for integrating C/C++ and Fortran code

- Functions that are useful in linear algebra, Fourier transform, and random number generation

SciPy [JOP2007] is an open source Python library that is built on top of NumPy for scientific computing. Utilizing SciPy, complex mathematic problems can be easily and systematically processed.

The Pandas library for Python [Mck2011] attempts to bridge the gap between scientific computing and statistical and database languages. Pandas provides functions equivalent to those in statistical languages (such as R) and database languages (such as SQL), but it also provides many features such as hierarchal indexing. Pandas provides support for structured tabular datasets (similar to R).

3.2 Software for Machine Learning

There are several software packages available for machine learning. Each of these packages has its own set of strengths and weaknesses. For our implementation, we chose scikit-learn [PVG2011] for k-means clustering, support vector machine and Gaussian naive Bayes. This is mainly because scikit-learn uses the Python programming language and is a mature and widely used framework for machine learning. We also chose Pybrain [SBW2010] for feed-forward neural networks. Pybrain is a mature framework for creating and simulating neural networks. In addition to scikit-learn, we use WEKA [HFH2009] because it has a large supporting community, and it is easy to add other machine learning techniques to it.

3.2.1 Scikit Learn

Scikit-learn [PVG2011] is a powerful machine learning library for the Python programming language. It provides a state of the art implementation of well-known and popular machine learning techniques. Unlike many other machine learning toolkit in Python, scikit-learn

incorporates compiled code for efficiency. Specifically, it uses the libSVM package for SVMs and the LibLinear package for generalized linear models. Scikit-learn was developed with the intentions of providing a solid implementation rather than a bunch of features.

3.2.2 PyBrain

PyBrain [SBW2010] is a versatile machine learning package written in python for application of and research on premier machine learning techniques such as deep-belief neural networks. The only packages required by PyBrain is SciPy and NumPy. The PyBrain library has several different types of trainable algorithms and architectural components as well as specialized data sets and benchmark tests and environments.

3.2.3 WEKA

The Waikato Environment for Knowledge Analysis (WEKA) [HFH2009] is a state-of-the-art unified workbench that gives researchers easy access to techniques in machine learning. WEKA provides a collection of machine learning and data preprocessing techniques for researchers and practitioners alike. One of the major advantages of WEKA is that it is modular, so new learning techniques can be added through the simple API.

3.3 Hierarchical Data Format 5

The hierarchical data format 5 (HDF5) [HDF2014] is a file format for storing and managing large quantities of data. HDF5 provides support for a large number of commonly used datatypes. It is designed for efficient I/O that is also fast and flexible. HDF5 has two major types of objects: 1) datasets and 2) groups. A dataset is a multidimensional array of homogeneous data. A group is a container structure that can hold datasets and other groups. Resources in a HDF5 can be accessed using the POSIX-style syntax /path/to/resource.

CHAPTER 4

Machine Learning

Following Axiom III from Worden and his colleagues [WFM2007], our research to date has used two unsupervised and three supervised learning techniques for different aspects of the SHM problem. The results of machine learning allow us to look at the problem of damage identification at a more sophisticated level. We may then address a multitude of issues and provide diagnoses of the problems. The unsupervised learning techniques are k-means and self-organizing maps (SOM). Supervised learning techniques are support vector machines (SVMs), naive Bayes (NB) classifiers, and feed-forward neural networks (FNNs). For each technique except SOM, we tested a version with principal component analysis (PCA) as a frontend to reduce the dimensionality of the data (usually to three principal components), and we tested another version without PCA. The objective is to explore these techniques and note their characteristics so that various combinations of them may be used appropriately in various circumstances. The code for the supervised learning techniques is in Appendix B.

The approach followed here can be generalized for exploring the characteristics of machine-learning techniques for monitoring various kinds of structures. One must first determine what signals are appropriate for monitoring the structures, (For example, acoustic signals are appropriate for monitoring metallic structures while signals propagated through optical fiber are appropriate for bridge type structures.) One then determines the sensor and communication infrastructure. Finally, one determines the characteristics of various supervised and unsupervised learning techniques for monitoring the structures in question (given the signals and infrastructure chosen). Admittedly, the repertoire of techniques explored here is far from complete, but we have included the ones most often encountered in structural health monitoring. Our dataset

consists of a set of 60 samples from the work reported by Esterline and his colleagues [EKS2010].

4.1 Supervised Learning Results

We used a stratified 5-fold cross-validation on the 60 data point. To compare supervised learning techniques, we used classification accuracy, the number of samples classified correctly over the number of samples in the dataset. We also compared techniques on how long the classifiers they trained took to classify 12 data points in our test set.

We ran a SVM with four types of kernel functions: linear, radial basis (RBF, with $\gamma = 0.03125$), polynomial and sigmoid. A sigmoid function, known as a squashing function, maps an axis to a finite interval – see figure 4.

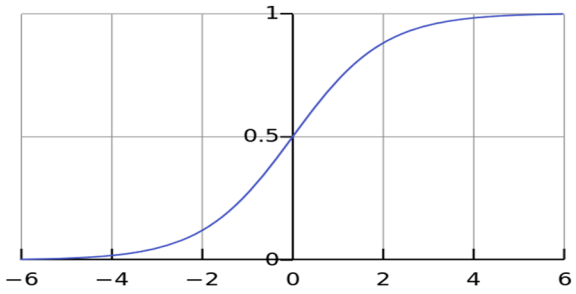

Figure 4: Sigmoid function [Wik2014c]

Table 4 displays the accuracy with which our SVMs classified 12 data points in our five validation sets. The SVMs were also trained with a PCA frontend and run on the same data. Table 5 displays the resulting average classification accuracy and timing. The mean for all our results is for 26 runs.

Table 4: Average accuracy and timing of the SVM without PCA (12 points, 26 runs)

Kernel Function	RBF		Linear		Polynomial		Sigmoid	
	Min	Max	Min	Max	Min	Max	Min	Max
Avg Accuracy	0.83	0.83	0.88	0.88	0.7	0.7	0.87	0.87
Accuracy Std	0.11	0.11	0.041	0.041	0.067	0.067	0.11	0.11
Timing Avg. (msec.) (12 classification)	0.078	0.14	0.074	0.15	0.075	0.089	0.092	0.11
Timing Std	0.0016	0.084	0.0017	0.11	0.0013	0.016	0.0032	0.026

Table 5: Average accuracy and timing of the SVM with PCA (12 points, 26 runs)

Kernel Function	RBF		Linear		Polynomial		Sigmoid	
	Min	Max	Min	Max	Min	Max	Min	Max
Avg Accuracy	0.78	0.78	0.85	0.85	0.62	0.62	0.77	0.77
Accuracy Std	0.15	0.15	0.06	0.06	0.08	0.08	0.12	0.12
Timing Avg. (msec.) (12 classifiations)	0.08	0.22	0.08	0.13	0.08	0.10	0.10	0.12
Timing Std	0.00031	0.26	0.00037	0.06	0.00024	0.03	0.00064	0.031

A Gaussian naïve Bayes classifier was trained and run with and without PCA. Table 6 shows the resulting average classification accuracy and timing.

