SANDIA REPORT
SAND2013-8910
Unlimited Release
Printed December 2013

Incremental Learning for Automated Knowledge Capture

Zachary O. Benz
Justin D. Basilico
Warren L. Davis
Kevin R. Dixon
Brian S. Jones
Nathaniel Martin
Jeremy D. Wendt

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-8910
Unlimited Release
Printed December 2013

Incremental Learning for Automated Knowledge Capture

Zachary O. Benz
Justin D. Basilico
Warren L. Davis
Kevin R. Dixon
Brian S. Jones
Nathaniel Martin
Jeremy D. Wendt

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Abstract

People responding to high-consequence national-security situations need tools to help them make the right decision quickly. The dynamic, time-critical, and ever-changing nature of these situations, especially those involving an adversary, require models of decision support that can dynamically react as a situation unfolds and changes. Automated knowledge capture is a key part of creating individualized models of decision making in many situations because it has been demonstrated as a very robust way to populate computational models of cognition. However, existing automated knowledge capture techniques only populate a knowledge model with data prior to its use, after which the knowledge model is static and unchanging. In contrast, humans, including our national-security adversaries, continually learn, adapt, and create new knowledge as they make decisions and witness their effect. This artificial dichotomy between creation and use exists because the majority of automated knowledge capture techniques are based on traditional batch machine-learning and statistical algorithms. These algorithms are primarily designed to optimize the accuracy of their predictions and only secondarily, if at all, concerned with issues such as speed, memory use, or ability to be incrementally updated. Thus, when new data arrives, batch algorithms used for automated knowledge capture currently require significant recomputation, frequently from scratch, which makes them ill suited for use in dynamic, time-critical, high-consequence decision making environments. In this work we seek to explore and expand upon the capabilities of dynamic, incremental models that can adapt to an ever-changing feature space.

ACKNOWLEDGMENTS

This work was funded under LDRD Project Number 151346 and Title "Incremental Learning for Automated Knowledge Capture".

CONTENTS

1. Introduction 9

2. Algorithms 11
   2.1. Overview 11
   2.2. Supervised Incremental Learning Algorithms 11
      2.2.1 Perceptron-Like Algorithms 11
      2.2.2 Confidence Weighted Algorithms 11
      2.2.3 Bounded Kernel Algorithms 11
      2.2.4 Bayesian Algorithms 11
   2.3. Streaming Random Forests 12
      2.3.1 OnlineCategorizationTree 12
      2.3.2 OnlineCategorizationTreeLearner 13
      2.3.3 OnlineCategorizationForestLearner 13
   2.4. Unsupervised Incremental Algorithms 13
      2.4.1 Lloyd's Algorithm 13
      2.4.2 K-Means++ 14
      2.4.3 Mini-batch K-Means 14
      2.4.4 Incremental K-Means Implementation 14

3. Experiments 17
   3.1. Overview 17
   3.2. Performance of Incremental Algorithms 17
      3.2.1 Windowing and Evaluation Techniques 17
      3.2.2 Ford Challenge 18
      3.2.3 Grant Prediction 18
      3.2.4 SNL Binary Classification Problem 19

4. Infrastructure 21
   4.1. Overview 21
   4.2. Data Streaming and Feature Extraction 21
   4.3. Experiment Harnesses 22
   4.4. Hyperparameter Optimization 23
      4.4.1 Inputs 23
      4.4.2 Running the tool 24
   4.5. Optimized Matrix Math 24
      4.5.1 Matrices – Dense, Sparse, and Diagonal 24
      4.5.2 Dense Implementation 25
      4.5.3 Diagonal Implementation 26
      4.5.4 Sparse Implementation 26
      4.5.5 Results 27
      4.5.6 Further Improvements 28

5. References 29

Appendix B: Installing Optimized BLAS 31

Distribution 32

FIGURES

Figure 1. Performance of algorithms on Ford Challenge data 18
Figure 2. Performance of algorithms on Grant Prediction data 19
Figure 3. Performance of algorithms on SNL Binary Classification Problem 20
Figure 4. Performance of Optimized Matrix Operations 28

Nomenclature

AUC	Area Under the Receiver Operator Characteristic Curve
DOE	Department of Energy
JDK	Java Development Kit
LDRD	Laboratory Directed Research and Development
ML	Machine Learning
SNL	Sandia National Laboratories

1. INTRODUCTION

In the first year of this project, we accomplished the primary goal of establishing a benchmark for existing incremental learning algorithms. To do this, we developed novel ways of measuring and comparing batch and incremental algorithms by developing experimental procedures that mimic typical contexts in which incremental algorithms are used for automated knowledge capture. We implemented a software harness for comparing algorithms using a wide variety of datasets and evaluation metrics. As part of this harness, we created a capability to handle a wide variety of datasets in different data formats. Within this harness, we implemented a large suite of classic and contemporary incremental learning algorithms for categorization and have benchmarked them on several real-world datasets.

In the second year of this project, we have investigated novel ways of combining incremental and batch algorithms, such as updating a knowledge model incrementally after batch learning has occurred on it with an iterative approach. Work has focused on leveraging streaming random forest algorithms to prepare experiments that seek to find balance between batch and incremental approaches. As part of this effort, internal Sandia optimization code is being leveraged to perform large-scale parameter optimization in order to characterize the behavior of the algorithms across a broad array of tasks. As before, we have leveraged data from the machine learning community, to include a variety of learning tasks hosted on Kaggle.

In the final year of the project, we focused specifically on streaming random forests as a learning construct in which to perform experiments. Additionally, we explored the space of unsupervised machine learning algorithms, focusing on variants of the K-Means algorithm.

All work is developed and deployed in Sandia's open source Cognitive Foundry software package, providing direct utility of the research to current and future users. A streaming data pipeline has been created within the Foundry, and is quickly reaching maturity as a platform for working with incremental data from a variety of sources.

This project supports the national security missions of DOE, DOD and DHS by providing science and technology in support of the human element that is critical to many mission areas. It addresses a fundamental barrier to using computational models of cognition by updating models with relevant knowledge in a timely manner to support a human's ability to understand and act in urgent and uncertain situations with nationally significant consequences. This will also increase their applicability to environments such as physical and cyber security that involve adversaries who constantly adapt tactics and strategies by also enabling the models to adapt.

2. ALGORITHMS

2.1. Overview

In investigating the space of incremental learning algorithms, we began with algorithms that are popular among researchers and practitioners. Initial benchmarking of these algorithms was performed as described in the Experiments section below. We then proceeded to investigate streaming random forest based algorithms and k-means based unsupervised incremental algorithms. Each of these is described in turn in this section. All code was implemented in the open source Cognitive Foundry library, discussed in the Infrastructure section below.

2.2. Supervised Incremental Learning Algorithms

For the purposes of benchmarking, we implemented a number of incremental learning algorithms from the literature, which we list here.

