FlexGP: a Scalable System for Factored Learning in the Cloud

by

Owen C. Derby

Submitted to the Department of Electrical Engineering
and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2013

Author .....
Department of Electrical Engineering
and Computer Science
May 23, 2013

Certified by .....
Kalyan Veeramachaneni
Research Scientist
Thesis Supervisor

Certified by .....
Una-May O'Reilly
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

FlexGP: a Scalable System for Factored Learning in the Cloud

by

Owen C. Derby

Submitted to the Department of Electrical Engineering
and Computer Science
on May 23, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This work presents FlexGP, a new system designed for scalable machine learning in the cloud. FlexGP presents a learner-agnostic, data-parallel approach to cloud-based distributed learning using existing single-machine algorithms, without any dependence on distributed file systems or shared memory between instances. We design and implement asynchronous and decentralized launch and peer discovery protocols to start and configure a distributed network of learners. Through a unique process of factoring the data and parameters across the learners, FlexGP ensures this network consists of heterogeneous learners producing diverse models. These models are then filtered and fused to produce a meta-model for prediction.

Using a thoughtfully designed test framework, FlexGP is run on a real-world regression problem from a large database. The results demonstrate the reliability and robustness of the system, even when learning from very little training data and multiple factorings, and demonstrate FlexGP as a vital tool to effectively leverage the cloud for machine learning tasks.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Research Scientist

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist

Acknowledgments

First and foremost, I would like to thank Una-May O'Reilly for her guidance and feedback over the course of my research and for enabling me to make this project my own. Likewise, the support and supervision I received from Kalyan Veeramachaneni was second to none. His insights and dedication were critical to formulating much of this work. Finally, I am grateful to Dylan Sherry and Alexander Waldin, my partners-in-crime in 32-D433, along with the rest of the ALFA group, for their support and suggestions this past year.

My friends and peers have been tremendously supportive and a constant source of inspiration. These past 5 years have been truly amazing and went by too quickly.

And last but definitely not least, I am forever grateful to my parents. Words simply are not enough to express my gratitude. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The past decade has seen tremendous growth in both the power and availability of computational resources. New technologies have emerged around these resources, bringing transformation to all parts of daily life. The cloud, a resource of seemingly infinite computational power, available on-demand and at your fingertips, stands out in particular as a new technology of importance for both professionals and academics alike. The growth and maturation of the cloud as a platform for computation has enabled machine learning at a scale never before seen. Simultaneously, we have seen an explosion of large, rich datasets; largely as a result of the increased availability of computers and the decreased cost of storage. Gone are the days when machine learning researchers were restricted to using small, hand-crafted datasets to train their models. Instead, the challenge now is constructing systems and algorithms for efficient learning in the cloud with these large datasets.

Developing systems for machine learning in the cloud presents several challenges. Taking advantage of the many compute resources available requires developing a system in a distributed computing environment. In addition to ensuring the algorithm is correct and performs well, concerns over shared memory, task distribution and coordination, and consistency arise in this environment. Addressing these challenges requires new approaches and often major refactoring of existing algorithms to fully and efficiently take advantage of the promise of more computational resources.

Oftentimes, the motivation for using clouds and other distributed computing platforms for learning is to deal with large datasets. Having a lot of data presents two major problems. First, it can be almost impossible, or so difficult as to be impractical, to learn on an entire dataset using just a single machine or simple cluster of machines. The data may simply be too big to fit into the available RAM. Second, the time taken to learn from a dataset is at least proportional to the size of the dataset. With such large datasets, learning time increases drastically, oftentimes so much that the learning may never finish. Further, existing methods for working with overly large datasets in the available RAM of a small cluster often introduces significant delays.

Amidst an era of "cloud computing" and "large-scale machine learning," many systems and platforms have emerged for distributing a given algorithm efficiently. Indeed, many common algorithms have been implemented on top of systems like GraphLab [11] or Mahout [13]. GraphLab, Mahout and similar distributed computing frameworks make strong assumptions about the organization of the algorithm or the structure of the datasets. Such assumptions enable these systems to run efficiently on the cloud. Unfortunately, different algorithms decompose in different, unique ways, often requiring extensive work and tweaking before they can run under these assumptions. When trying to use your algorithm with these systems, you are left either trying to fit a square peg (your algorithm + data) into a round hole (their system) or hacking your own solution. Neither option is very satisfactory.

This work presents FlexGP, a third option and potential solution for many researchers facing this problem. FlexGP[1] is an alternative approach for running machine learning in the cloud. Rather than decomposing and distributing a single learning algorithm across the cloud, FlexGP presents a learner-agnostic, data-parallel approach to cloud-based distributed learning, without any dependence on distributed file systems or shared memory between instances. With FlexGP, you can simply run your existing algorithm on hundreds of instances in the cloud with a large dataset and receive a fused model of significant improved quality.

[1]The "GP" in FlexGP stands for genetic programming. Even though the system is constructed to handle arbitrary learning algorithms, it currently runs genetic programming as the sample learner.

FlexGP addresses machine learning in the cloud by mirroring the full learner across the cloud and providing each instance with a subset of training data and algorithm parameters. In FlexGP, there is no single controller coordinating the system and the entire network is setup with peer-to-peer gossip-based protocols. Using simple fusion techniques, FlexGP produces a meta-model which enables predictions on withheld data. For historical and illustrative purposes, this work applies FlexGP to a real-world regression problem while using genetic programming (GP) as the learning algorithm.

We define and present a rigorous test framework for experimenting with FlexGP. Using this framework and our regression problem, we run a series of studies to analyze FlexGP, observing its performance and measuring the variance. Using these measurements, we infer the reliability of FlexGP and identify sources of variance in the performance. From these results, we show that FlexGP simultaneously provides improves performance while decreasing the variance when compared to a baseline single-machine learner.

The remainder of this work proceeds as follows. Chapter 2 discusses existing systems for distributed machine learning and work leading to the development of FlexGP. Chapter 3 gives an overview of the challenges FlexGP faces, its stated goals, and how the system is designed to meet those challenges and goals. Chapter 4 goes into the details of how the system is implemented, as well as briefly describing the GP library used. Chapter 5 discusses the regression problem, experimental setup and results from our studies. Finally, Chapter 6 concludes and discusses several ways in which FlexGP can be extended in the future.

Chapter 2

Related Work

The advent of today's cloud computing platforms has dramatically lowered the cost and increased the availability of large-scale computations. Previously, researchers would require large, expensive clusters to perform the same computation available now with a few clicks on a website. This has spurred researchers to revisit their algorithms, looking for ways to run them effectively in the cloud. Several systems for running machine learning on the cloud have been proposed as a result. While these systems each have their unique strengths and have seen wide-spread adoption, they only work well for particular types of algorithms and fail to provide a general purpose solution.

2.1 Systems for Distributed Machine Learning

MapReduce [2] is perhaps the most popular approach to running computation in parallel in the cloud. The MapReduce model requires users to specify a *map* function, which specifies how to decompose the input data into chunks of work for the many slaves to compute. After computation, the results are merged into a single set of answers using the user-defined *reduce*. Apache Mahout is an open source collection of machine learning algorithms which have been adapted to run within the MapReduce framework [13]. Systems implementing MapReduce, like Mahout, follow a master-slave model of computation where a special node is responsible for configuring and

coordinating an army of slaves. Further, MapReduce relies on a distributed file system for passing data between slaves and report results.

There are three challenges encountered when trying to complete machine learning tasks using the MapReduce model. First, the centralized architecture is prone to failure and can become a bottleneck in practice. A failed master, even in the presence of periodic snapshots, means hours of computation time lost. Further, with any centralized architecture, the scalability of MapReduce is limited in practice by the bandwidth and implementation of the master. Second, the reliance on a distributed file system proves difficult to support in practice and is not well-matched for running on the cloud. Configuring and maintaining a distributed file system is a non-trivial task, and something most researchers can't afford to bother with. Finally, MapReduce is a one-off approach, with no support for iterative processes, which is problematic for some machine learning algorithms. Each algorithm has several ways in which it can be broken down for parallelization in the cloud, but sometimes these sub-problems require an iterative approach, which is at odds with the MapReduce framework. Without support for iteration, it becomes a contortionist's act to run these learning algorithms on the cloud with MapReduce. Promising work has begun to add iteration to MapReduce [14], but it is still preliminary.

GraphLab [11] and Pregel [12] take a different approach to distributed machine learning. These systems adopt a *graph-parallel* view of the world. By this, they assume the problem can be broken down into a graph abstraction, where data resides on vertices and/or edges between vertices and each vertex executes a local program using its data and the data of its edges and its neighbors. These systems extend the Bulk Synchronous Parallel (BSP) [18] model for parallel computation with this graph abstraction. The BSP model supports message passing and distributed computation free of deadlocks with barrier synchronization. This model supports iterative algorithms, and as such is a better match for distributed machine learning.

Unfortunately, there are some severe practical limitations to GraphLab and similar systems. The graph abstraction does not work with all machine learning algorithms, in particular GP. In such cases it becomes just as hard to use as MapReduce. Further,

Pregel relies on a distributed file system, like Hadoop, and suffers from similar problems Mahout. While GraphLab can work without a distributed file system, it instead requires a shared or distributed memory system. Such systems are complex and expensive to set up, and have a strict limit on their scalability. To use shared memory, all the cores must reside in the same machine, restricting the size of a computation to the maximum number of cores available on a cloud machine, which is around 128 cores on most public cloud platforms. With distributed memory, the number of nodes isn't a restriction, but the overhead involved with maintaining memory consistency quickly dominates the algorithm's run time. Finally, despite using a graph abstraction, these systems still heavily rely on a centralized control structure for synchronization, and therefore suffer from the same drawbacks as Mahout in this respect.

2.2 Distributing Evolutionary Computation

There is a large body of existing research in distributed evolutionary computation [4, 25, 21, 22, 5] focused on modeling evolutionary dynamics and improving collaborative solution building by enabling communication between multiple evolving islands. Much of this work ignore the problem of matching the distributed system to a particular resource type or communication layer. Thus, much of this work is only tangentially related to FlexGP, as we are focused on writing a platform which takes advantage of the cloud platform. Further, these systems rely on MapReduce for parallelization and therefore suffer from the problems outlined in Section 2.1.

FlexGP's IP discovery is like other EC peer-to-peer systems. For example, the EvAg system [7, 10] also relies upon gossiping for node discovery. Little information is available on its startup method. It is not specialized to run on particular resource types whereas it is designed to investigate topology and a fine grained distribution model. EvAg and FlexGP differ in how they introduce evolutionary diversity: EvAg employs different operators across randomized neighbourhood whereas FlexGP factors each island with differentiation of data, GP objective function, operator set and input variables.

Eureqa [15] is a cloud-based system for evolving natural laws of complex systems with GP for symbolic regression. Because the system has since been commercialized, it is unclear what paradigm Eureqa uses for running in the cloud and whether it relies on a distributed file system, shared memory, neither or both. However, Eureqa does not support data sub-sampling or different learners, making it difficult to operate with large datasets.

2.3 Previous Work Leading to FlexGP

Historically, FlexGP was preceded by FlexGP-ECJ [16], a pilot exercise to explore adapting an existing system to the cloud. FlexGP-ECJ attempted to retrofit an existing grid-based evolutionary computation framework, called ECJ [15], to run in the cloud. Several difficulties were encountered. A grid-based system assumes all resources and their IPs are available at its startup so it starts with centralized migration topology initialization. In FlexGP-ECJ all islands start running genetic programming (GP) only once the topology is completely communicated to the entire network. When resource acquisition was later prefixed to the code, a bottleneck arose. The centralized start node had to wait (sometimes quite a while) until the final acquired instance sent in its IP before it established the topology and told each island to start GP.

The original intent of the exercise was to try and simply run ECJ in the cloud instead of on a cluster. However, several unique features of how ECJ was implemented made the process of porting it over extremely difficult. As with other existing distributed evolutionary computation models, ECJ was not designed for machine learning tasks. It was designed primarily to simulate the evolutionary dynamics of multiple independently evolving populations trying to maximize a fitness function. Because of this, most of the flexibility in the system focused on setting up different configurations of populations and evolutionary strategies, not system configuration. In the end, the work reduced to rewriting much of the underlying system, such that it became clear the work would be better achieved by starting from scratch. Thus, FlexGP was born.

Chapter 3

FlexGP Overview

FlexGP as a system is designed to run in the cloud. Through a carefully considered design, it takes full advantage of the advantages offered by the cloud platform while accommodating its failures and drawbacks. It is designed with an understanding that the cloud supplies sufficient computational resources upon request, yet expects those resources might fail or be delivered with unknown latency. It is conceived as a long-running computational learner, evolutionarily adapting and continuously improving its model whilst allowing for drastic changes in supplied cloud resources and network topology. It is realized as a collaboration of many heterogeneous FlexGP instances, independently learning a model and observing the topology of the network.

3.1 Running in the Cloud

The term "cloud" has become ubiquitous. Before proceeding with an overview of the design of FlexGP, it is necessary to refine what we mean by the term and its connection with FlexGP.

A cloud is a platform providing practically infinite computational resources on-demand. These resources are partitioned into virtualized instances, running on top of commodity hardware, typically running a full-fledged operating system. Because they are virtualized, instances can be created with various numbers of CPUs and amounts hard drive and RAM available. The cloud provides an API for starting, managing,

and stopping instances. Users can also change and configure the operating system, or "image," instances start up with, allowing for easy customization of what software instances have installed when they start.

With this platform construction comes various challenges and cautions to consider:

Distributed Environment Launched instances normally run independent of each other – it is left to the user to setup a network and coordinate work between them. Managing and executing a system in such a distributed landscape is very different from managing a local machine or cluster of machines.

Variable Startup Time Because instances are virtualized and clouds handle requests from many users at once, launching a new instance is a complicated process and is opaque to the user. From the user's perspective, requests for cloud instances are subject to arbitrary delays and failures. This means that it may take a while before a requested instance starts, if it even starts at all. See Appendix A for a more comprehensive discussion of this point.

Random Failures Most clouds are built using commodity hardware, which makes maintenance, repair and upgrade easy and cost effective. However, it also means hardware failure is a common occurrence and instances can crash without warning.

Virtual Hardware Unlike a physical CPU, the processing power of a virtual CPU is not very well defined and tends to fluctuate. Anything requiring precise timing or exact processor specifications will have trouble.

Price Fluctuation Pricing schemes for running on clouds tend to be complicated. Generally, the price rate (cost per unit time) for running an instance depends on the type of instance selected and the time at which it is run. Minimizing this cost is important.

3.2 Goals

The task for building FlexGP was undertaken with several goals in mind. These goals were conceived with respect to the presented cloud challenges and how the system will be used.

Graceful Scaling Running with 10 nodes should be as easy as running with 1000. When new instances start up, they should be quickly and effortlessly incorporated into the network.

Zero-Delay Computing Despite running on the cloud, the system should feel responsive. The user shouldn't have to wait for the last node to start for computation to begin. The current best results should be available at any time.

Fault Tolerant As with any distributed system on the cloud, failures will occur. FlexGP needs to tolerate such failures without compromising the system or computational results.

Robust Learning The user ought to be able to trust the results from a single run of FlexGP. Despite the stochastic nature of the system, the system ought to produce reliable results, with low variance across trials.

Heterogeneous Learning There is little value to be gained by running the same computation on 100 cloud instances. Instead, every instance should be computing something different, contributing in a valuable way to the end result.

Elastic Resource Allocation The cloud enables seamless launching of new instances. FlexGP ought to provide this functionality as well, enabling users to seamlessly add to or remove new instances from the computation as needed.

