The Cognitive Foundry: A Flexible Platform for Intelligent Agent Modeling

Justin Basilico, Zachary Benz, Kevin R. Dixon
Sandia National Laboratories
P.O. Box 5800 MS 1188
Albuquerque, NM 87185-1188
{jdbasil, zobenz, krdixon}@sandia.gov

Keywords: cognitive model, machine learning, agent simulation

ABSTRACT: *The Cognitive Foundry is a unified collection of tools for Cognitive Science and Technology applications, supporting the development of intelligent agent models. The Foundry has two primary components designed to facilitate agent construction: the Cognitive Framework and Machine Learning packages. The Cognitive Framework provides design patterns and default implementations of an architecture for evaluating theories of cognition, as well as a suite of tools to assist in the building and analysis of theories of cognition. The Machine Learning package provides tools for populating components of the Cognitive Framework from domain-relevant data using automated knowledge-capture techniques. This paper describes the Cognitive Foundry with a focus on its application within the context of agent behavior modeling.*

1 Introduction

The Cognitive Foundry is a unified collection of software tools for Cognitive Science and Technology (CS&T) applications. CS&T is a developing field at the intersection of cognitive science, computer science, and engineering that takes fundamental concepts from cognitive science and neuroscience and deploys systems implementing these ideas. To further the goals of this multidisciplinary field, we have designed the Foundry to be a robust, extensible platform to support research, rapid prototyping, and system deployment, while adhering to rigorous software-engineering principles. Instead of pushing a single theory of cognition, the Foundry contains reusable software components and algorithms designed to support a wide variety of development needs. The software architecture of the Foundry promotes reusability, maintainability, and cross-platform compatibility, without sacrificing computational resources by leveraging best-in-class numerical packages.

2 Why the Cognitive Foundry

As the Sandia National Laboratories CS&T program grew from its infancy, the use cases for our cognitive-modeling software evolved as well, driven by both researcher- and customer-centric needs. For basic-research experimentation, researchers wanted a reusable toolkit that allowed the rapid prototyping and visualization of new ideas and hypotheses in modeling cognition, as well as statistical-validation techniques to compare performance against a standard battery of existing results from the literature. Our customers have expressed an increasing interest in automatically populating cognitive models through automated knowledge-capture algorithms, processing large amounts of data efficiently, parallel and distributed computation, and verifiable software-development processes. We meet these seemingly divergent requirements by creating a graduated set of programmer interfaces that enable both research experimentation and system deployment, and the Foundry assists users by providing a set of tools that accompany those interfaces. For example, if a particular project would benefit from parallel computation, then the user can implement the rather simple methods associated with the Concurrent Cognitive Module interface. The Foundry then automatically provides the mechanisms to execute the code in a parallel fashion, with no additional burden placed on the user. We chose this graduated-interface strategy to support both the general case by providing a robust set of core functionality while also providing the infrastructure for rapidly constructing special-purpose applications that may require more intricate or onerous functionality. The manifestation of the Cognitive Foundry philosophy is that we provide a number of interfaces, some of which are easy to implement, while others may be more time consuming. The more interfaces, or functionality, that a Foundry developer can implement, the more Foundry tools can be brought to bear on the problem. Thus, users can select the parts that provide the best benefit to a specific project. The Foundry's Cognitive Framework provides a reusable framework for building agents and experimenting with cognitive simulation. The Machine Learning package provides a large library of powerful learning algorithm implementations that can be used on their own or to create

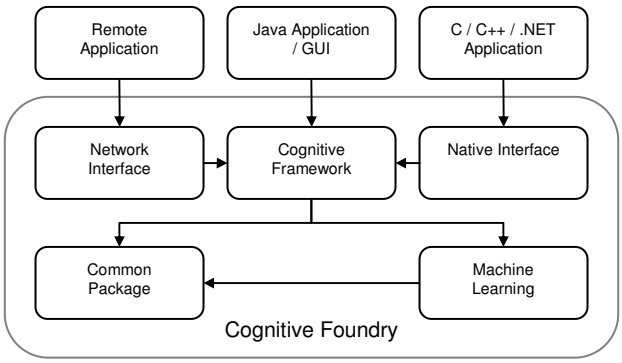

Figure 1. *Dependency graph diagram of Cognitive Foundry components and interoperability methods.*

components of the Cognitive Framework. Each component in the Cognitive Foundry is a tool that we have found useful for building cognitive models and cognitive systems.

2.1 Benefits of the Cognitive Foundry

One of the primary lessons learned from the maturation of Sandia's CS&T program is that the Cognitive Foundry must provide coverage and support of a cognitive system from idea to deployment, not just a cognitive simulation. The Cognitive Foundry's modularity allows users to determine which components are necessary, or provide value, to a particular project by selecting the tools used to solve common cognitive-systems tasks, while being assured that rigorous software-engineering quality processes have been employed. The Foundry also provides a well-defined path for components to incorporate the latest research ideas and transition them into a deployed system. Applications built on the Cognitive Foundry's Framework and Machine Learning packages can immediately make use of new modules for cognitive simulation and new algorithms that conform to the common set of interfaces.

