

Scaling Up Classifiers to Cloud Computers

Christopher Moretti*, Karsten Steinhaeuser*, Douglas Thain, Nitesh V. Chawla
Department of Computer Science & Engineering
University of Notre Dame Notre Dame, IN 46556, USA
{cmoretti,ksteinha,dthain,nchawla}@cse.nd.edu

Abstract

As the size of available datasets has grown from Megabytes to Gigabytes and now into Terabytes, machine learning algorithms and computing infrastructures have continuously evolved in an effort to keep pace. But at large scales, mining for useful patterns still presents challenges in terms of data management as well as computation. These issues can be addressed by dividing both data and computation to build ensembles of classifiers in a distributed fashion, but trade-offs in cost, performance, and accuracy must be considered when designing or selecting an appropriate architecture. In this paper, we present an abstraction for scalable data mining that allows us to explore these trade-offs. Data and computation are distributed to a computing cloud with minimal effort from the user, and multiple models for data management are available depending on the workload and system configuration. We demonstrate the performance and scalability characteristics of our ensembles using a wide variety of datasets and algorithms on a Condor-based pool with Chirp to handle the storage.

1 Introduction

The last decade witnessed a surge in the availability of massive datasets. Data collected from various scientific domains and real-world applications is quickly overwhelming computing systems and data mining algorithms, presenting a challenge for theoreticians and practitioners alike. Parallel and distributed data mining [19] have afforded us with scalable implementations of various learning algorithms, allowing a capability to scale to massive datasets while also enabling a significant improvement in accuracy.

Distributed data mining is a particularly attractive solution as one can partition a dataset into subsets, distribute them across multiple processors, and learn independent classifiers before coalescing them as an ensemble. An ad-

vantage of distributed data mining approaches is that the partition size of the learning task can be broken down to fit the available (commodity) computational resources. One can easily imagine a divide-and-conquer approach in which a dataset is distributed to a group of processors. Each of those processors learns a classifier concurrently, and reports its classifier to a central processor. The central processor can then process the predictions of the independent classifiers learned. Distributed data mining leads to a creation of an ensemble or committee of “diverse” classifiers. Each classifier is given a smaller sub-task of the learning task to learn, and hence the complexity of the learning task at hand is reduced. It also introduces diversity among the classifiers, leading to an improvement in accuracy. Moreover, learning on the entire very large training set, without partitioning, can force the inductive learner to over-fit the problem as it will try to model the entire training set; the learned classifier will then tend to lose its generality.

Of course, the open question is what sized subsets or data partitions to create. There really is “no known method of sample selection and estimation which ensures with certainty that the sample estimates will be equal to the unknown population characteristics” (p. 26) [12]. To do any intelligent subsampling, one might need to sort through the entire dataset, which could take away some of the efficiency advantages of distributing the workload in the first place.

Contributions While it has been shown that ensemble classifiers generally improve accuracy over the single classifier and offer computational advantages, various questions remain: 1) how to appropriately partition the data into subsets for learning? 2) what are the limits of scalability? 3) how to best exploit the available resources? Thus, the key contributions of the paper are as follows: 1) a scalable and efficient abstraction for distributing data to different sites; 2) a thorough comparison of multiple ways of partitioning and distributing data; 3) a scale of datasets to evaluate the performance three different learning algorithms – decision trees, k-nearest neighbors, and support vector machines – under the distributed setting.

*Denotes equal contribution.

2 Related Work

The problem we address lies at the intersection of data mining and high-performance computing. Accordingly, we provide a survey of relevant work from both areas.

Dataset sizes that exceed the memory capacity of a desktop computer pose a major challenge for data mining. This limitation can be mitigated through optimized algorithm design [21] and the use of sampling [6] or ensemble methods [4]. With improvements in multi-processor machines, and more recently multicore technology, greater scalability can be achieved by effectively parallelizing algorithm implementations [8, 17, 24, 31]. But these approaches remain limited because (i) performance gains often cannot be realized beyond 8-16 cores due to communication overhead and (ii) dataset sizes are restricted to the total memory available in the system, generally on the order of a few Gigabytes.

To overcome these hurdles and achieve not incremental improvements, but drastically increased scalability, the workload can be divided across a much larger distributed system, or computation grid [3, 11, 14, 20, 23, 27]. This approach has proven successful for certain tasks [2, 9], but such systems often require an application-specific design and implementation. In contrast, general-purpose systems may require less effort from the programmer and/or user but still cannot scale beyond several Gigabytes of data [10].

Our work bridges this gap by providing a generic abstraction for large-scale data mining, enabling the user to run his own algorithms with minimal programming effort. The abstraction is capable of managing both data and computation on various types of distributed systems ranging from small clusters to large dynamic computing clouds.

3 Abstraction for Distributed Data Mining

Modern computing systems provide the user with a large amount of parallelism. Despite many years of research into multi-threaded, message passing, and parallel programming languages, harnessing this parallelism remains very difficult for the non-expert user. Parallel machines commonly used today include:

Multicore computers: machines with multiple CPUs on a single chip that share a common RAM and run a single operating system image. At the time of writing, most desktop machines are two- or four-way multicore CPUs, and it is expected that future machines will have many more cores.

Cluster computers: collections of tens to thousands of individual machines, each with their own (perhaps multicore) CPU, RAM, and disk, all connected by a fast switch to some type of centralized filesystem. A cluster is typically homogeneous, reliable, and dedicated to a single user at a time. A user that requests 16 CPUs will have sole access to exactly those 16 CPUs for the length of the request.