3.3 Design

At its core, FlexGP constructs a simple workflow for machine learning. The input to the workflow is a desired learner library L, a dataset D to learn on, and a distribution

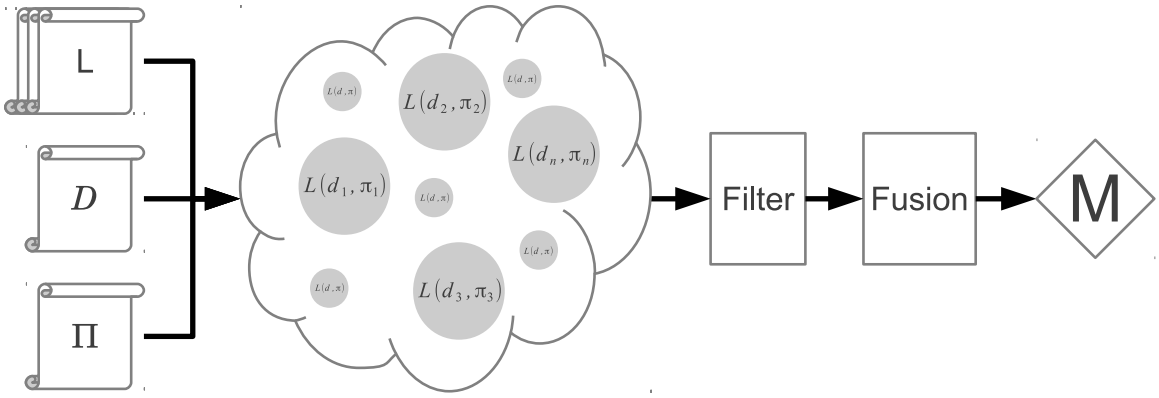


Figure 3-1: Overview of workflow in FlexGP. Given a learner L, a dataset D and parameter distributions Π, FlexGP launches n cloud instances to run L with data subset d_i and parameters π_i. The resulting models are collected, filtered and fused to produce a single model M.

of parameter settings Π. These inputs are fed to cloud instances in parallel, with each instance running the learner on a randomly chosen subset of the dataset and randomly selected parameters from the distribution. Using the data, the learners perform the typical machine learning task of producing and refining models. The objective for each learner is to produce models which can predict the output for unseen data. The models can be collected from every learner to form an ensemble of learners. This ensemble goes through a filtering stage, to remove models which perform poorly or are very similar to other models. A fusion technique is then applied to the ensemble, producing a single meta-model which can be used to give individual prediction values for data input. This workflow is depicted in Figure 3-1.

What follows is a discussion of the central design concepts behind FlexGP, building off of the goals from Section 3.2.

3.3.1 Computational Dichotomy

The computational components of FlexGP are split between two systems. One component runs locally, on the user's machine. The other runs in the cloud, across hundreds of instances. The user of FlexGP provides a learner library, required configuration information, the dataset to use, and his/her cloud credentials. From the user's machine, FlexGP initiates the cloud component, consistent with the provided

configuration, with the learner library and dataset. The cloud component then expands to the desired capacity (number of nodes) and learning begins. At any point, the user can begin construction of the ensemble of learners. This entails the local component initiating a process for collecting and downloading the models from all instances in the cloud component. Then filtering and fusion can be run locally.

It is important to remember that the cloud component is a collection of autonomous, independent instances in the cloud. There is no controller or master directing or controlling their collective action.

3.3.2 Learners

In FlexGP, a learner is the conceptual entity responsible for learning a model given some data. FlexGP operates on learners, while the cloud platform provides instances, a single unit of computational resource. To run efficiently on the cloud, FlexGP adopts a simple one-to-one mapping of learners to instances. This choice of mapping provides several benefits. First, it means the granularity of learning matches the granularity of compute resources, so FlexGP never has to deal with partial failures. If an instance fails, the learner fails completely as well. Second, it simplifies the considerations of how many instances to launch vs. how large to make each instance. The computational requirements of the learner dictate instance size, while dataset size and user requirements specify how many instances to launch. Finally, it makes system elasticity simple to implement and reason about. Shrinking the computation by n instances will simply remove n learners.

Once started, learners are completely autonomous units of learning, operating independently from each other, and communicating asynchronously. They fail independently[1] and can be queried independently of each other. Further, each learner is constructed (and potentially configured) identically. That is, every learner runs on the same instance type with the same user-provided library, has access to the same dataset and parameter space, and has the same capability to spawn new learners.

[1]This is approximately true. The frequency of the underlying hardware failing in a cascading or other dependent manner is low.

The learners form a true network of peers. This is not to say they are all performing the same work. Indeed, each learner may randomly partition the data and choose different parameters to learn with, as explained in Section 3.3.4.

3.3.3 Asynchronous Message Passing Network

While having independent and autonomous learners provides a nice abstraction to build upon, it may still be necessary to enable inter-learner communication. Learners may want to collaborate or compare models periodically or simply inform each other of progress. For example, a popular strategy for distributed genetic programming is the "island model," wherein each learner is an island evolving a population in isolation. Occasionally, individuals migrate between islands, establishing a loose collaboration network between the islands, leading to improved global results. With an established network, the GP island model could easily be built on top of FlexGP. A communication network is also necessary for any sort of reporting or monitoring to be implemented.

Most existing systems achieve networking with a centralized architecture, as discussed in Section 2.1. While this allows for the creation of arbitrary network topologies by the master, the nodes cannot begin computing until they receive parameters and IP lists from the master. However, on a cloud the master cannot know the IP addresses until all the instances have started. Because the latency for acquiring hundreds of instances in the cloud can be arbitrarily large[2], such an architecture does not scale well [16].

FlexGP is an entirely peer to peer (P2P) system, where the network is established organically and peers (learners) communicate asynchronously via message passing. This is a good match for the cloud, because instances can discover peers as they start up and gradually construct the network.

[2]Some of the requested nodes might even fail before reporting to the master, complicating matters further.

3.3.4 Factored Learning

To enable learning from very large datasets, where the data is too large for a single learner to feasibly learn from, several or many learners need to be used to learn anything in a reasonable amount of time. These learners can look at subsets of the data, and their resulting models can be combined.

As described so far, FlexGP provides a way for learning with many learners and fusing their results. To get each learner in FlexGP to use a different subset of the data, each learner must locally generate a subset of the dataset before learning starts. This sub-sampling step is performed randomly, according to a user-provided distribution and with a user-provided resampling method[3].

Because it is a stochastic process, this step does not guarantee that every learner runs with non-overlapping subsets of the data. However, when viewed collectively across the entire FlexGP network of learners, we expect to see the distribution of subsets to converge to the user-provided sampling distribution. And this will only improve as the number of learners increases.

FlexGP abstracts this process of learner-local sampling according to a user-defined distribution in a process labeled "factoring." Factoring is used to achieve varied learners and can be applied to any of the inputs to the learners. If the dataset is viewed as a matrix, where rows correspond to different samples and columns correspond to the features of each sample, the sub-sampling process above can be viewed as selecting a subset of the rows. However, a subset of the columns could also be selected. These form two axis along which the learners can be factored. Combining the two produces random partitions of the dataset. Further, the setting of various algorithm parameters of the learner can be factored.

It is this abstraction of factoring, paired with a distributed launch protocol and peer discovery algorithm, that makes up the heart of FlexGP and makes it so well adapted to running on the cloud. The independent learner assumption enables it to shamelessly start up in the cloud and remain tolerant of failures (when running

[3]Depending on the resampling method, in particular if samples are drawn with or without replacement, this process can be seen as generating many bootstraps or jackknifes of the data[26].

hundreds of nodes, losing a few doesn't matter), while factoring ensure every learner is providing a unique contribution towards the overall solution, and contributes to a greatly diverse ensemble for fusion.

3.3.5 Filtering and Fusing

To produce a final model with FlexGP, the models produced by each learner need to be collected, filtered and fused. The collection step is achieved by simply retrieving the current model, training data and parameter settings from every instance in parallel[4]. This forms an ensemble of models, which can then be filtered. The filtering step is user specified. At a minimum, it should remove any models which perform poorly on validation data (data which was not selected for training at each instance). More sophisticated filtering, such as removing duplicates or highly-correlated models could also be done.

After the filtering step, the remaining models in the ensemble are fused. This fusion process is also user-specified, and takes an ensemble of models and returns a single model which can produce a predicted output value given some data. Because we treat the learning algorithm as a black box, this fusion is performed using the model predictions (and associated errors), and is expressly not a fusion of model parameters[5] A more extensive discussion of how fusing can be performed and what is produced can be found in Section 4.7.

[4]This retrieval step does not interfere with the learners' progress. Therefore we can perform collection, filtering and fusion again later on if need be.

[5]Nevertheless, model parameter fusion could be performed if an appropriate model was used instead of GP.

Chapter 4

Implementation

This chapter focuses on the details of how FlexGP is implemented. It is broken down into several sections, according to the different points along the FlexGP workflow. Section 4.1 provides a description of the learning algorithm used in this work. Sections 4.3, 4.4 and 4.5 detail how the system is launched and configured on the cloud. Finally, Sections 4.6 and 4.7 discuss how the results are collected and analyzed to produce a final model. A diagram summarizing the entire FlexGP system is provided for reference in Figure 4-1. Each section explains a particular part of this diagram.

4.1 Learning with Genetic Programming

FlexGP is a platform for machine learning on the cloud. As such, it assumes nothing about the learning algorithm used. However, for this work, FlexGP was run with Genetic Programming (GP) as the learner as a concrete example. Although it is not important to understand all the details of how the GP learner was implemented, there are a few points worth noting.

In this case, GP was used in a symbolic regression task. Therefore, we can consider GP to be an evolutionary algorithm which optimizes a set of executable expressions encoding nonlinear functions mapping the data samples in the training dataset to their associated output variables. These expressions are considered "individuals" in

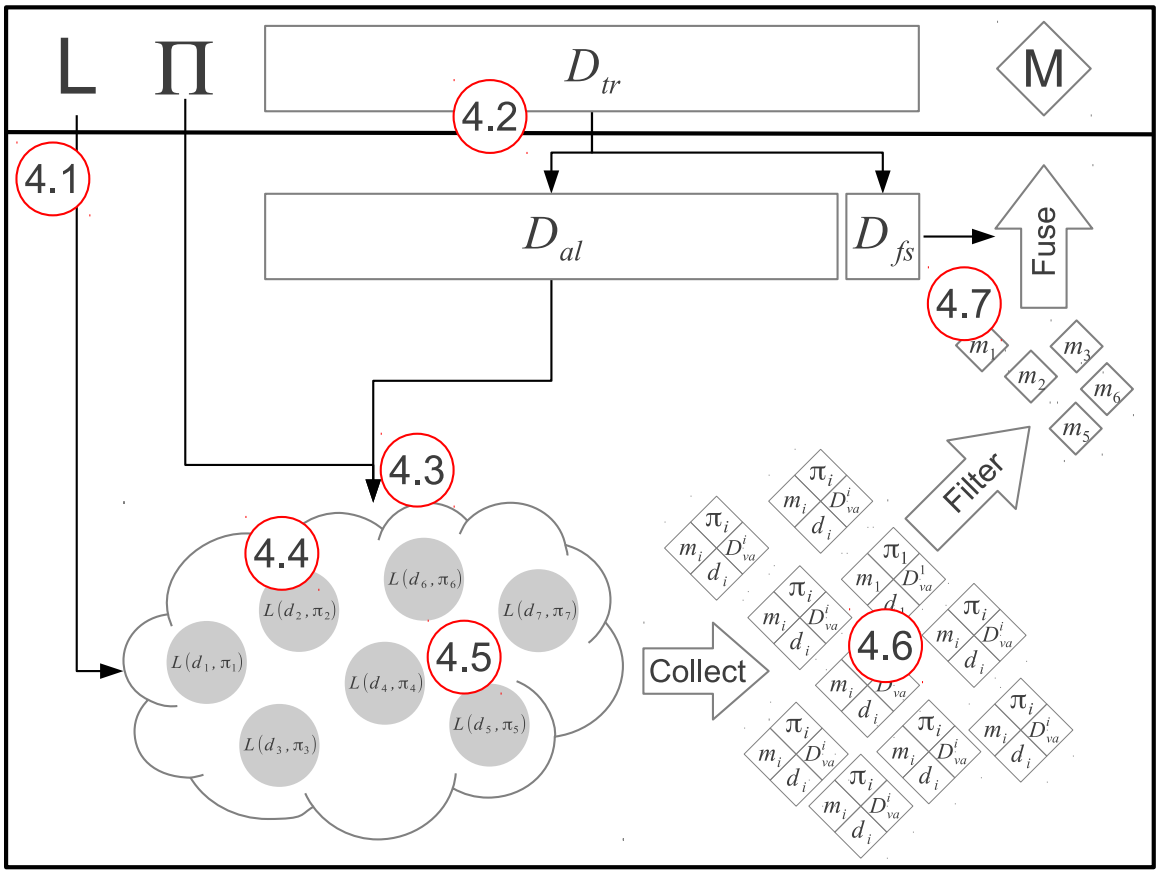


Figure 4-1: Detailed overview of how the FlexGP system runs. The top bar represents the system inputs (learner L, settings Π and data D_{tr}) and output (model M).

a "population"[1]. These individuals are represented as trees, where the terminals are features of the data samples and non-terminals are simple functions (arithmetic, trigonometric, power, etc). A sample individual is given in Figure 4-2.

GP progresses in an iterative fashion, moving through "generations" of the population, in much the same way a population of animals progresses via evolutionary adaptation from one generation to the next: individuals in the previous generation recombine and random mutations occur, giving rise to the next generation of the population. This process continues, until one of the individuals satisfies some goodness of fit cutoff or the learning is halted.

When running GP, there are many parameters to be set. The number of genera-

[1]The terminology of GP and the larger family of evolutionary algorithms borrow heavily from the field of evolutionary biology.

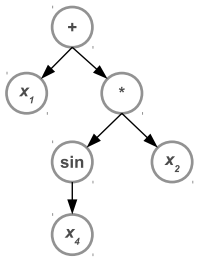


Figure 4-2: A sample individual in genetic programming.

tions to run for; the number of individuals in the population; the maximum size of any individual tree; how individuals are recombined; etc. For most of these parameters, there are commonly recommended values to use that seem to work well, regardless of the dataset being used. However, the objective function to use when measuring the goodness of fit and the set of functions to select non-terminals from often do depend on the dataset used. Often, these parameters are set arbitrarily or after a brief, superficial explorations of their possible settings. However, with FlexGP, we can explore the effect of setting these parameters to different values, as described in Section 4.4. A full description of the remaining parameters and the settings selected for them, as well as more information about the implemented GP learner, can be found in Appendix B.

4.2 Splitting the Data

The user provides the FlexGP system with the training dataset D_{tr}. Immediately, D_{tr} is split into two separate subsets: D_{al} for passing to the FlexGP instances in the cloud for training and D_{fs} for training the fusion model. The i^{th} FlexGP instance may[2] further split D_{al} into two more datasets: d_i for training a model with and D^i_{va} for validation of the produced model. Note that if all samples from D_{al} are to be used for training the model, d_i is just D_{al} and D^i_{va} is empty. This process is summarized in the top part of Figure 4-1.

[2]As described in Section 4.4, if data factoring is not turned on, this step is not performed.

Algorithm 1 NodeStart(n, R)

n: nodes to launch, R: list of ancestor IP addresses
Ψ: launch parameters, Π: FlexGP meta-parameters
$ip \leftarrow$ last(R)
retrieve(ip, Ψ, Π)
$R \leftarrow$ cat(R, MyIP())
$n \leftarrow n - 1$
if $n \leq \Psi.k$ and $n \geq 1$ **then**
  for $i = 1$ to n **do**
    $c_i \leftarrow$ BootNode(1, R)
else
  for $i = 1$ to $\Psi.k$ **do**
    $k \leftarrow \lfloor \frac{n}{\Psi.k-i+1} \rfloor$
    $c_i \leftarrow$ BootNode(k, R)
    $n \leftarrow n - k$
IPDiscovery(R)
GPMLCompute()

4.3 Parallel Asynchronous Startup

Given the goals of scaling gracefully and zero-delay learning from Section 3.2 and the charge from 3.3.3 for a peer-to-peer system, FlexGP implements a decentralized, peer-to-peer (P2P) startup algorithm. Every FlexGP instance is capable of launching other FlexGP instances. Immediately after booting, every FlexGP instance retrieves parameters from the node which started it. The parameters $\Psi.k$ and $\Psi.p$ indicate the number of nodes to start and the target IP list size (see Sect. 4.5), respectively. The FlexGP meta-parameters, Π, are used to determine the parameterization of each FlexGP learner (see Sect. 4.4). These steps are detailed in the NodeStart function in Algorithm 1.