2.2 Communicating with the Cognitive Foundry

The Cognitive Foundry is written in the Sun Java 1.5 programming language. We have also developed several other ways to interoperate with the Cognitive Foundry from non-Java applications, as shown in Figure 1. For example, we have created a native-machine interface that allows applications written in other programming languages, such as ANSI C/C++ or Microsoft .NET (C#, Visual Basic) to call directly into the Foundry API. The Cognitive Foundry also has a Network Interface library to facilitate connecting to, viewing, controlling, and launching models over a network. Finally, The Cognitive Foundry has a graphical user interface to support the inspection or manipulation of cognitive models during creation and execution.

2.3 Design Methodology

On a philosophical level, the design of the Cognitive Foundry has followed a graduated interface approach. That is, the Cognitive Foundry is built on top of a set of well defined, hierarchical interfaces. For example, the Cognitive Foundry defines the functionality that a Cognitive Model (a memory space, collection of modules, etc.) and Multivariate Minimization Algorithm (an objective function, an iteration loop, etc.) must possess. The Cognitive Foundry then provides one or more default implementations of these interfaces. However, developers can always create their own tailor-made implementations if existing ones do not meet their needs, allowing researchers to test new ideas and hypotheses quickly. Since other tools in the Cognitive Foundry provide functionality at the interface level, new implementations can automatically exploit existing functionality provided by other components in the Foundry by conforming to a defined interface. There are several benefits to this interface-centric component-based approach. It provides an easy mechanism for customizing existing object implementations in the Foundry. It also gives the ability to pick the specific objects from the Foundry that are useful for a certain application. Finally, it creates an integration point for many applications, which defines an easy transition path from research to deployment.

3 Cognitive Framework

The Foundry's Cognitive Framework is a modular software architecture for cognitive simulation designed for use in CS&T applications. The Cognitive Framework itself is a collection of interfaces, which allows Framework users to either leverage the existing tools in the Framework or specify different implementations to fit their specific needs in order to test new ideas and hypotheses.

3.1 Cognitive Model

The Cognitive Framework is designed so that different, and possibly competing, elements of a "theory of cognition" can be instantiated as desired, as shown in Figure 2. This is accomplished by having a Cognitive Module perform some aspect of a psychologically plausible cognitive process. A Cognitive Model, then, contains a collection of Cognitive Modules whose purpose is to instantiate some aspect of cognition. The main components of a Cognitive Model are shown in Figure 3. Conceptually, Cognitive Modules are the workhorse classes inside a Cognitive Model. A Cognitive Model and its corresponding Cognitive Modules use a single central container to store all state information, called a Cognitive Model State. The state contains the sufficient information needed to allow a Cognitive Model

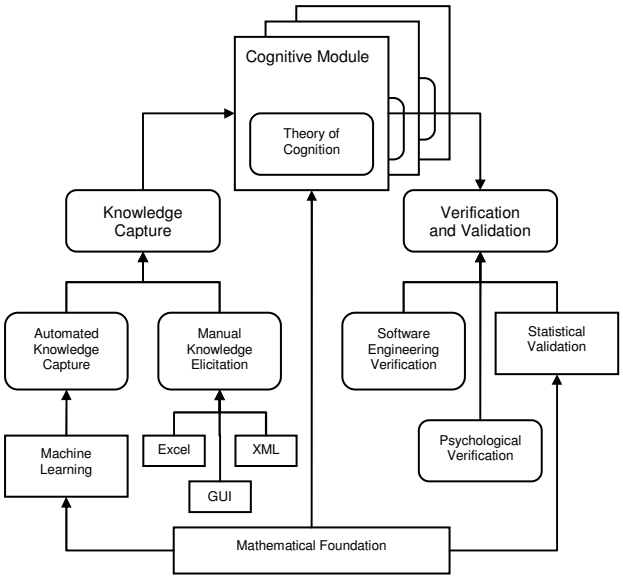

Figure 2. *Life-cycle diagram of a Cognitive Module. Knowledge capture techniques are used to populate a module from data. Verification and validation is done on a completed module.*

to resume execution later, or on another machine, without altering the results of a simulation. Furthermore, the Cognitive Model State can be sent across a network to distribute computation and exploit the parallelization inherent in many cognitive agent simulations. The Cognitive Framework supports the serialization of Cognitive Models using a binary format, human-editable XML, or comma-separated values.

3.2 Cognitive Module

The Cognitive Module interface gives users fine-grained control of what aspects of a "theory of cognition" are incorporated into a particular Cognitive Model. The primary functionality of a Cognitive Module is contained in its "update" method. The update method is given the current Cognitive Model State, along with the previous state of the Cognitive Module and the current set of sensory inputs. The update method of each Cognitive Module returns its state for the next time step. Cognitive Modules can pass information to one another through the Cognitive Model State object, which contains a blackboard-like component where information can be posted and read by any Cognitive Module.

