DOI 10.1007/s10723-014-9320-9

FlexGP

Cloud-Based Ensemble Learning with Genetic Programming for Large Regression Problems

Kalyan Veeramachaneni · Ignacio Arnaldo · Owen Derby · Una-May O'Reilly

Received: 23 June 2014 / Accepted: 17 October 2014 / Published online: 18 November 2014

Abstract We describe FlexGP, the first Genetic Programming system to perform symbolic regression on large-scale datasets on the cloud via massive data-parallel ensemble learning. FlexGP provides a decentralized, fault tolerant parallelization framework that runs many copies of Multiple Regression Genetic Programming, a sophisticated symbolic regression algorithm, on the cloud. Each copy executes with a different sample of the data and different parameters. The framework can create a fused model *or ensemble* on demand as the individual GP learners are evolving. We demonstrate our framework by deploying 100 independent GP instances in a massive data-parallel manner to learn from a dataset composed of 515K exemplars and 90 features, and by generating a competitive fused model in less than 10 minutes.

Keywords Cloud computing · Ensemble learning · Genetic programming · Symbolic regression

1 Introduction

The increased availability and cost effectiveness of data storage has allowed the collection of many large datasets. However, the exploitation of large data requires scaled machine learning solutions. We present FlexGP, the first Genetic Programming system to perform symbolic regression on large-scale datasets on the cloud. Genetic Programming continues to mature as a technique, with the emergence of products such as DataModeler [1] and Eureqa [2]. The main advantages of GP are its flexibility and is its embarrassingly parallel nature. The first allows non-linear symbolic models of the data to be obtained while the latter can be exploited to harness the computational power provided by clouds.

The increased size of datasets represents a challenge in the context of GP for two reasons. First, the size of the training dataset may now exceed the capacity of main memory. And second, the computational expense of the GP learner scales with the quantity of training data, as the score or *fitness* assigned to each candidate model depends on its error on the data.

To overcome these limitations, we implement a parallelization framework that performs an efficient decomposition of the computation required for learning. The proposed method splits the data into multiple subsets and learns a large quantity of models via independent learners. This allows the computation to execute on many instances in parallel and is known as a data-parallel approach (see [3]). Each model is used

K. Veeramachaneni · I. Arnaldo (✉) · O. Derby ·
U.-M. O'Reilly
Massachusetts Institute of Technology,
32, Vassar Street, Cambridge,
MA, 02139, USA
e-mail: iarnaldo@mit.edu

K. Veeramachaneni
e-mail: kalyan@csail.mit.edu

U.-M. O'Reilly
e-mail: unamay@csail.mit.edu

to make a prediction and a meta-model or *ensemble* is developed to fuse these predictions.

In this paper we present an end result of a three year project that resulted in FlexGP. Our contributions and the challenges we address are:

Multiple Regression Genetic Programming (MRGP) learner: FlexGP incorporates MRGP, a sophisticated learner that hybridizes tree-based GP and Least Absolute Selection and Shrinkage Operator (LASSO) introduced in [4]. In previous work, we have shown that MRGP outperforms both multiple linear regression and traditional GP-based symbolic regression methods [5].

Factoring: FlexGP employs factoring. Each learner can execute with a different set of machine learning parameters. For large data problems, each learner trains on a factored subset of the data. The subset can be on the basis of both number of training examples and number of features. All factoring is done in a probabilistic manner controlled by a simple user configuration. To improve speed and address memory limitation, we reduce the data size at each instance by a factor of 10.

Cloud-Scale Ensemble Learning: FlexGP is a cloud scale regression ensemble learning system. Models are generated by multiple cloud-backed virtual machines each supporting an independent parametrized GP learner. Under this scenario, FlexGP creates a fused model (ensemble) on demand at different points in time while the individual GP learners are evolving. The stochasticity of GP and the varying computation speed of virtual machines on the cloud induces variability in the learning rate of each learner. In this paper, we examine how the performance of the fused model changes in real time.

Decentralized Machine Learning platform: In FlexGP, there is no single controller coordinating the system. It launches via a cascaded asynchronous startup protocol and runs a completely decentralized neighbor discovery process at its IP layer. The protocol establishes the network simultaneously with the cascaded launch and integrates new instances into the network. It is resilient to instance failure and allows communication to continue even when instances disappear.

The paper is organized as follows. Section 2 provides an overview of FlexGP. We introduce MRGP, the regression algorithm at the core of FlexGP, in Section 3. In Section 4, we explain the adopted ensemble approach while in Section 5, we detail FlexGP's communication layer. Section 6 describes the experimental setup and we present the results in Section 7. Finally, Section 8 presents a review of related work and we conclude in Section 9.

2 FlexGP Overview

The availability of massive on-demand computational resources via the cloud enables us to learn many models in parallel. Bagging, boosting or simple parameter variation are now feasible at an unprecedented scale. We set FlexGP to run many instances of a sophisticated Genetic Programming algorithm in parallel in the cloud, thus generating *an ensemble of models*. Figure 1 provides an overview of FlexGP:

Multiple Regression Genetic Programming Each cloud node executes in a multi-threaded fashion a copy of Multiple Regression Genetic Programming [5]. MRGP is a hybrid method that combines tree-based Genetic Programming with LASSO.

Ensemble Learning The independent copies of the regression algorithm learn from different samples of the data and run with different parameters. At any moment of the run, it is possible to retrieve the best models of the run and build a meta-model by means of a model fusion process.

Cloud Layer FlexGP is a framework for mining large datasets on the cloud. The platform implements a distributed launch protocol and a decentralized, fault tolerant communication layer.

3 Multiple Regression Genetic Programming

MRGP is a hybrid method that combines tree-based Genetic Programming with LASSO. MRGP targets the minimization of two objectives. The first, *multiple regression error*, is an innovative accuracy measure that involves a LASSO process and is explained in detail in this section. The second objective is the model subtree complexity measure introduced in [6]. The algorithm implements Single Point crossover,