Figure 4-3a illustrates how FlexGP would launch the 7 instances in Figure 4-1 when $\Psi.k = 2$. Node A is launched and runs NodeStart(7, []), where [] indicates an empty list. A then boots nodes B and X, each of which will run NodeStart(3, $[IP_A]$), and will go on to boot 2 more nodes each. Figure 4-3b details the timeline of two nodes during startup, illustrating the concurrency present in the FlexGP startup. As soon as node A finishes executing NodeStart and started nodes B and X, it starts a new thread to begin running GP computation and then continues into the IPDiscovery algorithm, as described in Section 4.5. This enables us to run GP

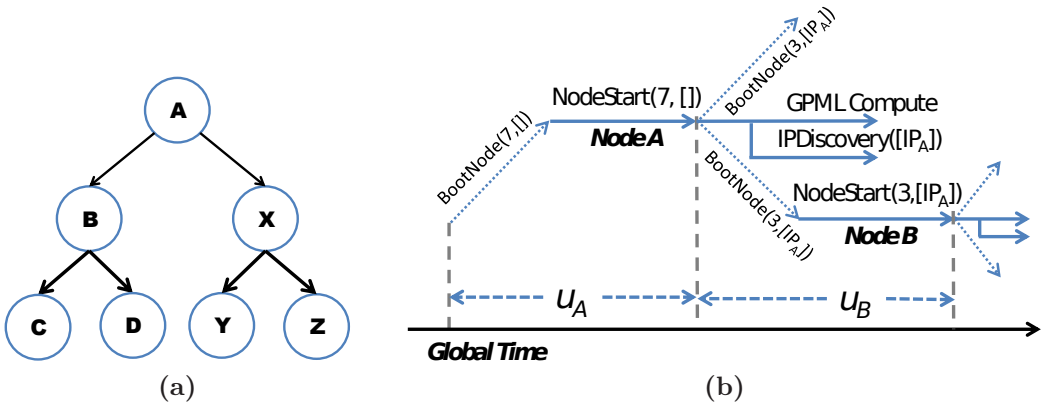


Figure 4-3: A view of the launch of FlexGP for the 7 nodes in Figure 4-1. Left: An initial node is launched and it brings up 2 more, which in turn bring up 2 more each, in a cascading fashion. Right: Timeline of booting and launching of instances. After starting more nodes, node A begins computation. The quantities u_A and u_B are described in Appendix B.

concurrent with IP discovery and network discovery.

To illustrate the benefits of this design decision, we look at how much progress each FlexGP instance makes. Of particular interest is how much progress is made by all instances when the last instance starts, which is when most centralized architectures would begin computing. Figure 4-4 presents two different views of this question. The first, presented in Figure 4-4a, measures total progress as the number of individuals evaluated across all instances as time passes. The second, presented in Figure 4-4b, examines the distribution of completed generations across all instances when the last instance starts (around the 1800th second).

The cumulative effect of this is that by the time the last instance starts, some instances have completed as many as 30 generations and some 2.2 million individuals have been evaluated across the FlexGP system. Since each evaluation requires a pass through the training data points, this corresponds to at least 2.2 million passes through the problem dataset[3].

[3]Note that our instance is a single core machine, but for more complex problems we could use a instance with 8 cores and run GP via multithreading allowing us to finish ~ 8 times as many fitness evaluations. Additionally if the instances include GPUs the number of fitness evaluations would be much higher.

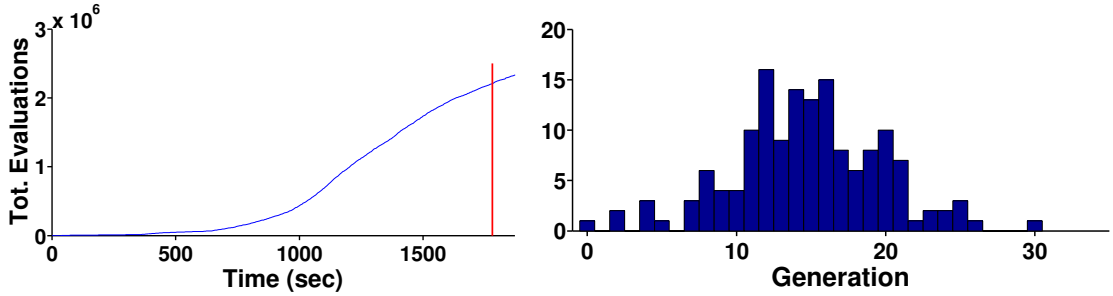


Figure 4-4: Progress of GP on each node during system launch. Left: Cumulative fitness evaluations completed by all FlexGP instances. Right: Histogram of number of generations completed when the last instance starts (marked as red line on left figure).

4.3.1 Tolerating Failures

Another goal from Section 3.2 is for FlexGP to be resilient to failures of cloud instances. With this asynchronous startup protocol, fault tolerance is achieved. The failure of one node interrupts the acquisition of further instances by that node, but does not hinder launches by other running nodes. For example, in Figure 4-3a, if node X failed to launch properly, nodes Y and Z will never be requested, but there is no affect on the acquisition of nodes B, C or D. In general, while the actual number of acquired nodes may not meet the requested N, GP (and IP discovery) can execute on all nodes that have been acquired. We have taken the view that N will usually be large enough and failure will be sufficiently infrequent that we do not need to be concerned about any reporting, tracking and explicit recovery of node failures.

There may still be cases where the launch did not acquire a sufficient proportion of N instances. This may occur in the unlikely event that a node crashes very early on in the launch or in the face of intermittent cloud service interruptions. If such a scenario arises, we can simply tap an existing node and have it run the startup with new parameters which will try to populate the network with more resources. This same strategy can also be used to increase the number of running instances after startup. We might want to do this at night, when cloud instances become cheaper to run.

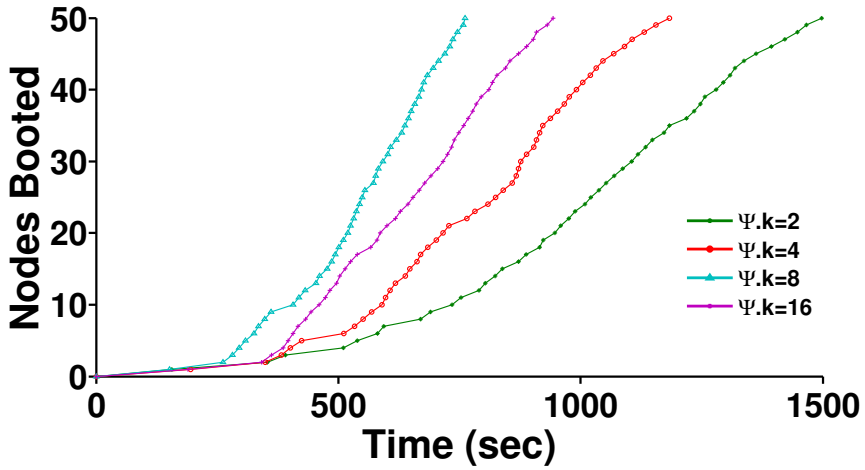


Figure 4-5: Time to acquire 50 instances at different values of $\Psi.k$. Values reported are averages taken over 30 trials at each value.

4.3.2 Selecting $\Psi.k$

An important parameter of the launch process is $\Psi.k$, which controls how many new instances are started by each instance. Figure 4-3a illustrates a launch with $\Psi.k = 2$.

In Fig. 4-5 we compare how long it takes to start up 50 nodes for $\Psi.k \in [2, 4, 8, 16]$. As expected, the time decreases as $\Psi.k$ increases, until $\Psi.k = 8$. Then the time gets worse for a value of 16. This is likely due to the wider variation in latencies for larger batch request sizes, as discussed in Appendix A. Note that the specific tradeoff point at $\Psi.k = 8$ is largely dependent on the properties of our cloud, how cloud instances are scheduled to be launched and the load it is under at the time of measurement. Therefore, we expect this point would change over time or if measured on a different cloud system.

4.4 Factored Learners

FlexGP generates a large set of diverse models for ensemble learning. This is achieved by varying the data partition and parameters each GP learner starts with. This leads to a set of factored learners, working in parallel to learn a diverse set of models.

FlexGP factors up to 4 different parameters at each FlexGP instance i: The training data samples (d_i), the training data features (F), the non-terminal functions

Parameter	Value	Definition	Default
Operator Set (L)	W X Y Z	$\{+, -, /, *\}$ $\{exp, ln\}$ $\{sqrt, x^2, x^3, x^4\}$ $\{sin, cos\}$	$W \cup X \cup Y \cup Z$
Objective Function (O)	Norm-1 Norm-2 Norm-3 Norm-4 Norm-inf	Mean absolute error Mean squared error Mean |cubed| error Mean error[4] Max error	Norm-2
Training Cases (d)	n	Subset of D_{al}, of size n	user-defined
Feature Set (F)	m	Subset of features, of size m	all features

Table 4.1: Data and GP parameters and their definitions, possible values and default values.

of GP trees (L) and the objective function used by GP (O). Factoring each of these parameters consists of selecting a particular setting for each parameter by drawing from a user-defined distribution over possible parameter settings. Note that while for some parameters like objective function, this setting is a particular value[4], for others it may be a set of values[5]. The different options available for these parameters are summarized in Table 4.1. If a particular factoring is not enabled in a run, then the default setting is used instead.

We take Π, the set of meta-parameters retrieved from the parent (see Sect. 4.3), to define the distributions over these options, guiding how FlexGP selects the values for each parameter. We will use the notation $p(O)$ to represent the distribution over the options for O. $p(L)$ gives probabilities for each of the 8 possible values of L.[6] $p(O)$ gives probabilities for each of the five $Norm$s defined. L and O are each chosen as a single sample from $p(L)$ and $p(O)$, respectively. $p(D_{al})$ defines probabilities of selecting each training case (from D_{al}, the set of all training cases). d is constructed by sampling without replacement n times from $p(D_{al})$. The distribution for F is split into two parts. $p_1(F)$ gives probabilities for the number m of features to use. $p_2(F)$

[4]Actually the specific objective function to use.

[5]i.e. a set of non-terminal functions to use while building GP trees.

[6]These values being W, $W \cup X$, $W \cup Y$, $W \cup Z$, $W \cup X \cup Y$, $W \cup X \cup Z$, $W \cup Y \cup Z$, and $W \cup X \cup Y \cup Z$.

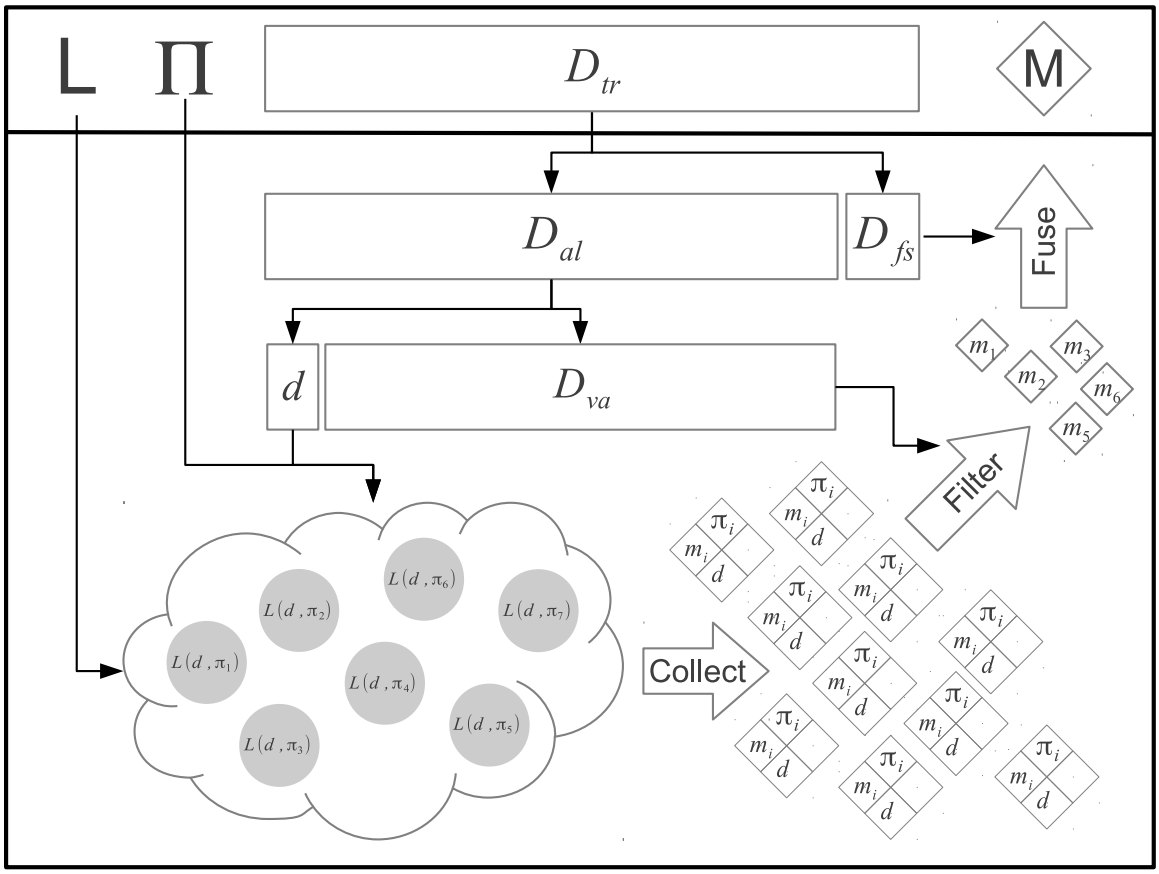


Figure 4-6: Detailed summary of how FlexGP runs, when data factoring is disabled.

defines the probabilities of using each feature. F is constructed by sampling m from $p_1(F)$ and then drawing m samples from $p_2(F)$ without replacement. Then Π is the set $\{p(L), p(O), p(D_{al}), p_1(F), p_2(F)\}$ and the settings of parameters sampled from Π by instance i is π_i. For example, π_i could be $\{W, Norm-2, d_i, \{x_1, x_2\}\}$ for instance i, while another instance j might have $\pi_j = \{\{W \cup X\}, Norm-4, d_j, \{x_1, x_4\}\}$.

Note that in the special case when the training data is not being factored, the process outlined in Figure 4-1 can be simplified to Figure 4-6. The main difference here is that since every instance will learn with the same training set, D_{al} is split into d and D_{va} before the cloud is launched and only d is passed to each instance and D_{va} is passed directly to the filter step.

Algorithm 2 IPDiscovery(R)

```
Λ ← R
loop
  λ ← set of new messages received
  for m in λ do
    if m.type is RequestIPList then
      Λ ← merge(Λ, m.Λ)
      RespondIPList(m.ip, Λ)
    else if m.type is RespondIPList then
      Λ ← merge(Λ, m.Λ)
  if len(Λ)< Ψ.p then
    ϵ ← random(Λ)
    RequestIPList(ϵ)
```

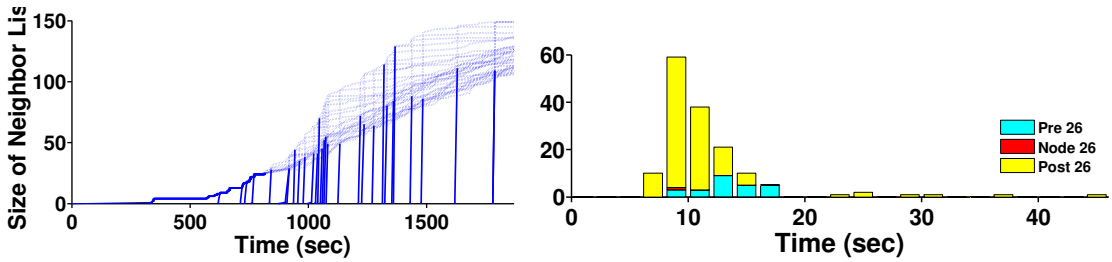
4.5 Peer Discovery

As discussed in Section 3.3.3, a key requirement of any cloud-based ML application is the support for communications between learners. Further, cloud-scale systems need an established network to robustly extract information and results. To address these requirements, FlexGP has a distributed IP discovery protocol. Note, the focus here is on the initial bootstrapping of the network - the "IP discovery" problem. This is separate from the problem of creating particular topologies in P2P networks [6].