3.3 Cognitive Element (Cogxel)

The Framework operates on a key data structure interface: the *cogxel*. Cogxel stands for "cognitive element" and is modeled after the word "pixel," which means picture element. A cogxel is the fundamental unit of data in the Cognitive Framework. Cogxels normally reside in the Cognitive Model State (blackboard), as they represent the overall state of the model. Cogxels are accessed by a high-level "semantic label" that describes the data contained by cogxel, such as "Heart Rate" or "Context12345." Cogxels are in turn stored in a manner that allows constant-time lookup of the data contained by the cogxel, allowing efficient retrieval along with semantically meaningful storage. A default cogxel implementation consists of a semantic label and a scalar activation level. However, being an interface, other applications have created their own cogxel implementations that use bindings, activation flags, etc. Thus, cogxels can be adapted to fit the specific needs of an application or a theory of cognition, such as to model symbolic, non-symbolic, or structured information.

3.4 Lightweight Implementation

The Cognitive Framework Lite is an implementation of the Cognitive Framework interfaces that abide by the specifications set forth by some of our customers and stakeholders. It is specifically designed for having many agent models within a larger simulation. It is a "lite" version of the interfaces because it does not allow for the dynamic addition or removal of modules while the model is running. This means that all modules must be added to the model at its creation time. This allows for compact data structures and a simple, fast update loop within the model. The "lite" version of the model can run within a high-performance computing (supercomputer) environment, with many models executing on a single processor with a minimal memory footprint. The "lite" implementation also contains lightweight implementations of some basic cognitive modules, such as a semantic network and perception modules.

3.5 Concurrent Implementation

Cognitive Models requiring significant computational resources can employ a concurrent implementation of the Cognitive Framework. For example, we are currently developing a theory of analogical reasoning using physiological models of visual and prefrontal cortex that

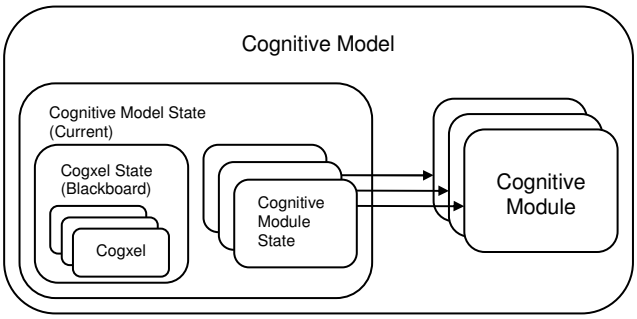

Figure 3. *Cognitive Model components. One object encapsulates the entire state of the model. The design emphasizes modularity and state encapsulation.*

employs large numbers of computational units representing cortical columns in the human brain. The concurrent implementation provides a means for distributing computation in parallel across available computing resources on local machines, namely multi-core and multi-processor computers. It also provides the basis for future extensions of the Cognitive Framework to support distributed computation across networked computational resources.

The concurrent implementation breaks up the update method of a Cognitive Module into three steps: read state, evaluate, and write state. A concurrent cognitive model can then operate as follows: all modules read required input state information sequentially, followed by parallel execution of each module's evaluate method, with the update method completed by having each module write out its state information in sequence. The Cognitive Foundry provides a default implementation of this parallel computation, which employs a thread pool with a user-defined number of threads onto which module evaluations are scheduled for execution. In this way, a user may fully utilize multiple cores and processors on a desktop computer for model execution. Future implementations will allow distributed computation through evaluations on network compute resources.

3.6 Perception Module

Every CS&T application needs a way to glean information from the environment. The Cognitive Framework accomplishes this by specifying a perception module, capable of taking some external data source and transforming relevant information from it into a form that other modules can process (*i.e.*, cogxels). Perception is represented as a module such that different approaches to perception can be experimented with, and possibly combined.

3.7 Manual Knowledge Elicitation and Automated Knowledge Capture

Perhaps the most intricate and application-specific step in creating a Cognitive Model is providing the proper set of parameters needed to configure the Cognitive Modules. Some Cognitive Modules can use predefined parameter sets, but most need to be loaded with domain-specific information. In general, there are two approaches to solve this problem: manual knowledge elicitation and automated knowledge capture. Manual knowledge elicitation involves a structured interview with a subject matter expert and the subsequent encoding of this information into a form that the Cognitive Foundry can understand. The Cognitive Foundry provides support for manual knowledge elicitation through both user-interface components and human-editable input file formats, such as Microsoft Excel and XML. Not surprisingly, manual knowledge elicitation is quite labor intensive and intractable for application domains where there are many interacting factors, as the combinatorics get out of hand quickly. Furthermore, slight changes in the application domain require a complete retooling of the elicitation process, generally resulting in another interview of the subject matter expert. The alternative approach, automated knowledge capture, involves collecting a relevant set of data in the application domain and then using that data along with a set of machine-learning, or data-mining, algorithms to create the necessary information needed for the Cognitive Module. The Cognitive Foundry provides support for this approach through the Machine Learning package, discussed in section 4.