Table 6: Average accuracy and timing of the Gaussian naive Bayes classifier (12 points, 26 runs)

	Gaussian NB w/o PCA		Gaussian NB w/ PCA	
	Min.	Max.	Min.	Max.
Avg Accuracy	0.78	0.78	0.78	0.78
Accuracy Std	0.085	0.085	0.1	0.1
Timing Avg. (msec.) (12 classifications)	0.18	0.25	0.19	0.23
Timing Std	0.0035	0.097	0.0032	0.042

A FNN classifier was trained and run with and without PCA. Table 7 shows the resulting average classification accuracy, timing, giving means and standard deviations.

Table 7: Average accuracy and timing of the FNN (12 points, 26 runs)

	FNN w/o PCA		FNN w/ PCA	
	Min.	Max.	Min.	Max.
Avg Accuracy	0.50	0.73	0.50	0.83
Accuracy Std	0.0	0.12	0.0	0.0
Timing Avg. (msec.) (12 classifications)	1.62	2.45	1.16	1.50
Timing Std	0.0052	0.85	0.023	0.23

Regarding accuracy, not that the minimum and maximum averages over twelve data points in the validation sets were the same for all techniques except FNN. Generally, the highest accuracy was with the SVMs. The best overall was the SVM with linear kernel function and without PCA (88% accuracy), the second best was the SVM with sigmoid kernel function and without PCA (87%), and the third best was the SVM with linear kernel function and with PCA (85%). Of the SVMs, the worst was the one with polynomial kernel function, both without (70%) and with (62%) PCA. All SVMs did worse with PCA but not by much in the case of the linear kernel function. Gaussian NB had the same accuracy with and without PCA (viz., 78%), slightly better than the SVM with polynomial kernel function and without PCA, about the same as SVM with sigmoid kernel function and with PCA. The minimum average accuracy of FNN over 26 runs was the same with and without PCA (viz., 50%), well below the accuracy of the other methods. Its maximum average accuracy over 26 runs, however, was comparable to that of the others; this improved with PCA: from 73% to 83%.

We turn now to the average time to classify twelve data points (both the minimum average over 26 runs and the maximum average over the 26 runs). SVM without PCA and with RBF, linear, or sigmoid kernel function took in range 0.078-0.15 msec. SVM with polynomial kernel function and without PCA took at most (averaging over the 12-point validation sets) 0.089 msec. PCA slowed down the SVMs by a negligible amount except in the case of the RBF kernel

function, where the maximum average went from 0.14 msec. without PCA to 0.22 msec. with PCA. Gaussian NB performed about the same with and without PCA, in the range 0.18 to 0.25 msec., which is slightly slower than the SVMs. FNN without PCA took in the range 1.62 to 2.45 msec. on average to classify twelve data points, more than ten times slower than the SVMs. PCA sped up FNN by 30-40%, to times in the range 1.16 to 1.50 msec.

Regarding the impact of PCA, it slows down and reduces the accuracy of the SVMs, has little effect on NB, and speeds and improves the accuracy of FNN. Reducing the dimensionality appears to lose information used by the highly accurate SVMs but actually improves the less accurate FNN, for which the lost information was apparently obfuscating. These results confirm that FNN is more susceptible to the curse of dimensionality. The linear transformation done by PCA, which reduces dimensionality, actuality slowed things a bit for the SVMs, which do their own dimensionality transformation. The time to reduce the dimensionality, however, results in faster classification for the more dispersed FNN.

4.2 Unsupervised Learning Results

Regarding unsupervised learning techniques, we first ran k-means clustering without PCA with *k* equal to 3, 4, 5 and 6. Our dataset is split into 30 samples in our training set and 30 samples in our test set. Then we ran k-means clustering with PCA with *k* again equal to 3, 4, 5 and 6. Table 8 shows the times taken to classify 30 data points without PCA, and Table 9 shows the same times with PCA. All times are in the narrow range 0.16-0.18 msec.

Table 8: Time (msec.) for Kmeans w/o PCA to classify 30 points (26 runs)

Technique	Kmeans 3	Kmeans4	Kmeans5	Kmeans6
mean	0.16	0.18	0.17	0.16
st. dev.	0.03	0.07	0.03	0.02

Table 9: Time (msec.) for Kmeans w/ PCA to classify 30 points (26 runs)

Technique	Kmeans 3	Kmeans 4	Kmeans 5	Kmeans 6
mean	0.17	0.17	0.18	0.17
st. dev.	0.07	0.03	0.03	0.02

Figures 4-7 show the clusters for k = 3-6, respectively, when PCA with three components was used so that points may be plotted in three dimensions. Note that k = 4 identifies four visually convincing clusters.

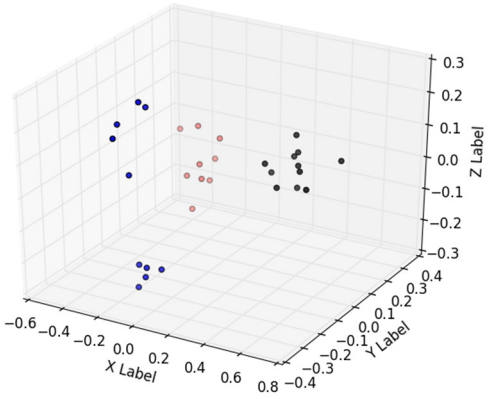

Figure 5: k-means with 3 clusters with PCA (30 data points)

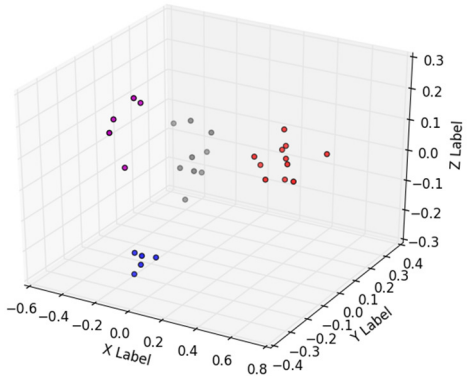

Figure 6: k-means with 4 clusters with PCA (30 data points)