Recall that as part of startup a parent node shares its IP list with all its children. A node at level i therefore has i IP addresses at startup. We then use a *gossip* protocol to populate the neighbor list at each node. First, we set a lower limit, $\Psi.p$, for the number of IP addresses a node needs to acquire. It generally is a function of the total number of nodes. We then follow an address passing protocol per Algorithm 2. In this protocol's active phase, each node selfishly tries to increase its IP addresses up to its limit by requesting more IP addresses from its neighbors while it shares with its neighbors its IP addresses in exchange. After it meets or exceeds the limit, in its passive phase, it serves any request it receives in exchange for their IP addresses.

Although our application does not require networking between instances, it is still important to examine the dynamics of the network constructed, especially how quickly instances connect to others after starting. To explore the network, we ran

FlexGP to launch 150 instances, with $\Psi.p = 25$. Recall $\Psi.p$ controls how many other instances each instance actively seeks to establish a link with before entering the passive phase of discovery. Thus, $\Psi.p$ controls the connectivity of the FlexGP network. The plots in Fig. 4-7 show global time progressing along the x-axis. In Fig. 4-7a, the y-axis denotes the number of other instances each instance has connected with. Each trace represents an instance and, for clarity, we only show a subset of the instances. Observe that all instances eventually acquire IP addresses of over half the network. Each trace changes from solid to dashed when that instance switches from the *active* to the *passive* phase of discovery (discussed in Section 4.5. Interestingly, the later instances to start discover a very large number of instances very shortly after launch. In fact, the last instance discovers nearly 150 instances almost immediately. This ensures connectivity if more instances are added later.



(a) (b)

Figure 4-7: IP discovery through gossip. Left: Progress of IP discovery as a function of global system time. Each line represents the number of IP addresses a node accumulates as time progresses. Right: Time it took for each node to acquire $\Psi.p$ IP addresses.

Figure 4-7b shows the distribution of delays for nodes to enter the passive phase of IP discovery. This delay for the i^{th} node is calculated as follows. Let S_i be the time at which the i^{th} node started and let T_i be the time at which the i^{th} node discovered its 25^{th} IP address. Then the latency for node i is given as $T_i - max(S_i, S_{26})$. We notice that almost 130 nodes take less than 25 seconds to enter the passive phase.

4.6 Gathering and Filtering Models

Given that the FlexGP instances in the cloud have been running for a while and have produced results, it is then necessary to retrieve the results to the user's local machine for fusion. Part of retrieving the models from the instances is recording what subsets of the data each instance used for training. This retrieval activity can be performed in parallel and at any time, satisfying the zero-delay computing goal of Section 3.2.

Once the models and training data descriptions have been retrieved, the ensemble needs to be cleaned up and possibly sub-selected. This process is called filtering. On average, during a typical 2 hour run of FlexGP, each instances will generate between 15 and 30 models, due to the iterative nature of the GP learner. This gives approximately 2000 models. A primary purpose of filtering is to remove models which overfit the training data, by trying them with the validation subset ($D^i_{va} = D_{al} \setminus d_i$), removing any which produce invalid results, or are duplicated models (since we collect the best model from every generation of each GP learner, it's possible to get the same best model for several generations). Further filtering can be done to sub-select for more diverse models or simply fewer models.

FlexGP provides 3 simple filter methods for now. The simplest method simply removes invalid models, and otherwise tries to include as many models as possible in the ensemble. For simplicity, this method is referred to as "all." The remaining two build off of "all" and present simple variations on ways of sub-selecting the ensemble to contain fewer, better models. The first, "per_node," selects the single best model produced by each instance. The second, "best_n," selects the best n results, for some predetermined value of n, based again on the performance of the models on D^i_{va}. Once the ensemble has been filtered, it is ready for fusion.

Variable	Notation	Definition
Data	D_{al}	GP training data
	d_i	Subset of D_{al} used by instance i for training
	D^i_{va}	Subset of D_{al} used by instance i for validation
	D_{fs}	Fusion training data
	D_{te}	Testing data
Data sample	$\overline{\mathbf{x}}_j$	$\overline{\mathbf{x}}_j = \{x_l \mid l = 1...\gamma\}$
Output variable	z_j	$z_j \in \mathbb{R}$, for $\overline{\mathbf{x}}_j$
Model	m	Model m
Prediction	$\hat{y}_{mj} = f_m(\overline{\mathbf{x}}_j)$	Model m's prediction, non-linear in $\overline{\mathbf{x}}_j$
Candidate models	Ω	Set of models for fusion
Predictions for $\overline{\mathbf{x}}_j$	$\overline{\mathbf{y}}_j$	$\overline{\mathbf{y}}_j = \{\hat{y}_{mj} \mid \forall m \in \Omega\}$
Predictions of model m, given D	$\overline{Y}^m_D$	$\overline{Y}^m_D = \{\hat{y}_{mj} \mid \forall \overline{\mathbf{x}}_j \in D\}$
Output estimate	$\hat{z}_j$	ensemble's estimate of z_j

Table 4.2: Problem Notation

4.7 Combining Models

4.7.1 Fusion Methods for GP

Since GP produces non-parametric models, we must rely on predictions for analyzing the model performance constructing meta-model. Each GP learner produces an output $\hat{y}_{mj} = f_m(\overline{\mathbf{x}}_j)$ for each input data sample $\overline{\mathbf{x}}_j$, where m and other notational conventions are defined in Table 4.2. An example of a GP model m is

$$\hat{f}_m(\overline{\mathbf{x}}) = log(x_1) + x_4 x_5 + \frac{x_6}{e^{x_9}}$$

We propose and define three methods for combining ensembles of GP models for regression problems. These methods all produce predictions by fusing the predictions from a mixture of models in the ensemble. They vary in how they mix the models and combine those models' predictions.

Average Ensemble Prediction (AVE)

The simplest fusion method is to generate $\overline{\mathbf{y}}_j$ for a new query $\overline{\mathbf{x}}_j$ and then report

the average of those predictions. That is, $\hat{z}_j = \frac{1}{m}\sum_{i=1}^{m}\hat{y}_{ij}$ for each test point $\overline{\mathbf{x}}_j$[8, 20].

Median Average Model (MAD)

A variant of AVE, MAD finds the *median* prediction among $\overline{\mathbf{y}}_j$ and a prediction for a new query is computed as the average of the prediction of this median model and those of its two neighbors in the prediction space [23].

Adaptive Regression Mixing (ARM)

In Adaptive Regression by Mixing (ARM), each model m is assigned a weight, W_m, which is used to compute $\hat{z}_j$ via a weighted average [27]. The fusion process consists of learning the weight for each model. Let $r = |D_{fs}|$, the size of the fusion training set, and $o = |\Omega|$, the number of models in the ensemble. Here, we assume that the errors for each model are normally distributed. We then use the variance in these errors to identify the weights by executing the following steps:

Step 1: Split D_{fs} randomly into two equally sized subsets $D^{(1)}$ and $D^{(2)}$.

Step 2: Evaluate σ_m^2 which is the maximum likelihood estimate of the variance of the errors, $\overline{e}_m = \{\hat{y}_{mj} - z_j | \overline{\mathbf{x}}_j, z_j \in D^{(1)}\}$. Compute the sum of squared errors on $D^{(2)}$, $\beta_m = \sum_{j=\frac{r}{2}+1}^{r}(\hat{y}_{mj} - z_j)^2$.

Step 3: Estimate the weights using:

$$W_m = \frac{(\sigma_m)^{-r/2} exp(-\sigma_m^{-2}\beta_m/2)}{\sum_{j=1}^{o}(\sigma_j)^{-r/2} exp(-\sigma_j^{-2}\beta_j/2)} \tag{4.1}$$

Step 4: Repeat steps 1-3 for a fixed number of times[7]. Average the weights from each iteration to get the final weights for the models.

Given a test sample $\overline{\mathbf{x}}_j$, predict $\hat{z}_j$ as the weighted average of model predictions: $\hat{z}_j = \sum_{m=1}^{o} W_m \hat{y}_{mj}$.

Transformation for large r: For large values of r, the calculation of the weights as given by equation 4.1 encounters an underflow error. To avoid this problem we equivalently compute the weights using equations 4.2 and 4.3.

[7] We used a fixed number of 100. However, a more intelligent stopping criteria could be used instead.

$$A_m = -\frac{r}{2} log(\sigma_m) + log(\frac{-{\sigma_m}^{-2}\beta_m}{2}) \tag{4.2}$$

$$W_m = exp(A_m - log(\sum_{q=1}^{o} A_q)) \tag{4.3}$$

4.7.2 Advantages and limitations

Each of the fusion approaches we presented above has certain advantages and limitations. AVE is the simplest of all but could bias the estimation in the presence of many correlated models producing similar outputs for a training sample. It could also be affected by outliers. MAD, though robust to outliers, ignores a lot of the information embedded in our large ensembles. Both of these techniques do not differentiate between models based on their performance on the training data and consider the models themselves to be independent.

ARM presents a unique way of identifying the weights for each model however it can become computationally intensive. The approach is also sensitive to the amount and order of data presented for training the weights. Once the weights are identified, real time execution of the deployed model is extremely efficient.

ARM, AVE and MAD require an outlier detection algorithm to remove any outliers before fusion. The outlier detection algorithm has to be run in real time. In our approach we estimate the minimum and maximum values for the output variable $z = (z_{min}, z_{max})$ and any model producing predictions that are outside these bounds are removed before fusion. For ARM, the weights are renormalized after removing the weights that correspond to the outlier models.

4.7.3 Selecting Methods for Filtering and Fusion

In order to determine which of the methods to use for the remainder of the experiments, we performed a brief study of the results achieved by all 9 combinations of the 3 filtering strategies and 3 fusion strategies from Sections 4.6 and 4.7.1, respectively. Previous work with FlexGP [19] has already taken a comprehensive look at

this question, so we only perform a cursory study of it here. We perform this study on the models collected for the *FGP* experiments, repeating the 9 filtering/fusing pairs for all possible inputs. We construct a simple plot of the results[8] in Figure 4-8 to compare the performance achieved with each pair. From this, we can easily see that MAD and AVE achieve good results when the best nodes are selected (the "best" filter method), while ARM achieves similar results regardless of which filter method is used. In fact, it appears that the simplest filtering method is best with ARM. Because ARM assigns a weight for each model based during training, it is essentially performing its own filtering. Therefore, we select the minimalist filtering method, "all," to use with ARM in the rest of the analysis.

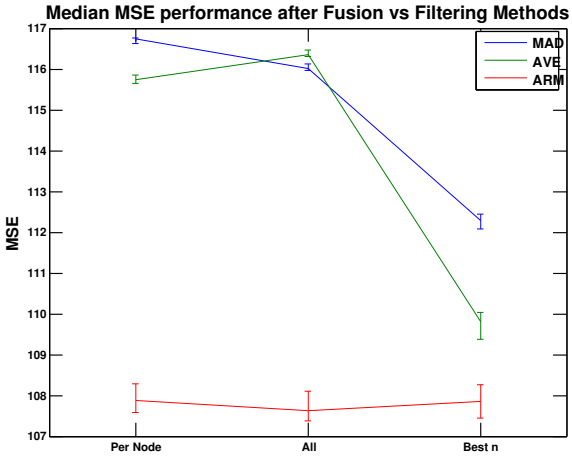


Figure 4-8: Comparison of system performance for different choices of filtering and fusion methods.

4.7.4 Producing a Meta-Model

After filtering and fusing the ensemble, we have produced the meta-model M which is modeled as $\hat{f}_M(\overline{\mathbf{x}}) = \sum_{m=1}^{o} W_m \hat{f}_m(\overline{\mathbf{x}})$. We can now use this model to produce predictions on D_{te} and assign a test MSE to M, which forms the result of a run. This completes the specification of the implementation of FlexGP system, and now we are ready to analyze it.

[8] Results reported in Table C.1.

Chapter 5

Evaluating FlexGP

Chapter 4 described the full implementation of FlexGP. We now begin a comprehensive examination of the overall performance of this implementation. To perform this in a rigorous manner, we outline a reusable framework for experimentation with FlexGP, which accounts for data and systemic sources of variance. Using this framework, we analyze results from large scale experiments on a real world dataset. The cloud component of the experiments took over 40,000 node-hours of compute time (the equivalent of running a single machine with just one CPU for 40,000 consecutive hours).

We begin with a description of the particular cloud platform we used, followed by a summary of the dataset selected for our experiments. We then present the our experimental framework and finally our experiments. Through these experiments, we see that FlexGP performs better and with less variance than stand-alone GP.

5.1 Cloud Infrastructure

Testing and development was completed using a private cloud. This cloud runs OpenStack[1], which is free and open source software for building both private and public clouds. Our cloud currently has 768 physical cores consisting of Intel Xeon 2.27GHz chips configured as 1536 virtual nodes with 3+ terabytes of available RAM. The

[1] http://www.openstack.org

OpenStack software provides cloud instance control (starting, stopping, and configuring individual instances) via the open-source *eucatools*[2] package, which is based on the API provided by the Amazon Elastic Compute Cloud[3] (EC2) cloud service.

Like with EC2, there are a variety of instance sizes available on the private cloud, with varying numbers of cores, RAM and hard drive capacity. For these experiments, the smallest instance size was chosen, to demonstrate the flexibility of the system and for ease of comparison across implementations. This instance has 2GB RAM, 30 GB of ephemeral storage and 1 virtualized CPU. This is very similar to the EC2 *m1.small* instance type, but with more RAM and significantly less storage. Note that there is nothing special about this instances size and FlexGP users could select larger instances if their problem required it and their budgets allowed.

Each instance runs Ubuntu 12.04 and has Java JRE installed. To integrate timestamps across nodes, we rely on Network Time Protocol (NTP), standard on Ubuntu 12.04, to provide accurate time synchronization. This is sufficient for our purposes, as FlexGP operates on the scale of many seconds to minutes, and is not affected by microsecond variations.

5.2 Dataset Description

For these experiments, we use the Million Song Dataset [1] (MSD for short) from the *music information retrieval* (MIR) community. This community has created a dataset of one million tracks (songs) and their associated meta-data, to push the focus towards solving large scale MIR problems. In particular, we use the *year prediction problem*[4] from this dataset, where the task is to predict the release year of a song, given a set of 90 track features. Of the million tracks in the dataset, 515,564 have valid year data.

Because of the 5-fold cross-validation (Section 5.3.1) in use, D_{te} is 20% of D (103,113 tracks). The remaining 80% constitutes D_{tr}. If running FlexGP, 12.5% of

[2] http://www.eucalyptus.com/download/euca2ools
[3] https://aws.amazon.com/ec2
[4] http://labrosa.ee.columbia.edu/millionsong/pages/tasks-demos#yearrecognition

D_{tr} (10% of D) is split off for D_{fs} (producing 51,556 tracks) and the remaining D_{al} (a total of 360,895 tracks) is further split (independently) by each node, as described in Section 4.2.

5.2.1 Splitting the Data

As discussed in Section 4.2 and later on in Section 5.3.1, there are several places where the data needs to be split into distinct subsets. With most typical datasets, generating splits is trivial as one can just randomly sample from the dataset n times to select a subset of size n. However, with MSD and the year prediction task, there is a complication. When splitting a set of tracks into two or more splits, one needs to be careful to avoid the "producer effect" [1]: since a single artist may author multiple tracks, having tracks from the same artist in different subset needs to be avoided. If not, it's possible to learn a model of the (hidden) artists instead of the tracks and achieve artificially better performance on the withheld subset with common artists.