The main constraint that automated knowledge-capture techniques have is that they all require relevant data for the problem they are addressing, usually in large amounts. If no relevant data exists, or can be readily collected, then automated knowledge-capture techniques may not be of much help in solving the problem. However, if a user has access to, or a mechanism to collect, relevant data, then an automated knowledge-capture technique probably exists that can accurately populate a cognitive model to predict or categorize the problem at hand. In some cases, real data may have to be augmented with synthetic data to provide enough support for the automated knowledge-capture algorithms. However, there exist procedures for this augmentation as well.

The Foundry contains a Framework-Learning integration package that was designed so that components of the Cognitive Framework package could be used in conjunction with machine-learning algorithms. In other words, this provides users of the Cognitive Framework inline access to the large collection of automated knowledge-capture algorithms from the Machine Learning package by automatically populating Cognitive Modules from data without the tedious process of manual knowledge elicitation. By using the modules of the Framework-Learning package, a developer can automatically create models of behavior and cognitive processes gathered from disparate data sources.

3.8 Cognitive Model Factory

A Cognitive Model Factory is a container that holds the complete recipe for making a Cognitive Model: the full list of modules and all of their parameters. Having a factory allows multiple copies of the same model to be instantiated and provided different inputs. It also provides a mechanism for modules to share static information across Cognitive Model instances so that the data does not have to be copied, thus saving memory in large-scale

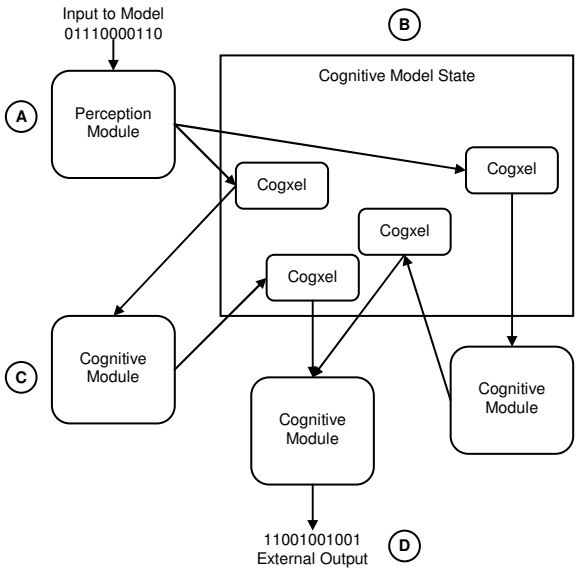

Figure 4. *Operation of a Cognitive Model. A) A perception module takes an external input, processes it, and produces outputs which are stored in cogxels in the Cognitive Model State. B) The Cognitive Model State contains current cogxel values that change as the model is run across multiple update steps. C) Cognitive modules take inputs in the form of cogxels from the shared Cognitive Model State, perform implementation specific processing, and then typically write outputs back to cogxels in the shared state. D) The model's state and the associated activity of the modules represent its embodiment of a model of cognition; some modules may produce external outputs that can be used for purposes such as model evaluation or for allowing a model to control external components, such as in an embedded system*

simulations. To borrow an example from physics, all oxygen-16 molecules are identical; they have the same static parameterization. To create a simulation of many interacting oxygen molecules, it is not necessary to copy the parameterization of an oxygen molecule for each model; they can share this static information. This saves memory and the needless computation used to copy the redundant parameterization. Cognitive Model Factories provide an analogous functionality, allowing Cognitive Modules to share static information across many instances.

3.9 Using the Output of a Cognitive Model

The final step in integrating a Cognitive Model into a CS&T application is to determine how to use the output of the model for the application. By this, we mean using information contained within cogxels to perform some sort of behavior or action for an agent within its environment. Typically, this is accomplished by determining the semantic labels for the cogxels containing the relevant output information for generating actions. However, an application can access and make use of any cogxel in the Cognitive Model State, which means it has a vast amount of information regarding the internal state of the Cognitive Model to make use of when generating an action. Figure 4 illustrates the overall operation of a Cognitive Model.

The Cognitive Foundry contains graphical user-interface tools that can display the outputs, and internal state, of a Cognitive Model. While a predefined user interface may not be applicable for a particular end-user application, developers often find it helpful for rapid-prototyping or debugging purposes. Following the design philosophy of the Foundry, the piece components of the user interface may serve as the basis for various end-user applications.