Cloud computers: collections of hundreds to tens of thousands of machines, different from clusters in two key respects. First, few centralized filesystems scale to cloud size, so a cloud makes use of individual disks on each node for both temporary and permanent storage. Second, because a cloud naturally has a high failure rate, it does not allocate specific nodes to users, but assigns resources dynamically.

To exploit the physical parallelism in these systems, we advocate *abstractions* that join together simple sequential programs into data parallel graphs. This allows rapid reuse of existing data mining codes without confronting the substantial challenges of writing applications using multi-threaded or message-passing libraries. This approach has been used successfully in systems such as Map-Reduce [7], Dryad [15] and All-Pairs [22]. In this work, we define the abstraction *Classify* as follows:

Classify(D, T, P, N, F, C) returns R: D - Training set: list of (name,properties) T - Testing set: list of (name,properties) P - Partitioning method. N - Number of partitions. F - Classifier function. C - Collection process. R - Result set: list of (name,class)

As shown in Figure 1, the *Classify* abstraction feeds dataset *D* into process *P*, which creates *N* partitions *D1...DN*. These are fed into *N* copies of *F* in parallel along with testing set *T*, generating results *R1...RN*. Results are combined by process *C* by majority voting into a final result *R* returned to the user. Classifier function *F* is simply an existing sequential classifier with the following signature:

F(D, T) returns R: D - Training set: list of (name,properties) T - Testing set: list of (name,properties) R - Result set: list of (name,class)
--

The user may choose from a variety of partitioning techniques for the training set. *Shuffle* selects data items one at a time and sends each to a random partition, resulting in roughly equal-sized partitions. A shuffle partition may also be *M-overlapping*, in which an item may appear in *M* partitions, allowing for more accurate sampling of minority classes but increasing data sizes and runtimes. *Chop* divides the training set into equal pieces, preserving the existing order. This is typically only appropriate when the data is pre-randomized, or when the user wishes to reproduce runs exactly. We will show that the choice of partition can have a significant effect on the implementation.

Classify appears similar to the abstraction Map-Reduce [7]. Our assignment of tasks *F* onto *D1...DN* is completed by the Mapper function, and *C*, the collection

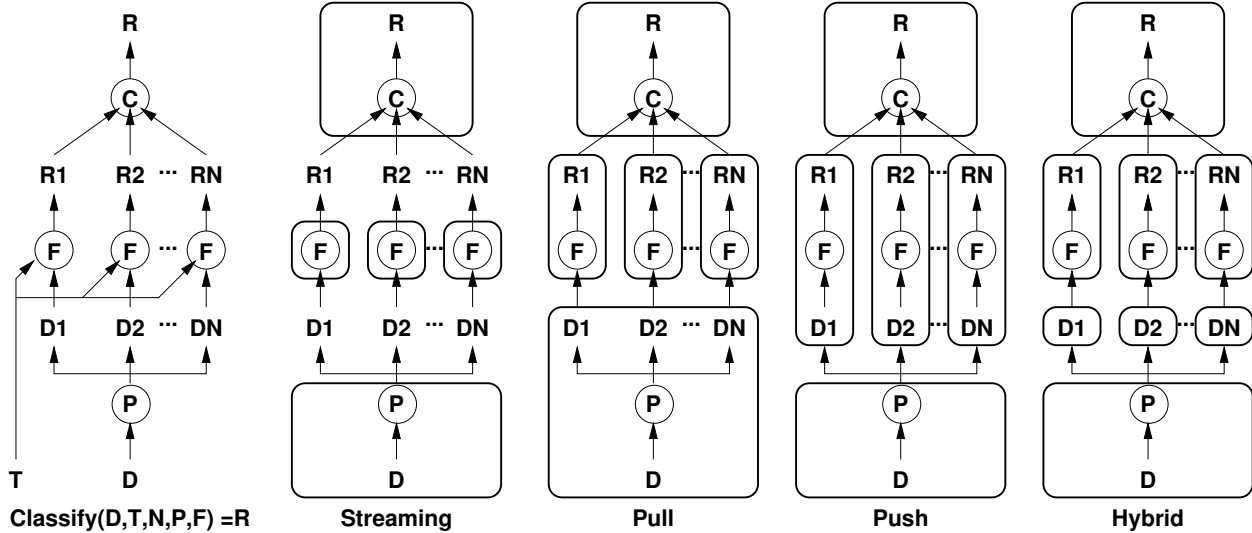


Figure 1. Four Implementations of the Classify Abstraction

This figure shows four possible ways of implementing the Classify abstraction by varying the placement of data and functions on the nodes of the system. Rounded boxes show the boundaries of one node in the system, which has both a CPU and local storage. For example, in the Pull implementation, the partition function P reads the training data D and writes the partitions D1...DN back to the same node. Each of the classifiers F run on separate nodes and pull the data over the network. But in Push, the partition function P reads the data D from one node and writes the partitions directly to the execution nodes, where the classifiers F read the local copy. Full details are given in Section 4.

of results of the subclassifiers into a final classification, is the job of the Reducer function. But several components of classification are not strictly accounted for by the Map-Reduce abstraction. The Map-Reduce model does not consider logical partitioning as a first-class component of the model, rather it delegates partitioning as an implementation detail of physical partitioning of the underlying filesystem.

Some Map-Reduce implementations [13, 5, 26] adapt the Map-Reduce model to recognize logical partitioning in various ways, such as allowing for custom partitioning algorithms or actually including partitioning as primitive in their adjusted models. Mapping logical partitions onto physical partitions within the filesystem, however, remains a characteristic highly dependent on the implementation rather than strictly defined within the Map-Reduce abstraction.