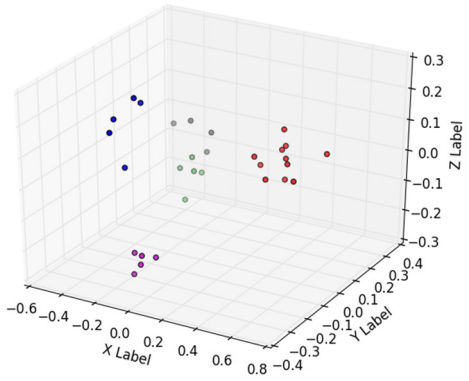

Figure 7: k-means with 5 clusters with PCA (30 data points)

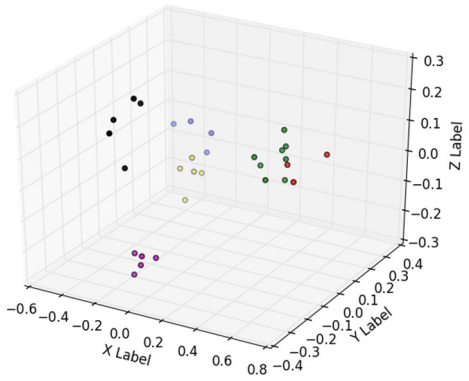

Figure 8: k-means with 6 clusters with PCA (30 data points)

We ran a SOM with 3, 4, 5 and 6 output neurons. Table 13 shows the time (in milliseconds) it took each of the SOM instances to classify all 30 data points in our test set. Clearly, the SOMs took much longer than did the k-means classifiers.

Table 10: Timing of the SOM

Technique	SOM 3	SOM4	SOM5	SOM6
mean	409.62	445.38	617.31	600.00
st. dev,	13.11	17.49	8.74	11.66

Table 11 shows the sizes of the clusters for the SOMs with 3, 4, 5, and 6 output neurons. Twenty-six runs produced no variation in cluster sizes for any of the SOMs. In the table, the right-side column gives the sizes of the clusters in the order cluster 0, cluster 1, and so on. Clusters of exactly the same size are found with three and four output neurons, but the four case adds an empty "cluster". Indeed, the partition of the set of data points into three clusters gives the least variation in cluster size.

Table 11: Mean cluster sizes for various SOMs (26 runs), ordered as cluster 0, cluster 1, etc.

SOM3:	14, 7, 9
SOM4:	14, 7, 0, 9
SOM5:	7, 2, 5, 5, 11
SOM6:	5, 11, 5, 0, 2, 7

The four sharp clusters (as per k-means with $k = 4$) can be anticipated given our experimental set-up, as we expect signals from four different sources. First of all, there are signals from the crack growth itself, but incremental crack growth deeper in the specimen produces waveforms with rather different characteristics from those produced on the surface, and this difference is enough to pull the crack data into two distinct clusters. Next, there is friction where the specimen is attached, accounting for a third cluster. Finally, the electrical environment produced a consistent kind of noise, giving rise to a fourth cluster. We conjecture that the SOMs tend to find only three clusters because there is a sort of continuum of data points between those related to deep crack growth and those related to crack growth at the surface; a SOM preserves the topological properties of the input space, and these intermediate points apparently pull together otherwise disparate clusters.

CHAPTER 5

Architecture

The problem with monitoring the structure of an airplane is that data must be processed not only in an intelligent and flexible way but also in near real-time. Our solution involves a multiagent system that sends a message to a workflow engine with the specification of the tasks. We set up our workflow in a sequence (pipeline) pattern where each task (stage) is its own individual process. In our setup, agents are advocates for certain computation-intensive techniques. These agents have enough intelligence so that they can work out which techniques are appropriate for the current situation and how these techniques must connect to each other in a workflow. An agent submits a multitask request to the workflow system and attaches input and output sources. So agents are the brains and the workflow is the brawn. The first section below presents the entire target architecture, which is under development. The second and last section presents what has actually been implemented.

5.1 Target Architecture

The prototype of the workflow currently handles the processing of a datastream in terms of the feature extraction and classification techniques. The feature extraction techniques derive correlation coefficients computed between an observed waveform and six reference waveforms which are generated from numerical simulations of acoustic emission events. The classification techniques, as discussed in the previous chapter, are Gaussian naïve Bayes, support vector machine, feed-forward neural network, k-means clustering and self-organizing maps. The feature extraction and classification techniques that get used are determined by two instances of the contract net protocol. Eventually, the system will be set up so that, after the contract net protocol has finished, a monitor agent will send the names of the feature extraction and classification

techniques to the data agent, which will send these names to the workflow engine through web services. Then the workflow engine will put the ID of the job, the IP address or host name of the machine and the port number of the running workflow engine into a NoSQL DB, as shown in figure 9.

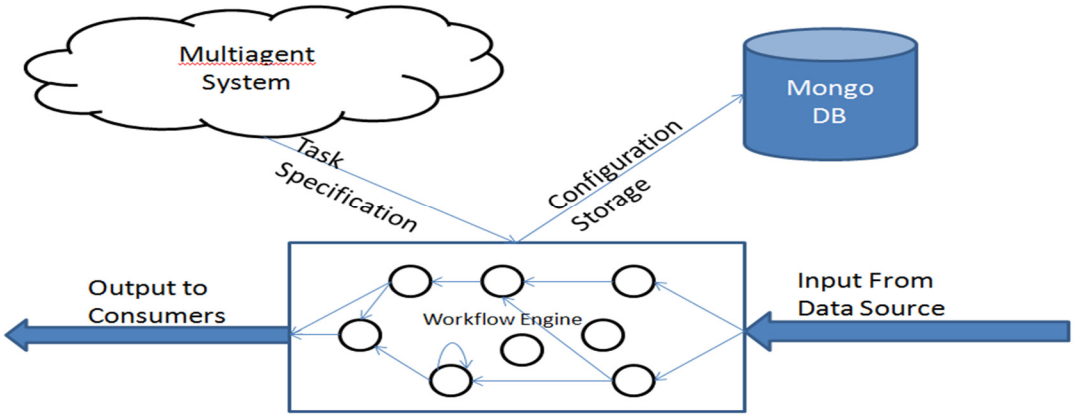

Figure 9: The workflow system in context

Then the workflow engine will send the ID of the job to the data agent. The data agent will wake up the input and output daemon processes, sending them the ID of the job. The input and output daemon processes will query the DB for the address of the machine and the port number of the running workflow engine based on the ID of the job. The input daemon will read the data from the mote (small battery-operated sensor node that is part of a wireless sensor network) and write it to a UDP socket. The output daemon will read data from a TCP socket and write it to a database.

Each node in figure 10 is a process, and arrows show pipes from one process to the next.

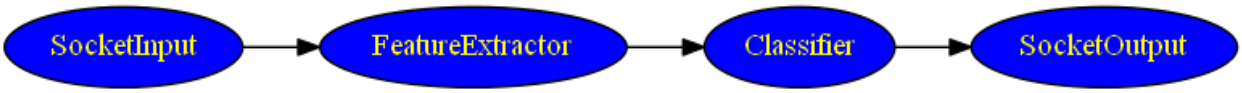

Figure 10: Workflow structure for classifying acoustic events, setup using the sequence pattern.