2.2.1 Perceptron-Like Algorithms

- Online Perceptron
- Margin Infused Relaxed Algorithm (MIRA) [Crammer, 2003]
- Relaxed Online Maximum Margin Algorithm (ROMMA) [Crammer, 2003]
- Aggressive Relaxed Online Maximum Margin Algorithm (AROMMA) [Crammer, 2003]
- Approximate Large Margin Algorithm (ALMA) [Gentile, 2002]
- Passive Aggressive Perceptron (multiple variants) [Crammer, 2006] [Wang, 2010]
- Shifting Perceptron [Cavallanti, 2007]
- Ballseptron [Shalev-Shwartz, 2005]

2.2.2 Confidence Weighted Algorithms

- Variance Drop [Dredze, 2008]
- Variance Project [Dredze, 2008]
- Standard Deviation Drop [Dredze, 2008]
- Standard Deviation Project [Dredze, 2008]
- Adapative Regularization of Weights (AROW) [Crammer, 2009]

2.2.3 Bounded Kernel Algorithms

- Stoptron [Orabona, 2009]
- Remove Oldest Kernel Perceptron (ROKP)
- Randomized Budget Perceptron [Cesa-Bianchi, 2006]
- Forgetron (multiple variants) [Dekel, 2008]
- Projectron (multiple variants) [Orabona, 2009]

2.2.4 Bayesian Algorithms

- Online Naïve Bayes

2.3. Streaming Random Forests

Streaming random forests are a relatively new concept in the area of machine learning. Being in its infancy, there are many new areas of study, though a few salient paradigms are emerging. For the purposes of our research, we wanted to create an architecture that would support current developments in streaming random forests, while giving us the flexibility to explore and implement new algorithms in the area. The components of our design are listed below:

2.3.1 OnlineCategorizationTree

The OnlineCategorizationTree is a decision tree classifier structure that can grow incrementally as new data arrives. The main difference between the nodes in this tree and a normal categorization tree is that, in our case, the frontier nodes store category counts and weak learners (NodeSplitters) which can be used by the overall streaming random forest learner (OnlineCategorizationTreeLearner) in the tree growing process. This allows for intra-node decision mechanisms that are more intricate than using something like the Gini coefficient (CART) or information gain (ID3, C4.5). This style of incremental tree has been described in papers such as [Domingos, 2000], [Saffari, 2009], and [Abdulsalam, 2011].

An alternate design could use standard CategorizationTreeNodes throughout the tree, and store frontier node state in the learner object, for example in a Map<Node, FrontierState>.

Additional sub-components of the OnlineCategorizationTree are as follows:

- OnlineCategorizationNode: an extension of CategorizationTreeNode to hold an online split (OnlineNodeSplitter), as well as count data.
- OnlineNodeSplitter: an interface for an object that decides when a node is ready to be split, and returns a BranchDecider that defines the split.
- OnlineNodeSplitterFactory: an object that creates OnlineNodeSplitters when the tree learner needs a new one (for newly-created frontier nodes).

The flexibility in the current design comes from OnlineNodeSplitter and OnlineNodeSplitterFactory, which let you plug in new strategies for determining node splits. The design breaks this down into two sub-packages:

- *staticfunc*: static functions – node splits come from a pool of pre-determined split functions like axis-aligned splits (standard CART) or random projections
    - OnlineHistogramSplits: uses an online histogram to keep track of each class distribution for determining split threshold.
    - ExtremelyRandomSplits: In the spirit of Extremely Randomized Trees [Geurts, 2006], selects the split point randomly within the feature range [min,max] that has been observed so far. It therefore needs a “warmup” stage to establish initial feature ranges.
- *dynamicfunc*: dynamic functions – the split function is updated dynamically over time, allowing the insertion of algorithms such as the Passive Aggressive learner [Crammer, 2006]. Linear learners in this role are sometimes called “oblique” decision trees.

- BinaryLearnerSplitterFactory: produces OnlineNodeSplitters that use an underlying SupervisedIncrementalLearner on an (optional) subspace of features.

2.3.2 OnlineCategorizationTreeLearner

This class is responsible for growing an OnlineCategorizationTree instance. It does this by accumulating statistics at frontier nodes and using a SplitDecision object to determine when to make a new split. When a frontier node is split, it is replaced in the tree with a normal CategorizationTreeNode, and new child frontier nodes are added to it as children. These new frontier nodes are given a set of competing NodeSplitter (weak learner) options by the NodeSplitterFactory. Nodes will not be split when they reach maxDepth.

2.3.3 OnlineCategorizationForestLearner

This is the top-level forest class that holds the individual tree learners. During learning, this module passes samples down to the relevant frontier in the tree.

2.4. Unsupervised Incremental Algorithms

This section describes our implementation of an incremental k-means in which clustering is done incrementally. Our implementation begins with the Online Lloyd's Algorithm and also adds optional modifications including K-Means++ [Arthur 2007] and Mini-batch K-Means [Sculley 2010].

2.4.1 Lloyd's Algorithm

Lloyd's Algorithm is an iterative refinement technique used for grouping data points into clusters. The first step in the algorithm is to obtain an initial set of k means. There are numerous methods that can be used for the initialization step such as the Forgy methods or the Random Partition method [Andberg, 1973]. The algorithm then repeats two steps until it converges. The first of these steps is an assignment step in which each data point is assigned to a cluster whose mean results in the least within-cluster sum of squares (WCSS):

$$\arg\min \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in S_i} \|\boldsymbol{x} - \boldsymbol{u}_i\|^2$$

where S_i is the set of data points assigned to cluster i, $\boldsymbol{x}$ is the data point represented as a vector, and $\boldsymbol{u}_i$ is the mean of the points in S_i.

The next step is to update the means for each cluster by assigning it to be the centroid of all of the data points assigned to the cluster using the following formula:

$$m_i^{(t+1)} = \frac{1}{|S_i^t|} \sum_{\boldsymbol{x} \in S_i^t} \boldsymbol{x}$$

where i is the index of the cluster, t is the current iteration, , and $\boldsymbol{x}$ is the data point represented as a vector. The algorithm reaches convergence when no data points change clusters during the assignment step.

The online version of Lloyd's algorithm behaves in a similar way, but data points are received one point at a time rather that all at once. Each time a cluster is updated, it is updated from only one data point. The first step is again to create an initial set of k means. Then, for each new data point, two steps are followed which are very similar to above. In the assignment step, the new data point is assigned to the nearest cluster, which minimizes the WCSS. The next step then updates only the cluster that the data point was assigned to. The resulting cluster mean is updated according to

$$\boldsymbol{c} = \boldsymbol{c} + \left(\frac{1}{v_c}\right)(\boldsymbol{x} - \boldsymbol{c})$$

where $\boldsymbol{c}$ is the cluster center, v_c is the count of points the cluster has been updated by, and $\boldsymbol{x}$ is the sample vector.

2.4.2 K-Means++

K-Means++ is an augmented version of k-means in which the first step of seeding the algorithm with the initial clusters is modified. This algorithm has been shown to be O(log k)—competitive with the optimal clustering. As previously mentioned, K-Means++ follows all of the steps of the standard k-means algorithm except for the first step. In K-Means++, the first cluster to use is chosen uniformly at random from the full set of data points. The next step is to choose another cluster from the set of data points with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$ where D(x) is the shortest distance to any of the cluster centers already chosen. This step is then repeated until k data points have been selected to be the initial clusters. From this point on, K-Means++ follows the exact same steps as k-means.