The testing set also does not fit into the Map-Reduce abstraction well. It must either be encapsulated in the Mapper and Reducer functions – a departure from the logical description of the Map-Reduce abstraction – or it must be stored on the distributed filesystem at a cost of multiple replicas and significant metadata for each instance of this one-time-use file.

Our intent is careful study of data placement and access. Instead of attempting to derive Classify from the general Map-Reduce, we chose to implement an abstraction that considers classification elements relating to data placement directly as first-class components of the abstraction model.

4 Implementing the Abstraction

There are many possible ways to implement Classify in a parallel or distributed system. An implementation must choose how many nodes to use for computation, how many to use for data, and how to connect the two. Figure 2 shows several possibilities we have explored, differing only in where data is placed in the system. Below we will explore the consequences of each of these choices on performance.

Streaming. The simplest implementation of Classify connects each process in the system at runtime via a *stream* such as a TCP connection or a named pipe. Data only exists in memory between processes and, except for some minimal buffering, a writer must block until a reader clears the buffer of data. However, this requires that all processes be ready to run simultaneously and affords no simple recovery from failure. If one process or stream fails, the entire abstraction must start from the beginning. Thus, it is an appropriate implementation for a multicore machine when the number of partitions is less than or equal to the number of processes. Except for very small workloads, it is not practical for a cluster or a cloud where the possibility of network or node failure is very high. To make the abstraction robust, we must make use of some storage between processes.

Pull. In this implementation, P reads data from the source node and writes partitions back to the same node. When the various Fs are assigned to CPUs, they connect to

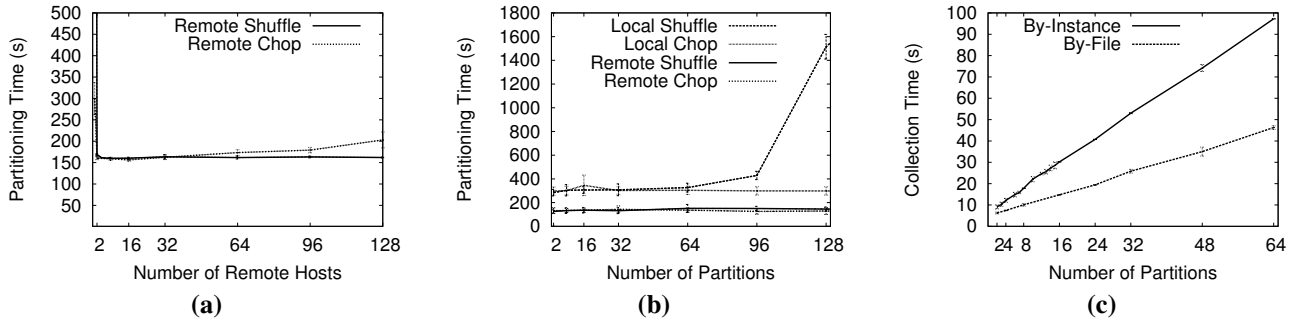


Figure 2. Performance of Partitioning and Collecting

2(a) shows the time to partition 5.4GB of data into 256 partitions on a single local disk or a varying number of remote disks. Figure 2(b) shows the time to partition 5.4GB of data into a varying number of partitions, using a single local disk and writing to 16 remote disks. 2(c) shows the time to collect classifier output (3.2MB per partition) from each of a varying number of remote disks. By-file collection uses 91MB, while by-instance uses less than 1KB.

the source node and *pull* in the proper partition. This provides maximum runtime flexibility as there is no constraint on where an F may run. Because each partition is stored on disk, individual Fs may fail and restart without affecting the rest of the computation. However, as we will show, this places a significant I/O burden on the source node in both the partitioning and classifying stages. The technique may be appropriate for a cluster with a large central file server, but is not likely to scale to a cloud of any significant size.

Push. In this implementation, P chooses *in advance* which nodes will be responsible for working on each partition. As it reads data items from the training set, they are *pushed out* directly to the assigned nodes. The Fs are then dispatched for execution. In “Pure Push”, each F must run only on the node where data is located. This may not be possible in a cloud, where that node may have been dynamically assigned to an unrelated task. Therefore we also define “Relaxed Push”, where each F prefers to run on the node with its partition but may also run on another node and access that partition remotely. This technique can (significantly) improve the performance of partitioning and the overall I/O rate as the number of nodes increases, but also increases the exposure of the system to failed, slow, or otherwise misbehaving disks.

In large clusters or clouds, we would like to Push data to a number of remote nodes equal to the number of partitions to maximize parallelism. Figure 2(a) shows, however, that chop partitioning to a large number of remote resources begins to reduce performance due to moving beyond homogeneous clusters and encountering a greater variety of hardware. Shuffle partitioning has its own drawback in a cloud environment, because it requires remote connections to remain open to every remote node throughout the entire partitioning.

Hybrid. To address the limitations of Push and Pull, we also define Hybrid. In this mode, P chooses a small set of intermediate nodes known to be fast, reliable, and of sufficient capacity to write the partitioned data. At runtime, each F then reads its partition over the network from these nodes. This combines advantages of Pull (flexible allocation of CPUs, reliable partitioning) with advantages of Push (increased I/O performance). However, it requires the implementation to have some knowledge of the reliability of the underlying system, which may not always be possible. Figure 2(b) shows that remote partitioning even to a modest set of reliable nodes is faster than local partitioning, without the pitfalls of Pushing data to unreliable environments.