To avoid the producer effect, splits need to occur between artists, not tracks[5]. However, since the number of tracks per artists is not constant, it is impossible to guarantee that a split will produce a subset with exactly n tracks. Therefore, when a split is stated as producing a subset of size n, it should be read instead as a subset of *target* size n.

5.3 Experimental Setup

5.3.1 System Configurations

FlexGP is a complex system with many parameters. Before examining the various experiments conducted and analyzing the results, Table 5.1 reviews those parameters which have been set to a definite value for the experiments, and points to where they are discussed. Note that some values were selected based on previous work.

[5]In particular, we sort the artists by the average year of all their tracks. Then we iterate over this sorted list with a window of size n, randomly picking one from each window to be in the subset. n is just the inverse of the target proportion of tracks to be in the subset.

Parameter	Notation	Description	Value	Reference
# instances	–	Number of instances to launch in the cloud	100	–[a]
Instance type	–	Type of instance to launch in the cloud	–	5.1
Branching factor	$\Psi.k$	Maximum number of instances launched by each instance	8	4.3.2, [3]
Peer list size	$\Psi.p$	Minimum number of peers to discover before an instance switches to passive phase	25	[3]
Collection method	–	Method for deciding what models from each instance are included in ensemble	The best model per generation	[19]
Filter method	–	Method for filtering collected ensemble	"all"	4.7.3
Fusion method	–	Method for fusing models from filtered ensemble	ARM	4.7, 4.7.3
Norms dist.	$p(O)$	Distribution to select O from	uniform	4.4
Function set dist.	$p(L)$	Distribution to select L from	uniform	4.4
Training cases dist.	$p(D_{al})$	Distribution to select d from	uniform	4.4
Data features dist.	$p_1(F)$	Distribution over how many features to use	–[b]	4.4
Data features dist.	$p_2(F)$	Distribution over feature indices	uniform	4.4

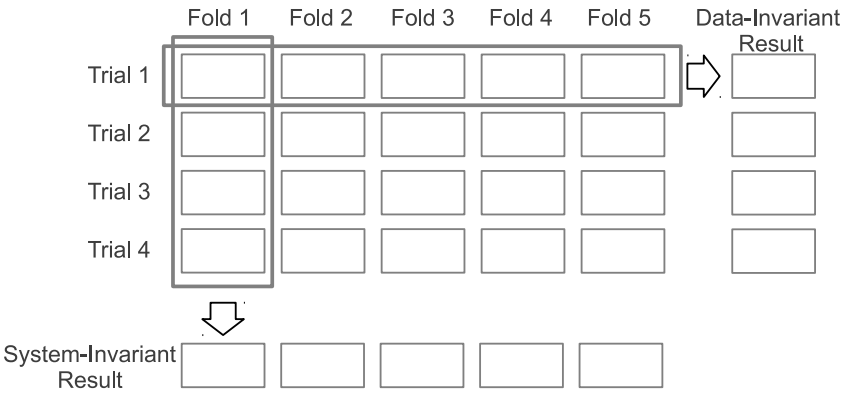
Table 5.1: Values used for various parameters.

[a] This is mostly arbitrary, but was also an artifact of being resource limited on our cloud.
[b] The probability mass function with equal mass at the values (5, 10, 20, 40, 60, 80, 90).

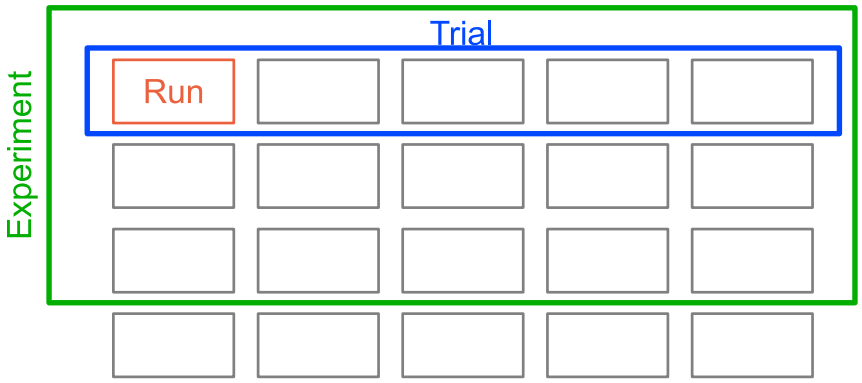
5.3.2 A Framework for Experimentation

FlexGP is a highly stochastic system - not only is the GP learner non-deterministic, but the factoring mechanism (Section 4.4) is also stochastic. This randomness is an important part of FlexGP, but can make it challenging to properly test and analyze the system. This section presents an overview of the process followed to run the experiments presented in Section 5.4. Each experiment specifies a specific Π_e to use.

One concern for these experiments is isolating the variance in system performance due to how the data is split. That is, for any given split of the dataset, numerous



(a) Matrix representation of results.



(b) Terminology of experiments.

Figure 5-1: Description of how results are presented.

possible subsets could result. Some of those subsets will be easier or harder to learn with, which can bias the results. To control for this, we run a 5-fold cross-validation over the entire FlexGP system[6]. To perform the cross-validation, D was split into 5 equal subsets[7] (D_{te}^1, D_{te}^2, D_{te}^3, D_{te}^4, D_{te}^5), which we refer to as **folds**.

The results are reported in a matrix, as depicted in Figure 5-1a. We refer to a single execution of FlexGP for a given fold and system settings Π_e as a **run**. The five runs of FlexGP with a given set of system settings over the different folds is a **trial**. Finally, the set of one or more trials with the same system settings forms an **experiment**. Figure 5-1b illustrates these distinctions in the context of our results

[6]It would be better to run cross-validation for $k > 5$, but it every trial requires k runs, and so it turns into a tradeoff with computation time.

[7]As discussed in Section 5.2.1, these will not be equally-sized subsets. However, the largest and smallest subsets differ in size by less than 0.1%, allowing us to treat them as equal for all intents and purposes.

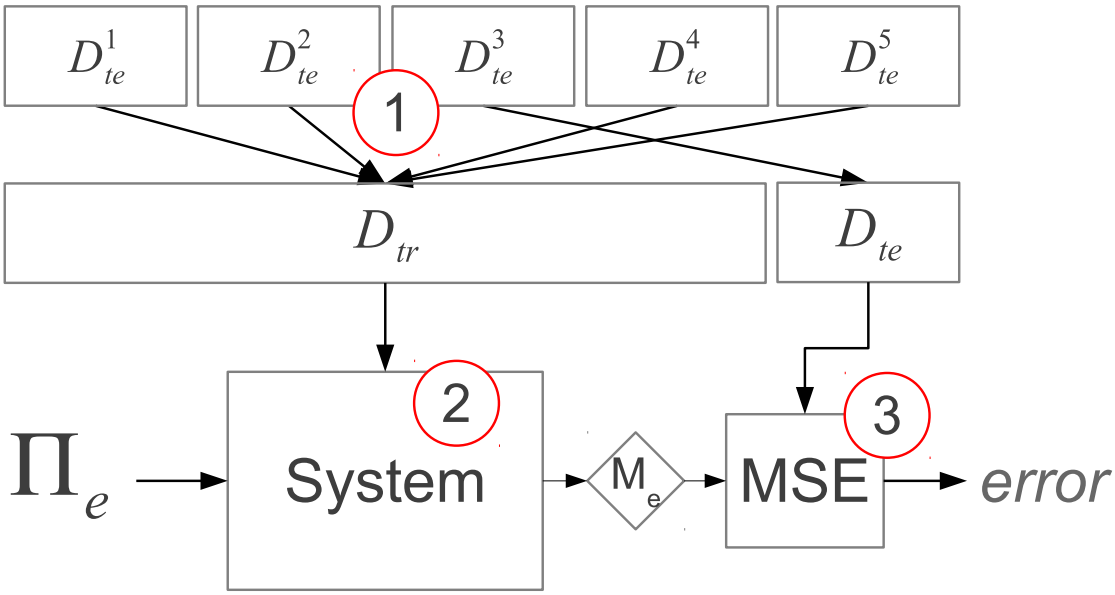


Figure 5-2: The flow of an experimental run. Here, fold 3 is being tested, with D^3_{te} being withheld for testing. Note that system is FlexGP for all but *SGP*.

matrix. The result of a trial is reported as the mean of the MSE from each of the five runs it is composed of. Because this result is averaged over the different folds of the data, it can be considered invariant to the effect of different data splits. Orthogonally, averaging the results of runs from the same experiment for a given split (averaging down a column instead of across a row) produces a result which is invariant to the randomness of the FlexGP process, as noted in the result rows of Figure 5-1a.

To execute FlexGP for fold j and trial i of experiment e, we proceed as follows:

Step 1 Construct $D_{tr} = D \setminus D^j_{te}$.

Step 2 Run FlexGP on D_{tr} with settings Π_e, as outlined in Figure 4-1, producing model M_e.

Step 3 Compute the MSE of M_e on D^j_{te}, recording the result.

Figure 5-2 illustrates this procedure, with the three steps highlighted in red. This procedure is repeated for every fold, for every trial, for every experiment.

Variance in FlexGP performance might arise when running with different datasets. Unfortunately, such a study is outside the scope of this work. However, the experimental framework outlined here could easily be repeated for different datasets in the

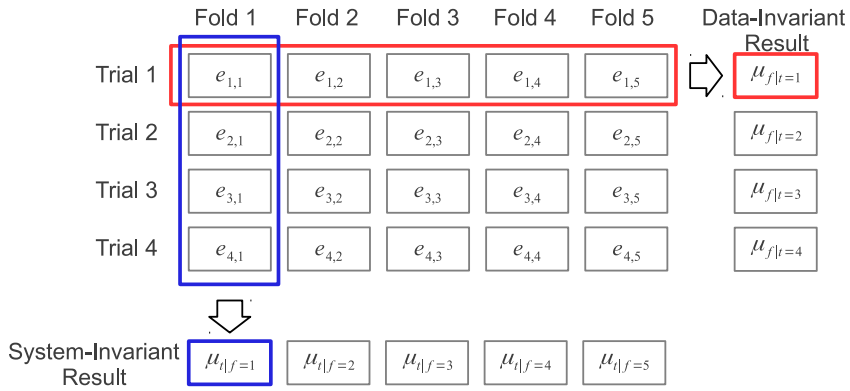


Figure 5-3: Notational convention for reporting results.

future. Then the results could be compared to those presented here to gain insight into how FlexGP performance varies with different datasets.

5.3.3 Establishing Notation

To aid in the following discussion, we define some additional notation by expanding the matrix representation of results from Section 5.3.1.

For a given experiment, the run of trial i on CV fold j produces an ensemble of models $\Omega_{i,j}$. After filtering and fusing this ensemble, we report the MSE of the fused model on D_{te}, the test dataset, as $e_{i,j}$. We then can compute summary statistics per trial or per fold (row or column, respectively). Let μ represent a mean and σ^2 a variance, then we define the mean MSE of trial i as $\mu_{f|t=i} = 1/5 \sum_{j=1}^{5} e_{i,j}$ and likewise for the variance $\sigma^2_{f|t=i}$. We can also compute the mean MSE for a given CV fold j, across the n trials of a given experiment, as $\mu_{t|f=j} = 1/n \sum_{i=1}^{n} e_{i,j}$ and for variance $\sigma^2_{t|f=j}$ as well. Figure 5-3 illustrates where these values can be found in reported tables.

Additionally, we occasionally need to refer to aggregate statistics of the models in the ensemble $\Omega_{i,j}$. Let $m_{i,j,k}$ denote the k^{th} model from ensemble $\Omega_{i,j}$, $r_{i,j,k}$ be the MSE of $m_{i,j,k}$ on the training dataset associated with it[8], and $t_{i,j,k}$ be the MSE of $m_{i,j,k}$ on D_{te}. Then we define the mean MSE on training data for ensemble $\Omega_{i,j}$ to be

[8]Recall that if data factoring is turned on, instance i is trained with a unique dataset d_i. Otherwise, all instances train with the same dataset d.

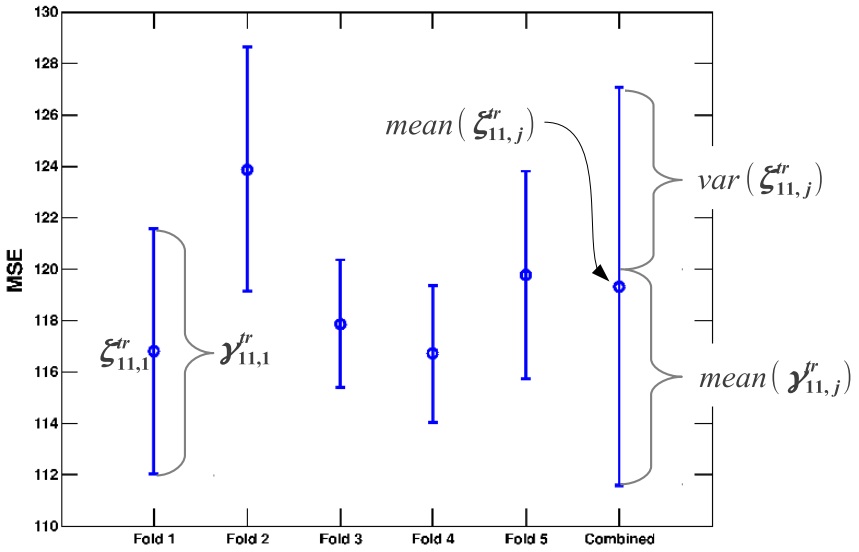


Figure 5-4: Illustration of notations for second order statistics of distribution of errors in ensembles for *FAC11*.

$\zeta^{tr}_{i,j} = \frac{1}{|\Omega_{i,j}|} \sum_{k=1}^{|\Omega_{i,j}|} r_{i,j,k}$. Similarly, let the variance of the MSE on training data for $\Omega_{i,j}$ be $\gamma^{tr}_{i,j} = \frac{1}{|\Omega_{i,j}|} \sum_{k=1}^{|\Omega_{i,j}|} (r_{i,j,k} - \zeta^{tr}_{i,j})^2$. Additionally, let $\zeta^{te}_{i,j}$ and $\gamma^{te}_{i,j}$ be the same statistics, but calculated from $t_{i,j,k}$ instead of $r_{i,j,k}$. Notice how we have one value for each of $\zeta^{tr}_{i,j}$, $\gamma^{tr}_{i,j}$, $\zeta^{te}_{i,j}$, and $\gamma^{te}_{i,j}$ per run. Finally, we can define several second order statistics, resulting from computing the *mean* and variance (*var*) of these values along a row or column. For example, $mean(\zeta^{tr}_{1,j}) = \frac{1}{5} \sum_{j=1}^{5} \zeta^{tr}_{1,j}$ would be the mean over all j (the CV folds) of the mean MSE on training data of each ensemble in the first trial of an experiment. Figure 5-4 illustrates this with the results from *FAC11*.