2.4.3 Mini-batch K-Means

Mini-batch k-means is a hybrid of the standard batch k-means algorithm and the stochastic gradient descent (SGD) variant of k-means developed by [Bottou, 1995]. The mini-batch algorithm, as described by [Sculley, 2010], attempts to improve upon the O(kns) computation time of the standard batch k-means where n is the number of data points and s is the maximum number of non-zero elements in a data point's vector while avoiding the early convergence of SGD due to stochastic noise. The algorithm begins by initializing the set of k means as done in Lloyd's algorithm. The following steps are then repeated for the specified number of iterations. First, b samples are selected from the full set of data points where b is the size of the mini-batch. For each of the selected samples, the nearest center is calculated and cached. Next, each of the samples update the cached closest center using the following update:

$$\boldsymbol{c} = \left(1 - \left(\frac{1}{v_c}\right)\right)\boldsymbol{c} + \left(\frac{1}{v_c}\right)\boldsymbol{x}$$

where $\boldsymbol{c}$ is the cluster center, v_c is the count of points the cluster has been updated by, and $\boldsymbol{x}$ is the sample vector.

2.4.4 Incremental K-Means Implementation

Our implementation of the incremental k-means algorithm follows the standard online Lloyd's algorithm where data points are fed one at a time and the nearest cluster to that point is updated. As previously mentioned, there are numerous methods that can be used to create the initial clusters. Our implementation extracts an interface allowing for easy swapping out of the way the

initial clusters are created. Currently, there is a basic implementation of the interface in which the first k points are used as the initial cluster centers. There is also a slightly modified version of the k-means++ initialization method. Our k-means++ implementation differs from the original implementation due to the fact that we are learning incrementally and do not have the full data set up front. To handle this, we buffer a specified number of data points before initializing the clusters. Once enough data points have been buffered, the k-means++ algorithm is run as described by [Arthur and 2007] except that instead of choosing points from the full data set, points are chosen from the buffered set of points.

We also implemented a slightly modified version of the mini-batch k-means algorithm similar to the one described by [Sculley 2010]. Again, the modification that was made is due to the fact that the algorithm was designed for when all of the data points are obtained upfront, while our k-means implementation feeds data in incrementally. To account for not having all of the data up front, a buffer needs to be created to store the received data points until the batch size has been reached. Once the batch size is reached, the buffer can be fed into the mini-batch method to update the clusters as previously described. The main difference in our implementation being that instead of randomly selecting points to be included in the batch, each batch is the next b obtained points where b is the batch size.

3. EXPERIMENTS

3.1. Overview

Experiments were performed to baseline performance of incremental algorithms, and to vet the capabilities of algorithms coming out of research and development. All work was performed within the open source Cognitive Foundry library, described in the Infrastructure section below.

3.2. Performance of Incremental Algorithms

In an effort to understand how incremental algorithms perform empirically and to provide a guidepost for algorithm research and development, the team performed a number of experiments with data sets from Kaggle.com. Experiments with two such data sets are presented here. In addition, data from another SNL project's binary classification problem was used for experiments. In the Infrastructure section below, we discuss the additions made to the Cognitive Foundry to support data streaming, feature extraction, and experiment harnesses.

3.2.1 Windowing and Evaluation Techniques

To facilitate comparison of incremental algorithms to their batch counterparts, and to provide a mechanism for evaluating performance continuously (simulating a streaming set of data) multiple approaches were devised for running training – testing cycles.

3.2.1.1 Batch Cross Block Performance Battery

As opposed to a traditional cross fold based analysis, cross block does not randomize the order of the data before dividing it into equal sized blocks. The intent is to maintain the temporal nature of a data stream, to include concept drift over time, allowing incremental algorithms to leverage their ability to adapt over time to changing inputs. In addition, for each block, a portion of the data is withheld as test data. Thus, the algorithm is trained on a given block's training data, then tested on that block's test data, and then the process is repeated with the next block.

3.2.1.2 Growing Batch Performance Battery

The growing batch methodology is similar to the batch cross block methodology, but each subsequent block of training data is larger in size than the previous one.

3.2.1.3 Interleaved Performance Battery

In the interleaved methodology, we do not attempt to mimic batch learning through repeated bulk train and test iterations, but instead evaluate the algorithm's performance after the algorithm updates on each data point.

3.2.1.4 Evaluation Techniques

Our primary measure of performance was area under the receiver-operator characteristic curve (AUC). In addition, overall accuracy (including a full confusion matrix), memory usage, and wall clock run time were measured. Student's T with a 0.95 confidence level was used to derive confidence intervals for results.

3.2.2 Ford Challenge

In the "Stay Alert! The Ford Challenge" on Kaggle, the stated task was to "design a classifier that will detect whether the driver is alert or not alert, employing data that are acquired while driving". The data consists of 30 real-valued features per sample, plus binary alert labels for the training data. Approximately 120,000 data samples were made available. Results for the top performing algorithms from the suite implemented during this project are shown in the figure below, with variants on the Passive-Aggressive algorithm showing the best overall performance. The results shown are for a Batch Cross Block Performance Battery.

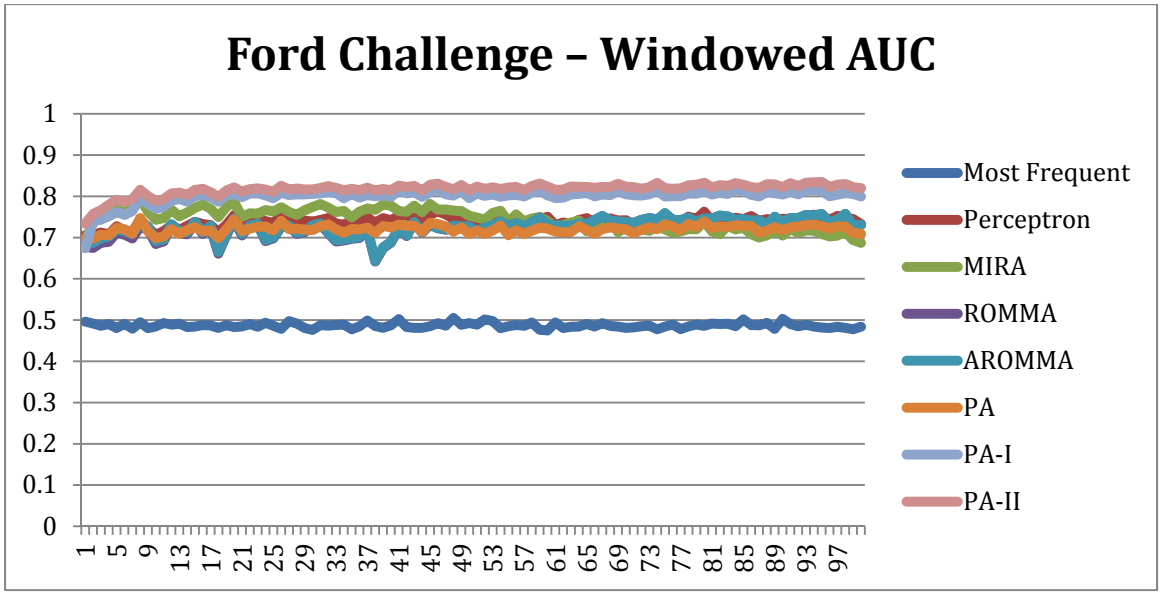

Figure 1. Performance of algorithms on Ford Challenge data

3.2.3 Grant Prediction

In the "Predict Grant Applications" challenge on Kaggle, the stated task was to "predict the outcome of grant applications for the University of Melbourne". The data consists of approximately 8,700 binary labeled samples (1 = grant won, 0 = grant not won) with approximately 250 features per sample, although there are many missing values. Features include relative value of the grant, information about the people on the grant application, dates associated with the grant, and information about the past performance of the people on the grant

in terms of receiving grants and publishing papers. Results for the top performing algorithms from the suite implemented during this project are shown in the figure below, with variants on the Passive-Aggressive algorithm again showing good overall performance, with good performance also seen from the Perceptron-like algorithm class of MIRA, ROMMA, and AROMMA. The results shown are for a Batch Cross Block Performance Battery.