4.1 Implementation Structure

We implemented Classify using Condor [29] to harness computing resources, and Chirp [28] to allow remote filesystem-like access to the storage at each node.

The source node is responsible for several tasks: partitioning the data, configuring local state to describe the batch jobs, submitting the batch jobs, and collection after all jobs have completed. The remote cloud or cluster nodes are responsible for executing the classifier instances and generating the prediction output. Partitioning is described above, so here we describe the remaining structures:

Local State. Local state requirements include an execution directory, the training and test set definitions required by all classifiers, and the batch job definition files. The test set and .names dataset definition are not replicated on the local disk, but rather shared efficiently. The job definition files are created after the data partitioning, and the batch jobs are submitted using these definitions.

Remote Structure. Within the batch jobs themselves, we use a hierarchical architecture of processes. The batch

job that is run on each remote node is the *wrapper*, a standard piece of code that is the same for all instances of Classify. The wrapper is responsible for setting up the execution environment on the remote compute node. The wrapper’s principal job is to execute the *function*, a user-provided, application-specific piece of translational middleware. The function executes the underlying data mining executable (the *application*) and maps application-specific output to the structure expected by the wrapper. The function allows execution of any underlying classifier without having to change core pieces of the abstraction framework.

Collection. We consider two approaches for collection. The first, *by-file*, is analogous to chop partitioning. The algorithm completes one prediction file at a time, maintaining a plurality-determining data structure for each test instance. After all files are processed, each data structure contains the combined final prediction. The overall accuracy, accuracy per class, and other statistics can be computed from these data structures. As the number of instances in the test set increases, this version needs more memory to maintain data structures for each instance, with memory requirements totaling a factor of the product of the number of test instances and the number of classes in the dataset.

The alternative, collecting *by-instance*, is akin to shuffle partitioning. All prediction files are accessed concurrently, and only one data structure is needed as each instance is tallied serially. Memory for this version remains constant as the number of instances increases, since the memory requirement is only a factor of the number of classes in the dataset. On the other hand, it requires more files open at once and accesses prediction files less efficiently.

An abstraction may decide the trade-off between file resources accessed concurrently and memory used for concurrent tallying data structures. For datasets few classes, concurrent data structures for each partition fit in memory easily even when the test set is large. However, for very large numbers of classes or very large numbers of instances in the test file, it is possible for the collection to exceed main memory capacity. Figure 2(c) shows the time required to collect results of a distributed ensemble of classifiers using these two approaches, varying the number of partitions. The input data is the set of prediction files from a run of the KDDCup data, chosen because it the largest by-file memory requirement among our datasets (approximately 91MB).

Because the largest set of prediction files for any configuration we tested consisted of less than 10MB of output, and thus disk space was not a concern, our implementation allows the batch system to return all prediction files to the submitting node, instead of using a separate file server or distributed filesystem. Because the largest collection memory requirement of any dataset we used was less than 100MB, all of our results use by-file collection.

5 Experimental Setup

To evaluate the performance and scalability characteristics of the data mining abstraction described in the previous section, we conduct experiments on a diverse body of datasets using a variety of popular learning algorithms.

Dataset	Training Instances (Size on Disk)	Test Instances (Size on Disk)	Attributes
Protein	3,257,515 (170 MB)	362,046 (20 MB)	20
KDDCup	4,898,431 (700 MB)	494,021 (71 MB)	41
Alpha	400,000 (1.8 GB)	100,000 (450 MB)	500
Beta	400,000 (1.8 GB)	100,000 (450 MB)	500
Syn-SM	10,000,000 (5.4 GB)	100,000 (55 MB)	100
Syn-LG	100,000,000 (54 GB)	100,000 (55 MB)	100

Datasets. We use a combination of real and synthetic datasets with varying dimensions covering a wide range of sizes. The Protein dataset is real data describing the folding structure of different amino acids; the task is to predict the structure of new sequences. The second dataset stems from the 1999 KDD-Cup¹ and contains real network data; the task is to distinguish the “good” ones from the “bad” (intrusion detection). The next two datasets, Syn-SM and Syn-LG, were produced with the QUEST generator [1] using a perturbation factor of 0.05 and function 1 for class assignment. The last two datasets, Alpha and Beta, are taken from the Pascal Large Scale Learning Challenge², which were deemed more appropriate for support vector machines. We found that the other datasets required significant tweaking of SVM parameters even on much smaller subsamples. The focus of our paper is primarily on scalability studies and less on parameter sweep for improvements in accuracy, hence we only use the Alpha and Beta datasets with SVMs.

Algorithms. We include three traditional learning methods for the evaluation of our abstraction framework:

- Decision trees (popular C4.5 implementation [25])
- SVMs (efficient implementation [18])
- K-nearest neighbor classification (our implementation)

The algorithms cover a range of computational complexities and rank among the most popular learning methods. For decision trees and support vector machines, we used the default parameters provided by the respective implementations. For k -nearest neighbor classification we used $k = 5$ neighbors. All of the algorithms were compiled for 32-bit x86 systems with *g++* v3.4.6 using optimization *-O3*.

We selected these algorithms because they naturally fit the distribute-compute-collect paradigm. However, it is worth noting that with only minor modifications to the abstraction we could accommodate other learning methods

¹<http://www.sigkdd.org/kddcup/index.php>

²<http://largescale.first.fraunhofer.de/>

as well, for example Distributed K-Means Clustering [16] or finding frequent itemsets using Apriori-Based methods [30], which may require multiple distributed stages.