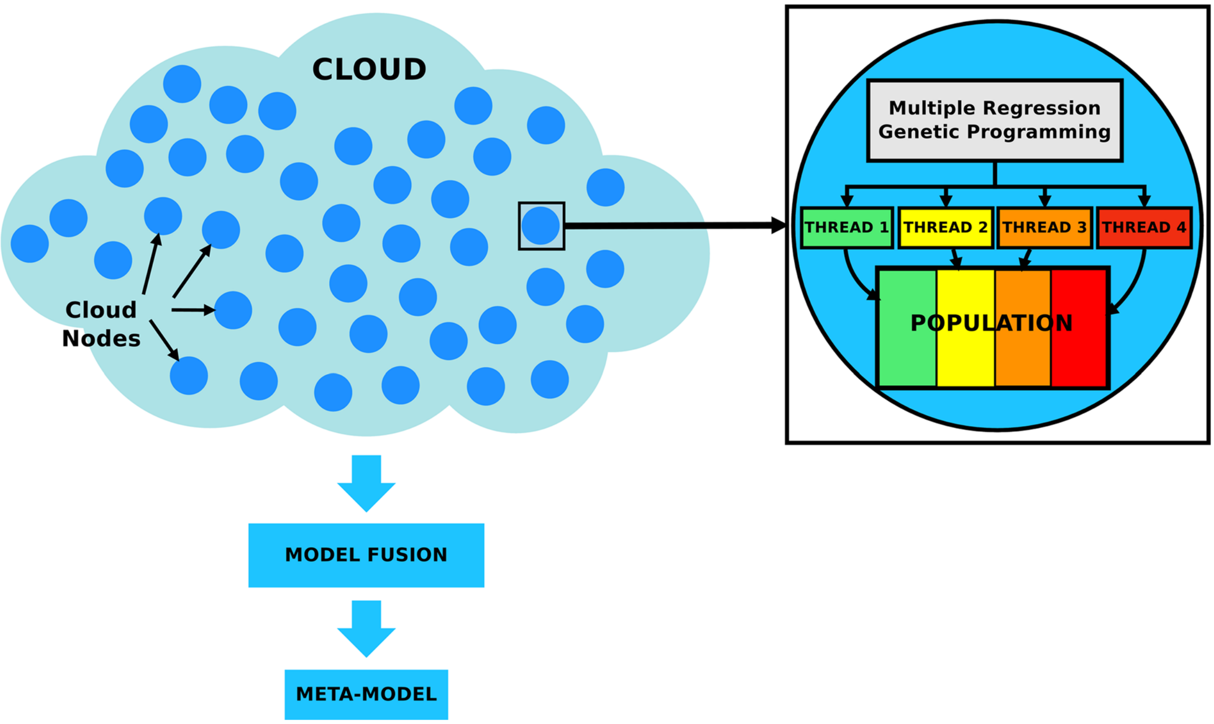


Fig. 1 FlexGP overview: each cloud node executes a copy of Multiple Regression Genetic Programming in a multi-threaded fashion. The models generated at each node can be retrieved online to build a meta-model via a fusion process

Subtree mutation, and a selection strategy based on Non-Dominated Sorting Genetic Algorithm II (NSGA-II) introduced by Deb et al. [7].

3.1 Terminology

The objective in a regression task is to find a *model* that maps one or more *input variables* onto a single *target variable* (desired output). When solving such a task using GP-based symbolic regression, models are instantiated by *programs* (expression trees in case of tree-based GP), where *program inputs* (*terminals*, *leaves*) map to the input variables' values, and *program output* is the expression's value when the root node is executed.

3.2 Multiple Regression Error

MRGP differs from conventional GP primarily in eliminating direct comparison of the final program output against the target variable, y. Instead, we tune in linear combination all subexpressions of a program with respect to the target output y. Then, we compare y to the output of the regression model. Given a dataset D (also known as a set of fitness cases) composed of m columns of input variables and n rows of examples, and a target vector y with target values for each example, we proceed as follows:

1. We step by step execute the program (with the conventional inorder tree parse) and store the output of each subexpression after it is executed. For tree-based GP, this means pausing the program execution process at each tree node (including leaves and the root node) and storing the value calculated at that node. By doing this for each training example, we obtain an $n \times k$ matrix of subexpressions F, where k is the size of the GP tree and n is the number of exemplars of D.
2. We map the values of F onto the desired output y using multiple linear regression (MR), which produces an optimal linear combination that minimizes the prediction error of $\hat{y}$. Multiple regression determines the vector of coefficients β that minimizes the sum of squares of residuals e of mapping the k subexpressions (predictors) onto

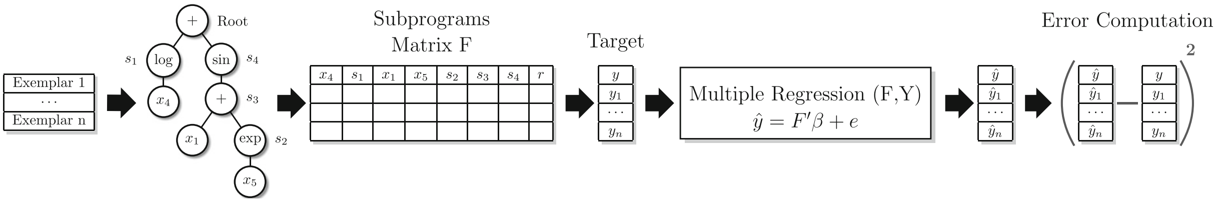


Fig. 2 The outline of model evaluation in Multiple Regression Genetic Programming

the desired output y:

$$y = F'\beta + e$$

where F' is a $n \times (k+1)$ matrix obtained from F by prepending it with an additional column of ones, so that the corresponding coefficient β_1 implements the intercept of the linear model.

3. To assess the quality of the regressed model, we compute its output as $\hat{y} = F\beta$ and compare it to the original targets of the dataset in a conventional way, i.e., as $(y - \hat{y})^T(y - \hat{y}) = e^T e$.

Figure 2 outlines this process for tree-based GP. Major challenges with multiple regression (step 2) can arise because the least squares approach fails if the matrix F is not of full rank. Moreover, this step arguably introduces a computational overhead with respect to standard Genetic Programming. To avoid rank deficiency issues and alleviate the computational burden, we resort to an efficient implementation of regularized linear regression [8]. The employed implementation is based on a cyclical coordinate descent method first proposed in [9]. Cyclical coordinate descent methods work on large datasets and can solve the family of regression problems written as follows:

$$\underset{\beta}{min}\ \frac{1}{2}||X\beta - y||_2^2 + \lambda_1||\beta||_1 + \frac{1}{2}\lambda_2||\beta||_2^2$$

where β is the vector of regression coefficients. We set $\lambda_2 = 0$ so the solution is the LASSO (L1-constrained) linear fit. The employed algorithm returns an array of models corresponding to different values of the parameter λ. We select the value of λ that maximizes the variables included in the model. Without loss of generality we refer to this as Multiple Regression (MR). For further details on the employed cyclical coordinate descent method, the reader is referred to the work by Friedman et al. [9].

3.3 Population Initialization

MRGP allows the initial population to be seeded with a linear combination of the input features (see Fig. 3). With such model, the multiple regression process involved in the evaluation step of MRGP obtains the LASSO linear fit of the problem. This initialization strategy together with elitism ensures that the solution provided by MRGP will be at least as accurate as the LASSO fit with respect to training data.

3.4 Evaluation Parallelism

The evaluation of the population of candidate models is computed in a multi-threaded fashion following a Master-Worker model. Each worker is charged with the evaluation of a subset of the population and is executed by a different CPU thread. Once the evaluation of the subpopulation is performed, the worker returns the corresponding fitness values. The number of threads (4 in Fig. 1) is set as a parameter and allows to exploit the multi-core *flavors* offered by cloud computing providers.

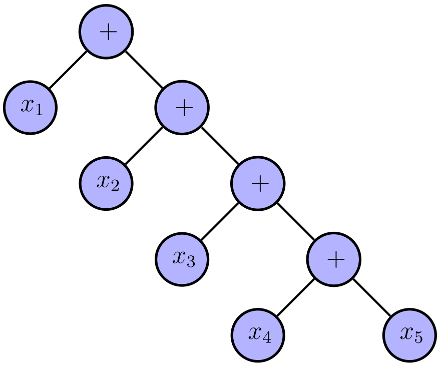


Fig. 3 MRGP allows to seed the initial population with a linear combination of the variables of the problem. In this case, the depicted problem is composed of 5 explanatory variables

4 Ensemble Learning

FlexGP executes independent copies of the MRGP algorithm, each trained with a different sample of the data and run with different parameters. At any moment of the run, it is possible to build a meta-model (ensemble) by means of a model fusion process.

