Genetic Programming and Evolvable Machines
https://doi.org/10.1007/s10710-020-09383-4

Learning feature spaces for regression with genetic programming

William La Cava[1] · Jason H. Moore[1]

Received: 15 October 2019 / Revised: 17 January 2020

Abstract
Genetic programming has found recent success as a tool for learning sets of features for regression and classification. Multidimensional genetic programming is a useful variant of genetic programming for this task because it represents candidate solutions as sets of programs. These sets of programs expose additional information that can be exploited for building block identification. In this work, we discuss this architecture and others in terms of their propensity for allowing heuristic search to utilize information during the evolutionary process. We investigate methods for biasing the components of programs that are promoted in order to guide search towards useful and complementary feature spaces. We study two main approaches: (1) the introduction of new objectives and (2) the use of specialized semantic variation operators. We find that a semantic crossover operator based on stagewise regression leads to significant improvements on a set of regression problems. The inclusion of semantic crossover produces state-of-the-art results in a large benchmark study of open-source regression problems in comparison to several state-of-the-art machine learning approaches and other genetic programming frameworks. Finally, we look at the collinearity and complexity of the data representations produced by different methods, in order to assess whether relevant, concise, and independent factors of variation can be produced in application.

Keywords Representation learning · Feature construction · Variation · Regression

✉ William La Cava
lacava@upenn.edu

Jason H. Moore
jhmoore@upenn.edu

[1] University of Pennsylvania, 3700 Hamilton Walk, Philadelphia, PA, USA

1 Introduction

Genetic programming (GP) is a method that attempts to solve problems by identifying and integrating the components of programs that contribute to good solutions. When applied to classification and regression problems, the components of programs, i.e. its building blocks, are analogous to engineered features. Because of this, we expect GP solutions to classification and regression problems to contain building blocks that explain the underlying factors of variation producing the observed response that is modelled. In the broader machine learning (ML) community, automatic engineering of feature spaces is referred to as *representation learning* [4]. Representation learning is a fundamental challenge in ML due to its computational complexity and the importance of data representation to the quality of models that can be trained. Our interest here is a variant of GP we refer to as multidimensional GP, i.e. MGP. MGP makes the relationship between building block discovery and representation learning explicit by optimizing a set of programs, each of which is an independent feature in the ML model. In this paper we discuss why the MGP architecture is suited to the task of representation learning, and study several techniques for improving the quality of data representations that MGP can learn.

In order to assess the quality of data representations, we first must establish a notion of what makes a representation good. First and foremost, a good representation allows a model to be trained that generalizes to unseen data better than a model trained directly on the raw attributes. Second, a good representation identifies independent components of variation in the data that cause the process response. Third, an ideal representation is succinct to aid in interpretation and intelligibility. Ideally, a representation only has as many features as there are independent factors controlling the process. Both the methods developed in this paper and the related discussion are centered around these three motivations.

The paper summarizes and extends our previous work [44, 46] in which we proposed and developed a multidimensional GP framework called the Feature Engineering Automation Tool (FEAT). The paper is organized as follows. First we present a brief background on the many methods that have been proposed to apply GP to feature construction/representation learning, focusing on those techniques that use ML as a heuristic for identifying and promoting building blocks. We discuss different architectures that motivate our focus on MGP. We consider FEAT in the context of related GP and neural network methods. In Sect. 3 we propose a set of methods hypothesized to improve our ability to identify accurate, succinct and disentangled representations. These methods consist of new multi-objective approaches as well as new semantic crossover operators. We conduct a series of experiments within this framework to study the effect of these methods on representation quality. We conduct an experiment at first on eight regression problems, considering full hyperparameter tuning, and analyze the representations that are produced with and without the new crossover methods. Finally, we benchmark the new methods against many ML and GP methods on more than 100 open source regression problems. We find that the new methods of crossover lead to state-of-the-art results for regression. We discuss the implications of these results and offer viewpoints for further analysis in the conclusions.

2 Background

The GP community has long been interested in feature construction, and it has been studied with various architectures. In fact, if the goal is to identify a single feature (or multiple features in the multiclass case [58]), GP can be applied directly without major changes [55]. These approaches make use of information-theoretic measures to estimate how good a program is likely to be as a feature in a larger model. Despite requiring minimal changes to GP's methodology, the optimization of single features lacks the ability to control for the multivariate context in which they are typically used.

An alternative approach that has been studied is to treat each individual in the population as a feature, and to optimize an ensemble model of the entire population [1, 2, 12, 42, 43, 52]. With this approach, only a single regression model must be trained per generation, which demands minimal overhead. However, it is not well understood how to properly select and vary the features evolved by such a process. Since each individual is a feature, its fitness depends heavily on the current population. Furthermore, a desirable set of features should be orthogonal to each other so that the representation is well-conditioned; in contrast, convergent evolutionary processes aim to make each individual, i.e. feature, the same. To overcome issues of collinearity and a convergent search process, the following ideas have been proposed. In evolutionary feature synthesis (EFS) [2], features are selected proportionally to their coefficient in a regularized linear model; in order to prevent multicollinearity, correlation thresholds are implemented during variation to keep children different from their parents. In the feature engineering wrapper (FEW) [42, 43], multicollinearity is selected against by using a survival version of ϵ-lexicase selection to choose features. In Kaizen GP [12, 52], individuals are only added to the model if they pass a significance test, in a hill climbing fashion. Another option is to not use an evolutionary updating scheme at all, but rather to create a large set of random features and fit an ML model to this, as in Fast Function Extraction (FFX) [50]. More recently, Vanneschi et. al. explored one step linear combinations of random programs [78], experimentally showing that they often lead to overfitting.