Computing Environment The platform used as testbed for our experiments is a Condor pool of approximately 500 machines. The pool consists primarily of workstations in a university environment with both 32-bit and 64-bit x86 processors and memory capacities ranging from 512MB to 4GB. Although the pool as a whole is a computation cloud with limited control for the user, a 48-node subset is in our possession, giving us more power over the environment (e.g. reliability of resources, priority status for execution). The machines in this dedicated cluster are dual-core 64-bit x86 architectures with either 2GB or 4GB of total memory (1GB or 2GB per core, respectively). Jobs were instructed to prefer this cluster over other nodes when available.

For multicore experiments we used 64-bit dual-core AMD Opterons with 2GB of total memory.

Practical Considerations Our main goal is to evaluate scalability with increasing system size, so we cover the range from 1 to 128 nodes for the five smaller datasets. With Syn-LG the memory requirements for each individual partition are much larger, hence we use 48 to 256 nodes instead. In addition, linear support vector machines are only tractable for the Alpha and Beta datasets. For k-nearest neighbor classification, we reduced the test set size to 1,000 instances for the synthetic datasets and to 10,000 instances for all other datasets to keep computation feasible within the system.

6 Results & Analysis

We performed a large number of experiments across datasets, algorithms, and system sizes as described above. In this section, we summarize the results and provide analyses and insights based on our findings with respect to the trade-offs discussed earlier.

6.1 Execution Time: Cloud

Since our primary interest lies in the scalability analysis, we start by examining the trends in execution time. Figure 3 show the execution time for decision trees, k-nearest neighbor classification, and support vector machines on multiple datasets for varying number of partitions. Within the grid of plots, rows correspond to datasets and columns correspond to learning algorithms. Each individual plot contains three lines for the different data distribution methods.

The results for Syn-LG with decision trees and k-nearest neighbors are omitted for space reasons as the trends observed are very similar to Syn-SM, albeit at a larger scale.

In addition, for massive datasets it is difficult to measure Push partitioning. This task is feasible for smaller datasets and controlled environments, but becomes more difficult as the size of the dataset or number of hosts and diversity of the system increases. Next, we examine the results for each of the algorithms in more detail.

Decision Trees The first column of Figure 3 shows strong parallelizability of decision trees across all datasets. In most of the experiments, the data distribution does not significantly influence the execution time through 16 or 32 partitions, demonstrating extensive, though not exclusive, use of the 48-node dedicated cluster. Beyond that threshold, performance diverges as jobs begin utilizing unreliable, heterogeneous nodes from the computing cloud. Even beyond the cluster/cloud threshold, however, we are able to continue to get improved turnaround times for several algorithms using the Hybrid approach.

As an example of a case where additional parallelism did *not* provide any added benefit, the KDDCup plot for decision trees shows that no improvements in execution time are achieved beyond 32 partitions. For decision trees in particular, the small workloads result in very minimal classifier training times. In addition, smaller jobs yield more relative overhead and higher costs to complete the serial stages of the process. It is unsurprising, then, that almost exactly the same amount of time is required for the execution phases when exceeding 32 partitions. For instance, doubling the collection time (twice as many predictions to process per instance) requires more time than is saved by the marginal improvement in execution time afforded by the resources.

Another factor impacting the scalability of executions is the data set size. The Syn-SM set continues to improve execution time using Hybrid through 128-way parallelism, whereas a smaller dataset, Beta, achieves limited further improvement beyond 32 nodes. The primary difference here is that for small data sets, further partitioning results in no effective gain when balancing batch job execution time against additional overhead from greater parallelism (partitioning, collection, and batch system overhead).

For almost all configurations the Hybrid approach yielded shortest turnaround times, and Pull yielded the longest turnaround times. Combining the advantages (and mitigating the disadvantages) of the Push and Pull techniques is particularly apparent as the number of partitions gets larger, and for the larger datasets.

K-Nearest Neighbor Classification The results in the second column of Figure 3 also show encouraging trends in execution time with respect to the number of partitions. For all datasets, we observe consistent improvements in execution time while staying within the small cluster (up to 32 nodes) and with one exceptions also with 64 partitions.