The feature extractor, or (better) the feature master, will control one or more slaves, depending on the choice of feature extraction techniques. Figure 10 shows the Feature master and its slaves.

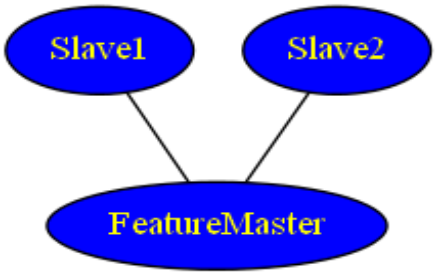

Figure 11: Example of the feature master and its slaves

The output of each slave (sub-vectors of the overall feature vectors) will go into another buffer, where the feature master will assemble the full feature vectors by matching timestamps. The full vectors will be written on the pipe to the classifier process.

After the data is sent to the output daemon, it may be stored along with the configuration in the database, where it could be accessed by the multiagent system or anybody who would like to view it. With our workflow system, multiple workflows can be run at the same time. And a given workflow can run multiple tasks at the same time. Figure 12 shows the flow of the overall architecture.

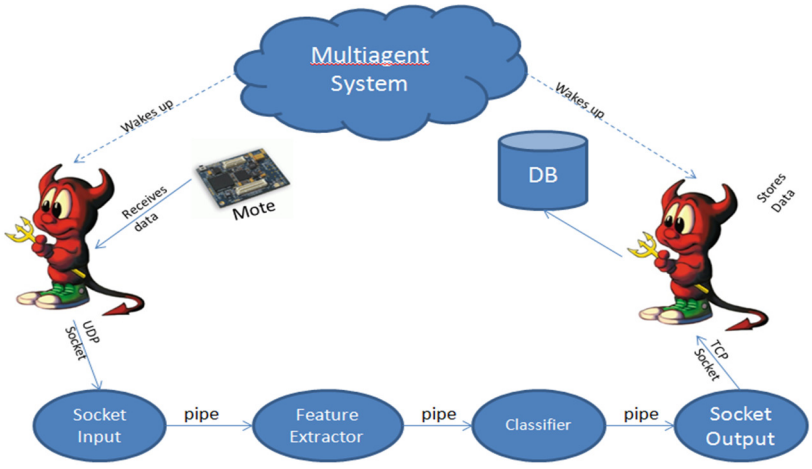

Figure 12: Flow of the overall architecture, with a workflow shown at the bottom.

5.2 Implementation

In our implementation, the daemon processes that we defined in our target system were not implemented nor were the connections to the multiagent system. Our Pattern Driven Multiprocess Workflow (PaDMuW) engine was implemented in Python. An agent or user can submit the setup of their desired workflow to the workflow engine via web services.

The WSDL file contains the description of the web service as well as any ports or complex types required by the methods exposed by the web service--see Appendix A. The workflow is set up using web services. The WSDL file defines two complex types: 1) **`WorkflowModule`** and 2) **`Connection`**. The WSDL file also defines an array for each of these two types: 1) **`WorkflowModuleArray`** and 2) **`ConnectionArray`**. The **`WorkflowModuleArray`** describes all of the active modules that are in the workflow. The **`ConnectionArray`** describes the connections to the active modules. After the web service is called, the **`WorkflowModuleArray`** and **`ConnectionArray`** are passed to an instance of the bootstrap class is created.

Figure 13 shows the definition of the **`WorkflowModule`** type as defined by the web service (in the WSDL file). The **`WorkflowModule`** complex type defines the interface expected for a module in the workflow. This includes the canonical name for the module (viz., **`"name"`**), the name of the class (viz., **`"module_class"`**), the name for the module's location on the physical disk (**`"module_location"`**), and the arguments to the module (**`"module_args"`**). We used the spyne package for python to implement our web

service.

```
<xs:complexType name="WorkflowModule">
    <xs:sequence>
        <xs:element name="name" type="xs:string" minOccurs="0"
            nillable="true"/>
        <xs:element name="module_class" type="xs:string"
            minOccurs="0" nillable="true"/>
        <xs:element name="module_location" type="xs:string"
            minOccurs="0" nillable="true"/>
        <xs:element name="module_args" type="tns:stringArray"
            minOccurs="0" nillable="true"/>
    </xs:sequence>
</xs:complexType>
```

Figure 13: The definition of the ***WorkflowModule*** *type as defined by the web service*

Also in the WSDL file, the **Connection** complex type defines the interface expected for the connections coming into and out of each module. Each connection is identified by the canonical name given in the workflow module of the module to which it connects. Figure 14 shows the definition of the **Connection** type as defined in the WSDL file.

```
<xs:complexType name="Connection">
    <xs:sequence>
        <xs:element name="out_ports" type="xs:string" minOccurs="0"
            nillable="true"/>
        <xs:element name="in_ports" type="xs:string" minOccurs="0"
            nillable="true"/>
    </xs:sequence>
</xs:complexType>
```

Figure 14: The definition of the ***Connection*** *type as defined by the web service*

PaDMuW has various module classes for the various kinds of tasks (such as data input, file output, feature extraction, and classification of feature vectors). Each module class inherits from the **Module** abstract base class, shown in Figure 15. The **Module** abstract base class inherits from the **Process** class, which spawns processes. In our system, each of the created modules can have one of three statuses: 1) Created, 2) Running and 3) Exited. The **Module** class also takes in an instance of the **Messaging** class, which encapsulates the communication

between modules. The administrator provides a concrete **`run()`** method to override the abstract **`run()`** method; the concrete method has all of the code that is required to run the concrete module.

```
import abc
from multiprocessing import Process
import types
from const import *
from Message import *

class Module(Process):
    __metaclass__ = abc.ABCMeta
    def __init__(self, name, messaging):
        Process.__init__(self)
        if not isinstance(name, types.StringType):
            raise Exception("Name must be a string")
        self.status = STATUS_CREATED
        self.name = name
        self.mess = messaging
    @abc.abstractmethod
    def run(self):
        return
```