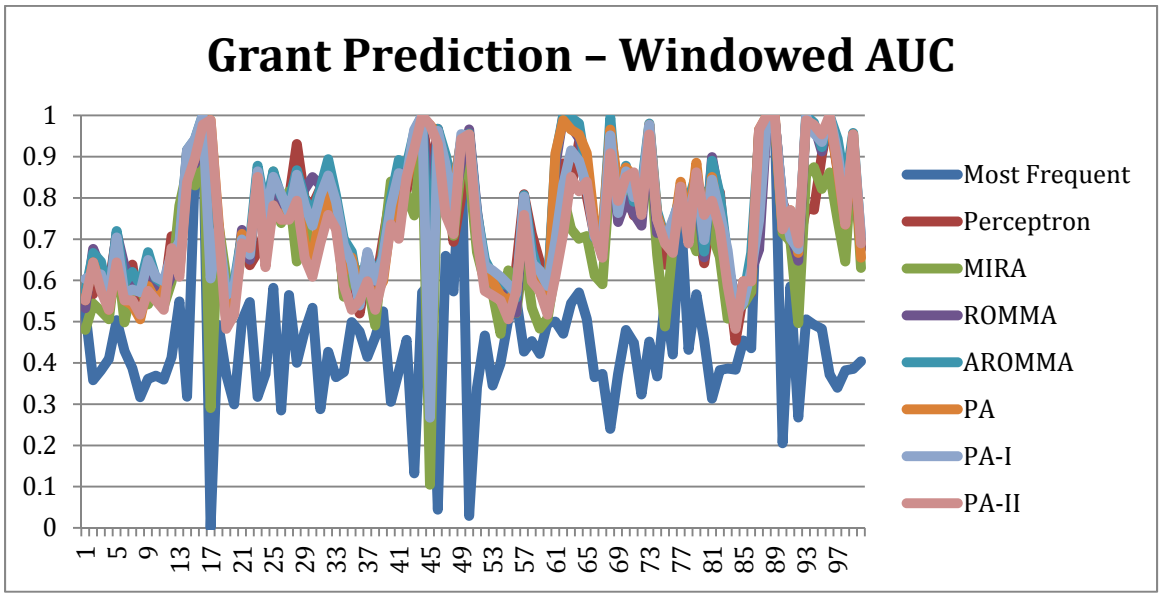

Figure 2. Performance of algorithms on Grant Prediction data

3.2.4 SNL Binary Classification Problem

An internal SNL project provided us with data for a binary classification problem. In this data set, approximately 5% of the labels were Class A, with the remainder being Class B. In such an unbalanced data set, a significant challenge is in minimizing the number of errors made in labeling Class A instances, typically at the tradeoff of mislabeling instances of Class B. Within class accuracy (rather than overall accuracy) was used as a performance metric in order to avoid the pitfall of perceived high accuracy for an unbalanced data set. As can be seen in the figure below, using Most Frequent as a baseline gets a 100% accuracy rate for Class B (purple X), as would be expected, but in this case Class A accuracy is the worst it can be, at 0% (green triangle). To give a sense for overall performance, we also measure the overall class average accuracy (blue diamond). Thus, our goal is to find an algorithm that maximizes our ability to correctly label Class A instances (get the green triangle as high as possible) while minimizing the tradeoff of reduced accuracy of Class B labeling (minimize how far the purple X falls). As can be seen from the results, Perceptron like algorithms (MIRA, ROMMA, AROMMA) provide the best overall performance. The relatively poor performance of the confidence weighted class of algorithms is somewhat surprising, but may be attributed to the relatively small size of the data set used.