Rather than building a model from the entire population, one could apply an ML method to the entire program trace as a means of identifying building blocks [38]. Multiple regression GP (MRGP) [1] defines a program's behavior as the Lasso [76] estimate generated over the entire program's trace. One downside of this approach is the likely presence of highly correlated features in the program trace, leading to an ill-conditioned regression matrix. In a similar vein to MRGP, Behavioral GP [39] extracts information from the entire program trace, this time using a decision tree algorithm to identify important building blocks, which are stored in an archive for re-use. In both algorithms, the key insight is to use ML with program traces to undo the complex masking effect that program execution has on the behavior of building blocks that are downstream from other operations in the program (for further discussion on the topic of program traces see [37]).

The multidimensional framework used by our studied approach, MGP, is in some sense in between the ensemble techniques and the program trace techniques described above. Individuals are represented as sets of separate subprograms, usually trees. Unlike population-wide models, the fitness of each individual is directly related to its model predictions, and individuals in the population benefit from typical evolutionary optimization processes. Unlike program trace-based methods, by using multi-output individuals, MGP exposes *independent components* of the total program behavior to the ML process that produces the model. As a result, building blocks are easier to isolate and share among the population in direct ways.

Examples of MGP include Krawiec's method [36], multigene GP [22, 68, 69], M2GP [31], M3GP [56], e-M3GP [70], M4GP [45], and FEAT [46]. In all of these methods, individuals in the population produce a set of corresponding outputs that are then fed into a deterministic ML method to produce the program's regression or classification estimates. In the case of M2GP, M3GP, and M4GP, classification proceeds using a nearest centroid classifier [77], whereas linear regression methods are used for regression with M3GP [57] and FEAT.

Although a number of methods have been proposed in the MGP paradigm, they have not made much use of the semantics of independent building blocks in each program that this architecture creates. An exception is our work on FEAT [44, 46], in which we use the coefficient magnitude to weight probabilities of mutation. In our first study on FEAT [46], we proposed using multiple objectives to leverage the architecture of MGP to a larger degree than in previous studies. In our second study [44], we looked at semantic variation operators to achieve the same goal. The main contributions of this paper are (1) to summarize the proposed methods and findings of previous papers, (2) extend the analysis and description of FEAT, and (3) empirically compare all the variants of FEAT that have been proposed to each other, and to state-of-the-art GP and ML methods.

3 Methods

The goal of regression is to build a predictive model $\tilde{y}(\mathbf{x})$ using N paired examples $\mathcal{T} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. The regression model $\tilde{y}(\mathbf{x})$ associates the inputs $\mathbf{x} \in \mathbb{R}^d$ with a real-valued output $y \in \mathbb{R}$. The goal of feature engineering/representation learning is to find a new representation of $\mathbf{x}$ via a m-dimensional feature mapping $\boldsymbol{\phi}(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}^m$, such that the model $\hat{y}(\boldsymbol{\phi}(\mathbf{x})) : \mathbb{R}^m \rightarrow \mathbb{R}$ outperforms the model $\tilde{y}(\mathbf{x})$ by some pre-defined metric.

In MGP, each individual in the population is a candidate representation, $\boldsymbol{\phi}(\mathbf{x})$, consisting of a list of programs $[\phi_1, \dots, \phi_m]$. As an example, the individual

$$[(+\ (x_1)\ (x_2)),\ (\cos\ (x_3)\ ),\ (\exp\ (\text{cube}\ (x_1)))]$$

would encode a representation with three features: $(x_1 + x_2)$, $\cos(x_3)$, and $\exp(x_1^3)$. Throughout the paper, we refer to these subprograms ϕ as *features*, and use the word *attribute* to refer to the independent variables in $\mathbf{x}$.

MGP methods share this representation in common, and differ in terms of (1) the ML method used to generate the model prediction, i.e. $\hat{y}(\boldsymbol{\phi})$; (2) the crossover and mutation operators used; (3) the treatment of internal program weights; and 3) the selection process used.