3.10 Example Implementation of a Cognitive Model

The design of the Cognitive Foundry supports cognitive simulation at many levels of fidelity from low-level connectionist networks to high-level, abstract symbolic approaches. We have used the Cognitive Foundry for a variety of research projects, including building a model of driving difficulty, agents in economic and social simulations, and evaluating trainee performance in a simulated environment. Here we present a brief overview of an application of the Cognitive Foundry for a current project underway to better elucidate how the Foundry helps support cognitive simulations. The project entails constructing a model of human analogical reasoning based on published psychological and physiological literature and research into human performance on a visual test of intelligence. For this model, the level of fidelity to be simulated is that of a single cortical column within the human brain. Currently we are employing Fuzzy Adaptive Resonance Theory (ART) models as an abstract representation of a cortical column. Individual cortical column models are assembled in a connectionist manner that plausibly models the human visual processing stream and prefrontal cortex. Following training, the model is presented with representative visual problems from the visual test of intelligence and for each the model's output is its answer to the problem, chosen from a multiple choice set.

The Foundry enables the straightforward construction, training, and evaluation of this model. A perception module models the human primary visual cortex's orientation columns. This module takes as an input a rasterized visual intelligence test problem and outputs real-valued cogxels representing the degree to which a given line orientation is detected in the input image in each of a set of retinotopic receptive areas. Cortical columns are implemented as cognitive modules. Each cognitive module contains a single Fuzzy ART as well as

a description of what cogxels the module should use as its input. The input cogxel description provides the mechanism for determining the wiring of the overall connectionist network of cortical columns. The overall Cognitive Model contains the perception module and each of the cortical column modules, and is implemented using the concurrent model implementation previously described.

Running the model is handled by the Foundry, and involves presenting an input image to the model during an update step. Refer to Figure 4 for an overview of the operation of the model. Internally, the Cognitive Framework presents the image as input to the perception module, causing it to produce output cogxels. These cogxels reside in the blackboard-like Cognitive Model State. As the model is run across multiple update steps, each module retrieves the appropriate input columns from the Cognitive Model State, and produces its own output cogxels. In this model, these outputs are real-valued vectors produced by the Fuzzy ART contained within a given module.

By employing the Foundry, the team was able to focus on the implementation details of the cortical columns and overall wiring of the model without having to separately devote effort to creating the framework for constructing, training, and evaluating the model. Furthermore, the concurrent implementation provided by the Foundry allows for a significant reduction in computation time for model training and evaluation. Finally, the robust set of machine learning algorithms included in the Foundry (described in section 4 below) provided the team with a strong foundation for implementing project specific functionality.

4 Machine Learning Package

The Cognitive Foundry's Machine Learning Package provides a wide variety of optimized, verified, and validated general- and special-purpose algorithms for machine learning: the analysis and characterization of large datasets, function minimization, parameter estimation, prediction, and categorization. The package is highly extensible, meant for allowing the rapid-prototyping of applications based on machine learning and the development of new or experimental algorithms and architectures. Typically, in machine learning, there are various conflated components: the object being created, the learning algorithm used to create the object, the data upon which the algorithm operates, the performance measure, and statistical validation. For example, we can create a neural network using gradient descent with a mean squared-error cost function. However, there are many neural-network architectures (feedforward, recurrent, different activation functions, etc.), many different learning algorithms (conjugate gradient, Levenberg-Marquardt, Quasi-Newton, etc.), many different cost functions, and many validation techniques. Unless these components are decoupled, the combinatorics quickly becomes onerous. In keeping with the design philosophy of the Cognitive Foundry, the Machine Learning package separates each of these components and eliminates the need for Foundry users to write special-purpose code. This allows users of new functions (*e.g.*, neural-network architectures) to use existing learning algorithms and, conversely, creators of new learning algorithms to test their ideas on different functions.

4.1 Why a Machine Learning Package?

We created the Machine Learning package to support using automated knowledge-capture techniques to populate cognitive models. Many such techniques are based on machine-learning algorithms. Furthermore, our research group had several implementations of similar machine-learning algorithms written in different programming languages, with slight variations on similar approaches. The Machine Learning package is a common repository of these algorithms, so that they may be easily integrated into different applications and projects.

Due to the decoupling of the learning algorithm from the object being learned, the Machine Learning package allows rapid prototyping and experimental testing of different algorithms, approaches, and function approximators and categorizers. The package accomplishes this through the systematic use of interfaces and generics to encapsulate the needs of each algorithm, including their inputs, outputs, and parameterizations. We followed an object-oriented design for the entire package so that the different algorithms utilize common, interchangeable subcomponents, such as cost functions and statistical validation. This approach greatly simplifies the integration of exiting machine-learning algorithms to new problems and, conversely, to apply new machine-learning algorithms to existing problems and datasets. We did this to focus on experimenting with different algorithms and parameterizations to create machine-learning systems embedded into CS&T applications. The design of the Machine Learning package makes wide use of Java generics and, together with the decoupling of the piece components of a machine-learning system, the source code tends to be very similar to pseudo-code from textbooks and research papers. This has the result of greatly increasing the reusability of elements in the package and increasing the level of verification and validation of the algorithms.