4.1 Data Management

The targeted data D is split into training set D_{tr}, fusion training data D_f, and test set D_{te}. GP instances learn from a sample of D_{tr} while D_f is employed to filter and fuse the models obtained in the different cloud nodes. Finally, D_{te} is reserved to test the accuracy of the fused model.

4.2 Data and Parameter Factoring

We define two parameters n and p for the local copies of the regression algorithm (see Table 1). The parameter n corresponds to the size of D^i_{tr}, the training data used at the ith copy of the regression algorithm. D^i_{tr} is constructed by sampling without replacement n times from D_{tr}. The other parameter, p, is the size of the feature set F presented to the local learner.

We build F by drawing p samples from the features of the dataset without replacement. This strategy promotes the exploration of different combinations of the variables of the problem in our cloud runs.

Data parallel strategies result in an ensemble of models, each obtained with a different subset of the data. The isolation of the different copies of the algorithm helps diversifying our runs and provides robustness to the learning process.

4.3 Building the Final Model

FlexGP provides capability to generate *online*, i.e. at any moment of the run, a meta-model that combines the predictions of the models retrieved from the cloud nodes. To obtain the final model, we perform a two-step process: *model filtering* and *model fusion*.

Table 1 FlexGP parameters and their definition

Parameter	Definition
n	number of examples
p	size of feature set

4.3.1 Model Filtering

Each GP node stores the best model per generation, i.e. the model exhibiting the lowest error with respect to the received training data. The motivation to save the best model per generation is that models from advanced generations can overfit the data while some of the models obtained previously might exhibit a better generalization capability, i.e. a better accuracy with respect to unseen data. To build a meta-model, the stored models (best per generation and cloud node) are retrieved and evaluated against the fusion training data D_f to obtain their Mean Squared Error. The o models exhibiting the lowest error with respect to D_f are then selected as the best models of the run and will be used in the fusion process.

4.3.2 Model Fusion

In [10], we implemented and compared a number of fusion techniques and finally decided upon the algorithm Adaptive Regression by Mixing (ARM) introduced by Yang [11]. ARM allows to fuse a set of models M according to an estimation of their accuracy. The fused model z obtained with ARM is a linear combination of the models $m \in M$. Given a test sample $\overline{X_j}$, the prediction $\hat{z}_j$ issued by the fused model is the weighted average of model predictions $\hat{z}_j = \sum_{m=1}^{o} W_m \hat{Y}_{mj}$. Thus, the fusion process consists of learning the weight W_m for each model. Let $r = |D_f|$ be the size of the fusion training set, and $o = |\mathrm{M}|$ be 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_f randomly into two equally sized subsets $D_f^{(1)}$ and $D_f^{(2)}$.

Step 2: For each model m, evaluate σ_m^2 which is the maximum likelihood estimate of the variance of the errors $\overline{e}_m$ on $D^{(1)}$, $\overline{e}_m = \left\{\hat{Y}_{mj} - Y_j | \overline{X_j}, Y_j \in D_f^{(1)}\right\}$. Compute the sum of squared errors on $D^{(2)}$, $\beta_m = \sum_{j=\frac{r}{2}+1}^{r} \left(\hat{Y_{mj}} - Y_j\right)^2$.

Step 3: Estimate the weights using:

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

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

Transformation for large r: For large values of r, the calculation of the weights as given by (1) encounters an underflow error. To avoid this problem we equivalently compute the weights using (2) and (3).

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

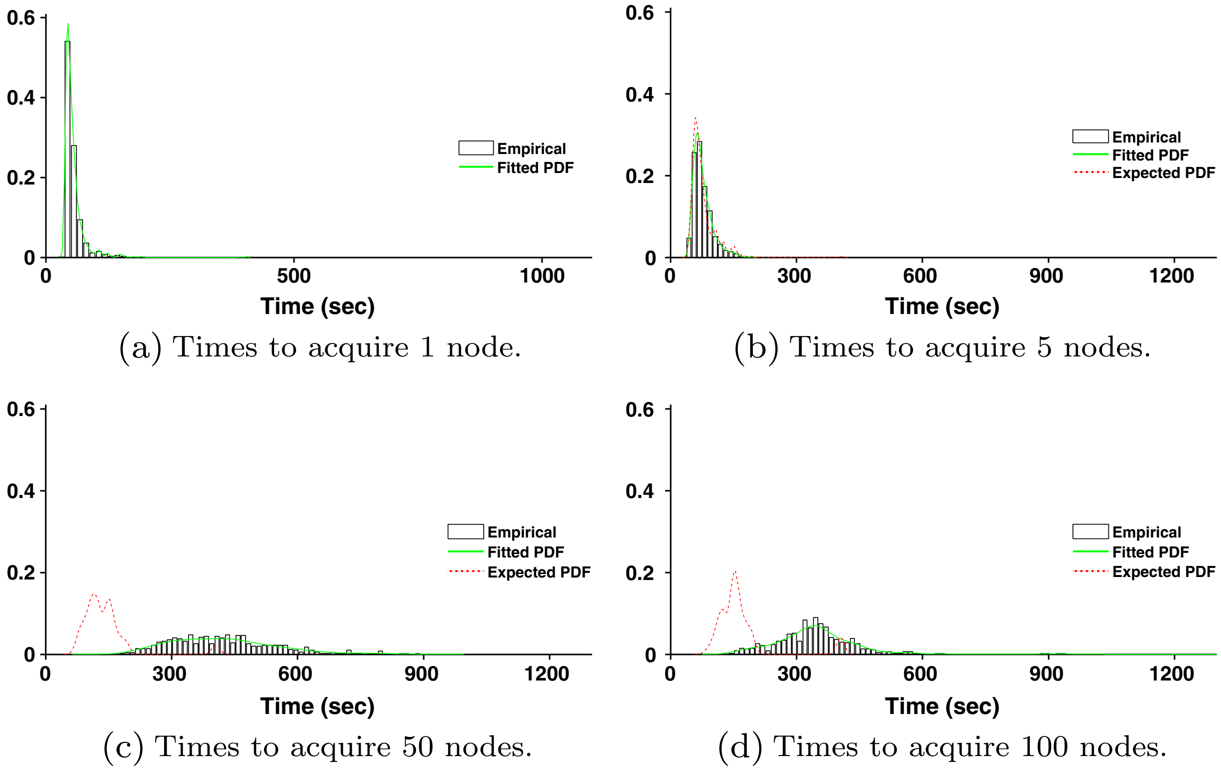
$$W_m = exp\left(A_m - log\left(\sum_{q=1}^{o} A_q\right)\right) \quad (3)$$

5 Decentralized Machine Learning Platform for Running MRGP

FlexGP implements a distributed launch protocol and a decentralized, fault tolerant communication layer. For extensive details of the methods described in this section, the reader is referred to the work by Derby [12].

5.1 Launch Protocol

In designing FlexGP's launch protocol, we started by studying the severity of latency in acquiring cloud instances. 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. 4a. If we optimistically assume a batch request of n instances



(a) Times to acquire 1 node.