We study a recent MGP method named FEAT [46], in which candidate features are parameterized by weights, $\boldsymbol{\theta}$, and used to fit a linear model

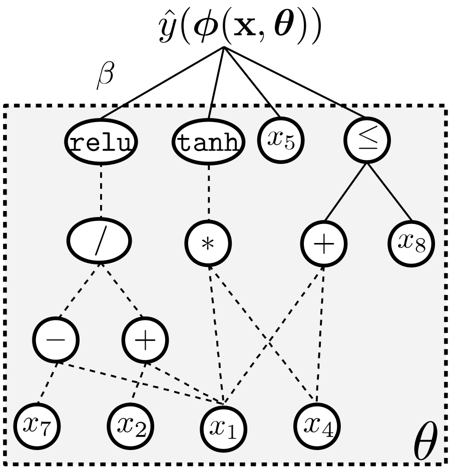
$$\hat{y} = \sum_{i=1}^{m} \beta_i \phi_i(\mathbf{x}, \boldsymbol{\theta}) \tag{1}$$

The coefficients $[\beta_1, \dots, \beta_m]$ are determined using ridge regression [29]. Note that each ϕ is normalized to zero mean, unit variance before ridge regression is applied. The fitness of each individual in FEAT is its mean squared error (MSE) on the training set.

FEAT constructs trees from elementary boolean- and continuous-valued functions and literals (see Table 1). FEAT differs from traditional symbolic regression

Table 1 Functions and terminals used to develop representations

Continuous functions	$\{+, -, *, /, ()^2, ()^3, \sqrt{()}$, sin, cos, exp, log, exponent, logit, tanh, gauss, relu$\}$
Boolean functions	$\{$and, or, not, xor, $=, <, \leq, >, \geq\}$
Terminals	$\{\mathbf{x}\}$



$$\hat{y}(\boldsymbol{\phi}(\mathbf{x}, \boldsymbol{\theta})) =$$

$$\beta_0[\mathtt{max}(0,\ \theta_0 \tfrac{\theta_1 x_7 - \theta_2 x_1}{\theta_3 x_2 + \theta_4 x_1})] + \beta_1[\tanh(\theta_5(\theta_6 x_1 * \theta_7 x_4)] + \beta_2(x_5) + \beta_3[(\theta_8 x_1 + \theta_9 x_4) \leq x_8]$$

Fig. 1 Example model representation in FEAT, illustrating the connection to neural networks. The input variables ($\mathbf{x}$) are represented as leaves, and each other node is an operator from Table 1. Weights are attached to each edge connecting a differentiable operator to its arguments within the gray box, shown with dotted lines. Those weights, $\boldsymbol{\theta}$, are learned via gradient descent using backpropagation. The final layer is fed into a linear model, $\hat{y}(\boldsymbol{\phi}(\mathbf{x}))$, parameterized by β. This example model is composed of four sub-programs, or features, varying in depth and composition

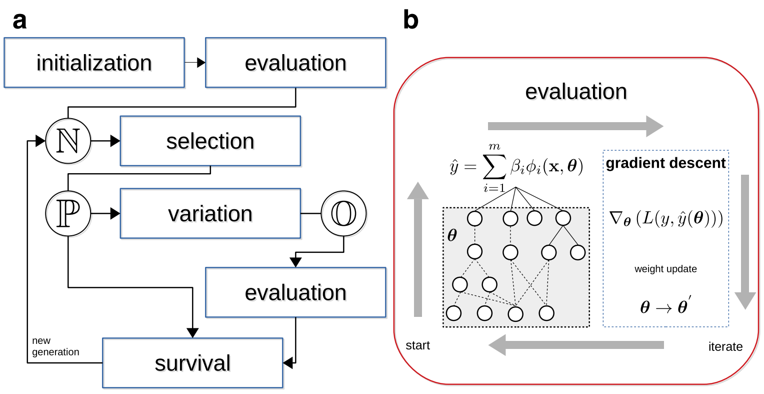

a
initialization
evaluation
ℕ
selection
ℙ
variation
𝕆
evaluation
new generation
survival
b
evaluation
$\hat{y} = \sum_{i=1}^{m} \beta_i \phi_i(\mathbf{x}, \boldsymbol{\theta})$
$\boldsymbol{\theta}$
gradient descent
$\nabla_{\boldsymbol{\theta}} (L(y, \hat{y}(\boldsymbol{\theta})))$
weight update
$\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}'$
start
iterate

Fig. 2 **a** The steps of the FEAT algorithm. Execution begins with initialization and evaluation, selection, variation, evaluation of offspring, and finally survival. **b** The evaluation process is shown. Individuals are evaluated and used to fit a linear model, and then gradient descent is performed for a set number of iterations to learn the internal weights $\boldsymbol{\theta}$. The final model and its loss, $L(y, \hat{y})$, are returned

(SR) which treats the parameters $\boldsymbol{\theta}$ as leaves. Instead, these weights are attached to the edges of all differentiable operators and updated each generation via gradient descent (see Fig. 1).