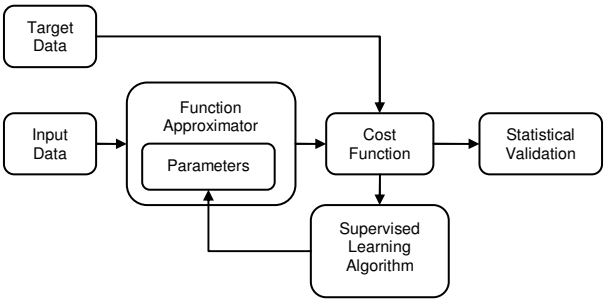

Figure 5. *Typical supervised learner design pattern.*

4.2 Learned Functions

One of the key concepts of the Machine Learning package is the separation of the machine-learning algorithm from the function created by the algorithm, which is typically some form of function approximator or categorizer. For example, consider a prototypical function approximator, the artificial neural network. Many machine-learning algorithms can estimate locally optimal parameters for neural networks, such as gradient descent, genetic algorithms, and inverted quadratic line search. However, each of those machine-learning algorithms can estimate the parameters of a much broader class of function approximators. Separating the function approximator from the machine-learning algorithm means that we only have to write a learning algorithm once, instead of the combinatorial explosion that occurs in the cross product of function approximators and learning algorithms. The package contains many standard and specialty function approximators such as artificial neural networks, linear systems, dynamical systems, k-nearest neighbors, polynomials, categorizers, decision trees, support vector machines, and mixture of Gaussians. Furthermore, the Cognitive Foundry allows developers to create their own functional forms, or to chain together existing functions, to create new functional architectures, making the Machine Learning Package highly extensible. A fundamental ability for function approximators and categorizers is the ability to express itself as a vector of tunable parameters. In this form, many learning algorithms can automatically tune the parameters of function approximators to achieve some desired result.

4.3 Learning Algorithms

The main paradigm of the learning package is that a learning algorithm is given a dataset from which to create a new learned object, typically a function approximator or categorizer. This creates a clear separation between the input to the algorithm, the output from the algorithm, and the parameters of the algorithm. There exist two primary learning interfaces: the Batch Learner interface and the Online Learner interface. Batch Learners operate only after all data have been collected, while Online Learners can operate concurrently as data are being collected. Each of these learners operates in a supervised or unsupervised manner. Supervised-learning algorithms learn to generalize from example input-output pairings, while unsupervised-learning algorithms attempt to discover patterns in an unlabeled dataset to achieve some goal (Duda, Hart, & Stork, 2001). We will discuss each of these techniques in the following sections.

4.3.1 Supervised-Learning Algorithms

Supervised-learning algorithms take a dataset of input-output pairs and generalize them to as-yet-unseen inputs by finding parameter sets that minimize a cost function. Commonly these algorithms yield "function approximators" or "regression" when the outputs are real valued and "categorizers" when the outputs are discrete valued. We provide a wide variety of supervised-learning algorithms such as derivative-free algorithms, gradient-based methods, linear regression, algebraic solvers, kernel methods for both regression and categorization, and decision trees for both regression and categorization. We also have implemented meta-learning algorithms such as ensemble methods, which demonstrate the power of having a unified set of interfaces implemented by a variety of learning algorithms. A typical design pattern for a supervised learning algorithm is shown in Figure 5.

4.3.1.1 Derivative-Free Algorithms

There is a class of parameter-estimation algorithms that do not require gradient information to find a (locally) minimum-cost parameter set of a function. That is, to find the optimal solution, these algorithms only require function evaluations. While these algorithms are the most general, and can find optimal parameters for functions that are highly nonlinear with nonanalytic or inexistent derivatives, they tend to be less efficient than gradient-based algorithms when gradient information can be computed or approximated.

We have implemented standard derivative-free algorithms including Direction Set (Powell's) Method, Downhill Simplex (Nelder-Mead) algorithm, Genetic Algorithms, and Simulated Annealing.

4.3.1.2 Gradient-Based Algorithms

A more restrictive class of parameter-estimation algorithms are those that require gradient information to find a (locally) minimum-cost parameter set of a function. These algorithms usually perform better than non-gradient-based methods.

We have implemented standard gradient-based algorithms including Quasi-Newton Minimization (BFGS), Levenberg-Marquardt Estimation, Conjugate-Gradient Minimization, and Gradient Descent. Many of these

algorithms are also guaranteed to converge using approximated differentiation procedures by estimating the parameter Jacobian from function evaluations alone. Oftentimes, gradient-based algorithms with approximated derivatives are more efficient than derivative-free algorithms. We have implemented automated differentiation procedures for arbitrary functions to give users the ability to try different gradient-based and derivative-free algorithms.

4.3.1.3 Special-Purpose Solvers

Some special-case functions have closed-form or iterative optimal solutions. These functions may be used with the general-purpose algorithms mentioned above or with solvers that exploit particular features of these special functions. For example, we have implemented solvers for linear regression, linear systems, linear dynamical systems, multivariate Gaussians, and so forth. Not surprisingly, these special-purpose solvers are typically more efficient than general-purpose solvers for the same functional form.