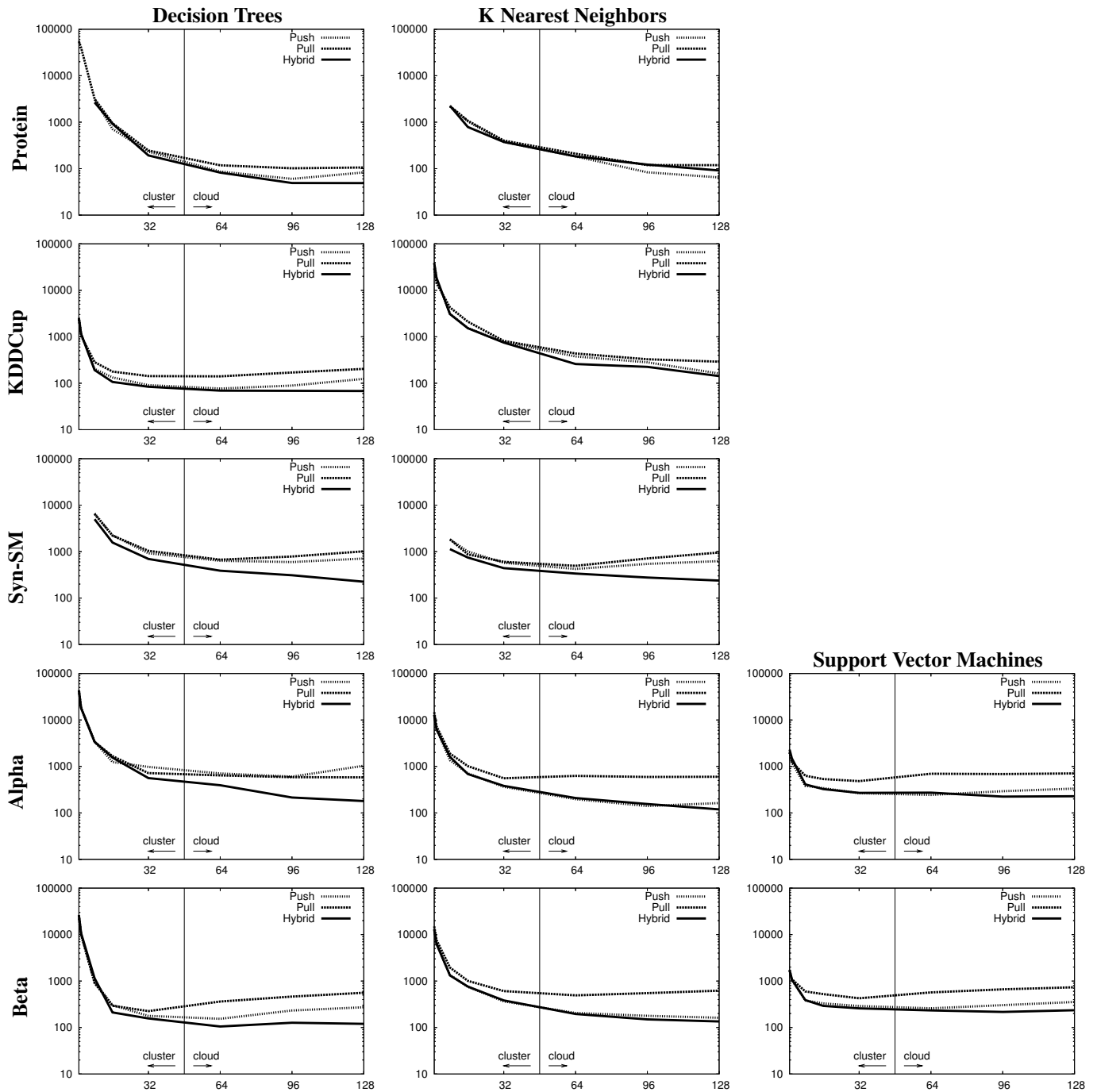


Figure 3. Scalability of Classifiers from a Cluster to a Cloud

This figure shows the runtime of executing Classify on five different datasets with three different classifiers. Each configuration is scaled up from 1 to 32 nodes on a homogeneous reliable cluster, and then up to 128 nodes on a dynamic computing cloud. Each abstraction is run in three different configurations: Push, Pull, and Hybrid, as shown in Figure 1. (Results for SVM are not shown on the first three datasets, because the algorithm does not converge.) Each graph shows the number of hosts on the X axis and the execution time in seconds on the Y axis. Generally speaking, the hybrid implementation is the most robust across the various configurations.

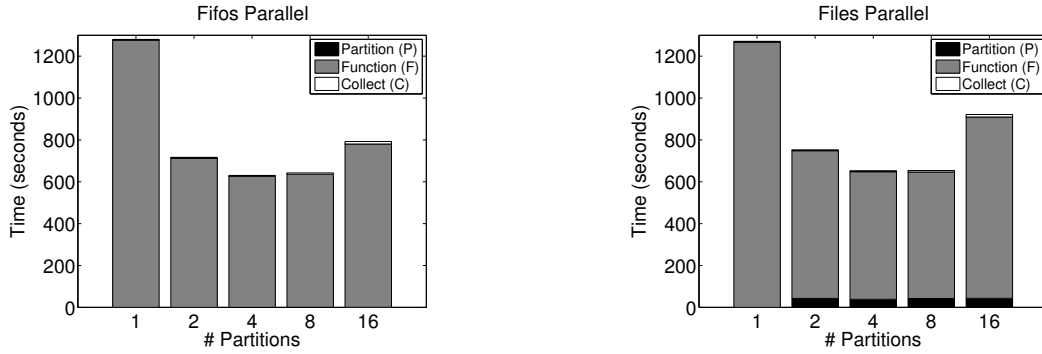


Figure 4. Scalability of Decision Trees on a Multicore Processor

Only for 128 partitions do we see increased execution times in several cases, most notably for the Push method. This behavior is due to some jobs getting placed on slower machines in the computation cloud. In addition, the plots only show times for successful runs, but it is worth noting that with Push it sometimes took several attempts to complete the task without experiencing a failure in the cloud.

The aforementioned trade-offs are also apparent in these results, in particular with dataset Syn-SM. Neither Push nor Pull are able to improve beyond 64 partitions, and in fact both achieve significantly worse performance. However, the flexibility of the Hybrid method allows it to efficiently distribute data and computation, resulting in additional gains when going to 128 partitions.

Dataset size should also be taken into consideration when determining the appropriate configuration for a given problem. For smaller datasets, the choice of data distribution method is largely irrelevant, as all three lines exhibit very similar behavior. But for large problems the Push and especially Hybrid models are better suited as using the maximum number of available partitions achieves the best performance and therefore is advisable.

Support Vector Machines As shown in the right column of Figure 3, support vector machines exhibit behavior different from the other algorithms. Most notably, the majority of experiments do not achieve the best execution time for the largest number of partitions. And with SVMs this is not only due to heterogeneity in the computation cloud, but also to the strong dependency of the algorithm runtime on the characteristics of the data.