In Fig. 2 we show the execution steps involved in FEAT, described here. FEAT uses a typical $\mu + \lambda$ evolutionary updating scheme, where $\mu = \lambda = P$. The method optimizes a population of potential representations, $\mathbb{N} = \{n_1 \dots n_P\}$, where n is an "individual" in the population, iterating through these steps:

1. Fit a linear model $\hat{y} = \mathbf{x}^T \hat{\beta}$. Create an initial population $\mathbb{N}$ consisting of this initial representation, $\boldsymbol{\phi} = [x_1, \dots, x_d]$, along with $P - 1$ randomly generated representations. To initialize a random representation, a number of features (m) is chosen uniform-randomly from a user-specified max dimensionality. For each feature, the well-known "grow" method is used to build a program, with one notable change: the leaves of each program are sampled from $[x_1, \dots, x_d]$ proportionally to $\hat{\beta}$. See Sect. 3.1 for more details.
2. While the stop criterion is not met:
   - (a) Select parents $\mathbb{P} \subseteq \mathbb{N}$ using a selection algorithm (see Sect. 3.3).
   - (b) Apply variation operators to parents to generate P offspring $\mathbb{O}$; $\mathbb{N} = \mathbb{N} \cup \mathbb{O}$ (see Sect. 3.2).
   - (c) Reduce $\mathbb{N}$ to P individuals using a survival algorithm (see Sect. 3.3).
3. Select and return $n \in \mathbb{N}$ with the lowest error on a hold-out validation set.

Individuals are evaluated using an initial forward pass, after which each representation is used to fit a linear model (Eq. 1) using ridge regression [29]. The weights of the differentiable features in the representation are then updated using stochastic gradient descent.

In the following sections, we describe the specific components of FEAT, including the feedback mechanism, variation, and the selection and survival algorithms. We also present the variations of these methods that are the subject of experiments later in the paper.

3.1 Feedback

In order to promote building blocks, FEAT uses feedback from the ML process to bias the variation step. In a nutshell, the probability of a feature in $\boldsymbol{\phi}$ being mutated or replaced by crossover is inversely related the magnitude of its coefficient β in Eq. 1. Let $\tilde{\beta}_i(n) = |\beta_i| / \sum_i^m |\beta_i|$. The normalized coefficient magnitudes $\tilde{\beta} \in [0, 1]$ are used to define softmax-normalized probabilities. The probability of mutation for feature i in program n is denoted $PM_i(n)$, and defined as follows:

$$\begin{aligned} s_i(n) &= \exp(1 - \tilde{\beta}_i) / \sum_i^m \exp(1 - \tilde{\beta}_i) \\ PM_i(n) &= \gamma s_i(n) + (1 - \gamma)\frac{1}{m} \end{aligned} \tag{2}$$

Here, γ is a parameter that controls the amount of feedback from the weights that is used to bias the selection of feature i for mutation. When γ is zero, the $1/m$ term in Eq. 2 gives uniform mutation probability across features. In our experiments, we tune γ, and also test whether the softmax normalization of $s_i(n)$ is useful.

3.2 Variation

During variation, the representations are perturbed using a set of mutation and crossover methods. The baseline version of FEAT chooses among 6 variation operators that are as follows.

- *Point mutation* changes a node type to a random one with matching output type and arity.
- *Insert mutation* replaces a node with a randomly generated subtree of depth 1.
- *Delete mutation* removes a feature or replaces a sub-program with an input node, with equal probability.
- *Insert/delete dimension* adds or removes a new feature.
- *Sub-tree crossover* replaces a sub-tree from one parent with the sub-tree of another parent.
- *Dimension crossover* swaps two features between parents.

The exact probabilities of each variation operator will affect the performance of the algorithm, but for the sake of simplicity we use each operator with uniform probability. In the following section, we describe semantic crossover operators that are analyzed for their usefulness in our experiments.

3.2.1 Semantic crossover

The following two crossover methods are called *semantic* because they use information about the program's outputs to determine the recombination that occurs to produce a child from two parents. Both operators are based on the following observations. We have two parent representations, $\boldsymbol{\phi}_{p1}$ and $\boldsymbol{\phi}_{p2}$, with corresponding model outputs $\hat{y}_{p1}$ and $\hat{y}_{p2}$ that are linear combinations of their respective representations, as in Eq. 1. We want to produce the best combination of $\boldsymbol{\phi}_{p1}$ and $\boldsymbol{\phi}_{p2}$ for the child representation $\boldsymbol{\phi}_{c}$. Basically we can treat this as a feature selection problem, where we have features $\boldsymbol{\phi}_A = \boldsymbol{\phi}_{p1} \cup \boldsymbol{\phi}_{p2}$ and we want to pick the best. On one hand we could simply concatenate the feature sets, and generate a new model $\hat{y}(\boldsymbol{\phi}_A)$, which is the linear model fit to all features of both parents. This approach would lead to exponential growth in offspring, which would run against our goal of lowering complexity.

In lieu of that approach, we propose here what are essentially regularized versions of geometric semantic crossover [54] that constrain the number of features in the offspring to be of equal cardinality to $\boldsymbol{\phi}_{p1}$, i.e. $|\boldsymbol{\phi}_c| = |\boldsymbol{\phi}_{p1}|$. The first operator, best residual fit crossover (ResXO), chooses a feature from $\boldsymbol{\phi}_{p1}$ to be replaced, and then chooses the feature in $\boldsymbol{\phi}_{p2}$ that best approximates the residual of the model after removing this feature. The second operator, stagewise crossover (StageXO), uses forward stagewise regression [32] as a feature selection method to iteratively construct the offspring.