4.3.1.4 Kernel Methods

We have extensive support for kernel-based methods, including a set of useful kernel functions and tools for composing kernel functions. Kernels allow certain machine-learning algorithms to extend to nonvector data by defining a similarity measure between two inputs that fulfill the properties of a kernel function. As such, the library utilizes generics for kernels, which promotes the creation of kernels for new data types, which may in turn be plugged into existing kernel-based algorithms. We have implemented several kernel-based learning algorithms for categorization, regression, and clustering.

4.3.1.5 Ensemble Methods

Ensemble methods typically take a simple function and combine several together to create sophisticated responses to novel inputs, similar to voting schemes, game theory, or stock-market collaboration. Ensemble methods are a natural fit for the Cognitive Foundry's Machine Learning package because each machine-learning algorithm conforms to a standard interface that allows algorithms that create the same function to be interchanged, meaning that algorithms conforming to the same interface can be automatically combined using an ensemble-learning algorithm that utilizes the output of each learning algorithm. We have implemented several ensemble methods including Bagging and AdaBoost.

4.3.2 Unsupervised Learning Algorithms

Unsupervised-learning algorithms take a high-dimensional space, potentially a "Big Data" problem, and map it to a low-dimensional and (hopefully) simpler space. These algorithms are useful for understanding complicated relationships, identifying statistical regularities, and visualization. Our set of unsupervised algorithms emphasizes principal components analysis and clustering algorithms, such as k-means clustering, agglomerative clustering, reductionist clustering, and affinity propagation. The clustering algorithms are based on a user-defined distance metrics that allow all unsupervised algorithms to be easily adapted to new types of data. We also exploit singular value decomposition and eigenvector decomposition for dimensionality reduction. For automated knowledge capture, unsupervised learning algorithms, such as the clustering algorithms listed above, are used to discover contexts, latent patterns, and relationships for a Cognitive Model automatically.

4.4 Experiments and Performance Evaluation

We have created objects that automatically evaluate the performance of learning algorithms against a dataset using statistical-hypothesis testing techniques. These "Experiment" objects automatically provide performance confidence bounds using generally accepted validation techniques, such as n-fold, leave-one-out (jackknife), and bootstrap validation. For each experiment, a user specifies the dataset, learning algorithms, validation methodology, performance criteria, and summary statistics. The output of an experiment is a confidence interval describing the performance range and statistical confidence in the experiment, making it easy to compare different learning algorithms, algorithm parameters, or functional forms in a statistically significant manner.

4.5 Statistics Package

The Cognitive Foundry also includes a comprehensive Statistics package for performance analysis and statistical-hypothesis testing, in addition to providing many probability distributions for modeling purposes.

4.5.1 Null-Hypothesis Testing

The goal of null-hypothesis testing is to determine if two distributions of data are different in a statistically significant sense. In other words, can the observed results be due to chance? This gives a user of the Cognitive Foundry a quantifiable confidence on the performance of the system. Different statistical tests have different assumptions, and it is necessary to find the test appropriate for the problem at hand.

This package contains the standard statistical tests such as Student-t Test, Analysis of Variance (ANOVA or F-Test), z Test, Kolmogorov-Smirnov Test, Fisher Sign Test, Wilcoxon Signed Rank Test, Mann-Whitney U Test (Wilcoxon Rank-Sum Test), Receiver-Operator Characteristic, and others. Given the well-designed structure of the Cognitive Foundry, it is easy to use the

appropriate null-hypothesis test for the given problems facing a project.

4.5.2 Confidence Bounds

From estimates based on different datasets, it is often useful to determine what possible values a parameter could take. For instance, what is the likely room temperature from a set of noisy thermometer readings? Likewise, how many experiment subjects do we need to achieve a margin of error of at most ±3%? To answer these types of questions, we have implemented the standard confidence-bounds routines such as Student-t, Gaussian, Chebyshev, Markov, and Bernoulli.

5 Common Math Package

The main purpose of the Common Math package is to provide a common foundation of useful mathematical routines for building applications. Much of our research and many applications require mathematical rigor, and the main component of the Common package is to facilitate matrix and vector computation, decompositions, and solver routines, in both dense and sparse representations. Dense-matrix computation tends to be faster than its sparse-matrix counterpart, however, many "Big Data" applications simply cannot use a dense-matrix representation. The Cognitive Foundry gives users the flexibility to choose the representation that best suits their needs. The basic definitions of a matrix and vector are interfaces, which gives Foundry users the freedom to write their own Matrix package. The default Matrix package in the Cognitive Foundry is based on the publicly available Matrix Toolkits for Java (MTJ) library. MTJ performs its computations, decompositions, and solvers using callbacks into the best-in-class native libraries LAPACK and BLAS, resulting in computational speeds competitive with other heavily optimized computational packages. If these native libraries are not available, MTJ will redirect the calls to platform-independent Java versions of LAPACK and BLAS, which are slower than the native versions. This flexibility allows Cognitive Foundry applications to use the most efficient computational engine available, while preserving cross-platform compatibility.