Figure 15: The implementation of the abstract Module class

Modules communicate using pipes. Each module has zero or more pipes into and out of it. This communication between modules is encapsulated in class **`Messaging`**, shown in Figure 16. The **`Messaging`** class has two pipe attributes, one for the pipes coming into a module, and one for the pipes going out of it. Pipes are used in our case as a way to pass data between modules. Method **`send()`** sends an object to the other end of the pipe, where **`recv()`** receives it; **`recv()`** blocks until there is an object to be received.

```
from  multiprocessing import Pipe
import types
class Messaging(object):
    def __init__(self, inports, outports):
        if len(inports) == 1:
            self.inports = inports[0]
        elif len(inports) > 1:
            self.inports = inports
        if len(outports) == 1:
            self.outports = outports[0]
        elif len(outports) > 1:
            self.outports = outports
    def recv(self):
        if isinstance(self.inports, types.ListType):
            temp = []
            for i in self.inports:
                temp.append(i.recv())
            return tuple(temp)
        else:
            return self.inports.recv()
    def send(self, data):
        if isinstance(self.outports, types.ListType):
            for i in self.outports:
                i.send(data)
        else:
            self.outports.send(data)
```

Figure 16: The implementation of the Messaging class

An instance of class **Bootstrap** is created for each job. It is passed a specification of the workflow: the modules and how they are connected. An instance of **Bootstrap** sets up each module, with the connections into and out of it, and it starts each module. The code for class **Bootstrap** is shown in Figure 15. An instance of **Bootstrap** takes in the **WorkflowModuleArray** and the **ConnectionArray** and sets up the workflow.

```
import types
import sys
from multiprocessing import Pipe
from Message import *
class Bootstrap(object):
    def __init__(self, modules, connections):
        self.modules = modules
        self.conns = connections
        self.mod = []
        self.inPorts = {}
        self.outPorts = {}
        for i in self.modules:
            self.inPorts[i.name]=[]
            self.outPorts[i.name]=[]
    def run(self):
        for i in self.mod:
            i.start()
    def setup(self):
        for i in self.conns:
            p1,p2 = Pipe()
            self.inPorts[i.in_ports].append(p1)
            self.outPorts[i.out_ports].append(p2)
        for i in self.modules:
            if i.module_location not in sys.path:
                sys.path.append(i.module_location)
            module = __import__(i.module_name)
            class_ = getattr(module, i.module_class)
            if i.module_args is not None:
                instance = class_(i.name, Messaging(self.inPorts[i.name],
                                    self.outPorts[i.name]), *i.module_args)
            else:
                instance = class_(i.name, Messaging(self.inPorts[i.name],
                                    self.outPorts[i.name]))
            self.mod.append(instance)
```

Figure 17: The implementation of the Bootstrap class

Data streams from the wireless sensor network are simulated with data streams from an hdf5 file. The **`Input`** module reads acoustic emission data from an HDF5 dataset. Figure 18 shows the **`run()`** method of the **`Input`** module. The **`Output`** module takes the classification and writes it out to a file.

```
def run(self):
  self.status = STATUS_RUNNING
  f = h5py.File("Waveforms.hdf5", "r")
  data = f.get('CrackGrowth')
  for i in data:
      self.mess.send((i, time.clock()))
  self.mess.send(-1)
  self.status = STATUS_EXIT
  self.f.close()
```

Figure 18: The implementation of the run method for the ***`Input`*** *module*

CHAPTER 6

Conclusion

For structural health monitoring (SHM), a structure is monitored in near real-time. SHM involves observing the civil or mechanical structure over time using periodically spaced measurements, extracting the damage-sensitive features, and statistically analyzing these features to determine the current state of system health. With regard to maintenance, SHM is used to screen the condition and provide real-time information on the integrity of the structure.

The target architecture described here aims to provide a workflow engine that allows for data to be processed in near real-time. Our target setup has a multiagent system with agents that advocate for various feature extraction and machine learning techniques. The agents bid on which techniques should be connected together in a workflow. The lead agent then communicates this specification to the workflow engine. The workflow engine sets up the workflow then attaches input and output sources.

We have currently implement key parts of the target architecture in Python. One job has multiple modules and connections (pipes). Module classes are extension of the **`Module`** abstract base class. For each job, a user or agent uses web services to tell the workflow engine what arguments it takes: where the modules reside on disk, the name of the module classes, the canonical names of the modules. In the larger implementation, there will be daemon process controlling data input and information output and a NoSQL database to store the information and for job setup.

The basic data that we work with consists of acoustic signals that we interpret regarding the events that produce them (e.g., crack growth). At the lowest level, for a given data stream, the agents in the target system will find techniques to extract features from the stream and

classification techniques that take vectors of these features as input and produce classifications of the corresponding events. The classifiers are trained using machine-learning techniques.

Following Axiom III from Worden et al [WFM2007], we used two unsupervised and three supervised learning techniques to look at the different aspects of SHM. The results of these learning techniques will allow us to look at the problem of damage identification at a higher level of sophistication. The unsupervised learning techniques are k-means and self-organizing maps (SOM). Supervised learning techniques are support vector machines (SVMs), naive Bayes classifiers, and feed-forward neural networks (FNNs). We tested a version with principal component analysis (PCA) as a frontend to reduce the dimensionality of the data (usually to three principal components) for each technique with SOM being the exception and we tested another version without PCA. Of all of the supervised learning techniques the SVM with the RBF kernel was the most accurate, but the SVM with a linear kernel function performed the fastest. Of the unsupervised learning techniques, the SOM with four clusters had the three clusters the SOM with three clusters had. Its fourth cluster was empty, suggesting that there really are just three clusters among the data.

Future work will investigate other types of machine learning techniques such as weak, ensemble and semi-supervised learning. We will also look into machine learning techniques that work for other types of material such as carbon fiber laminate. We also plan on investigating other agent collaboration patterns. Finally, we will look into dimensionality reduction techniques and try to determine what features are the most important and how to make them dominate.

References

[ABJ2004] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludaescher, B., & Mock, S. (2004, March). Kepler: Towards a grid-enabled system for scientific workflows. In the Workflow in Grid Systems Workshop in GGF10-The Tenth Global Grid Forum, Berlin, Germany.

[Bis2006] Bishop, C. M. (2006). Pattern recognition and machine learning (Vol. 4, No. 4, p. 12). New York: springer.

[BCG07] Bellifemine, F.L., Caire, G., and Greenwood, D. Developing Multi-Agent Systems with JADE. Wiley, 2007.

[BFH2005] Brown, J. L., Ferner, C. S., Hudson, T. C., Stapleton, A. E., Vetter, R. J., Carland, T., & Wood, M. (2005). Gridnexus: A grid services scientific workflow system. International Journal of Computer Information Science (IJCIS), 6(2), 72-82.

[BVi2005] Buhler, P. A., & Vidal, J. M. (2005). Towards adaptive workflow enactment using multiagent systems. Information technology and management, 6(1), 61-87.

[CKR2007] Couvares, P., Kosar, T., Roy, A., Weber, J., & Wenger, K. (2007). Workflow management in condor. In Workflows for e-Science (pp. 357-375). Springer London.