Best residual fit crossover (ResXO) Given parents $p1$ and $p2$, ResXO swaps a feature in $p1$ with the feature in $p2$ that most closely approximates the residual error of $p1$ with the selected feature removed. The child representation is denoted as $\boldsymbol{\phi}_c$. The steps are as follows:

1. Pick ϕ_d from $\boldsymbol{\phi}_{p1}$ using probabilities given by Eq. 2.
2. Calculate the residual of $p1$ without ϕ_d:

$$r = y - \hat{y}_{p1} - \beta_d \phi_d$$

3. Choose ϕ^* from $\boldsymbol{\phi}_{p2}$, which is the feature most correlated with r.
4. $\boldsymbol{\phi}_c = \boldsymbol{\phi}_{p1}$ with ϕ_d replaced by ϕ^*.

ResXO can be likened to a special case of semantic backpropagation [17, 26, 62], since it seeks to replace a component of the parent program with a subprogram most closely matching the desired semantics, given by r. Within the MGP framework, this backpropagation is very simple, and does not require complex inversion operations to be introduced. We expect that ResXO will also lead to lower correlations between features in $\boldsymbol{\phi}_c$ than in $\boldsymbol{\phi}_{p1}$. To understand why, consider that

$$r = y - \sum_{\phi_i \in \boldsymbol{\phi}_{p1} \setminus \phi_d} \beta_i \phi_i$$

Therefore r should have low correlation with the rest of $p1$'s representation. Assuming the replacement feature from $\boldsymbol{\phi}_{p2}$ closely matches r, it should also be uncorrelated with $\{\boldsymbol{\phi}_{p1} \backslash \phi_d\}$. Note that ResXO may produce an individual with higher squared error than its parents, since ϕ_d may be more correlated with r than ϕ^*.

Forward stagewise crossover (StageXO) Rather than restricting crossover to the replacement of a single feature, the crossover operator can be used to compile the set of features that iteratively reduce the target error using a forward stagewise crossover method we call StageXO. The procedure is as follows:

1. Set the initial residual equal to the target: $r = y$. Center means around zero for all ϕ.
2. Set $\boldsymbol{\phi}_A$ to be all subprograms in $\boldsymbol{\phi}_{p1}$ and $\boldsymbol{\phi}_{p2}$.
3. While $|\boldsymbol{\phi}_c| < |\boldsymbol{\phi}_{p1}|$:
    (a) Pick ϕ^* from $\boldsymbol{\phi}_A$ which is most correlated with r.
    (b) Compute the least squares coefficient b for ϕ^* fit to r.
    (c) Update $r = r - b\phi^*$
    (d) Add ϕ^* to $\boldsymbol{\phi}_c$.
    (e) Remove ϕ^* from $\boldsymbol{\phi}_A$.

Unlike feature selection methods like forward/backward stepwise selection, forward stagewise selection only calculates the weight of a single feature at a time, and is thus more lightweight. The downside of this approach in the context of regression is that it generally takes more iterations to reach the least squares coefficients of the complete model [21]. In our case this is unimportant, since we are only interested in quickly choosing the most important features, which are then used to fit a multiple linear regression model. We expect the child representation returned by StageXO to contain uncorrelated features since the residual is updated each iteration to remove the portion of the response explained by previous features.

Forward stagewise regression, and therefore the StageXO operator, is closely related to boosting [20]. In both cases the residual is iteratively reduced by adding model components (weak learners in the case of boosting, and features/building blocks in our case). The relationship between forward stagewise regression, boosting, and regularized linear models is expounded upon in [21]. The stagewise additive modeling paradigm is also used by a recent GP technique called Wave [51], in which GP runs are iteratively trained on residuals of previous runs. The insight here is that the unique representation of programs in MGP allows the same general methodology to be exploited for combining partial solutions during crossover, rather than as a post-run ensemble method.

Let us briefly consider the computational complexity of these operators. Both operators scale linearly with dataset size, N. Let M be the maximum dimensionality of an individual, a user-specified parameter defined for the experiments in Table 2. ResXO scales linearly with M (due to step 3), whereas StageXO scales quadratically with M [due to Step 3.(a)]. Therefore we expect ResXO to be quicker in practice, a hypothesis we test in the experiments of Sect. 4.

Table 2 Hyperparameter values for FEAT in the experiments

Hyperparameter	Values
Probability of crossover (complement: mutation)	[0, 0.25, 0.5, 0.75, 1.0]
Feedback (γ, Eq. 2)	[0, 0.25, 0.5, 0.75, 1.0]
Type of feature crossover	[Standard, ResXO, StageXO]
Probability of feature crossover (complement: subtree crossover)	[0.5, 0.75, 1.0]
Feedback softmax normalization	[on, off]
Population size	500
Generations	100 (200 for PMLB)
Max stalled generations	10
Max depth	6
Maximum dimensionality	$\min(50, 2 * |\mathbf{x}|)$
Iterations of gradient descent	10

"Maximum dimensionality" refers to the largest allowed number of features (m)

3.3 Selection and survival

The selection step selects P parents that will be used to generate offspring. Following variation, the population consists of $2P$ representations of parents and offspring. The survival step is used to reduce the population back to size P, at which point the generation is finished. We empirically compared five algorithms for selection and survival: (1) ϵ-lexicase selection (Lex) [47], (2) non-dominated sorting genetic algorithm (NSGA2) [13], (3) a novel hybrid algorithm using Lex for selection and NSGA2 for survival, (4) simulated annealing [34], and (5) random search. These comparisons are described in "Comparison of selection algorithms" of "Appendix". We found that the hybrid algorithm (3) performed the best; it is described below.