Once again, the data distribution method is less of a factor than the amount of parallelism in determining the execution time, although the pull method is consistently the worst performer. In our experiments, we also observe a tendency towards a smaller number of partitions than the other algorithms. More specifically, the best performance was achieved with 8 to 16 partitions in all configurations.

6.2 Execution Time: Multicore

The same constructs that apply to running on clusters or clouds also apply to a multicore environment on a single machine. Figure 4 shows the runtime of the abstraction applied over varying numbers of partitions in a dual-core environment. When streaming using fifos, the partitioner and the classifiers run in parallel. For files, the partitioner runs first, placing the files, which are then accessed by the classifiers after all partitions are created.

Streaming using files results in marginally faster turnaround times. A machine with more cores would clearly allow for greater scalability up to the limit at which the abstraction is bound by data rather than by computation. As expected, once beyond the number of cores, efficiency decreases, as each classifier is fighting for limited resources. Beyond 16 concurrent classifiers, progress slows significantly and the turnaround time is much longer than the serial execution. However, processors with a large number of cores are on the horizon, and future work should evaluate the Classify abstraction in such environments.

6.3 Accuracy

It is generally established that ensemble learning can result in improved accuracy [4]. Our fundamental goal in this paper is to work with that assumption and evaluate the system aspects of distributed data mining. For the experiments we consider primarily synthetic datasets, and therefore observe only modest improvements.

Figure 5 shows the trends for each classifier on all applicable datasets. We see that, in most cases, accuracy is quite stable with an increasing number of partitions. Notable exceptions are increased accuracy for decision trees on the Alpha and Syn-SM datasets, and decreases for decision trees on the Beta dataset as well as k-nearest neighbors on the Syn-SM dataset with 8 partitions.

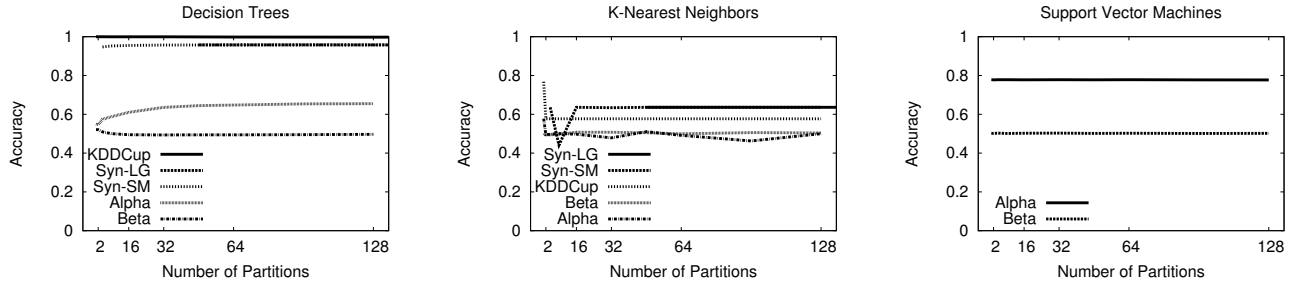


Figure 5. Trends in Accuracy with a Varying Number of Partitions.

	Cluster	Cloud
Pull	<ul style="list-style-type: none"> - chop is necessary for large number of partitions - for large clusters, submitting node can become a bottleneck as the data server - worst turnaround time in most experiments 	<ul style="list-style-type: none"> - chop is necessary for large number of partitions - for large clusters, submitting node can become a bottleneck as the data server - less concern about heterogeneity (fast nodes run bigger share), reliability (data not on remote nodes)
Hybrid	<ul style="list-style-type: none"> - shuffle is preferred partitioning method (can randomize, overlap, etc.) - less risk of bottleneck in large clusters where submitting node has limited resources - sweet spot trading off parallelism for robustness 	<ul style="list-style-type: none"> - shuffle is preferred partitioning method (can randomize, overlap, etc.) - not reliant on central file server during execution - best choice for turnaround for most configurations (mitigates disadvantages of the other two methods)
Push	<ul style="list-style-type: none"> - good for small runs with limited parallelism available - shuffle is preferred partitioning method (can randomize, overlap, etc.) - good for algorithms with super-linear complexity - brittleness less concern in controlled environment 	<ul style="list-style-type: none"> - trade-off between partitioning robustness (chop) and performance (shuffle) - trade-off between parallelism and reliability (more available resources but less reliable “in the wild”)

Table 1. Analysis of Trade-Offs Between Different Criteria Based on Empirical Observations

7 Conclusion

We started the paper with three fundamental questions regarding distributed data mining from a cluster to a cloud. To that end, we proposed a scalable and efficient abstraction, called *Classify*, to knit together sequential programs into data parallel graphs, allowing for a seamless deployment on clusters or clouds or multi-processor machines. We evaluated three different and popular learning algorithms with varying degrees of complexity on datasets with varying sizes up to 54 GB. Table 1 summarizes the key results. We reposition them with respect to our three questions below.

1. How to partition the data into subsets for learning?

The Hybrid method is most appropriate for completing runs of the abstraction in flexible environments, as it exhibited the most benefits for both Cluster and Cloud. It is more amenable to shuffle mode than the other methods, because it allows the performance advantage of remote partitioning and mitigates the lack of robustness of the shuffle algorithm.

2. What are the limits of scalability?

We observe that fundamental limits of scalability are, as one would expect, available memory on each commodity workstation and convergence properties of the algorithm. We observed that for some datasets, SVMs failed to converge in reasonable time, even for much smaller samples. For the largest dataset, we were unable to learn on partitions that were less than $1/16^{th}$ of the original data, largely due to memory issues.