[DBG2004] Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., & Livny, M. (2004, January). Pegasus: Mapping scientific workflows onto the grid. In Grid Computing (pp. 11-20). Springer Berlin Heidelberg.

[DHS1999] Duda, R. O., Hart, P. E., & Stork, D. G. (1999). Pattern classification. John Wiley & Sons.

[DSm1983] Davis, R., & Smith, R. G. (1983). Negotiation as a metaphor for distributed problem solving. Artificial intelligence, 20(1), 63-109.

[DSS2005] Deelman, E., Singh, G., Su, M. H., Blythe, J., Gil, Y., Kesselman, C., & Katz, D. S. (2005). Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming, 13(3), 219-237.

[EKS2010] Esterline, A., Krishnamurthy, K., Sundaresan, M., Alam, T., Rajendra, D., & Wright, W. (2010). Classifying Acoustic Emission Data in Structural Health Monitoring using Support Vector Machines. In Proceedings of AIAA Infotech@ Aerospace 2010 Conference.

[FWo2007] Farrar, C. R., & Worden, K. (2007). An introduction to structural health monitoring. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 365(1851), 303-315.

[FJK2004] Foster, I., Jennings, N. R., & Kesselman, C. (2004, July). Brain meets brawn: Why grid and agents need each other. In Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1 (pp. 8-15). IEEE Computer Society.

[FLi2007] Farrar, C. R., & Lieven, N. A. (2007). Damage prognosis: the future of structural health monitoring. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 365(1851), 623-632.

[FPF2011] Figueiredo, E., Park, G., Farrar, C. R., Worden, K., & Figueiras, J. (2011). Machine learning algorithms for damage detection under operational and environmental variability. Structural Health Monitoring, 10(6), 559-572. [LFi1998] Labrou, Y., & Finin, T. (1998). Semantics and conversations for an agent communication language. Readings in agents, 235-242.

[HDF2014] The HDF Group. (2014) “WELCOME TO THE HDF5 HOME PAGE!” available: http://www.hdfgroup.org/HDF5/

[HFH2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. ACM SIGKDD explorations newsletter, 11(1), 10-18.

[IEEE2009] IEEE FIPA Standards Committee. (2009) "FIPA Contract Net Interaction Protocol Specification" available: http://www.fipa.org/specs/fipa00029/SC00029H.html

[JOP2007] Jones, E., Oliphant, T., & Peterson, P. (2007). SciPy: Open source scientific tools for Python, 2001–. URL http://www. scipy. org, 73, 86.

[Kra1992] Kramer, M. A. (1992). Autoassociative neural networks. Computers & chemical engineering, 16(4), 313-328.

[KLa1980] Klema, V., & Laub, A. J. (1980). The singular value decomposition: Its computation and some applications. Automatic Control, IEEE Transactions on, 25(2), 164-176.

[Las2005] Laszewski, G. V. "Java cog kit workflow concepts for scientific experiments." Argonne National Laboratory, Argonne, IL, USA Technique Report (2005).

[LMP2003] Luck, M., McBurney, P., & Preist, C. (2003). Agent technology: Enabling next generation computing (a roadmap for agent based computing). AgentLink/University of Southampton.

[Mck2011] McKinney, W (2011). McKinney, W. (2011). pandas: a Foundational Python Library for Data Analysis and Statistics. Python for High Performance and Scientific Computing, 1-9.

[Num2012] Numpy. (2012) "Numpy" available: http://www.numpy.org/

[Oli07] Oliphant, Travis E. "Python for scientific computing." Computing in Science & Engineering 9.3 (2007), pp. 10-20.

[PJL2005] Patel, J., Teacy, W. L., Jennings, N. R., Luck, M., Chalmers, S., Oren, N., & Thompson, S. (2005). Agent-based virtual organisations for the grid. Multiagent and Grid Systems, 1(4), 237-249.

[PTT2004] Poggi, A., Tomaiuolo, M., & Turci, P. (2004, June). Extending JADE for agent grid applications. In Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004. 13th IEEE International Workshops on (pp. 352-357). IEEE.

[PVG2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. The Journal of Machine Learning Research, 12, 2825-2830.

[San1999] Sanner, M. (1999). Python: a programming language for software integration and development. J Mol Graph Model, 17(1), 57-61.

[Smi1980] Smith, R. (1980). Communication and control in problem solver. IEEE Transactions on computers, 29, 12.

[SBW2010] Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., ... & Schmidhuber, J. (2010). PyBrain. The Journal of Machine Learning Research, 11, 743-746.

[SVo1996] Singh, M. P., & Vouk, M. A. (1996, May). Scientific workflows: scientific computing meets transactional workflows. In Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions (pp. 28-34).

[VVa2004] Van Der Aalst, W., & Van Hee, K. M. (2004). Workflow management: models, methods, and systems. MIT press.

[VBS2004] Vidal, J. M., Buhler, P. A., & Stahl, C. (2004). Multiagent systems with workflows IEEE Internet Computing, 8(1), 76.

[Wik2014a] Wikipedia. (2014) “Wikipedia: Artificial neural network” available: http://en.wikipedia.org/wiki/Artificial_neural_network

[Wik2014b] Wikipedia. (2014) “Wikipedia: Support vector machine” available: http://en.wikipedia.org/wiki/Support_vector_machine

[Wik2014c] Wikipedia. (2014) “Wikipedia: Sigmoid function” available: http://en.wikipedia.org/wiki/Sigmoid_function

[Woo2009] Wooldridge, M. (2009). An introduction to multiagent systems. John Wiley & Sons.

[WFM2007] Worden, K., Farrar, C. R., Manson, G., & Park, G. (2007). The fundamental axioms of structural health monitoring. Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science, 463(2082), 1639-1664.

[WRX1999] Wan, F., Rustogi, S. K., Xing, J., & Singh, M. P. (1999, August). Multiagent workflow management. In IJCAI Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business, Stockholm, Sweden.