Parents are selected using Lex. Lex was proposed for regression problems [41, 47] as an adaption of lexicase selection [71] for continuous domains. The version of ϵ-lexicase selection we refer to and describe here is the "semi-dynamic" version proposed previously [41]. Under ϵ-lexicase selection, parents are chosen by filtering the population according to randomized orderings of training samples, with the ϵ threshold defined relative to the sample loss of the population. This filtering strategy scales the probability of selection for an individual based on the difficulty of the training cases on which the individual performs well. Lex has shown strong performance among SR methods in recent tests, motivating our interest in studying it [61].

Survival is conducted using the survival sub-routine of NSGA2, a popular strategy for multi-objective optimization [13]. NSGA2 applies preference for survival using Pareto dominance relations. An individual (n_i) is said to *dominate* another (n_j) if, for all objectives, n_i performs at least as well as n_j, and for at least one objective, n_i strictly outperforms n_j. The Pareto *front* is the set of individuals in $\mathbb{N}$ that are non-dominated in the population and thus represent optimal trade-offs between objectives found during search. Individuals are assigned a Pareto *ranking* that specifies the number of individuals that dominate them, thereby determining their proximity to the front.

The survival step of NSGA2 begins by sorting the population according to their Pareto front ranking and choosing the lowest ranked individuals for survival. To break rank ties, NSGA2 assigns each individual a crowding distance measure, which quantifies an individual's distance to its two adjacent neighbors in objective space. If a rank level does not completely fit in the survivor pool, individuals of that rank are sorted by highest crowding distance and added in order until P individuals are chosen.

3.3.1 Objectives

In our study, we consider three objectives corresponding to three goals:

1. Reduce model error.
2. Minimize complexity of the representation.
3. Minimize the entanglement of the representation.

The objectives related to the first two goals are used whenever NSGA2 is used for selection or survival in any version of FEAT explored in this paper. For the final goal of minimizing entanglement, we experiment with the addition of one of two different objectives, described at the end of this section.

The first objective is always the mean squared loss for individual n, and the second is the complexity of the representation. Regarding complexity, many definitions come to mind: one could look at the number of operations in a representation, or look at the behavioral complexity of the representation (e.g. using the order of a best-fit polynomial [79]). The complexity definition we use is similar to that used by Kommenda et al. [35]. The basic notion is to assign a complexity weight to each operator (see Table 1), with higher weights assigned to operators considered more complex. If the weight of operator o is c_o, then the complexity of an expression tree beginning at node o is defined recursively as

$$C(o) = c_o \sum_{a=1}^{k} C(a) \tag{3}$$

where node o has k arguments, and $C(a)$ is the complexity of argument a. The complexity of a representation is then defined as the sum of the complexities of its output nodes. The goal of defining complexity in such a way is to discourage deep subexpressions within complex nodes, which are often hard to interpret. It is important to note that the choice of operator weights is bound to be subjective, since we lack an objective notion of interpretability.

We test the third objective using two different metrics: the correlation of the transformation matrix $\phi(\mathbf{x})$ and its condition number. These metrics are defined below.

Disentanglement is a term used to describe the notion of a representation's ability to separate factors of variation in the underlying process [4]. Although a thorough review is beyond the scope of this section, there is a growing body of literature addressing disentanglement, primarily with unsupervised learning and/or image analysis [24, 27, 28, 40, 53, 80]. There are various ways to quantify disentanglement. For instance, Brahma et al. [5] proposed measuring disentanglement as the difference between geodesic and Euclidean distances among points on a manifold (i.e. training instances). If the latent structure is known, the information-theoretic metrics proposed by Eastwood and Williams [15] may be used. In the case of regression, a disentangled representation ideally contains a minimal set of features, each corresponding to a separate latent factor of variation, and each orthogonal to each other. In this regard, we attempt to minimize the collinearity between features in ϕ as a way to promote disentanglement. We tested two measurements of collinearity (a.k.a. multicollinearity) in the derived feature space. The first is the average squared Pearson's correlation among features of ϕ, i.e.,

$$Corr(\phi) = \frac{1}{N(N-1)} \sum_{\phi_i,\phi_j \in \phi, i \neq j} \left(\frac{\mathrm{cov}(\phi_i, \phi_j)}{\sigma(\phi_i)\sigma(\phi_j)} \right)^2 \tag{4}$$