(b) Times to acquire 5 nodes.

(c) Times to acquire 50 nodes.

(d) Times to acquire 100 nodes.

Fig. 4 Probability distribution functions (PDF) of times to acquire nodes

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. 4. 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.

This discrepancy indicates that smaller batch requests achieve closer to optimal latency than larger requests. In light of this observation, we draw two important guidelines to design the launch protocol:

1. Our system ought to emphasize small batch requests over large ones
2. Because acquiring many (50 or 100) instances may take significantly longer than acquiring the first 10 instances, we should start running MRGP 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.[1]

Algorithm 1 NodeStart(n, R)

```
$n$: nodes to launch, $R$: list of ancestor IP addresses
$\Psi$: launch parameters, $\Pi$: GP meta-parameters
$ip \leftarrow$ last($R$)
retrieve($ip$, $\Psi$, $\Pi$)
$R \leftarrow$ 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($R$)
MRGPCompute()
```

FlexGP implements a robust, 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 Section 5.2), respectively. The GP meta-parameters, Π, are used to determine the parameterization of each GP learner (see Section 4). These steps are detailed in the NodeStart function in Algorithm 1.

Figure 5 left illustrates how FlexGP would launch 7 instances 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 5 right 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 MRGP and then continues into the IPDiscovery algorithm, as described in Section 5.2. This enables us to run GP concurrent with IP discovery and network discovery.

The protocol is tolerant of node failures: 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 Fig. 5, if node X failed to launch properly, nodes Y and Z will never be requested, but there is no impact 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. However, 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, FlexGP enables us to ask an existing node to run the startup protocol with new parameters. This way, the node 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.

5.2 Distributed IP Discovery

Cloud-scale systems need an established network to robustly extract information and results. As we

[1] This is the case in our private OpenStack cloud where we experience frequent request failures.

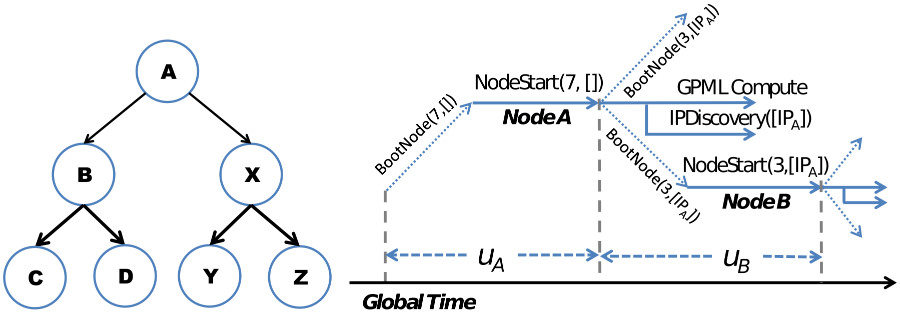


Fig. 5 A view of the launch of FlexGP for 7 nodes. Left: an initial node is launched and it brings up 2 more, which in turn bring up 2 more each, in a cascading fashion. Right: timeline of booting and launching of instances. After starting more nodes, node A begins computation

observed in Section 5.1, the latency for a many-node acquisition is quite large, therefore we need to establish a communication network and start the learning process before the last of the instances is acquired. To reduce the latency while still achieving the networking requirements of FlexGP, we design a distributed IP discovery protocol. Note, we focus here on the initial bootstrapping of the network, i.e. the "IP discovery" problem. This is separate from the problem of creating particular topologies in P2P networks as in [13].

Algorithm 2 IPDiscovery(R)

```
$\Lambda \leftarrow R$
loop
    $\lambda \leftarrow$ set of new messages received
    for $m$ in $\lambda$ do
        if $m.type$ is RequestIPList then
            $\Lambda \leftarrow$ merge($\Lambda$, $m.\Lambda$)
            RespondIPList($m.ip$, $\Lambda$)
        else if $m.type$ is RespondIPList then
            $\Lambda \leftarrow$ merge($\Lambda$, $m.\Lambda$)
    if len($\Lambda$)< $\Psi.p$ then
        $\epsilon \leftarrow$ random($\Lambda$)
        RequestIPList($\epsilon$)
```

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 the address passing protocol shown in 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.

6 Bases of Comparison

The experiments presented in this paper aim to compare FlexGP to state-of-the-art regression approaches. In this section, we provide concise descriptions of the compared approaches and the experimental parameters we used. It also reviews the dataset we used for comparison.

6.1 MRGP vs. Other Single-Desktop Regression Algorithms

As a foundation, we compare the learner currently used by FlexGP, MRGP, in standalone mode to other single-desktop regression algorithms to justify the investment in scaling it on the cloud. We consider Multiple Linear Regression, Vowpal Wabbit, Feed-Forward Neural Networks, and three validated GP techniques including the commercial tool Eureqa. This analysis will help the reader identify the best regression algorithm according to his/her own needs. For instance, Feed-Forward Neural Networks provide highly accurate predictions but sacrifice transparency of the models. The parameter settings of MRGP are summarized in Table 2.

Multiple Linear Regression: We obtain a linear model of the data using the least squares approach. In the following, this approach is referred to as *Multiple Linear Regression*.

Vowpal Wabbit (VW): Vowpal Wabbit is a fast out-of-core machine learning tool [14]. VW implements an online learning algorithm based on sparse gradient descent on a user-selected loss function (see [15]). VW generates linear models of large datasets (that might not fit in RAM) in minimal time. We consider three different configurations

Table 2 Parameters settings of the Symbolic Regression Strategies based on Genetic Programming

Parameter	DynEq-GP	MOGP/MOGP-opt	MRGP	Eureqa
pop size	1000	1000	1000	–
selection	Dynamic Eq. with tournament selection	NSGAII with crowded Tournament	NSGAII with crowded Tournament	–
crossover	Single Point Crossover	Single Point Crossover	Single Point Crossover	–
mutation	Subtree mutation	Subtree mutation	Subtree mutation	–
Error	MSE	MSE	Multiple regression error	MSE
Complexity	–	Subtree Complexity	Subtree Complexity	–
Threads	1	1/4	4	4

corresponding to three values of the *learning rate decay* parameter:

- VW-0.5D: VW with a *learning rate decay* of 0.5
- VW-0.1D: VW with a *learning rate decay* of 0.1
- VW-0.01D: VW with a *learning rate decay* of 0.01

Feed Forward Neural Networks: Neural Networks can be trained to mimic any input to output mapping. Feed Forward Neural Networks (FFNNs) are characterized by the number of hidden layers and the number of neurons per layer. The first layer has a connection to the network input. Each subsequent layer has a connection from the previous layer and the final layer produces the network's output. In this paper, we consider four different configurations (see Table 3):

- FFNN-1l-10n: FFNN with one hidden layer of 10 neurons
- FFNN-1l-20n: FFNN with one hidden layer of 20 neurons
- FFNN-1l-30n: FFNN with one hidden layer of 30 neurons
- FFNN-5l-20n: FFNN with five hidden layers of 20 neurons

We employ Matlab's Neural Network Toolbox [16] to train the FFNNs. The networks are trained via backpropagation with the Levenberg-Marquardt optimization.