5.4 Analysis of FlexGP

Having built and validated a robust, decentralized cloud-based system for machine learning in Chapter 4, and defined a rigorous framework for experimenting, we turn our focus to the dynamics of the whole FlexGP system. As discussed in Section 5.2, we apply FlexGP to the year prediction task from the Million Song Dataset (MSD). To evaluate the performance of FlexGP and better understand the dynamics of the system, we perform three studies, summarized in Table 5.2, to examine four questions:

1. **How does FlexGP improve overall performance for a given problem?** (5.4.1)

2. **How reliable are FlexGP results?** (5.4.1)

3. **How do different factorings impact the overall performance of the system?** (5.4.2)

4. **How much training data is enough for FlexGP to perform well?** (5.4.3)

	Exp.	Factoring				$\frac{\lvert D_{tr}^i \rvert}{\lvert D_{gp} \rvert}$	# Samples	# Trials	Run Time (hours)
		F	D	L	O				
Study 1	*SGP*					–	412451	10	21
	FGP	✓	✓	✓	✓	10%	36090	10	2
Study 2	*FAC01*					10%	36090	1	2
	FAC02	✓				10%	36090	1	2
	FAC03		✓			10%	36090	1	2
	FAC04			✓		10%	36090	1	2
	FAC05				✓	10%	36090	1	2
	FAC06	✓	✓			10%	36090	1	2
	FAC07	✓		✓		10%	36090	1	2
	FAC08	✓			✓	10%	36090	1	2
	FAC09		✓	✓		10%	36090	1	2
	FAC10		✓		✓	10%	36090	1	2
	FAC11			✓	✓	10%	36090	1	2
	FAC12	✓	✓	✓		10%	36090	1	2
	FAC13	✓	✓		✓	10%	36090	1	2
	FAC14	✓		✓	✓	10%	36090	1	2
	FAC15		✓	✓	✓	10%	36090	1	2
	FAC16	✓	✓	✓	✓	10%	36090	1	2
Study 3	*TDS01*	✓	✓	✓	✓	0.1%	361	1	6
	TDS02	✓	✓	✓	✓	1%	3609	1	6
	TDS03	✓	✓	✓	✓	10%	36090	1	6
	TDS04	✓	✓	✓	✓	25%	90224	1	6
	TDS05	✓	✓	✓	✓	50%	180448	1	6
	TDS06	✓	✓	✓	✓	100%	360895	1	6

Table 5.2: Settings of the different experiments. Note that *FGP*, *FAC16*, and *TDS06* have the same settings. Instead of running the same exact thing multiple times, the first trial of *FGP* was used for *FAC16* and *TDS06*.

SGP		*FGP*	
$\mu_{f\|t=i}$	$\sigma_{f\|t=i}$	$\mu_{f\|t=i}$	$\sigma_{f\|t=i}$
113.790	3.768	107.674	3.196
112.075	3.492	107.322	1.774
113.447	6.173	108.342	2.615
114.280	4.267	108.012	3.048
114.828	4.668	107.570	2.386
109.635	5.369	107.214	2.459
115.912	5.144	108.344	2.543
111.608	3.842	107.558	3.043
110.533	3.167	106.200	2.867
116.354	2.074	108.138	1.692

Table 5.3: Summary of results from *SGP* and *FGP* experiments.

5.4.1 Study 1: Comparing FlexGP with GP

To study the benefits gained from using FlexGP, we run two experiments. In the first, denoted *SGP* (for **S**tandard **GP**), we run the stand-alone GP learner (Section 4.1) with the full D_{tr} subset for each fold to establish a baseline of performance on the MSD problem. The second, denoted *FGP* (for **F**lex**GP**), runs the full FlexGP system with all factorings enabled and each instance selecting 10% of D_{al} for each fold. Both experiments are run for 10 trials.

To answer the first question, we directly compare the results from *SGP* and *FGP*. The mean and standard deviation of these results are reported in Table 5.3[9]. Each trial of *SGP* was run for 21 hours[10], while each trial of *FGP* was only run for two hours. Figure 5-5 compares the distribution of *SGP* results (left) with the distribution of *FGP* results (right). The middle distribution of Figure 5-5 is computed from the errors (on D_{te}) of every model generated by each run from *FGP* which was used for fusion.

From Figure 5-5 we see two things. First, FlexGP significantly outperforms a single learner, even when each leaner in FlexGP ran for $\frac{1}{10}$ as long and with $\frac{1}{10}$ as much training data. Further, when comparing the distribution of individual model

[9]The raw results for *SGP* and *FGP* are reported in Table C.2.

[10]To allow each run to complete a comparable number of generations to what *FGP* learners completed.

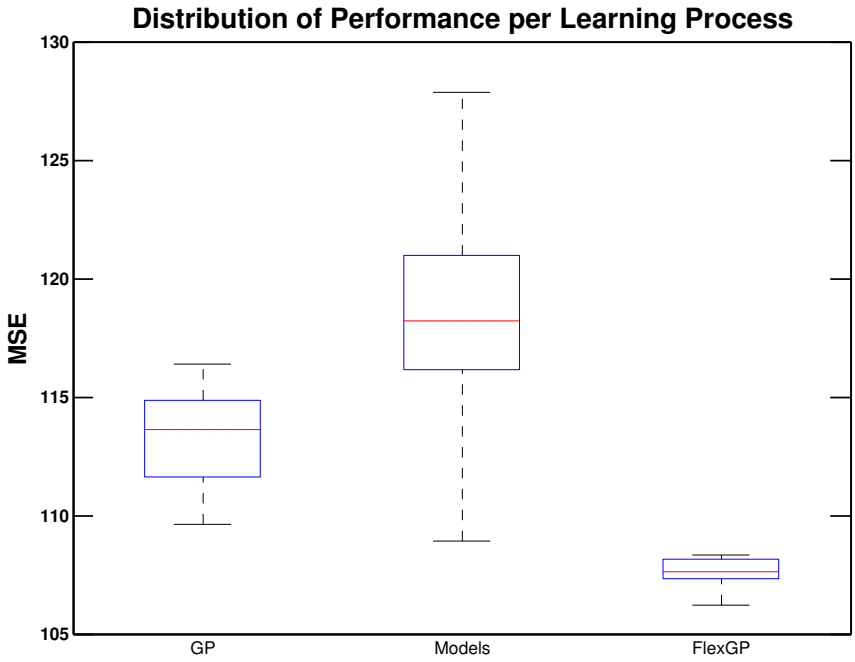
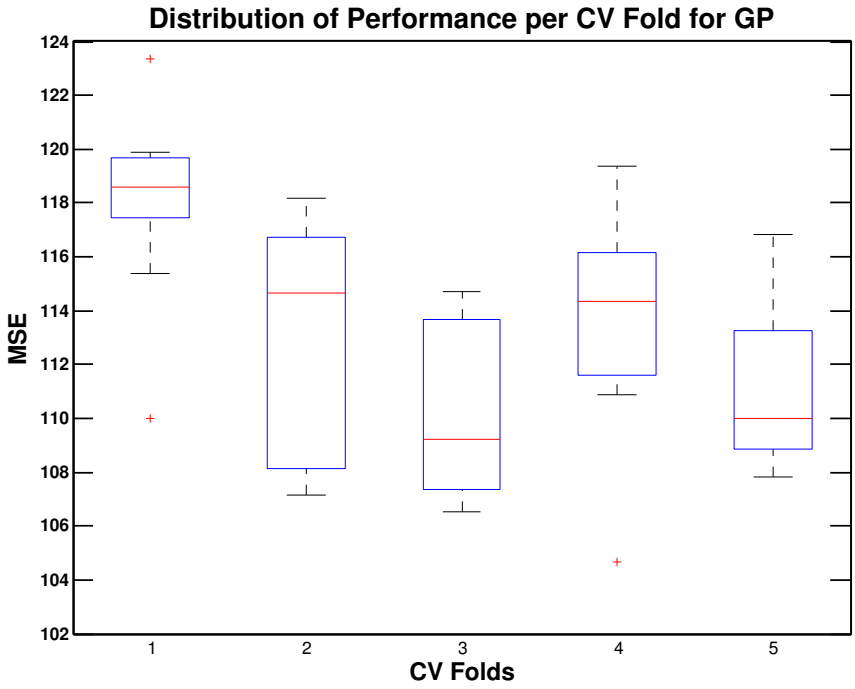


Figure 5-5: Boxplots of (left to right): $\mu_{f|t=i}$ for SGP, MSE of all ensemble models from every fold of every trial in FGP, and $\mu_{f|t=i}$ for FGP. The whiskers extend to 1.5 IQR and outliers are omitted.

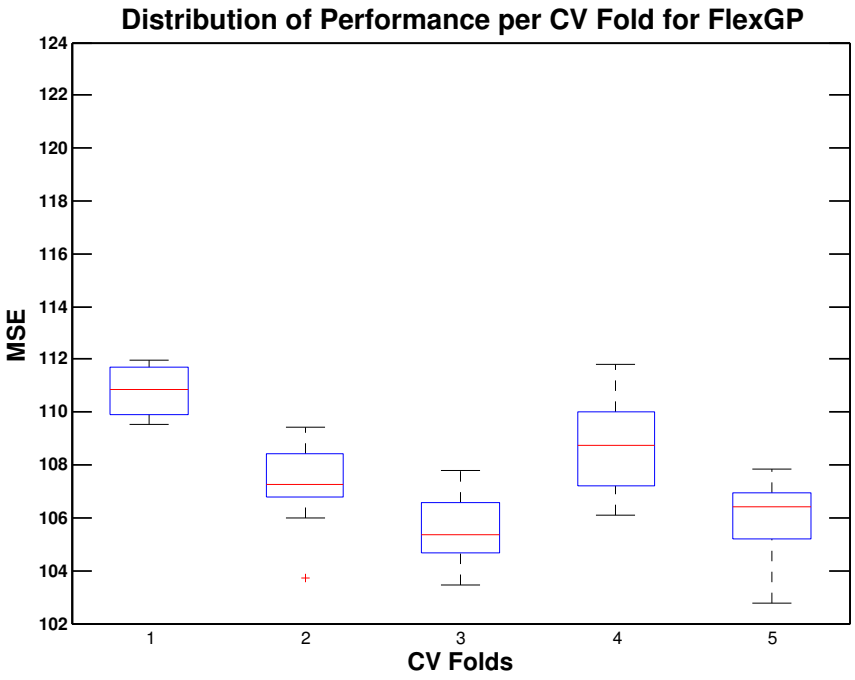
performances with the performance after fusion (compare the middle and rightmost distributions), we see just how powerful fusion can be. Individually, most of the models perform worse than in SGP and there's a huge variance in their performances. However, when fused, the ensembled models produce a result better than any individual model and even better than the baseline. Further, the FlexGP results have much less variance than SGP ($\sigma^2 = 0.42$ versus $\sigma^2 = 5.01$), indicating that FlexGP produces more consistent results than GP, even when learning with different splits of the data.

To answer the second question, we examine how FlexGP results vary for the same fold. By looking at the results within each fold from FGP, an estimate of the variance of the FlexGP system itself can be obtained. We can compute a similar estimate for SGP, and compare them.

Figure 5-6 compares the variance of $\mu_{t|f=i}$ across each of the five CV folds for each of SGP and FGP. This corresponds to reading down the columns of Table C.2. Notice that the median is less and the spread is smaller for each fold in FGP compared to



(a) *SGP*



(b) *FGP*

Figure 5-6: Boxplots of $\mu_{t|f=i}$ over CV folds for *SGP* and *FGP*. Notice how different folds are harder than others. Whiskers extend to 1.5 IQR.

that of *SGP*. This is consistent with Figure 5-5 and further confirms the benefit of running FlexGP.

However, what's also interesting to note is how the relative difficulty of different splits is nearly consistent between the two experiments. Both GP and FlexGP perform worst on fold 1, while they both performing best on folds 3 and 5. This suggests that performance depends on how the data is split and thus validating our use of cross validation to get a true estimate of system performance.

5.4.2 Study 2: Impact of Different Factorings

A key component of the FlexGP system is its ability to factor the learners over several parameters. This makes the system much more flexible than it otherwise would be. It also increases the diversity of the individual learners, which in turn impacts how they perform. To examine the role of these factorings in system performance, we run 16 experiments, each for one trial, with different subsets of factorings enabled. In particular, with the four factorings defined by Section 4.4, we examine all 2^4 combinations of enabling/disabling each factoring. Previous work of ours has shown that factorings do produce diverse learners, and that diversity may have an impact on performance [19]. These experiments are denoted as *FAC01* through *FAC16*.

The experimental results are reported in Table C.3 and summarized in Table 5.4 (along with the factoring settings). Each run within each trial lasted two hours. In Figure 5-7 we compare the fused performance ($\mu_{f|t=1}$) with that of the constituent ensembles ($mean(\zeta^{tr}_{1,j})$, $mean(\zeta^{te}_{1,j})$).

Looking at 5-7, we can see that FlexGP continues to give a significant improvement in performance over the models that form the ensemble. Notice, however, that the performance of FlexGP does not significantly vary with the different factorings explored. Given the low variance of FlexGP results demonstrated by the right-most boxplot of Figure 5-5, we do not expect to see varied results if we ran more trials for each experiment. Further, the FlexGP results are not at all correlated with the training MSE of the ensemble[11].

[11] Although highly correlated with the test MSE, at a correlation coefficient of 0.81.

Exp.	F	D	L	O	$\mu_{f\|t=1}$	$\sigma_{f\|t=1}$
FAC01					105.074	3.081
FAC02	✓				104.812	2.131
FAC03		✓			105.198	2.507
FAC04			✓		106.252	1.956
FAC05				✓	106.330	2.349
FAC06	✓	✓			106.460	2.076
FAC07	✓		✓		107.230	3.264
FAC08	✓			✓	107.476	1.599
FAC09		✓	✓		105.734	1.821
FAC10		✓		✓	105.754	3.092
FAC11			✓	✓	107.198	2.002
FAC12	✓	✓	✓		107.108	4.056
FAC13	✓	✓		✓	106.620	3.713
FAC14	✓		✓	✓	108.306	1.560
FAC15		✓	✓	✓	106.772	4.499
FAC16	✓	✓	✓	✓	107.674	3.196

Table 5.4: Summary of results from the *FAC* experiments.

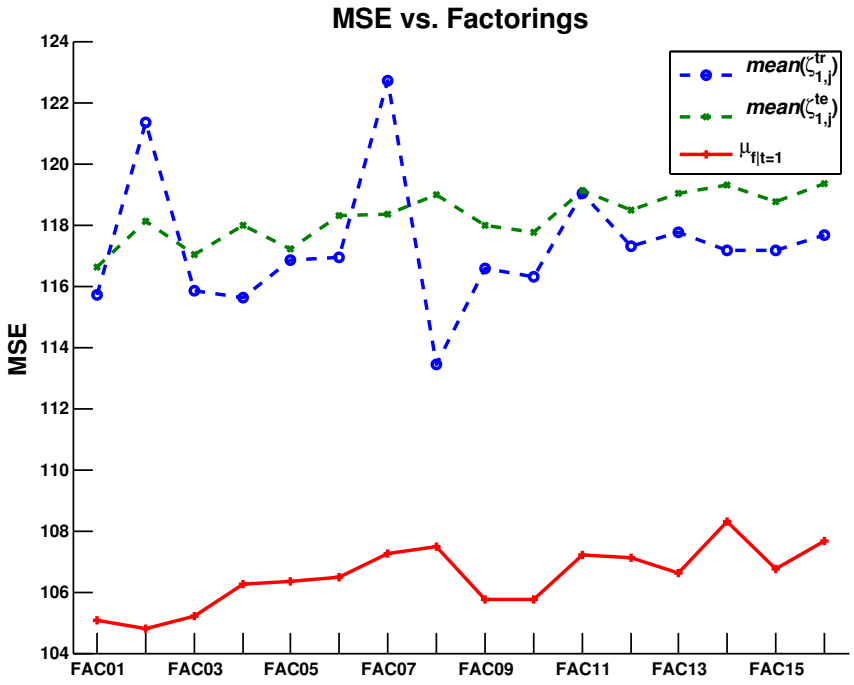


Figure 5-7: Plots of FlexGP ensemble and fused performance from the *FAC* experiments. The $mean(\zeta^{te}_{1,j})$ (green line) and $mean(\zeta^{te}_{1,j})$ (red line) for each experiment is plotted, along with the resulting $\mu_{f|t=1}$ after fusion (blue line).