The motivation to square the Pearson's correlation is the observation that two negatively correlated features are equally undesirable to two positively correlated features. Eq. 4 is relatively inexpensive to compute but only captures bivariate correlations in ϕ. As a result we also test the condition number (CN). Consider the $N \times m$ representation matrix $\boldsymbol{\Phi}$, where each column is the output of a feature. The CN of $\boldsymbol{\Phi}$ is defined as

$$CN(\phi) = \frac{\mu_{\max}(\boldsymbol{\Phi})}{\mu_{\min}(\boldsymbol{\Phi})} \tag{5}$$

where $\mu_{\max}$ and $\mu_{\min}$ are the largest and smallest singular values of $\boldsymbol{\Phi}$. Unlike *Corr*, *CN* can capture higher-order dependencies in the representation. *CN* is also related directly to the sensitivity of $\boldsymbol{\Phi}$ to perturbations in the training data [3, 9], and thus captures a notion of model invariance explored in previous work by Goodfellow et al. [25]. Another common measure of multicollinearity, the variance inflation factor [59], is likely to be too expensive to compute for our purposes.

3.4 Connection to neural networks

In addition to learning internal weights via gradient descent, FEAT includes instructions typically used as activation functions in neural networks (NN), e.g. tanh, sigmoid, logit and relu nodes, in addition to elementary arithmetic and boolean operators. Although a fully connected feedforward NN could be represented by this construction, representations in FEAT are biased to be thinly connected by their tree-based initialization. Because of this architecture, FEAT can be thought of as a method for evolving neural network architectures. The idea to evolve NN architectures is well established in literature, and is known as neuroevolution. Popular methods of neuroevolution include neuroevolution of augmenting topologies (NEAT [75] and Hyper-NEAT [74]), and compositional pattern producing networks [72] . The aforementioned approaches eschew the parameter learning step common in other NN paradigms, although others have developed integrations [16]. In addition, they have been developed predominantly for other task domains such as robotics and control [23], image classification [65, 66], and representation learning [10, 30]. Reviews of these methods are available [19, 73]. Neuroevolution is a part of a broader research field of neural architecture search (NAS) [48, 49, 81]. NAS methods vary in approach, including for example parameter sharing [64], sequential model-based optimization [49], representation learning [81], and greedy heuristic strategies [11].

4 Experiment

Our experiment consists of three stages. First, we conduct a robust study of FEAT with and without the semantic crossover operators introduced in Sect. 3.2.1. In this study we simultaneously vary the hyperparameters related to variation in order to analyze the results in detail for eight regression problems. In the second study, we compare the use of additional objectives (Sect. 3.3.1), the use of semantic crossover, and state-of-the-art ML methods across 100 benchmark regression problems from the Penn ML Benchmark (PMLB) [60]. The properties of these datasets are shown in Fig. 3. In the final study, we compare FEAT, FEATResXO, and FEATStageXO to results from a recent large benchmark study of SR methods [61] utilizing the same benchmark resource.

In addition to these studies, we examine the FEAT results in detail for one of the benchmark datasets. For this dataset we plot the final population of models, illustrate model selection and compare the resultant features to results from linear and ensemble tree-based results. This gives practical insight into the method and provides a sense of the intelligibility of an example representation. The results of this illustrative example are given in “Illustrative example” of “Appendix”.

4.1 Comparison of crossover methods

Despite several MGP methods having been proposed, there has not been a systematic study of the effect of variation operators on the performance of this family of methods. To fill this gap, and to properly analyze the new methods introduced in this paper, we performed a grid search of variation hyperparameters on eight regression problems. The hyperparameters that were varied are shown in Table 2.

Feedback softmax normalization refers to the softmax transformation in Eq. 2; we tested for whether this normalization, which assumes a multinomial distribution of probabilities, was useful. The eight comparison problems are listed in Table 3.

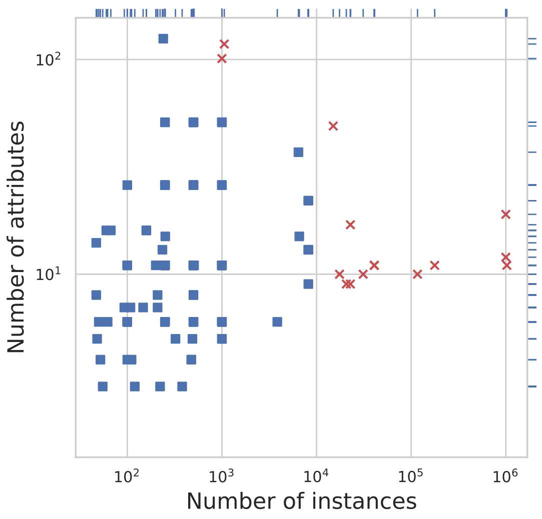


Fig. 3 Properties of the PMLB benchmark datasets [60]. The squares are datasets that were used to benchmark variants of FEAT in Sect. 4

Table 3 Regression problems used for method comparisons

Problem	Dimension	Samples
Airfoil	5	1503
Concrete	8	1030
ENC	8	768
ENH	8	768
Housing	14	506
Tower	25	3135
UBall5D	5	6024
Yacht	6	309

4.2 Comparison of all FEAT variants

In the second study we compare all FEAT variants to four state-of-the-art ML methods. This study includes the assessment of additional objectives that explicitly reward disentangled representations as described in Sect. 3.3.1. For each method we perform hyperparameter tuning as shown in Table 6 in “Additional experiment information” of “Appendix”. We compare all of the FEAT variants to XGBoost [7], multilayer perceptron (MLP), ElasticNet, kernelized ridge regression, and random forests (RF). We assess the methods in terms of their R^2 test set scores as well as the complexity of their solutions.