Dynamic operator equalization Genetic Programming: We built a tree-based GP system with subtree mutation, single point crossover and tournament selection. We incorporated linear scaling [17, 18] and implemented dynamic operator equalization [19], a technique that ensures an appropriate size distribution of the population. This approach is referred to as *DynEq-GP* in the remaining of this work. The parameter settings of this approach are summarized in Table 2.

Multi-Objective Genetic Programming (MOGP): MOGP is also a tree-based GP system with subtree mutation, single point crossover, and post-hoc linear scaling. However, in this case we do not use an equalization operator. Instead, we implement a multi-objective strategy based on Non-dominated Sorting Genetic Algorithm II (NSGA-II). The algorithm minimizes both the error of the models and the subtree complexity measure proposed

Table 3 Parameters settings of the Feed-Forward Neural Networks

Parameter	FFNN-1l-10n	FFNN-1l-20n	FFNN-1l-30n	FFNN-5l-20n
hidden layers	1	1	1	5
neurons per layer	10	20	30	20
Training algorithm	Backprop.	Backprop.	Backprop.	Backprop.
Error	MSE	MSE	MSE	MSE
Threads	4	4	4	4

in [6]. The parameter settings of this approach are summarized in Table 2.

Optimized Multi-Objective Genetic Programming (*MOGP-opt*): The learning strategy is identical to MOGP. This version optimizes speed via a population compilation technique and multi-threading.

Eureqa Desktop: Eureqa [20] is a commercial Symbolic Regression tool that obtains short, readable models by optimizing a ratio of accuracy versus model complexity. Although Eureqa offers distributed implementations that run on private servers or Amazon EC2, we employ the desktop release to make an appropriate comparison with MRGP. Table 2 shows the parameter settings of this approach.

Due to the different nature of the compared algorithms, it is not straightforward to ensure equal conditions for all the algorithms. Instead, we set the stop criteria summarized in Table 4 and focus the analysis on the trade-off between accuracy and waiting time of the studied algorithms. The limit of one hour for the runs is motivated by the fact that cloud computing providers such as Amazon AWS do not charge fractional hours [21]. This means that, cost-wise, there is no benefit in reducing the running time below one hour.

We run all the algorithms on the same machine, equipped with an Intel Core-i7-3930K composed of 6 cores with hyper-threading running at 3.20GHz, 32GB of RAM, and a SSD drive. Note that we run 10 replicas of the algorithms that present a stochastic nature, i.e. Feed-Forward Neural Networks, and all GP-Based Symbolic Regression methods.

6.2 FlexGP vs. Single-Desktop MRGP

We analyze whether exploiting the different levels of parallelism of Genetic Programming allows better solutions to be obtained in a shorter time. FlexGP runs many instances of Multiple Regression Genetic Programming on the cloud, each with a different subset of the data and a sample of the explanatory variables of the problem. Moreover, each instance of MRGP is executed in a multi-threaded fashion to exploit the evaluation parallelism of GP.

This experimental setup assumes the need of transparent, accurate, non-linear models in a reduced time and great availability of compute resources. The latter assumption is valid in cloud environments where compute resources are inexpensive and readily accessible in large quantities.

6.3 Million Song Dataset

We employ the Million Song Dataset (MSD) year prediction challenge introduced in [22]. It is a popular regression problem in which the goal is to predict the release year of a large set of songs. The size and dimensionality of the dataset are challenging, since it is composed of 515K songs, each described with 90 features and a year label. We generate the splits D_{tr}, D_f, and D_{te} accounting for 70 %, 10 %, and 20 % of the data respectively (see Table 5). Note that the *producer effect* issue has been taken into account to perform all the splits.

7 Results

We first compare MRGP, the core learner of FlexGP (Section 3), with state-of-the-art regression algorithms. Then, we analyze whether FlexGP improves the results obtained with the single-desktop version of MRGP and whether it generates accurate models in a shorter time.

Table 4 Stop criteria and number of replicas of the different regression algorithms

Method	Stop Criterion	replicas
Multiple Linear Regression	–	1
Vowpal Wabbit	100 passes or convergence	1
Feed-Forward Neural Networks	100 epochs or convergence	10
GP-Based Symbolic Regression	end of generation after 1 hour	10

Table 5 MSD splits

	D_{tr}	D_f	D_{te}	Total
Exemplars	362K	51K	102K	515K
	70 %	10 %	20 %	100 %
Features	90	90	90	90
use single-desktop	training		testing	–
use FlexGP	training	fusion train	testing	–

7.1 Analysis of Single-Desktop Regression Algorithms

Table 6 shows the Mean Squared Error (MSE) and training time of the approaches compared in this analysis. We also depict in Fig. 6 the trade-off between waiting time and error the models obtained with Multiple Linear Regression, Vowpal-Wabbit, Feed-Forward Neural Networks, and the GP-based Symbolic Regression methods MOGP, MOGP-opt, Eureqa, and MRGP.

Linear Regression: Linear Regression methods obtain accurate models in a very reduced time. It is worth noting that VW obtains an accuracy very close to the least-squares linear fit in only 9.18 seconds.

Feed-Forward Neural Networks: FFNNs provide the most accurate predictions, obtaining a MSE of only 75.117 in the case of the network composed of 5 hidden layers, each with 20 neurons. The training time of FFNNs depends on the structure of the network, which in turn determines the number of parameters that need to be learned during the training process.

GP-Based Symbolic Regression: GP-based methods present different behaviors. DynEq-GP is impractical because the first generation is extremely time consuming (48323.84 seconds) and makes it hard to conform to the provided computational budget. MOGP obtains an accuracy similar to DynEq-GP but in a significantly shorter time. MOGP-opt outperforms both DynEq-GP

Table 6 Testing set Mean Squared Error (MSE) and learning time of state-of-the-art regression techniques on the Million Song Dataset

Method	Method	MSE	Time (seconds)
Linear Regression	Multiple Linear Regression	87.225	31.04
	VW-0.5D	107.308	7.74
	VW-0.1D	89.706	7.73
	VW-0.01D	87.233	9.18
Feed-Forward Neural Networks	FFNN-1l-10n	77.015	1312.96
	FFNN-1l-20n	76.474	2122.00
	FFNN-1l-30n	76.454	3231.05
	FFNN-5l-20n	75.117	4208.61
GP-Based Symbolic Regression	DynEq-GP	112.563	48323.84
	MOGP	112.603	3600.00
	MOGP-opt	106.780	3600.00
	Eureqa	96.862	3600.00
	MRGP	85.666	3600.00