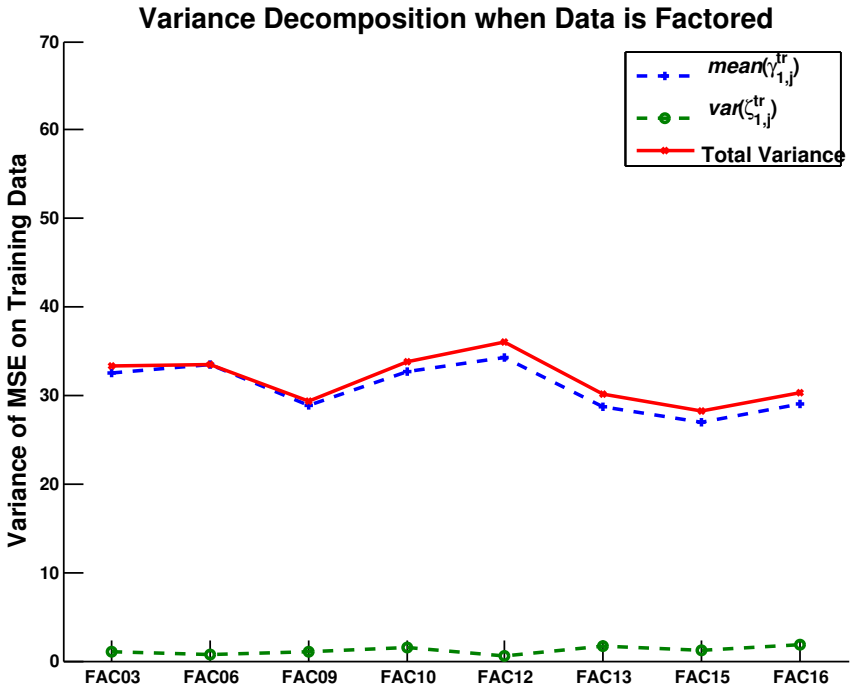
This shows us that varying the factorings has little impact on the average performance of the models in the ensemble. We next turn to examining the variance of the training performance of the ensemble. By drilling down and looking at the variance of the training MSE across models in the underlying ensemble, we can begin to see some effects from factorings. In particular, the enabling or disabling of data factoring has a very interesting effect. The two plots split the experiments between those which run with data factoring turned on (Figure 5-8a) and those which don't (Figure 5-8b). In Figure 5-8a you can see that when the d_i is sampled locally at every FlexGP instance, the total variance[12] (solid line) is relatively stable. However, when data is not factored and the training data is identical for every instance, the total variance is erratic.

Digging further, if we treat the MSE of the models from each run in a trial as (five) separate distributions, we can (approximately) break the total variance down into two components. The first component is $mean(\gamma^{tr}_{1,j})$, the mean of the variances of these distributions, which indicates how much of the variance is due to the different factorings. The second component is $var(\zeta^{tr}_{1,j})$, or the variance of the means of these distribution. This tells us how much of the variance can be explained by the CV folds, which we have already shown to be uneven (see Figure 5-6).

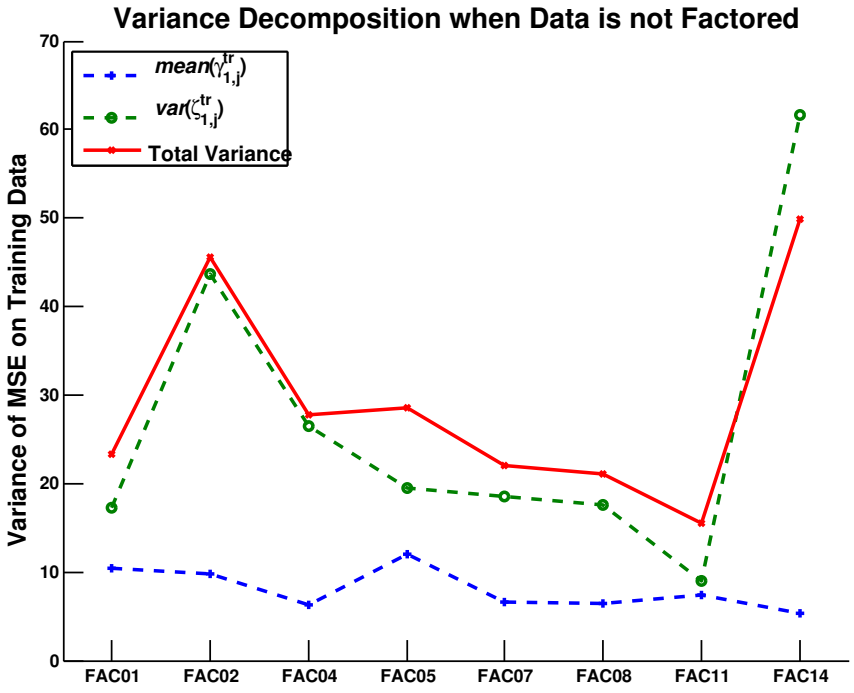
With this decomposition, we now see something extraordinary. With data factoring enabled, the effect of the splits ($var(\zeta^{tr}_{1,j})$) is minimized and the majority of the variance is due to the instances all learning on different subsets of the data ($mean(\gamma^{tr}_{1,j})$. Further, this result appears to be stable across the other factorings (suggesting that the data factoring is the predominant source of variance). However, as soon as the data is no longer factored, the ordering flips and the splits contribute a significant amount of variance, even moreso than the variance from the remaining factorings. Additionally, the amount of variance from the splits is itself erratic, contributing to the erratic nature of the total variance.

This result makes intuitive sense. If every instance is looking at a different subset

[12]The total variance is defined as the variance in the training MSEs of all models from all ensembles of a given trial (i.e. the concatenation of all five ensembles from a trial).



(a)



(b)

Figure 5-8: Total variance and its two components.

of the training data, then instances are resampling the data, ensuring they are learning in different areas of the problem and negating the variance arising from only studying the data within the provided split (D_{al}). However, as soon as data factoring is disabled, every instance learns from the same subset, which only acts to magnify the problems associated with using a particular split of the data.

In the end, despite the increased variance of the ensembles when data factoring is turned off, FlexGP improves performance over the ensemble average. This suggests that even though which factorings are used can have a dramatic effect on the performance within the ensemble, FlexGP is not as sensitive. However, this could be a dataset-dependent result and warrants further study with other datasets.

5.4.3 Study 3: Changing the Size of the Data

Another key aspect of FlexGP is the opportunity to vary the amount of training data used by the local learners. This enables faster learning and also leads to more diverse learners. However, learning on less data typically comes at the risk of overfitting. To explore this concern, we run six experiments, each for one trial, with varying sizes of d_i. In particular, for each experiment, all instances still sample a fixed-size training set, but between experiments that size is changed. Previous work of ours has shown that while individual model performance suffers, FlexGP performance remains stable as training set size decreases [19]. These experiments are denoted as *TDS01* through *TDS06*.

As before, raw results from the experiments are reported in Table C.3 and summarized in Table 5.5. Each of these experiments was run for six hours each[13]. Figure 5-9 is structured identically to Figure 5-7 and reports how the $\mu_{f|t=1}$ after fusion compares to the average train and test MSE of the ensembles.

Examining Figure 5-9 yields several interesting observations. First, unlike Figure 5-7, we see that the size of d has a significant impact on FlexGP performance. While fused performance still outperforms the average ensemble test MSE, it suffers when the training set is either too small or too large. In the case where the training set

[13]To account for the delayed learning in *TDS16* due to learning on over 350,000 datapoints

Exp.	$\frac{\lvert d_i \rvert}{\lvert D_{gp} \rvert}$	# samples	$\mu_{f\lvert t=1}$	$\sigma_{f\lvert t=1}$
TDS01	0.10%	361	113.644	3.557
TDS02	1.00%	3,609	104.464	4.622
TDS03	10.0%	36,090	107.674	3.196
TDS04	25.00%	90,224	107.080	3.545
TDS05	50.00%	180,448	110.398	1.700
TDS06	100.00%	360,895	112.258	3.049

Table 5.5: Summary of results from the *TDS* experiments.

is extremely small (0.1% of D_{al}) the ensemble models simply overfit and have poor generalization, as shown by the extremely low (good) train MSE performance but extremely bad test MSE performance at the 0.1% point. Conversely, as the size of d_i increases, there becomes too much data to learn effectively with and the models begin to bloat and underfit. Fortunately, even in the case of severe overfitting, FlexGP is still able to do something reasonable and produce improved results.

Following the discussion of the total variance decomposition in Section 5.4.2, Figure 5-10 examines this decomposition in the *TDS* experiments. And, consistent with the results from the previous section, we see that with all factorings on, the majority of the variance is due to the factorings and not the data splits. In fact, the amount of variance arising from the factorings increases exponentially as the training set size decreases. One likely explanation for this phenomenon is that as the dataset size decreases, there are more possible subsets to choose for training when data factoring is enabled. Thus, we see an increase in ensemble variance as the data factoring becomes more and more important. Further study of turning on and off data factoring while changing training data set amounts is needed to make any definitive conclusions, however.

5.5 Summary of Findings

We conclude briefly by summarizing the four questions explored above, and the insights we gained from our analysis:

Q1 How does FlexGP improve overall performance for a given problem?

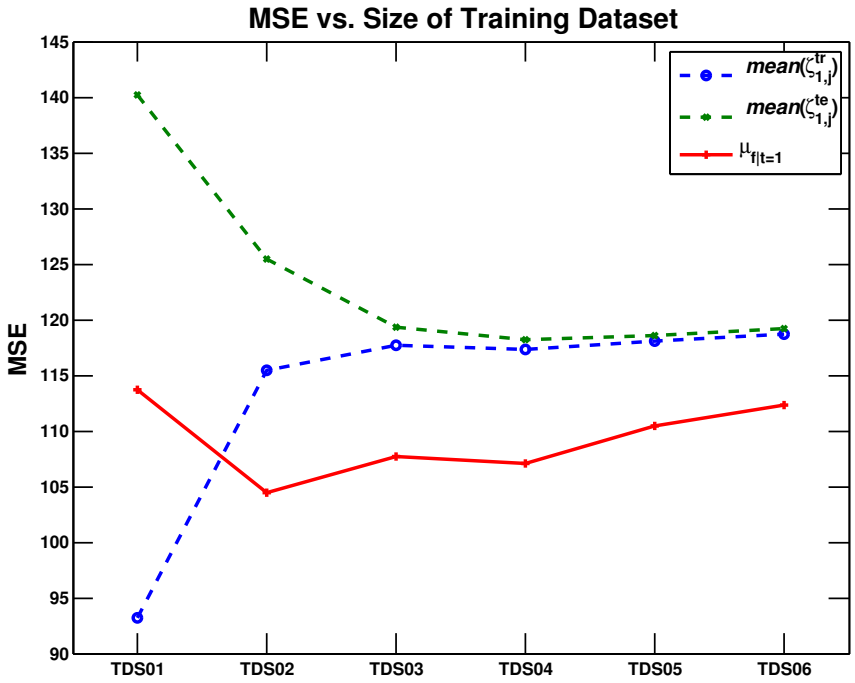


Figure 5-9: Plots of FlexGP ensemble and fused performance for *TDS* experiments. The $mean(\zeta^{te}_{1,j})$ (green line) and $mean(\zeta^{te}_{1,j})$ (red line) for each experiment is plotted, along with the resulting $\mu_{f|t=1}$ after fusion (blue line).

A1 FlexGP performs significantly better than a single learner, while also producing results with lower variance.

Q2 How reliable are the FlexGP results?

A2 FlexGP produces better results than a single learner in all folds, although those results vary with each split.

Q3 How do the different factorings impact the overall performance of the system?

A3 Different factorings have little impact on the mean performance of FlexGP. However, data factoring has a tremendous impact on the quality of the performance, acting to stabilize the results and greatly decrease the variance while increasing the diversity of the learners.

Q4 How much training data is enough for FlexGP to perform well?

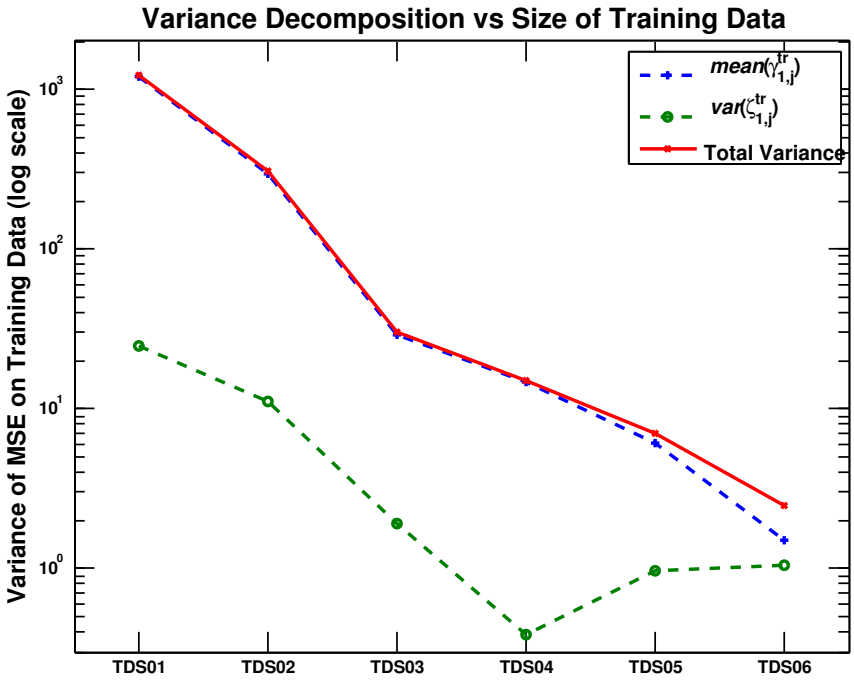


Figure 5-10: Components of total variance for the FAC experiments.

A4 When the target training data size is too small or too large, FlexGP performance is degraded compared to more optimal settings. However, the fused results are still better than the baseline and of significantly lower variance.

Chapter 6

Conclusion

6.1 Summary of Contributions

This work presented FlexGP, a scalable system for learning in the cloud. In today's environment of cheap, massive-scale computing resources, running existing machine learning algorithms in the cloud is a very appealing option for handling the increasing size of datasets. FlexGP provides a system for scaling any machine learning algorithm on the cloud effortlessly. By treating the learning algorithm as a black box which takes a data partition and algorithm parameter settings as input and outputs a model, FlexGP presents a unique approach to cloud-based distributed learning which will work for any existing algorithm.

Starting from a principled discussion of existing solutions, we outlined the ways in which MapReduce- and BSP-based distributed computing paradigms are insufficient for the general problem of scaling *all* types of machine learning algorithms with the cloud. We also described previous work to retrofit a cluster-focused system to the cloud and how that work motivated the development of FlexGP. We presented the challenges to any cloud-based learning system, and the goals we set for FlexGP to ensure it met those challenges.

Next, we presented a sketch of our design for FlexGP, shaped by those goals, to run efficiently in the cloud. This included:

- an efficient workflow for machine learning, where factored learners are run in parallel, and then collected and filtered to form an ensemble, which is then fused to produce a meta-model.

- a parallel, asynchronous and recursive startup protocol which allows for the distributed launching of hundreds of instances quickly.

- a distributed peer discovery protocol for the creation a communication layer between instances simultaneously with learning.

- a novel approach to factored learning, which enables FlexGP to run a network of diverse learners.

We then gave a detailed description of how each of these features was implemented. This description included empirical results demonstrating the efficacy of various critical components.

Finally, We outlined an experimentation framework for testing FlexGP which controls for the variance due to arbitrary splits of the data as well as variance due to single trials. This framework can be taken and used in future work involving FlexGP or adopted for other distributed computing systems. Using this framework, we investigated four overarching questions about the performance of FlexGP and demonstrated that FlexGP improves performance over a baseline learner. We also showed that factoring indirectly contributes to a lower variance by creating more diverse ensembles.

6.2 Directions for Future Work

The contributions of this work have been many. Yet, there remains a lot to do and many directions for future work and extensions to the FlexGP system. In this section several candidate areas for improvement are discussed.

Elastic Management

FlexGP supports arbitrarily starting and stopping learners to expand and contract the learning effort. However, it has to be done by hand and there is no automated way of retrieving the progress from the killed learners before they are stopped. Concurrent work is looking at how to automate this process, so the computation footprint can be automatically expanded and contracted according to an input signal or user command.

Restarting Learners

Once started, learners continuously compute until the learning algorithm terminates or they are forcefully stopped (by the user or some management process). If the learning algorithm terminates, the learner will just sit there, burning cpu time (and therefore $). Instead, the learner should restart, selecting a new set of algorithm parameters and data partition. Further, it might be desirable to restart learners which are either doing very poorly (they chose the "wrong" setting of parameters or data partition) or happen to be running with identical settings to another learner (depending on what is being factored, this could be more or less likely).