3. How to exploit the available resources?

We observe that using commodity machines, as in a cloud, via the proposed abstraction framework results in efficient utilization of available resources, most of which would otherwise remain idle. The abstraction also allows efficient use of multicore machines by employing parallel data mining. As a result of generating ensembles accuracy also improves, drawing another key highlight of exploiting available resources: being able to learn on the entire dataset in a reasonable time while providing improvements in accuracy.

Acknowledgments We want to thank Phil Snowberger for his contributions in the early stages of this research. This work was supported in part by National Science Foundation Grants CNS-06-43229, CCF-06-21434, and CNS-07-20813.

References

- [1] R. Agrawal, T. Imielinski, and A. Swami. Database Mining: A Performance Perspective. *IEEE Trans. Knowl. Data Eng.*, 5(6):914–925, 1993.
- [2] G. Buehrer, S. Parthasarathy, S. Tatikonda, T. Kurc, and J. Saltz. Toward terabyte pattern mining. In *Proceedings of ACM SIGPLAN PPOPP*, pages 2–12, 2007.
- [3] M. Cannataro, A. Conguista, A. Pugliese, D. Talia, and P. Trunfio. Distributed data mining on grids: services, tools, and applications. *IEEE Systems, Man, and Cybernetics, Part B*, 34(6):2451–2465, 2004.
- [4] N. V. Chawla, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer. Learning Ensembles from Bites: A Scalable and Accurate Approach. *Journal of Machine Learning*, 5:421–451, 2004.
- [5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2007.
- [6] S. Cong, J. Han, J. Hoefflinger, and D. Pdua. A sampling-based framework for parallel data mining. In *Proceedings of ACM SIGPLAN PPOPP*, pages 255–265, 2005.
- [7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large cluster. In *Operating Systems Design and Implementation*, 2004.
- [8] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForest – A Framework for Fast Decision Tree Construction of Large Datasets. *Dat. Min. Knowl. Disc.*, 4(2-3):127–162, 2000.
- [9] C. Giannella, H. Dutta, K. Borne, R. Wolff, and H. Kargupta. Distributed data mining for astronomy catalogs. In *SDM Workshop on Scientific Data Mining*, 2006.
- [10] L. Glimcher, R. Jin, and G. Agrawal. FREERIDE-G: Supporting applications that mine remote data repositories. In *ICPP*, pages 109–118, 2006.
- [11] L. Glimcher, R. Jin, and G. Agrawal. Middleware for data mining applications on clusters and grids. *Journal of Parallel and Distributed Computing*, 68(1):37–53, 2008.
- [12] B. Gu, F. Hu, and H. Liu. *Instance Selection and Construction for Data Mining*, chapter Sampling: Knowing Whole from its Part. Kluwer, 2001.
- [13] Hadoop. <http://hadoop.apache.org/>, 2007.
- [14] J. Hofer and P. Brezany. Digidt: Distributed classifier construction on the grid data mining framework gridminer-core. In *ICDM Workshop on Data Mining and the Grid*, 2004.
- [15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data parallel programs from sequential building blocks. In *Proceedings of EuroSys*, March 2007.
- [16] G. Ji and X. Ling. *Emerging Technologies in Knowledge Discovery and Data Mining*, chapter Ensemble Learning Based Distributed Clustering, pages 312–321. LNCS, Springer, 2007.
- [17] R. Jin, G. Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):71–89, 2005.
- [18] T. Joachims. Training linear svms in linear time. In *KDD*, pages 217–226, 2006.
- [19] H. Kargupta, K. Bhaduri, and K. Liu. The distributed data mining bibliography. <http://www.cs.umbc.edu/~hillol/DDMBIB/>, 2006.
- [20] A. Kumar, M. Kantardzic, and S. Madden. Distributed data mining: Framework and implementations. *IEEE Internet Computing*, 10(4):15–17, 2006.
- [21] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. In *5th Int'l Conference on Extending Database Technology*, pages 18–32, 1996.
- [22] C. Moretti, J. Bulosan, D. Thain, and P. Flynn. Allpairs: An abstraction for data-intensive cloud computing. In *IEEE/ACM International Parallel and Distributed Processing Symposium*, April 2008.
- [23] M. S. Perez, A. Sanchez, V. Robles, P. Herrero, and J. M. Pena. Design and implementation of a data mining grid-aware architecture. *Future Generation Computer Systems*, 23:42–47, 2007.
- [24] X. Qiu, G. C. Fox, H. Yuan, and S.-H. Bae. Parallel data mining on multicore systems. Technical report, Indiana University, May 2008.
- [25] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman Publishers, 1993.
- [26] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Symposium on High-Performance Computer Architecture (HPCA)*, 2007.
- [27] D. Talia, P. Trunfio, and O. Verta. The weka4ws framework for distributed data mining in service-oriented grids. *Concurr. Comp.-Pract. E.*, 2008.
- [28] D. Thain, C. Moretti, and J. Hemmes. Chirp: A practical global file system for cluster and grid computing. *Journal of Grid Computing*, to appear in 2008.
- [29] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurr. Comp.-Pract. E.*, 17(2-4):323–356, 2005.
- [30] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.
- [31] M. J. Zaki, C.-T. Ho, and R. Agrawal. Parallel Classification for Data Mining on Shared-Memory Multiprocessors. In *15th Int'l Conference on Data Engineering*, pages 198–205, 1999.