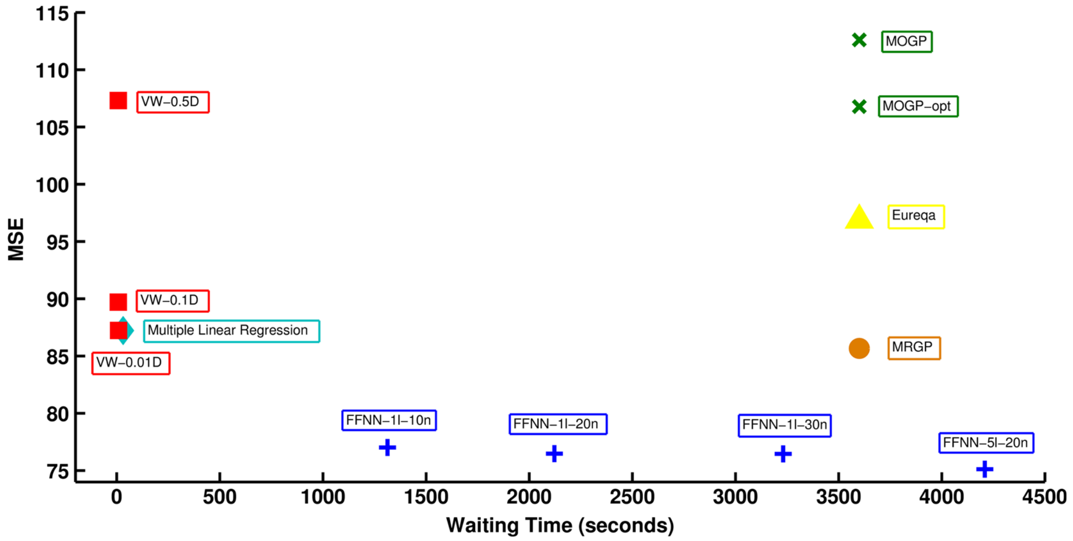


Fig. 6 MSE vs waiting time trade-off of the models obtained with Multiple Linear Regression, Vowpal-Wabbit, Feed-Forward Neural Networks, and GP-based Symbolic Regression methods MOGP, MOGP-opt,Eureqa, and MRGP

and MOGP but it remains highly time consuming and the achieved accuracy is far from that of linear models. Eureqa significantly outperforms the approaches based purely on Genetic Programming, i.e. DynEq-GP, MOGP, and MOGP-opt.

MRGP: FlexGP's local learner outperforms DynEq-GP, MOGP, MOGP-opt, and Eureqa given the training time limit of one hour. It also outperforms the accuracy achieved with linear regression methods. Therefore, it is the method that generates the most accurate transparent models.

A deeper analysis of the MRGP runs reveals that, due to the high cost of its fitness evaluation, fewer evaluations are executed with respect to the other GP-based methods. With this observation in mind, we posit that MRGP can benefit from a data-parallel deployment with our FlexGP framework to reduce its training time and hopefully improve its final accuracy.

It is worth noting that, in previous works, we have employed FlexGP to run different GP-Based Symbolic Regression methods, namely DynEq-GP [10, 23], MOGP, and MOGP-opt [24] on MSD and other benchmarks. Given that MRGP outperforms these three methods, we expect to improve upon previous results when we deploy MRGP with the FlexGP platform.

7.2 Ensemble Learning with FlexGP

We deploy the Multiple Regression Genetic Programming learner with FlexGP on our private OpenStack development cloud. We study two different FlexGP configurations corresponding to two learning strategies. The parameters of these two configurations are summarized in Table 7 and detailed in the following:

FlexGP-DATA: We run 100 copies of the algorithm, each learning from a different 10 % split of the training data. We refer to this configuration as *FlexGP-DATA*.

FlexGP-DATA-VARS: In addition, we analyze whether factoring explanatory variables helps improving the accuracy of the fused model to solve this particular problem. We also run 100 FlexGP nodes, each with a 10% split of the data and a random sample of 50 % of the variables of the problem. We call this second configuration *FlexGP-DATA-VARS*.

In both cases, all the cloud nodes are run with the *s1.4core* flavor, that is, a virtual machine with the following specs: 4 VCPUs, 2GB RAM, and 10.0GB Disk. We exploit the multicore flavor by running each of the local copies of the algorithm in a 4-threaded fashion. Note that the runs are replicated 10 times.

Table 7 FlexGP configurations

Parameter	cloud nodes	flavor	exemplars n	feature set size p
FlexGP-DATA	100	s1.4core	10%	100%
FlexGP-DATA-VARS	100	s1.4core	10%	50%

We retrieve the models generated after 5, 10, 15, 30, 45, and 60 minutes and perform the filtering and fusion processes at each time step. The average error of the fused model (or *meta-model*) at the different time steps is shown in Table 8.

FlexGP-DATA vs FlexGP-DATA-VARS: *FlexGP-DATA* clearly outperforms *FlexGP-DATA-VARS*. In the first case, the error is progressively reduced over time and reaches 84.304 at the end of the run (60 minutes). On the other hand, the final error of *FlexGP-DATA-VARS* is 96.606. It appears, in this particular problem, that larger subsets, perhaps only the full set, of variables are required.

FlexGP vs MRGP: With respect to the single-desktop version, *FlexGP-DATA* improves the final accuracy of the fused model (84.304 vs. 85.666). Moreover, as soon as in the first 10 minutes of the run, *FlexGP-DATA* obtains a fused model with an average MSE of 85.222, an error lower than the obtained with the single-desktop MRGP running for one hour.

The results presented in this paper show that significant speedup can be obtained by deploying MRGP in a data-parallel manner with FlexGP. Moreover, when large datasets that do not fit in RAM are targeted, the memory footprint and running time of each instance do not increase when the learning data at each instance is kept constant.

We have also shown that the fused model built with FlexGP outperforms the models obtained with the single-desktop version of MRGP. This difference was, however, more significant in previous works (see [10]). In the referred work, the core learner was *Dynamic operator Equalization Genetic Programming* and the retrieved models performed poorly when evaluated individually. The fusion process improved the accuracy significantly by assigning appropriate weights to the different weak models. In the experiments presented in this paper, the fusion process via ARM enhances only marginally the performance of individual models. Two observations explain the observed behavior. First, the core learner MRGP presents low variability between runs, and yields competitive models with correlated predictions. Therefore, weighting these models via ARM does not change significantly the predictions made by the individual models. Second, we have verified that the models trained with a reduced subset (only 10 %) of the training data achieve a performance on unseen data similar to that of models trained with the complete training set. This can be caused by the technique employed to sample examples at each instance, which takes into account the *producer effect* and, as a result, generates splits that maintain the original distribution of the data.

8 Related Work

There is a large body of distributed EC research which focuses exclusively on the design of distributed, algorithmic models, like island-based GP [25], but are not designed to take advantage of a particular resource type or communication layer. Much of this work is only tangentially related to FlexGP, as we developed an EC platform which takes advantage of the cloud platform. The systems in [27–30] rely upon

Table 8 MSE of the fused model at different time steps of the FlexGP runs. The errors are averaged over 10 runs

Approach	MSE@5	MSE@10	MSE@15	MSE@30	MSE@45	MSE@60
FlexGP-DATA	86.102	85.522	85.480	84.921	84.643	84.304
FlexGP-DATA-VARS	98.896	97.428	97.355	96.446	96.220	96.606

MapReduce for parallelization. MapReduce is a powerful platform for distributed computation, but its dependence upon a distributed file system, single point of failure in the master and synchronization bottlenecks are not a good match for an iterative approach like Genetic Programming-based symbolic regression.

FlexGP's IP discovery is like other EC peer-to-peer systems. For example, the EvAg system introduced in [31, 32] 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 and input variables. Folino et al. [33] introduced peer to peer based design for building classifier ensembles.