Sophisticated Logging and Monitoring

Currently, FlexGP implements rudimentary event-based logging of system info, learner progress and state, and network communications. These logs serve as the only form of system monitoring as well. This is sufficient for a developmental version of FlexGP, but for any non-developer users, advanced monitoring facilities will be needed. Such monitoring would enable the user to inspect the progress at each learner, assess overall system health, and make informed decisions as to when the learning should be expanded, shrunk, or stopped completely. Some work is being done to help with visualizing the log information in real time, which is a first step towards monitoring.

Online Fusion

In addition to adding monitoring facilities, the usability of FlexGP would be greatly improved if fusion could be performed online. This would make the system truly cloud-based, as opposed to the current version where the user needs sufficient memory and space on their local machine to compute the filter and fusion steps. The challenge here is to determine how best to share the work of computing the fusion across the learners.

Moving beyond GP

Currently, FlexGP has only been tested with genetic programming as the learning library. However, there is nothing about FlexGP which requires this. To demonstrate this, and to validate FlexGP as a generally useful tool, it should be run with several other learning libraries (support vector machines, k-nearest neighbors, neural networks, etc.) and performance should be examined.

6.3 Conclusion

FlexGP is designed explicitly to take advantage of the unique opportunities of the cloud. By combining the process of factoring algorithm parameters and data partitions with parallel startup and peer discovery protocols, FlexGP constructs a completely peer-to-peer network of heterogeneous learners, producing many diverse models. Further, FlexGP provides efficient mechanisms for collecting the learned models into an ensemble, which can then be filtered and fused into a single meta-model.

FlexGP is a robust system designed for scalable, factored learning on large datasets. As FlexGP continues to develop, we are excited to see FlexGP extended in new directions and solve new problems, on its path to becoming an essential tool for researchers the world over.

Appendix A

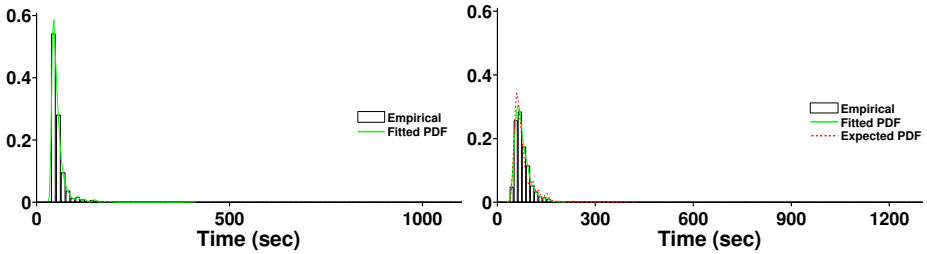
Starting Nodes in the Cloud

Applications typically request instances from a cloud in batches. The cloud possibly queues these batch requests and may decompose them; interleaving them with requests from other users. This might depend on batch size or the cloud's use of an internal fine-grained queue and a scheduler. Regardless of what a particular cloud does, the instance scheduler implementation should be treated as opaque by application designers.

To fully understand how FlexGP would need to interact with the underlying cloud platform, a study of the latency in acquiring cloud instances was undertaken. This is important to understand the choices made in designing FlexGP's launch protocol.

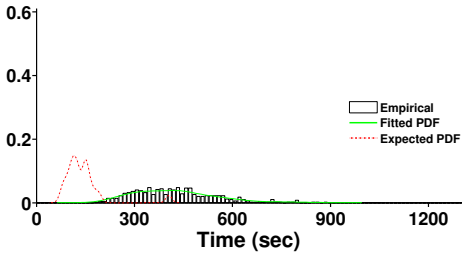
To begin, we develop a theoretical framework for analyzing our observations. We assume that the time elapsed between requesting an instance and when that instance has booted and begins running our code, the *latency*, is modeled by some distribution $P(u)$. We first estimated $P(u)$ by acquiring a single instance 1,000 times and measuring the latency, u, of each request. The data and its distribution are reported in Fig. A-1a. If we optimistically assume a batch request of n instances is served in parallel as n independent requests by the scheduler, then the total latency, v_n, of the request ought to be the maximum of n independent samples drawn from $P(u)$. We estimated $P(v_n)$ for $n \in [5, 50, 100]$ with 500 samples and then fit a non-parametric distribution to the data. We report the observed data and fitted distributions alongside the predicted distributions (based on our measured $P(u)$) in Fig. A-1. While the predicted

and empirical distributions for $P(v_5)$ are close, the actual latency distributions for $P(v_{50})$ and $P(v_{100})$ are significantly larger than predicted.

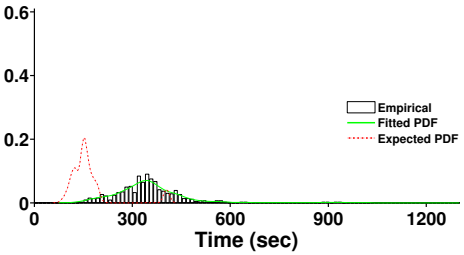


(a) Times to acquire 1 node.

(b) Times to acquire 5 nodes.



(c) Times to acquire 50 nodes.



(d) Times to acquire 100 nodes.

Figure A-1: Probability distribution functions (PDF) of times to acquire nodes.

This discrepancy indicates that smaller batch requests achieve closer to optimal latency than larger requests, and so our system ought to emphasize small batch requests over large ones. Futher, because acquiring many (50 or 100) instances may take significantly longer than acquiring the first 10 instances, we should start running GP on an instance immediately after it boots, long before the entire set of nodes is acquired. Another concern when computing using the cloud is failing nodes. Requested nodes may never be acquired and running nodes may fail. This necessitates an architecture which is resilient to failures.

Appendix B

GP Learner

This section presents the GP implementation used in detail, for the sake of reproducibility. The GP learner is implemented in Java.

This GP implementation is configured as described by Koza [9], with the following differences. The learner is set with a population of size 1000. The mutation rate is 0.2, the crossover rate is 0.6 and nodes were selected uniformly at random for crossover. The max depth is set to 6 for initialization and then trees are allowed to grow to a depth of 12 afterwards. Tournament selection was used, with a tournament size of 7. and trees were initialized with the ramped-half-and-half algorithm. The learners were allowed to run until they were stopped or ran through 50 generations, whichever occurred first. Bloating is prevented by using Silva's dynamic operator equalization [17] with a bin width of 5. During fitness evaluation, individuals' predictions are transformed with Vladislavleva's approximate linear scaling [24]. Finally, because the years in MSD are only reported as integers, fitness values are rounded to the nearest years before computing an error.

Appendix C

Experimental Results

For reference purposes, we report here the raw results of all experiments. The tables appear in the order they are referenced from the text.

Exp.	per_node	none	best_n
ARM	107.796	107.674	107.798
	107.356	107.322	107.21
	108.316	108.342	108.314
	108.346	108.012	108.674
	107.956	107.57	107.926
	107.51	107.214	107.352
	108.394	108.344	108.336
	107.778	107.558	107.708
	106.318	106.2	106.146
	108.15	108.138	108.122
AVE	115.878	116.502	110.122
	115.624	116.304	109.73
	115.898	116.54	110.088
	115.728	116.394	109.882
	115.598	116.302	110.118
	115.564	116.276	109.196
	115.748	116.48	109.874
	115.816	116.414	109.278
	115.934	116.304	109.292
	115.67	116.292	109.598
MAD	116.742	116.214	113.058
	116.744	115.962	112.24
	116.614	116.206	112.342
	116.532	116.134	112.48
	116.312	115.97	112.046
	116.774	116.054	111.75
	116.812	116.126	112.368
	116.632	115.99	112.51
	116.924	115.872	111.968
	116.71	115.978	112.164

Table C.1: Mean MSE performance ($\mu_{f|t=i}$) for each trial i in *FGP* per filter/fusion pair.

Exp.	CV Folds				
SGP	118.30	116.73	113.66	110.85	109.41
	117.43	107.73	112.46	111.66	111.10
	123.33	114.04	108.71	113.35	107.80
	119.90	116.38	114.69	111.60	108.85
	118.51	118.18	107.35	116.82	113.28
	118.72	107.15	108.21	104.69	109.42
	119.24	117.85	107.02	119.36	116.10
	115.37	108.13	109.70	116.15	108.69
	110.01	110.21	106.51	115.39	110.55
	119.68	115.25	114.27	115.75	116.81
FGP	111.72	109.27	105.70	108.22	103.46
	110.30	107.58	106.57	106.11	106.05
	109.89	107.19	104.97	111.81	107.85
	111.72	108.45	104.65	110.03	105.21
	111.77	105.98	106.23	106.96	106.91
	109.60	107.22	103.68	109.42	106.15
	111.41	109.45	104.93	109.25	106.68
	111.96	107.26	103.46	108.18	106.93
	109.56	103.71	107.78	107.20	102.75
	109.94	106.81	106.93	110.04	106.97

Table C.2: Raw MSE performance ($e_{i,j}$) for every fold of every trial from experiments *SGP* and *FGP*.

Exp.	CV Folds				
FAC01	109.82	103.04	102.30	106.45	103.76
FAC02	105.90	102.14	103.44	107.63	104.95
FAC03	108.40	104.49	103.71	107.11	102.28
FAC04	107.85	105.37	104.97	108.79	104.28
FAC05	107.74	105.17	103.51	109.55	105.68
FAC06	109.12	105.10	104.64	108.28	105.16
FAC07	110.59	106.28	104.09	110.78	104.41
FAC08	107.82	108.09	106.86	109.46	105.15
FAC09	108.08	104.71	104.02	107.29	104.57
FAC10	110.45	104.27	104.12	107.21	102.72
FAC11	109.91	106.50	105.54	108.66	105.38
FAC12	112.51	101.78	105.02	109.08	107.15
FAC13	111.04	107.29	101.91	108.97	103.89
FAC14	110.75	107.09	107.57	108.96	107.16
FAC15	112.35	107.21	100.43	109.10	104.77
FAC16	111.72	109.27	105.70	108.22	103.46
TDS01	118.40	108.98	111.78	115.30	113.76
TDS02	110.90	107.22	101.27	103.50	99.43
TDS03	111.72	109.27	105.70	108.22	103.46
TDS04	110.58	102.59	107.29	110.43	104.51
TDS05	112.17	110.17	109.48	112.00	108.17
TDS06	114.47	110.46	109.21	116.43	110.72

Table C.3: Raw MSE performance ($e_{i,j}$) for every fold of every trial from *FAC* and *TDS* experiments.

Bibliography

[1] T. Bertin-Mahieux, D.P.W. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *ISMIR 2011: Proceedings of the 12th International Society for Music Information Retrieval Conference, October 24-28, 2011, Miami, Florida*, pages 591–596. University of Miami, 2011.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. page 1, 2004.

[3] Owen Derby, Kalyan Veeramachaneni, and Una-May OReilly. Cloud driven design of a distributed genetic programming platform. In Anna I. Esparcia-Alczar, editor, *Applications of Evolutionary Computation*, number 7835 in Lecture Notes in Computer Science, pages 509–518. Springer Berlin Heidelberg, January 2013.

[4] Pedro Fazenda, James McDermott, and Una-May OReilly. A library to run evolutionary algorithms in the cloud using MapReduce. In Cecilia Chio, Alexandros Agapitos, Stefano Cagnoni, Carlos Cotta, FranciscoFernndez Vega, GianniA. Caro, Rolf Drechsler, Anik Ekrt, AnnaI. Esparcia-Alczar, Muddassar Farooq, WilliamB. Langdon, JuanJ. Merelo-Guervs, Mike Preuss, Hendrik Richter, Sara Silva, Anabela Simes, Giovanni Squillero, Ernesto Tarantino, AndreaG.B. Tettamanzi, Julian Togelius, Neil Urquhart, ima Uyar, and Georgios Yannakakis, editors, *Applications of Evolutionary Computation*, volume 7248 of *Lecture Notes in Computer Science*, pages 416–425. Springer Berlin Heidelberg, 2012.

[5] Di-Wei Huang and J. Lin. Scaling populations of a genetic algorithm for job shop scheduling problems using MapReduce. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 780–785, December 2010.

[6] Mrk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321–2339, 2009. Gossiping in Distributed Systems.

[7] JuanLuis Jimnez Laredo, Daniel Lombraa Gonzlez, Francisco Fernndez de Vega, Maribel Garca Arenas, and JuanJulin Merelo Guervs. A peer-to-peer approach to genetic programming. In Sara Silva, JamesA. Foster, Miguel Nicolau, Penousal Machado, and Mario Giacobini, editors, *Genetic Programming*, volume 6621 of *Lecture Notes in Computer Science*, pages 108–117. Springer Berlin Heidelberg, 2011.

[8] M. Kotanchek, G. Smits, and E. Vladislavleva. Trustable symbolic regression models. *Genetic Programming Theory and Practice V, Genetic and Evolutionary Computation*, pages 203–222, 2007.

[9] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[10] J.L.J. Laredo, A.E. Eiben, M. Steen, and J.J. Merelo. Evag: a scalable peer-to-peer evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11:227–246, 2010.

[11] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[12] Grzegorz Malewicz, Matthew H. Austern, Aart JC Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, page 135146, 2010.

[13] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Publications Co., 2011.

[14] Joshua Rosen, Neoklis Polyzotis, Vinayak Borkar, Yingyi Bu, Michael J. Carey, Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Iterative MapReduce for large scale machine learning. March 2013.

[15] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, April 2009. PMID: 19342586.

[16] Dylan Sherry, Kalyan Veeramachaneni, James McDermott, and Una-May OReilly. Flex-GP: genetic programming on the cloud. In Cecilia Di Chio, Alexandros Agapitos, Stefano Cagnoni, Carlos Cotta, Francisco Fernndez de Vega, Gianni A. Di Caro, Rolf Drechsler, Anik Ekrt, Anna I. Esparcia-Alczar, Muddassar Farooq, William B. Langdon, Juan J. Merelo-Guervs, Mike Preuss, Hendrik Richter, Sara Silva, Anabela Simes, Giovanni Squillero, Ernesto Tarantino, Andrea G. B. Tettamanzi, Julian Togelius, Neil Urquhart, A. ima Uyar, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation*, number 7248 in Lecture Notes in Computer Science, pages 477–486. Springer Berlin Heidelberg, January 2012.

[17] Sara Silva. Handling bloat in GP. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, GECCO '11, pages 1481–1508, New York, NY, USA, 2011. ACM.

[18] Leslie G. Valiant. A bridging model for parallel computation. 33(8):103111, 1990.

[19] Kalyan Veeramachaneni, Owen Derby, Dylan Sherry, and Una-May OReilly. Learning regression ensembles with genetic programming at scale. in press, 2013.

[20] Kalyan Veeramachaneni, Katya Vladislavleva, Matt Burland, Jason Parcon, and Una May O'Reilly. Evolutionary optimization of flavors. In Juergen Branke and et al, editors, *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1291–1298, Portland, Oregon, USA, 7-11 July 2010. ACM.

[21] A. Verma, X. Llora, D.E. Goldberg, and R.H. Campbell. Scaling genetic algorithms using mapreduce. In *Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on*, pages 13–18, December 2009.

[22] A. Verma, X. Llora, S. Venkataraman, D.E. Goldberg, and R.H. Campbell. Scaling eCGA model building via data-intensive computing. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, July 2010.

[23] C. Vladislavleva and G. Smits. Symbolic regression via genetic programming. *Final Thesis for Dow Benelux BV*, 2005.

[24] E.Y. Vladislavleva. *Model-based problem solving through symbolic regression via pareto genetic programming*. PhD thesis, CentER, Tilburg University, 2008.

[25] Shuaiqiang Wang, Byron J. Gao, Ke Wang, and Hady W. Lauw. Parallel learning to rank for information retrieval. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 1083–1084, New York, NY, USA, 2011. ACM.

[26] Larry Wasserman. *All of nonparametric statistics*. Springer, New York; London, 2006.

[27] Yuhong Yang. Regression with multiple candidate models: selecting or mixing? *Statistica Sinica*, 13(3):783–810, 2003.