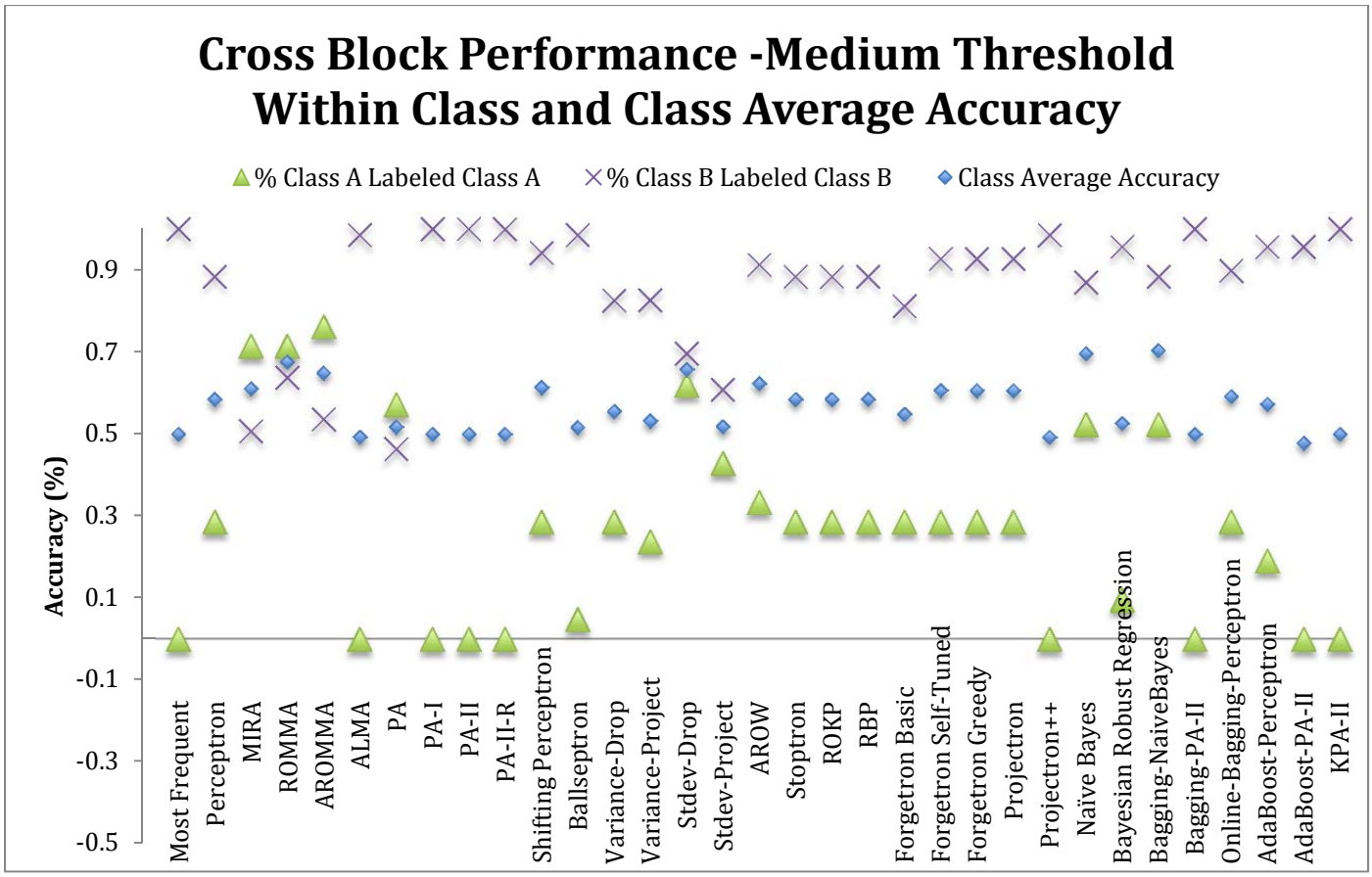

Figure 3. Performance of algorithms on SNL Binary Classification Problem

4. INFRASTRUCTURE

4.1. Overview

The Cognitive Foundry [Basilico, 2008] is a Java software library designed for the research and development of intelligent systems. It is composed of several packages, each targeted toward different research and application domains. In this work we expanded the Cognitive Foundry's Machine Learning package, an open-source Java library of machine learning algorithms designed to be easily embedded into applications. We initially created the Machine Learning package to quickly build new applications that require learning as a core component and to facilitate experimentation with a wide variety of machine-learning algorithms within a variety of new and existing applications. As the Foundry has grown, the Machine Learning package has matured into a complete framework for common aspects of machine learning and has become our most used package, based on the need for adaptive, data-driven behaviors in many systems.

A variety of libraries for data mining and machine learning already exist. Many of the most well-known, such as Weka [Hall, 2009] and RapidMiner [Mierswa, 2006], are primarily designed for data-mining experts to interactively explore existing data, although they also provide development interfaces. Our Machine Learning package takes a different approach that focuses on learning rather than data mining by specifically aiming to make it easy to use learning within the critical processing loop of applications, thus emphasizing the development interface as the primary use. One of the most similar libraries is Java- ML [Abeel, 2009], however we take a different design approach, provide a wider variety of learning algorithms, and provide multithreaded implementations. The Foundry's emphasis on interfaces and generics makes it easier for developers to use learning algorithms with their own application-specific data structures, rather than enforcing a predetermined data model. Furthermore, it supports the research of new machine learning algorithms by providing a large collection of common algorithmic building blocks. Lastly, the Foundry provides extensive developer documentation for every public class and method along with thorough unit tests to verify the quality of each algorithm's implementation.

The Cognitive Foundry is designed in an object-oriented manner to be reusable in a variety of applications, thus its design focuses on the concepts of modularity, componentization, extensibility, and embedability. For machine learning, these concepts are embodied in the design pattern of algorithm-as-object: the learning algorithm is separate from both the data and from the output that learning produces, such as a categorizer. To facilitate embedding in applications, the Foundry is designed to be interface-centric while providing default implementations and makes use of Java generics and common Java Development Kit (JDK) classes. Our design emphasizes implementations with composition, serialization, clear parameterization, and state-encapsulation in mind.

4.2. Data Streaming and Feature Extraction

For the purposes of supporting streaming data, a data streaming framework was added to the Coginitive Foundry. This framework allows for the mapping of data in Comma Separated Value (CSV) format into feature vectors with both labels (for classification problems) and identifiers.

A collection of utility classes provide for one-to-one, one-to-many, many-to-many, and many-to-one mappings on the fly from input data columns to elements of the feature vector. For example, an input column containing simple text based state descriptions can be automatically mapped to multiple real valued features using a binary coding methodology. This mapping infrastructure allows for on the fly mapping of input data files into a streaming data source. The implementation is abstract and generic, supporting the easy expansion beyond support for CSV files as inputs.

4.3. Experiment Harnesses

The Cognitive Foundry, through it's algorithm-as-object design pattern, makes it straightforward to swap in and out building blocks of an experiment, including the data to be used, algorithm to be tested, and methodology for evaluating performance. Multiple performance battery classes were created, one per type of data windowing, in order to support running a battery of algorithms through a test suite with a given set of input data. As an example, the execution code for Interleaved Performance Battery follows:

```
final OnlineLearnerValidationExperiment<InputOutputPair<Vectorizable, Boolean>, LearnedType,
Map<String, Object>, Map<String, Object>> experiment =
            new OnlineLearnerValidationExperiment<InputOutputPair<Vectorizable, Boolean>,
LearnedType, Map<String, Object>, Map<String, Object>>();

        CompositeSupervisedPerformanceEvaluator<Vectorizable, Boolean, Boolean>
supervisedCompositeEvaluator =
            new CompositeSupervisedPerformanceEvaluator<Vectorizable, Boolean, Boolean>();
        supervisedCompositeEvaluator.add("error", new MeanZeroOneErrorEvaluator<Vectorizable,
Boolean>());
        supervisedCompositeEvaluator.add("confusion", new
ConfusionMatrixPerformanceEvaluator<Vectorizable, Boolean>());

        CompositePerformanceEvaluator<LearnedType, Collection<? extends
InputOutputPair<Vectorizable, Boolean>>> compositeEvaluator =
            new CompositePerformanceEvaluator<LearnedType, Collection<? extends
InputOutputPair<Vectorizable, Boolean>>>();

        compositeEvaluator.add("supervised", supervisedCompositeEvaluator);
        compositeEvaluator.add("target-estimate", SingletonPerformanceAdapter.create(
            new BinaryEstimateAsDoubleEvaluator<Vectorizable>()));
        compositeEvaluator.add("memory", new MemoryUsageEvaluator(false));
        experiment.setPerformanceEvaluator(compositeEvaluator);

        CompositeSummarizer supervisedCompositeSummarizer =
            new CompositeSummarizer();
        supervisedCompositeSummarizer.add("error", new StudentTConfidence.Summary(0.95),
"error");
        supervisedCompositeSummarizer.add("confusion", new
DefaultConfusionMatrix.CombineSummarizer<Boolean>(), "confusion");
        supervisedCompositeSummarizer.add("accuracy-windowed", WindowedSummarizerAdapter.create(
            windowSize, new ConfusionMatrixAccuracySummarizer<Boolean>()), "confusion");
        CompositeSummarizer compositeSummarizer =
            new CompositeSummarizer();
        compositeSummarizer.add("auc", new AreaUnderCurveSummarizer(), "target-estimate");
        compositeSummarizer.add("supervised", supervisedCompositeSummarizer, "supervised");
        compositeSummarizer.add("memory", new StudentTConfidence.Summary(0.95), "memory");
        compositeSummarizer.add("auc-windowed", WindowedSummarizerAdapter.create(
            windowSize, new AreaUnderCurveSummarizer()), "target-estimate");

        experiment.setSummarizer(compositeSummarizer);

        InterleavedPerformanceExperimentListener listener = new
InterleavedPerformanceExperimentListener(
            experiment);
        experiment.addListener(listener);

        Map<String, Object> summary = experiment.evaluatePerformance(learner, data);
```

4.4. Hyperparameter Optimization

This section provides a description of the tool developed using the Cognitive Foundry in combination with internal Sandia software to do parameter studies on the hyperparameters used in machine learning algorithms contained in the Cognitive Foundry. This was developed to be easily configurable and setup for use on Sandia's high performance capacity clusters. The tool consists of several python scripts, configuration files, and xml files used for configuration.