Over the past two decades numerous researchers in the machine learning community have pursued the idea of generating a great quantity of models for the same data [34–38]. Similarly, the task of combining predictions from an ensemble of models has attracted attention in recent years. This follows from two observations: there is usually not a single explanation for the data, and multiple models cover the observation space in a more robust manner than a single model can. Initially researchers focused on methods which generated multiple models from the same data irrespective of what kind of learner was being used. These methods relied on repeated subset sampling methods with replacement, e.g. bagging introduced by Breiman [39] and iterative sampling methods, e.g. boosting proposed by Freund [40]. Other examples include random forests [41] and Adaboost [42]. In this work, in addition to these subset sampling techniques, we focus on how the changing the parameters of the base learner can generate multiple models. We also explore how subsampling can allow us to learn from smaller dataset that could potentially fit in the main memory in addition to reducing the time for each iteration in GP.

When looking at ensembles built using GP, most of the work has focused on classification [43–48]. In classification with GP, models are either built to output discrete class labels or are constructed to yield a continuous number which leads to a class label when converted into class probabilities. As demonstrated in [49], multiple class labels can be fused via majority vote or a sophisticated criterion.

Conversely, there has been very little research on regression ensembles; some examples include the works [50–52]. GP based regression ensembles present two challenges. First, in regression, due to the unconstrained nature of GP models one must perform multiple tests which can guarantee models' outputs are within a reasonable range for any unseen data point before the model is admitted into an ensemble. Second, in classical machine learning, many examine methods for combining ensembles of parametric models. These methods attempt to understand the differences in the models based on their parameters and/or produce a fused model by fusing the parameters. However, for a structure free, parameter free approach, one has to rely on developing a fusion model in the output space. To fuse outputs of models from such regression ensembles, most current approaches use simple averaging techniques. In FlexGP we utilize a fusion method called ARM introduced by Yang [11] which trains a meta model using a subset of data set aside for fusion training. The method is low overhead and produces a linear combination of the models in the ensemble. This achieves superior and more stable performance than simple averaging.

9 Conclusions

We have described FlexGP, the first Genetic Programming system to perform regression on large-scale datasets on the cloud via massive data-parallel ensemble learning. To overcome cloud failures, FlexGP implements an asynchronous, fault-tolerant cascaded launch protocol and a decentralized communication layer. It launches many copies of Multiple Regression Genetic Programming, a novel regression method that combines tree-based Genetic Programming with Lasso. The independent copies run with different parameters and learn from different samples of the data, thereby reducing the computational burden on each learner and generating a diverse ensemble of models. FlexGP allows the best models of the run to be retrieved *online*, and to build a meta-model by means of a model filtering and fusion process.

We demonstrate our approach with the Million Song Dataset year prediction challenge, a large regression problem in which the goal is to predict the release year of 515K songs. We first compare MRGP, i.e. FlexGP's local learner, against a variety of state-of-the-art regression methods. We show that MRGP outperforms all methods that provide transparent models, i.e. GP-based symbolic regression and linear regression methods, given a training time limit of one hour. Additionally, we deploy the MRGP learner with FlexGP in a massive data-parallel manner. The performed experiments show that exploiting the data, run, and evaluation levels of parallelism of Genetic Programming allows for more accurate solutions to be obtained in a shorter time.

We plan to release FlexGP and offer it to researchers in the need of a large-scale symbolic regression tool. We encourage the EC community to develop competitive regression methods and to use FlexGP to deploy them on the cloud in a data-parallel manner to tackle large-scale data problems.

Acknowledgements The authors would like to thank Dylan Sherry for his valuable contributions to the FlexGP project and Dr. Krzysztof Krawiec for his contributions to the MRGP method. The ALFA group gratefully recognizes the financial support of the Li Ka Shing Foundation and the G.E. Global Research Center. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of G.E.

References

1. Friese, M., Flasch, O., Vladislavleva, K., Bartz-Beielstein, T., Mersmann, O., Naujoks, B., Stork, J., Zaefferer, M.: Ensemble-based model selection for smart metering data. In: Proceedings of the 22nd Workshop Computational Intelligence, pp. 215–227. Dortmund, Germany (2012)
2. Schmidt, M., Lipson, H.: Distilling free-form natural laws from experimental data. Science **324**(5923), 81–85 (2009)
3. Choudhury, A., Nair, P.B., Keane, A.J., et al.: A data parallel approach for large-scale gaussian process modeling. In: Proceedings of the Second SIAM International Conference on Data Mining, pp. 95–111. SIAM (2002)
4. Tibshirani, R.: Regression shrinkage and selection via the lasso. J. R. Stat. Soc. Series B **58**, 267–288 (1994)
5. Arnaldo, I., Krawiec, K., O'Reilly, U.M.: Multiple regression genetic programming. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, GECCO '14, pp. 879–886. ACM, New York (2014)
6. Vladislavleva, E.: Model-based problem solving through symbolic regression via pareto genetic programming. Ph.D. thesis, Tilburg University, Tilburg, the Netherlands (2008)
7. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002). doi:10.1109/4235.996017
8. Ganjisaffar, Y.: Lasso4j. https://code.google.com/p/lasso4j/ (2014)
9. Friedman, J.H., Hastie, T., Tibshirani, R.: Regularization paths for generalized linear models via coordinate descent. J. Stat. Softw. **33**(1), 1–22 (2010)
10. Veeramachaneni, K., Derby, O., Sherry, D., O'Reilly, U.M.: Learning regression ensembles with genetic programming at scale. In: Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13, pp. 1117–1124. ACM, New York (2013)
11. Yang, Y.: Adaptive regression by mixing. J. Am. Stat. Assoc. **96**(454), 574–588 (2001)
12. Derby, O.: FlexGP: a scalable system for factored learning in the cloud. Master's thesis, Massachusetts Institute of Technology (2013)
13. Jelasity, M., Montresor, A., Babaoglu, O.: Gossiping in distributed systems. Comput. Netw. **53**(13), 2321 (2009). doi:10.1016/j.comnet.2009.03.013
14. Langford, J.: Vowpal wabbit. http://hunch.net/vw/ (2014)
15. Langford, J., Li, L., Zhang, T.: Sparse online learning via truncated gradient. J. Mach. Learn. Res. **10**, 777–801 (2009)
16. MathWorks: Neural network toolbox. http://www.mathworks.com/products/neural-network/ (2014)
17. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., Costa, E. (eds.) Genetic Programming. Lecture Notes in Computer Science, vol. 2610, pp. 275–299. Springer, Berlin / Heidelberg (2003)
18. Vladislavleva, C., Smits, G.: Symbolic regression via genetic programming. Final Thesis for Dow Benelux BV (2005)
19. Silva, S., Dignum, S., Vanneschi, L.: Operator equalisation for bloat free genetic programming and a survey of bloat control methods. Genet. Program Evolvable Mach. **13**(2), 197–238 (2012)
20. Eureqa desktop: http://www.nutonian.com/products/eureqa/ (2014)
21. Amazon web services (AWS): http://aws.amazon.com/ (2014)
22. Bertin-Mahieux, T., Ellis, D.P., Whitman, B., Lamere, P.: The million song dataset. In: Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011) (2011)
23. Sherry, D., Veeramachaneni, K., McDermott, J., O'Reilly, U.M.: Flex-GP: genetic programming on the cloud. In: Chio, C.D., Agapitos, A., Cagnoni, S., Cotta, C., Vega, F.F.d., Caro, G.A.D., Drechsler, R., Ekart, A., Esparcia-Alcazar, A.I., Farooq, M., Langdon, W.B., Merelo-Guervos, J.J., Preuss, M., Richter, H., Silva, S., Simes, A., Squillero, G., Tarantino, E., Tettamanzi, A.G.B., Togelius, J., Urquhart, N., Uyar, A., Yannakakis, G.N. (eds.) Applications of Evolutionary Computation no. 7248 in Lecture Notes in Computer Science, pp. 477–486. Springer, Berlin Heidelberg (2012)
24. Sherry, D.J.: FlexGP 2.0: multiple levels of parallelism in distributed machine learning via genetic programming. Master's thesis, Massachusetts Institute of Technology (2013)