Appendix A

This WSDL file with the definitions of the complex types **WorkflowModule, Connection, WorkflowModuleArray** and **ConnectionArray** with the description of the Web service.

```
<wsdl:definitions
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
  xmlns:tns="shm.wf"
  xmlns:plink="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
  xmlns:xop="http://www.w3.org/2004/08/xop/include"
  xmlns:soap12env="http://www.w3.org/2003/05/soap-envelope/"
  xmlns:senc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:s1="module"
  xmlns:s0="connection"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:senv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soap12enc="http://www.w3.org/2003/05/soap-encoding/"
  targetNamespace="shm.wf" name="Application">
    <wsdl:types>
    <xs:schema targetNamespace="module"
      elementFormDefault="qualified">
      <xs:import namespace="shm.wf"/>
      <xs:complexType name="WorkflowModule">
        <xs:sequence>
          <xs:element name="name" type="xs:string" minOccurs="0"
             nillable="true"/>
          <xs:element name="module_class" type="xs:string"
           minOccurs="0" nillable="true"/>
         <xs:element name="module_location" type="xs:string"
           minOccurs="0" nillable="true"/>
        <xs:element name="module_name" type="xs:string" minOccurs="0"
          nillable="true"/>
        <xs:element name="module_args" type="tns:stringArray"
          minOccurs="0" nillable="true"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="WorkflowModuleArray">
        <xs:sequence>
          <xs:element name="WorkflowModule" type="s1:WorkflowModule"
            minOccurs="0" maxOccurs="unbounded" nillable="true"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="WorkflowModule" type="s1:WorkflowModule"/>
      <xs:element name="WorkflowModuleArray"
        type="s1:WorkflowModuleArray"/>
```

```
  </xs:schema>
  <xs:schema targetNamespace="connection"
    elementFormDefault="qualified">
    <xs:complexType name="Connection">
      <xs:sequence>
        <xs:element name="out_ports" type="xs:string" minOccurs="0"
          nillable="true"/>
        <xs:element name="in_ports" type="xs:string" minOccurs="0"
          nillable="true"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ConnectionArray">
      <xs:sequence>
        <xs:element name="Connection" type="s0:Connection"
          minOccurs="0" maxOccurs="unbounded" nillable="true"/>
      </xs:sequence>
    </xs:complexType>
    <xs:element name="Connection" type="s0:Connection"/>
    <xs:element name="ConnectionArray" type="s0:ConnectionArray"/>
  </xs:schema>
  <xs:schema targetNamespace="shm.wf"
    elementFormDefault="qualified">
    <xs:import namespace="connection"/>
    <xs:import namespace="module"/>
    <xs:complexType name="stringArray">
      <xs:sequence>
        <xs:element name="string" type="xs:string" minOccurs="0"
          maxOccurs="unbounded" nillable="true"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="runWorkflowResponse">
      <xs:sequence>
        <xs:element name="runWorkflowResult" type="xs:integer"
          minOccurs="0" nillable="true"/>
      </xs:sequence>
    </xs:complexType>
    <xs:complexType name="runWorkflow">
      <xs:sequence>
        <xs:element name="mod" type="s1:WorkflowModuleArray"
          minOccurs="0" nillable="true"/>
        <xs:element name="conn" type="s0:ConnectionArray"
          minOccurs="0" nillable="true"/>
      </xs:sequence>
    </xs:complexType>
    <xs:element name="stringArray" type="tns:stringArray"/>
    <xs:element name="runWorkflowResponse"
      type="tns:runWorkflowResponse"/>
    <xs:element name="runWorkflow" type="tns:runWorkflow"/>
  </xs:schema>
</wsdl:types>
<wsdl:message name="runWorkflow">
  <wsdl:part name="runWorkflow" element="tns:runWorkflow"/>
```

```
  </wsdl:message>
  <wsdl:message name="runWorkflowResponse">
    <wsdl:part name="runWorkflowResponse"
      element="tns:runWorkflowResponse"/>
  </wsdl:message>
  <wsdl:service name="WorkflowService">
    <wsdl:port name="Application" binding="tns:Application">
      <soap:address location="http://localhost:8000/"/>
    </wsdl:port>
  </wsdl:service>
  <wsdl:portType name="Application">
    <wsdl:operation name="runWorkflow" parameterOrder="runWorkflow">
      <wsdl:input name="runWorkflow" message="tns:runWorkflow"/>
      <wsdl:output name="runWorkflowResponse"
        message="tns:runWorkflowResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="Application" type="tns:Application">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="runWorkflow">
      <soap:operation soapAction="runWorkflow" style="document"/>
      <wsdl:input name="runWorkflow">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="runWorkflowResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
</wsdl:definitions>
```

Appendix B

Supervised Machine Learning Code

We present code for the supervised learning techniques (Gaussian naïve Bayes, feed-forward neural net, and support vector machines with various kernel functions) with and without principal component analysis (PCA). There is a file for each technique, with and without PCA.

Gaussian Naïve Bayes with PCA

```
from sklearn.cross_validation import StratifiedKFold
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB
import time
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

classifier = GaussianNB()
X = np.loadtxt('testrun50.txt', delimiter=',', dtype=float)
y = np.loadtxt('testrun50class.txt', delimiter=',', dtype=float)

cv = StratifiedKFold(y, n_folds=5)

X_temp = PCA(n_components=3).fit_transform(X, y)

timing = []
accuracy = []
timingStd = []
accuracyStd =[]

i = 0
while i <= 25:
    tempTiming = []
    tempAccuracy = []
    for train, test in cv:
        classifier = classifier.fit(X_temp[train], y[train])
        t1 = time.clock()
        P = classifier.predict(X_temp[test])
        t2 = time.clock()
        tempAccuracy.append(accuracy_score(y[test], P))
        tempTiming.append((t2-t1)*1000)
    i += 1
    accuracy.append(np.array(tempAccuracy).mean())
    accuracyStd.append(np.array(tempAccuracy).std())
    timing.append(np.array(tempTiming).mean())
    timingStd.append(np.array(tempTiming).std())
```

Gaussian Naïve Bayes without PCA

```
from sklearn.cross_validation import StratifiedKFold
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB
import time
from sklearn.metrics import accuracy_score

classifier = GaussianNB()
X = np.loadtxt('testrun50.txt', delimiter=',', dtype=float)
y = np.loadtxt('testrun50class.txt', delimiter=',', dtype=float)

cv = StratifiedKFold(y, n_folds=5)

timing = []
accuracy = []
timingStd = []
accuracyStd =[]

i = 0

while i <= 25:
    tempTiming = []
    tempAccuracy = []
    for train, test in cv:
        classifier = classifier.fit(X[train], y[train])
        t1 = time.clock()
        P = classifier.predict(X[test])
        t2 = time.clock()
        tempAccuracy.append(accuracy_score(y[test], P))
        tempTiming.append((t2-t1)*1000)
    i += 1
    accuracy.append(np.array(tempAccuracy).mean())
    accuracyStd.append(np.array(tempAccuracy).std())
    timing.append(np.array(tempTiming).mean())
    timingStd.append(np.array(tempTiming).std())
```