4.4.1 Inputs

The configuration for this tool is in a series of xml files, which will be discussed in turn. The first of these files is the parameter study configuration file. This file is broken up into four main sections: a section for defining the dataset, a section for defining how the algorithms are evaluated, a section for defining the behavior on the cluster, and a section for configuring the different algorithms. This file is updated with each new parameter study as the algorithms and/or parameters to study are adjusted.

The dataset section defines what data source to use for the study. The only thing required for this section is the path to where the data source, which is typically a file of comma-separated values, is located. Example configuration files are provided in an appendix.

The next section defines how the algorithms are to be evaluated for the study. In this section, the main class for running the study is defined, along with the performance measure used to evaluate the algorithms. The main class is defined simply by the fully qualified path to the class, but the performance measure is a path to an xml file. This xml file is used by XStream, a third party library that serializes to/from xml, to instantiate an evaluator for use in the study. To allow use of a new performance measure, the only setup required is the creation of an xml file in the proper XStream format for the new performance measure along with the Java class it represents.

The section for configuring the cluster behavior simply consists of two parameters. The first of these is the number of nodes to allow the optimizer to use on the cluster. This number is on a per algorithm basis because each algorithm is scheduled as a different job. If there are, for example, three algorithms configured in this study, then up to thirty nodes could be used. The second of the parameters is the wall clock time, which is the maximum amount of time the cluster will allow the job to run. This, again, is on a per algorithm basis due to each algorithm being a different job.

The last section is where the actual behavior of the study is configured through defining which algorithms and parameters are to be studied. For each of the algorithms to be studied, there is a sub section where the parameters to study for that algorithm are defined. A grid search option is used to study these parameters and the values are defined in each sub section. For each parameter that is to be studied, there are two different options for defining the values: explicitly define each value to use or define a min/max and how many values you want to sample from in between. When defining a min/max, the values can either be sampled linearly (i.e. 1, 2, 3, 4, 5) from the min to max or on a log scale (i.e. 10^{-3}, 10^{-2}, 10^{-1}).

The rest of the input to the tool is in more xml files in XStream format for creating the machine learning algorithms and the performance measures. These files need only be created once and then don't need to be changed again unless the class the xml is representing changes.

4.4.2 Running the tool

The tool consists of three python scripts: one to run the study, one to post-process the results, and one to generate a template configuration file. The first step in running the tool is to create the input configuration file as described above. As a starting point, a copy can be made of a previously used configuration file or the python script can be run to generate one. If the python script is used to generate one, every machine learning algorithm that is configured for use will be included and the sections that aren't needed for the study can simply be removed.

The next step is to run the main python script to schedule the jobs on the cluster. This script will parse the xml configuration file and schedule the jobs to be run on the cluster. As mentioned above, each algorithm to be scheduled has a different job and will be subject to the job queue. As the jobs begin to finish, the results will be output into different folders for each algorithm in an internal output format. Each of the resulting output files has every combination of parameters that the optimizer used and the resulting objective function value as defined by the performance measure. If the number of parameters and algorithms used in the study gets too large, it can be hard to find the best set of parameters and determine which algorithm performed the best on the dataset. The last of the python scripts seeks to solve this by aggregating the results into a single file. This output file contains the best combination of parameter values for each algorithm that was run with the resulting objective function value.

4.5. Optimized Matrix Math

When this work began, the Foundry codebase contained built-in matrix arithmetic codes built on the Matrix-Toolkit Java (MTJ) codebase. We were leveraging this code for a relatively large system of equations (15K by 15K), and found that the code was running very slowly. When investigating, we found that the problem was in sparse and diagonal matrix multiplications. We found this very odd as sparse and diagonal matrix multiplications should be far faster than dense matrix multiplications. This section is organized as follows: First, we briefly describe matrices and the reason for sparse and diagonal matrix implementations. We then describe each of the implementations we built. We provide a brief analysis of the results of our implementations. Finally, we discuss further improvements that could be made to our implementations

4.5.1 Matrices – Dense, Sparse, and Diagonal

Matrices are a mathematical abstraction for representing a system of equations on a set of unknowns. In general, there are m equations and n unknowns (variables). Matrices store the multiplier applied to each variable in the same order for each equation as an element in a two-dimensional array. This results in each equation being stored in one row of the matrix, and all equations' multipliers for a single variable being stored in one column of the matrix.

In general, any element of this two-dimensional array can be non-zero. In many applications, nearly all of them are non-zero. These types of matrices are called "dense" and they require the

full two-dimensional array for storage. However, many very interesting problems are not dense. Two common classes are “diagonal” and “sparse”.

Diagonal matrices are systems of equations where each equation only has one unknown, and all equations have a different variable. Thus, the matrix has only one non-zero element in each column and has only one non-zero element in each row. Specifically, this matrix is set up so that the non-zero elements are stored down the main diagonal (from the upper-left corner to the bottom-right corner) and these matrices are square (m = n). Solving these matrices is trivial – and yet they are often a crucial component of a matrix problem.

Sparse matrices arise in many real-world matrix problems (for instance, graph representations, finite-element systems, and “bag of word” document representations). In a sparse matrix system, most equations have more than one unknown, but far fewer than n. The problem is still quite difficult to solve as the various equations overlap in which variables have non-zero multipliers, and so all equations are interdependent. However, since most elements of the two-dimensional representation are zero, storing a sparse matrix in a dense format is very memory intense and results in considerable wasted computation – multiplying or adding with zeroes. By switching to a sparse representation – implicitly storing the zeroes, you can solve a much larger system of equations on the same computer.

Returning to our original problem: We wanted to solve large problems (tens of thousands of equations of tens of thousands of unknowns). Building the system of equations required matrix multiplication and addition between various sparse and diagonal matrices. However, MTJ’s implementation was orders of magnitude slower for sparse and diagonal operations than it was for dense operations. We decided to see if we could build something faster than MTJ in a few days’ work.

We were able to build something far faster in a few days and then spent a few weeks making it robust, and adding all of the further features (matrix inversion, determinant, etc.) supported by MTJ’s matrices. Our final implementation was added to the Foundry as the “optimized” matrix package. We were unable to replace all of the Foundry’s dependencies to the MTJ matrices before funding ended, although future implementations and improvements can now depend on a more efficient matrix library.

4.5.2 Dense Implementation

Our dense matrix implementation is relatively straightforward. We represent the matrix as a two-dimensional array of double-precision floating-point values. We use Java’s fundamental `double` type instead of its Object-based `Double` type, as we don’t require the additional functionality from the Object-based type, and it requires considerably more memory per value.

To further speed up implementation, we rely on the Basic Linear Algebra Subprogram (BLAS) package for the operations it supports. BLAS is a set of operations that have been optimized for performance on various machines. There is a Java implementation that is our default package. BLAS can be optimized to specific machines – callable as native codes from Java. Those codes can be considerably faster than their Java counterparts. If you want the fastest versions of matrix

multiplication, you should follow the instructions in the appendix for compiling and integrating these versions locally on your machine.

We provide the same Java interface for the dense matrix class as that provided by MTJ's matrix classes. We found some further operations necessary to implement required functionality (QR decomposition for matrix inversion, SVD and LU for determinant), and provide those operations as non-interface methods. For these decompositions, we rely on the Linear Algebra Package (LAPACK). Unfortunately, we were unable to integrate machine-specific optimized versions of LAPACK before funding ended.

To further optimize this code, we created optimized codes for all operations by dense matrices on sparse or diagonal matrices. These codes are able to execute faster than Java's BLAS between two dense matrices.