25. Fernández, F., Tomassini, M., Vanneschi, L.: An empirical study of multipopulation genetic programming. Genet. Program Evolvable Mach. **4**(1), 21–51 (2003). doi:10.1023/A:1021873026259
26. Fazenda, P., McDermott, J., O'Reilly, U.M.: A library to run evolutionary algorithms in the cloud using MapReduce. In: Chio, C., Agapitos, A., Cagnoni, S., Cotta, C., Vega, F., Caro, G., Drechsler, R., Ekárt, A., Esparcia-Alcázar, A., Farooq, M., Langdon, W., Merelo-Guervós, J., Preuss, M., Richter, H., Silva, S., Simes, A., Squillero, G., Tarantino, E., Tettamanzi, A., Togelius, J., Urquhart, N., Uyar, A., Yannakakis, G. (eds.) Applications of Evolutionary Computation. Lecture Notes in Computer Science, vol. 7248, pp. 416–425. Springer, Berlin Heidelberg (2012)
27. Wang, S., Gao, B.J., Wang, K., Lauw, H.W.: 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, pp. 1083–1084. ACM, New York (2011)
28. Verma, A., Llora, X., Goldberg, D., Campbell, R.: Scaling genetic algorithms using MapReduce. In: Intelligent Systems Design and Applications, 2009. ISDA '09. Ninth International Conference on, pp. 13–18 (2009)
29. Verma, A., Llora, X., Venkataraman, S., Goldberg, D., Campbell, R.: Scaling eCGA model building via data-intensive computing. In: Evolutionary Computation (CEC), 2010 IEEE Congress on, pp. 1–8 (2010)
30. Huang, D.W., Lin, J.: 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, pp. 780–785 (2010)
31. Jiménez Laredo, J., Lombraña González, D., Fernández de Vega, F., García Arenas, M., Merelo Guervós, J.: A peer-to-peer approach to genetic programming. In: Silva, S., Foster, J., Nicolau, M., Machado, P., Giacobini, M. (eds.) Genetic programming. Lecture Notes in Computer Science, vol. 6621, pp. 108–117. Springer, Berlin Heidelberg (2011)
32. Laredo, J., Eiben, A., Steen, M., Merelo, J.: Evag: a scalable peer-to-peer evolutionary algorithm. Genet. Program Evolvable Mach. **11**, 227–246 (2010). doi:10.1007/s10710-009-9096-z
33. Folino, G., Forestiero, A., Spezzano, G.: A jxta based asynchronous peer-to-peer implementation of genetic programming. J. Softw. **1**(2), 12–23 (2006)
34. Perrone, M.P., Cooper, L.N.: When networks disagree: Ensemble methods for hybrid neural networks. In: Mammone, R. (ed.) Neural Networks for Speech and Image processing, pp. 126–142. Chapman and Hall (1993)
35. Krogh, A., Vedelsby, J.: Neural network ensembles, cross validation, and active learning. Adv. Neural Inf. Process. Syst. **7**, 231–238 (1995)
36. Quinlan, J.R.: Bagging, boosting, and C4.5. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI'96, vol. 1, pp. 725–730. AAAI Press (1996)
37. Dietterich, T.: An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. Mach. Learn. **40**(2), 139–157 (2000)
38. Dietterich, T.: Ensemble methods in machine learning. In: Multiple Classifier Systems. Lecture Notes in Computer Science, vol. 1857, pp. 1–15. Springer, Berlin Heidelberg (2000)
39. Breiman, L.: Bagging predictors. Mach. Learn. **24**(2), 123–140 (1996)
40. Freund, Y., Schapire, R.: Experiments with a new boosting algorithm. In: Machine learning international conference, pp. 148–156. Morgan Kauffman Publishers, Inc. (1996)
41. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
42. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. Syst. Sci. **55**(1), 119–139 (1997)
43. Imamura, K., Soule, T., Heckendorn, R., Foster, J.: Behavioral diversity and a probabilistically optimal GP ensemble. Genet. Program Evolvable Mach. **4**(3), 235–253 (2003)
44. Bhowan, U., Johnston, M., Zhang, M., Yao, X.: Evolving diverse ensembles using genetic programming for classification with unbalanced data. IEEE Trans. Evol. Comput. **17**(3), 368–386 (2013). doi:10.1109/TEVC.2012.2199119
45. Langdon, W., Barrett, S., Buxton, B.: Combining decision trees and neural networks for drug discovery. In: Foster, J., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A. (eds.) Genetic Programming. Lecture Notes in Computer Science, vol. 2278, pp. 60–70. Springer, Berlin Heidelberg (2002)
46. Johansson, U., Löfström, T., König, R., Niklasson, L.: Genetically evolved trees representing ensembles. In: Artificial Intelligence and Soft Computing–ICAISC 2006, pp. 613–22 (2006)
47. Folino, G., Pizzuti, C., Spezzano, G.: Mining distributed evolving data streams using fractal GP ensembles. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A. (eds.) Genetic Programming. Lecture Notes in Computer Science, vol. 4445, pp. 160–169. Springer, Berlin Heidelberg (2007)
48. Lanzi, P.L.: XCS with stack-based genetic programming. In: Sarker, R., Reynolds, R., Abbass, H., Tan, K.C., McKay, B., Essam, D., Gedeon, T. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, pp. 1186–1191. IEEE Press, Canberra (2003)
49. Kittler, J., Hatef, M., Duin, R., Matas, J.: On combining classifiers. IEEE Trans. Pattern Anal. Mach. Intell. **20**(3), 226–239 (1998)
50. Iba, H.: Bagging, boosting, and bloating in genetic programming. In: Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E. (eds.) Proceedings of the Genetic and Evolutionary Computation

Conference, vol. 2, pp. 1053–1060. Morgan Kaufmann, Orlando, Florida (1999)

51. Veeramachaneni, K., Vladislavleva, K., Burland, M., Parcon, J., O'Reilly, U.M.: Evolutionary optimization of flavors. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation, pp. 1291–1298. ACM (2010)

52. Kotanchek, M., Smits, G., Vladislavleva, E.: Trustable symbolic regression models: using ensembles, interval arithmetic and pareto fronts to develop robust and trust-aware models. In: Riolo, R., Soule, T., Worzel, B. (eds.) Genetic Programming Theory and Practice V. Genetic and Evolutionary Computation Series, pp. 201–220. Springer, US (2008)