Feed-forward Neural Net without PCA

```
import numpy as np
from pybrain.datasets            import ClassificationDataSet
from pybrain.tools.shortcuts     import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from sklearn.cross_validation import StratifiedKFold
import time
from sklearn.metrics import accuracy_score
def data(trainSet, trainClass, testSet, testClass):
    input_size = trainSet.shape[1]
    trndata = ClassificationDataSet(input_size, 1, nb_classes=6)
    input_size = testSet.shape[1]
    tstdata = ClassificationDataSet(input_size, 1, nb_classes=6)
    for i, k in zip(trainSet, trainClass):
        trndata.appendLinked(tuple(i), (k,))
    for i, k in zip(testSet, testClass):
        tstdata.appendLinked(tuple(i), (k,))
    trndata.assignClasses()
    tstdata.assignClasses()
    trndata._convertToOneOfMany()
    tstdata._convertToOneOfMany()
    return trndata, tstdata
X = np.loadtxt('testrun50.txt', delimiter=',', dtype=float)
y = np.loadtxt('testrun50class.txt', delimiter=',', dtype=float)
cv = StratifiedKFold(y, n_folds=5)
timing = []; accuracy = []; timingStd = []; accuracyStd =[]
i = 0
while i <= 25:
    fnn = buildNetwork( 6, 8, 6, bias=False, outputbias=False)
    tempTiming = []
    tempAccuracy = []
    for train, test in cv:
        trndata, tstdata = data(X[train], y[train], X[test], y[test])
        trainer = BackpropTrainer(fnn, dataset=trndata, momentum=0.1,
                                  verbose=False, weightdecay=0.01)
        trnerr,valerr = trainer.trainUntilConvergence(dataset=trndata,
                                                      maxEpochs=500)
        t1 = time.clock()
        P = trainer.testOnClassData(dataset=tstdata )
        t2 = time.clock()
        tempTiming.append((t2-t1)*1000)
        tempAccuracy.append(accuracy_score(y[test], P))
    i+=1
    timing.append(np.array(tempTiming).mean())
    timingStd.append(np.array(tempTiming).std())
    accuracy.append(np.array(tempAccuracy).mean())
    accuracyStd.append(np.array(tempAccuracy).std())
```

Feed-forward Neural Net with PCA

```
from pybrain.datasets            import ClassificationDataSet
from pybrain.tools.shortcuts     import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer
from sklearn.cross_validation import StratifiedKFold
import time
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA

def data(trainSet, trainClass, testSet, testClass):
    input_size = trainSet.shape[1]
    trndata = ClassificationDataSet(input_size, 1, nb_classes=6)
    input_size = testSet.shape[1]
    tstdata = ClassificationDataSet(input_size, 1, nb_classes=6)
    for i, k in zip(trainSet, trainClass):
        trndata.appendLinked(tuple(i), (k,))
    for i, k in zip(testSet, testClass):
        tstdata.appendLinked(tuple(i), (k,))
    trndata.assignClasses()
    tstdata.assignClasses()
    trndata._convertToOneOfMany()
    tstdata._convertToOneOfMany()
    return trndata, tstdata

X = np.loadtxt('testrun50.txt', delimiter=',', dtype=float)
y = np.loadtxt('testrun50class.txt', delimiter=',', dtype=float)

X_temp = PCA(n_components=3).fit_transform(X, y)

cv = StratifiedKFold(y, n_folds=5)

timing = []
accuracy = []
timingStd = []
accuracyStd =[]

i = 0

while i <= 25:
    fnn = buildNetwork( 3, 6, 6, bias=False, outputbias=False)
    tempTiming = []
    tempAccuracy = []
    for train, test in cv:
        trndata, tstdata = data(X_temp[train], y[train], X_temp[test],
                                y[test])
        trainer = BackpropTrainer(fnn, dataset=trndata, momentum=0.1,
          verbose=False, weightdecay=0.01)
        trnerr,valerr = trainer.trainUntilConvergence(dataset=trndata,
          maxEpochs=500)
```

```
    t1 = time.clock()
    P = fnn.activateOnDataset(tstdata)
    t2 = time.clock()
    tempTiming.append((t2-t1)*1000)
tempAccuracy.append(accuracy_score(y[test], P.argmax(axis=1)))
i+=1
timing.append(np.array(tempTiming).mean())
timingStd.append(np.array(tempTiming).std())
accuracy.append(np.array(tempAccuracy).mean())
accuracyStd.append(np.array(tempAccuracy).std())
```

Support Vector Machine without PCA

```
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA
from sklearn.cross_validation import StratifiedKFold
import time
from sklearn import svm

X = np.loadtxt('testrun50.txt', delimiter=',', dtype=float)
y = np.loadtxt('testrun50class.txt', delimiter=',', dtype=float)

kernels = ['rbf','poly','sigmoid','linear']
cv = StratifiedKFold(y, n_folds=5)
timing = {}
accuracy = {}
timingStd = {}
accuracyStd ={}

for j in kernels:

    classifier =  svm.SVC(kernel=j ,probability=True, C=2048,
                          gamma=0.03125)

    i = 0
    timing[j] = []
    accuracy[j] = []
    timingStd[j] = []
    accuracyStd[j] = []

    while i <= 25:
        tempTiming = []
        tempAccuracy = []
        for train, test in cv:
            classifier = classifier.fit(X[train], y[train])
            t1 = time.clock()
            P = classifier.predict(X[test])
            t2 = time.clock()
            tempAccuracy.append(accuracy_score(y[test], P))
            tempTiming.append((t2-t1)*1000)
        i += 1
        accuracy[j].append(np.array(tempAccuracy).mean())
        accuracyStd[j].append(np.array(tempAccuracy).std())
        timing[j].append(np.array(tempTiming).mean())
        timingStd[j].append(np.array(tempTiming).std())
```

Support Vector Machine with PCA

```
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.decomposition import PCA
from sklearn.cross_validation import StratifiedKFold
import time
from sklearn import svm

X = np.loadtxt('testrun50.txt', delimiter=',', dtype=float)
y = np.loadtxt('testrun50class.txt', delimiter=',', dtype=float)

X_temp = PCA(n_components=3).fit_transform(X,y)

kernels = ['rbf','poly','sigmoid','linear']
cv = StratifiedKFold(y, n_folds=5)


timing = {}
accuracy = {}
timingStd = {}
accuracyStd ={}

for j in kernels:


    classifier =  svm.SVC(kernel=j ,probability=True, C=2048,
                          gamma=0.03125)

    i = 0
    timing[j] = []
    accuracy[j] = []
    timingStd[j] = []
    accuracyStd[j] = []

    while i <= 25:
        tempTiming = []
        tempAccuracy = []
        for train, test in cv:
            classifier = classifier.fit(X_temp[train], y[train])
            t1 = time.clock()
            P = classifier.predict(X_temp[test])
            t2 = time.clock()
            tempAccuracy.append(accuracy_score(y[test], P))
            tempTiming.append((t2-t1)*1000)
        i += 1
        accuracy[j].append(np.array(tempAccuracy).mean())
        accuracyStd[j].append(np.array(tempAccuracy).std())
        timing[j].append(np.array(tempTiming).mean())
        timingStd[j].append(np.array(tempTiming).std())
```