6 Related Work

Agent-based (Wooldrige & Jennings, 1995) and cognitive simulation is an active area of research, and there are many cognitive architectures in existence, such as ACT-R (Anderson & Lebiere, 1993) and SOAR (Laird, Newell, & Rosenbloom, 1987). The Foundry's Cognitive Framework builds upon previous research in cognitive frameworks at Sandia by Forsythe and Xavier (2002). However, unlike cognitive architectures that are built around a single theory of cognition, the Foundry promotes modularity and experimenting with various aspects of different theories of cognition. This provides a value added over existing architectures in that the relevant cognitive components can be utilized to fill project specific needs. This includes the ability to choose the appropriate level of modeling fidelity for a given application. The Foundry is also different in its focus on integrated automated knowledge capture and the ability to create lightweight cognitive components that are easy to embed in agents and stand-alone CS&T applications.

An example of a project that incorporated different cognitive architectures is the Agent-Based Modeling and Behavior Representation (AMBR) Model Comparison project (Gluck & Pew, 2001), which involved comparing the performance of different cognitive architectures in modeling the behavior of an air traffic controller. The Foundry is designed to support combining and comparing existing models of cognition developed within its environment, similar to the comparisons performed using the HLA environment in AMBR. It also provides its own robust library for the development of new models, and for the combination of new and existing models.

There are also other libraries of machine-learning algorithms in existence, such as Weka (Witten & Frank, 2005) and RapidMiner, formerly YALE (Mierswa, Wurst, Klinkenberg, Scholz, & Euler, 2006). However, the Foundry's learning package differs from existing packages in several ways. First, the Cognitive Foundry does not force users to create datasets in a fixed data structure for the machine-learning algorithms, such as vector data. Instead, the Foundry algorithms are implemented so that they can be used with a variety of data structures by its use of generic type parameters. The Foundry's Machine Learning package is also different because it spans the entire development cycle of a learning system from data collection, analysis, experimentation, rigorous tools for performance validation, and deployment into applications, including embedded systems. The package is targeted at making it easy to embed the function created through learning into other applications, such as agent models, which distinguishes it from other packages that are focused primarily on data mining and visualization.

7 Conclusions and Future Work

We have presented the Cognitive Foundry and its two primary components that relate to agent behavior modeling: the Cognitive Framework and Machine Learning packages. For future work, we would like to create a graphical user interface to increase the ability of users with little computer-programming experience to create cognitive models and machine-learning systems.

We also plan on adding new cognitive modules, learning algorithms, and other packages, such as text and image analysis, to the Foundry.

8 References

Anderson, J., & Lebiere, C. (1993). *The Atomic Components of Thought*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Duda, R. O., Hart, P. E., & Stork, D. G. (2001). *Pattern Classification*. New York, NY: Wiley-Interscience.

Forsythe, C., & Xavier, P. G. (2002). Human emulation: Progress toward realistic synthetic human agents. In *Proceedings of the 11th Conference on Computer-Generated Forces and Behavior Representation*.

Gluck, K. A. and Pew, R. W. (2001) Overview of the Agent-based Modeling and Behavior Representation (AMBR) Model Comparison Project. In *Proceedings of the 10th Conference of Computer Generated Forces and Behavior Representation*.

Laird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An Architecture for General Intelligence. *Artificial Intelligence, 33 (1)*, 1-64.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M. & Euler, T. (2006). YALE: Rapid Prototyping for Complex Data Mining Tasks. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-06)*.

Witten, I. H. & Frank, E. (2005) *Data Mining: Practical machine learning tools and techniques*, 2nd Edition. San Francisco: Morgan Kaufmann.

Wooldridge, M., & Jennings, N. R. (1995). Intelligent Agents: Theory and Practice. In *Knowledge Engineering Review*.

Acknowledgements

This work was partially funded by the Office of Naval Research, Code 30. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

Author Biographies

JUSTIN BASILICO is a researcher at Sandia National Laboratories in Albuquerque, NM. He received his B.A. in Computer Science from Pomona College and his M.S. in Computer Science from Brown University. His research interests include machine learning, cognition, information retrieval, statistical text analysis, and human-computer interaction.

ZACHARY BENZ is a researcher at Sandia National Laboratories in Albuquerque, NM. He received his B.S. in Engineering at Harvey Mudd College and his M.S. in Computer Science at the University of New Mexico. His research interests are in cognition, cognitive modeling and statistical text analysis.

KEVIN R. DIXON is a researcher at Sandia National Laboratories in Albuquerque, NM. He received his B.S. and Ph.D. from Carnegie Mellon University in Electrical & Computer Engineering. His research interests are in statistical pattern recognition, dynamical systems, human-machine interaction and adaptive control.