4.5.3 Diagonal Implementation

As a diagonal matrix can only store non-zero entries along its main diagonal, we store only those entries as a one-dimensional array of `double` elements. Although the diagonal matrix class implements the same interface as all other matrix classes, some operations are not supported (such as setting off-diagonal elements to be non-zero). This class uses O(n) storage, and O(n) or $O(n^2)$ operations (compared to dense which is $O(n^2)$ storage, and $O(n^2)$ or $O(n^3)$ operations).

All code in this class is implemented in our own code (no calls to BLAS or LAPACK). This is possible as all diagonal operations are trivial.

4.5.4 Sparse Implementation

Sparse matrices can be non-zero at any element in the matrix, but are zero at most elements in the matrix. To allow this flexibility, our sparse matrix has m sparse rows. The sparse rows are represented as a map of integer/double pairs (the integer specifies which column is non-zero, and the double specifies the value). This layout allows quick setting of elements of the matrix. However, we found that when operating with this matrix, the code was very slow. The slowness is due to two issues: First, Java's map class requires Object-based `Double` and `Integer` classes. This increases the size of each element considerably. Second, the map stores each element in a different place on the heap, so running through the elements of the matrix requires jumping about in memory – removing the benefits of hardware caching for large matrices.

Because of these issues, we transform the matrix to the Yale format before any operation (see http://en.wikipedia.org/wiki/Sparse_matrix#Yale_format). The Yale format stores the non-zero elements in a row-order array of doubles, and the columns for those elements in a row-order array of integers. A third array stores the index into the first two arrays for the first element of each row. The benefit of this format are that it allows for memory-contiguous, and primitive-type double and int representations – significantly improving caching performance. The negative is that the matrix cannot be altered in this format. Thus, before operating with a matrix, we make sure it is in the Yale format, and before altering a matrix, we make sure it is the map-based format. There is a one-time cost when transitioning between the formats, but as long as

multiple operations are being performed in-a-row, or if the operation is a multiplication (which passes over all the elements multiple times), the transformation leads to a considerable savings.

We coded nearly all operations ourselves. However, some operations (such as inverting the matrix, and solving for the determinant) would result in dense intermediate formats. In such cases, we convert the sparse matrix to a dense matrix and call the dense-matrix-equivalent function. We realize that sparse matrices are often used for problems so large that a dense matrix can't represent them in memory – and so transitioning to a dense matrix to solve the system of equations is not practical. However, with sparse matrices, generally the system of equations is not directly solved – an iterative approximation solver (such as Steepest Descent or Conjugate Gradient) is used. We implemented these solvers for any matrix in the Foundry as well [Shewchuk, 1994]. If you are trying to solve a sparse system of equations that are symmetric positive definite (many are), we recommend these sparse solvers. If your system of equations is not symmetric positive definite, you can still use our Conjugate Gradient Minimizer solver that minimizes the error of $A^TAx = b$. If A is square and nonsingular, this still provides the solution to $Ax = b$.

4.5.5 Results

We performed some simple tests to see how quickly our operations are performing. We focused on matrix-matrix multiplication between all three matrix types. The below figure shows the wall-clock execution time for different size matrices and different types. When we tried repeating this with the MTJ matrix classes, we found their dense-dense multiplication was slightly faster (~3%), but that their dense-sparse multiplication on the smallest size didn't finish in minutes. We decided at that point to stop testing MTJ's multiplication speeds.

The following graph shows the log-log plot of square-matrix size (one dimension shown) and execution time. All matrices were filled with random numbers. The sparse matrices were 25% filled (75% zeroes) – a rather full sparse matrix compared to finite element matrices. Even so, dense-matrix/dense-matrix multiplication was consistently the slowest, with significant speedups for adding any sparse matrices, and massive speedups for diagonal matrices. In a separate study (not shown) we demonstrated that as the density of the sparse matrix decreases, the execution time also decreases.

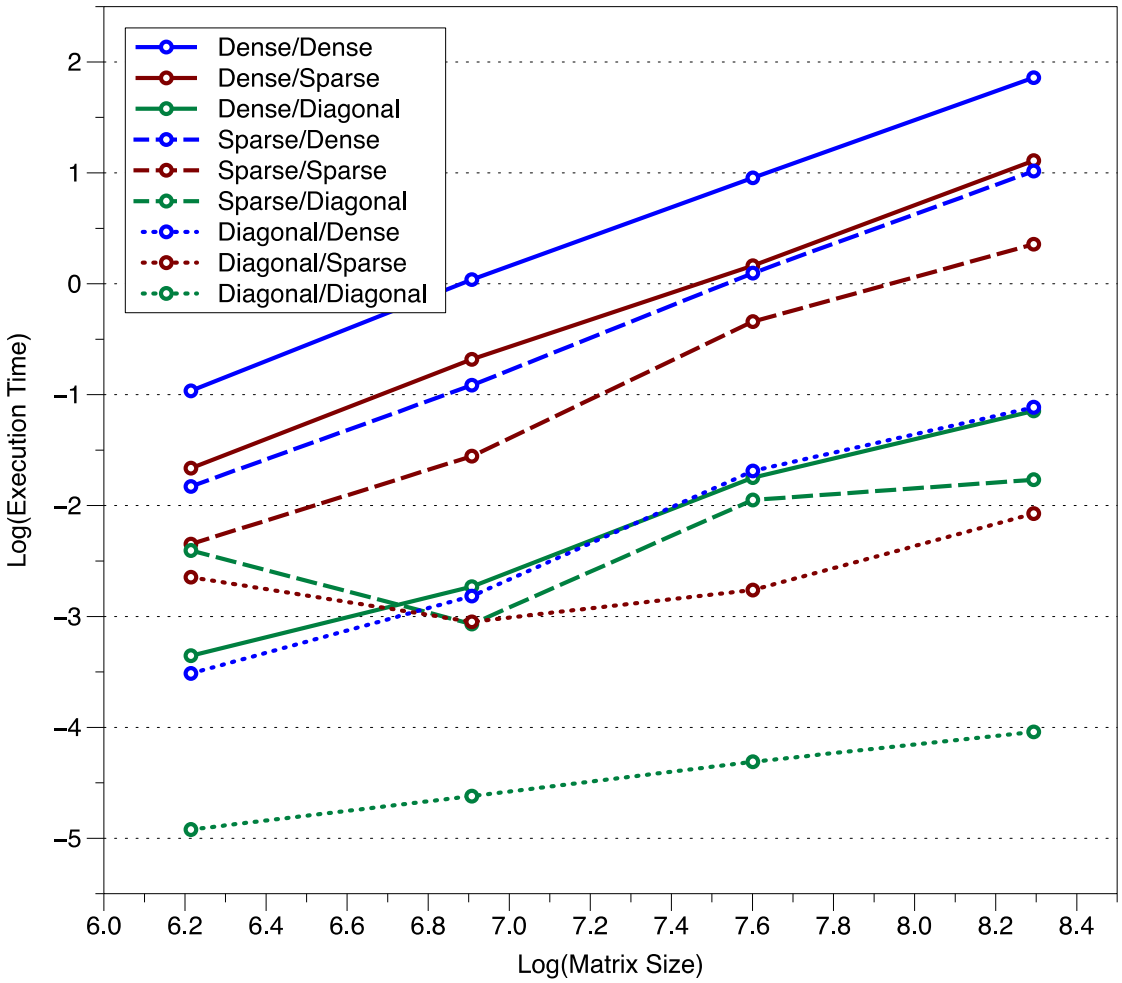

Figure 4. Performance of Optimized Matrix Operations

4.5.6 Further Improvements

Although we called the package containing our matrix classes optimized, we realize it can still be optimized further. Specifically, we recommend the following optimizations:

1) Use native LAPACK to optimize the dense-matrix decomposition methods if native LAPACK is available.
2) Multithreading most matrix operations should be trivial. Parallelizing to 4 or 8 threads should cause 3+ or 7+ times speedups (less than 4 or 8 due to overhead).
3) We did not have time to dig into machine-specific, cache-layout improvements. If we could discover the cache-page size for the resident machine, we might be able to layout the matrix in memory for more efficient operations.

5. REFERENCES

Abdulsalam, H., D. B. Skillicorn, P. Martin. Classification Using Streaming Random Forests, *Knowledge and Data Engineering, IEEE Transactions on* , vol.23, no.1, pp.22,36, Jan. 2011.

Abeel, Thomas, Yves Van de Peer, and Yvan Saeys. Java-ML: A machine learning library. The Journal of Machine Learning Research, 10:931–934, 2009.

Andberg, M.R., 1973. Cluster Analysis for Applications. Academic Press, New York.

Arthur, D., & Vassilvitskii, S. K-means++: the advantages of careful seeding. In 2007 ACM-SIAM symposium on discrete algorithm (SODA'07). 2007

Basilico, Justin, Zachary Benz, and Kevin R. Dixon. The Cognitive Foundry: A flexible platform for intelligent agent modeling. In Proceedings of the 17th Behavior Representation in Modeling and Simulation (BRIMS) Conference, 2008.

Bottou, L. and Y. Bengio. Convergence properties of the kmeans algorithm. In *Advances in Neural Information Processing Systems.* 1995.

Cavallanti, Giovanni. Nicolo Cesa-Bianchi, and Claudio Gentile. Tracking the best hyperplane with a simple budget Perceptron. Machine Learning, 143-167. 2007.

Cesa-Bianchi, Nicolo and Claudio Gentile. Tracking the Best Hyperplane with a Simple Budget Perceptron. Conference on Learning Theory, 483-298. 2006.

Crammer, Koby and Yoram Singer. Ultraconservative Online Algorithms for Multiclass Problems. The Journal of Machine Learning Research, 951-991. 2003.

Crammer, Koby, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. Online Passive-Aggressive Algorithms The Journal of Machine Learning Research, 551-585. 2006.

Crammer, Koby, Alex Kulesza, and Mark Dredze. Adpative Regularization of Weight Vectors. Advances in Neural Information Processing Systems. 2009.

Dekel, Ofer, Shai Shalev-Shwartz, and Yoram Singer. The Forgetron: A Kernel-based Perceptron on a Budget. SIAM Journal on Computing, 1342-1372. 2008.

Domingos, Pedro and Geoff Hulten.. Mining high-speed data streams. In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '00). ACM, New York, NY, USA, 71-80. 2000.

Dredze, Mark, Koby Crammer, and Fernando Pereira. Confidence-Weighted Linear Classification. International Conference on Machine Learning. 2008.

Gentile, Claudio. A new approximate maximal margin classification algorithm. The Journal of Machine Learning Research, 213-242. 2002.

Geurts, P., D. Ernst, and L. Wehenkel. Extremely randomized trees. Machine Learning, *63*(1), 3-42. 2006.

Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. SIGKDD Explorations Newsletter, 11(1):10–18, 2009.

Heimsund, Bjorn-Ove. Matrix toolkits for Java, 2011. http://code.google.com/p/matrix- toolkits-java.

Mierswa, Ingo, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. YALE: Rapid prototyping for complex data mining tasks. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2006.

Orabona, Francesco, Joseph Keshet, and Barbara Caputo. Bounded Kernel-Based Online Learning. The Journal of Machine Learning Research, 2643-2666. 2009.

Saffari, Amir, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line Random Forests, in 3rd IEEE ICCV Workshop on On-line Computer Vision, 2009.

Sculley, D. Web-Scale K-Means Clustering. In Proc. Of the 19th Int. Conf. on World Wide Web (WWW'10). 2010, pp. 1177-1178

Shalev-Shwartz, Shai and Yoram Singer. A New Perspective on an Old Perceptron Algorithm. Conference on Learning Theory, 815-824. 2005.

Shewchuk, Richard Jonathan. An Introduction to the Conjugate Gradient Method without the Agonizing Pain, Edition 1 ¼. August 1994.

Wang, Zhuang and Slobodan Vucetic. Online Passive-Aggressive Algorithms on a Budget. In Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS). 2010.

APPENDIX B: INSTALLING OPTIMIZED BLAS

You may wish to stop using Java's unoptimized implementations of BLAS. The following instructions assume Mac OSX. If you're using a different OS, these may not work. (These instructions derived from http://jeshua.me/blog/NetlibJavaJNI.)

We believe similar steps could be done to improve LAPACK functionality, but were unable to implement and validate those improvements by the end of funding.

1. Download and get into the latest Cognitive Foundry
   a. Download from http://foundry.sandia.gov/download.html
   b. Unzip it
   c. Open Terminal
   d. cd CognitiveFoundry-X.X.X/Dependencies/netlib-java-X.X.X/
2. Download and set up the correct paths for elements
   a. cp arpack-combo-0.1.jar arpack-combined.jar
   b. cp netlib-java-0.9.3.jar netlib-java-dev.jar
   c. cd jni
   d. JNI_DIR=/System/Library/Frameworks/JavaVM.framework/Versions/A/Headers/ (NOTE: Make sure jni.h is in that folder. If it's not do a search for jni.h on your machine and replace with the correct path.)
   e. Download http://www.caam.rice.edu/software/ARPACK/SRC/arpack96.tar.gz and http://www.caam.rice.edu/software/ARPACK/SRC/patch.tar.gz
   f. mv ~/Downloads/arpack96.tar.gz .
   g. mv ~/Downloads/patch.tar.gz .
   h. sh configure
   i. jni_replacement=$(printf "%s\n" "$JNI_DIR" | sed 's/[\&/]/\\&/g')
   j. sed -i -e "s/CPPFLAGS=/CPPLAGS= -I${jni_replacement} -I. /g" Makefile.incl
   k. sed -i -e "s/CFLAGS=/CFLAGS= -I${jni_replacement} -I. /g" Makefile.incl
3. Make sure you have the right Fortran tools (and link to the right ones in the code)
   a. sudo port install g95
   b. edit Makefile.incl change the FORTRAN_LIBS line to say the following –> FORTRAN_LIBS=-L/opt/local/lib -lf95 -Wl,-single_module
4. Compile
   a. make
5. Move the jnilib files to the right spot
   a. sudo cp libjni*.jnilib /System/Library/Java/Extensions/
6. In Java code, check to see if you're using the native version of BLAS (vs. Java's JVM version) – Foundry does this automatically.
   a. "org.netlib.blas.NativeBLAS".equals(BLAS.getInstance().getClass().getName());

DISTRIBUTION

1	MS1027	Zachary Benz	5635
1	MS0899	Technical Library	9536 (electronic copy)
1	MS0359	D. Chavez, LDRD Office	1911

Sandia National Laboratories