At the beginning of model training, FEAT sets aside 25% of the shuffled training data for validation and final model selection. The population’s median validation fitness is also used to terminate training if it stops improving for a set number of generations (see “max stalled generations” in Table 2). Otherwise, we limit FEAT’s optimization to 200 iterations or 60 min, whichever comes first. All runs are conducted on a heterogeneous computing cluster, with each training instance run on a single 2.6 GHz processor with a maximum of 12 GB of RAM. For each method, we use grid search to tune the hyperparameters with tenfold cross validation (CV). We use the mean cross-validation coefficient of determination (R^2) for assessing performance, defined as:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{6}$$

where $\bar{y}$ is the mean of the data labels y. In our results we report the CV scores for each method using its best hyperparameters. The algorithms are ranked on each dataset using their median CV score over 5 randomized shuffles of the dataset. For comparing complexity, we count the number of nodes in the final model produced by each method for each trial on each dataset. Note that this complexity definition is different (and simpler) than that used as an objective in FEAT, i.e. Eq. 3. To quantify the "entanglement" of the feature spaces, we report Eq. 4 in the raw data and in the final hidden layer of FEAT and MLP models. We also test two additional versions of Feat, denoted FeatCorr and FeatCN, that include a third objective corresponding to Eqs. 4 and 5, respectively.

4.3 Benchmark comparison

In the final study, we compared FEAT with each crossover variant to 15 other methods: 5 GP methods [1, 6, 41, 67] and 10 ML methods from scikit-learn [63]. The 5 GP methods we compared to are:

- Geometric semantic GP (GSGP) [6]
- MRGP [1]
- Age-fitness pareto optimization (AFP) [67]
- ϵ-lexicase selection (EPLEX) [41]
- ϵ-lexicase selection with 1 million evaluations (EPLEX-1M) [41]

These methods were benchmarked on 94 of the open-source datasets collected in the Penn ML Benchmark [60]. We used results from Orzechowski et al.'s benchmark analysis [61] as a comparison, and followed the same validation procedure. Each comparison method underwent hyperparameter tuning using fivefold cross validation on a 75% split of the training set, and was then tested on a 25% test fold. The hyperparameters are detailed in Table 1 of the original work [61] and Table 7 of "Appendix". This process was repeated for 10 trials. GP methods were given 100,000 evaluations, apart from EPLEX-1M which used 1 million. For FEAT, we did not re-tune the hyperparameters, instead using the values determined from the hyperparameter tuning experiment.

4.4 Metrics

As mentioned earlier, we consider there to be three over-arching goals when learning a representation. The first is that $\boldsymbol{\phi}(x)$ leads to a model with a low generalization error. To measure this, we compare the mean squared error (MSE) and coefficient of determination (R^2) of each model output on the test set. We also wish to minimize the complexity of the representation. To measure the complexity of solutions in FEAT, we count the total number of nodes in the final representation. For comparison to XGBoost, we count the number of nodes in the trees, and for comparison to MLP, we count the number of nodes in the network. Finally, we want a representation that is "disentangled", meaning that each feature of $\boldsymbol{\phi}$ is as orthogonal to the others as possible. We use this Eq. 4 to compare the entanglement of final representations across selection methods.

5 Results

The comparison of crossover methods are presented first. In addition to test score reporting, we plot various views of the data with respect to different hyperparameters, look at representation correlations in the resultant models, and conduct statistical comparisons. The subsequent section compares the use of additional objectives within FEAT to the use of semantic crossover; these comparisons include ML benchark methods as well. We include comparisons of the final model sizes and their correlations in this section. Finally, in the last section we compare FEAT, FEATResXO and FEATStageXO to the set of results from [61]. This section includes score comparisons and runtime comparisons.

5.1 Comparison of crossover methods

Prediction comparisons for each crossover method are shown in Fig. 4 for the eight tuning problems. The plot shows the mean test fold R^2 value for the tuned estimator, summarized across trials. In general one can see that StageXO produces the most accurate results, followed by ResXO. Across the eight problems, StageXO significantly outperforms standard crossover ($p < 0.035$); the pairwise statistical comparisons are given in Table 4.

We also looked at the correlation of the representations produced by the different crossover methods, shown in Fig. 5. We confirmed our hypotheses that ResXO and StageXO would produce less correlated representations than the traditional crossover operator.

The best values for each tuned parameter is shown in Table 5. We found that softmax normalization did not improve the feedback probabilities. Across problems, the best crossover/mutation fraction was found to be 0.75 (Fig. 6), with a feature crossover rate of 0.75 for Feat and 0.5 for ResXO and StageXO. The best

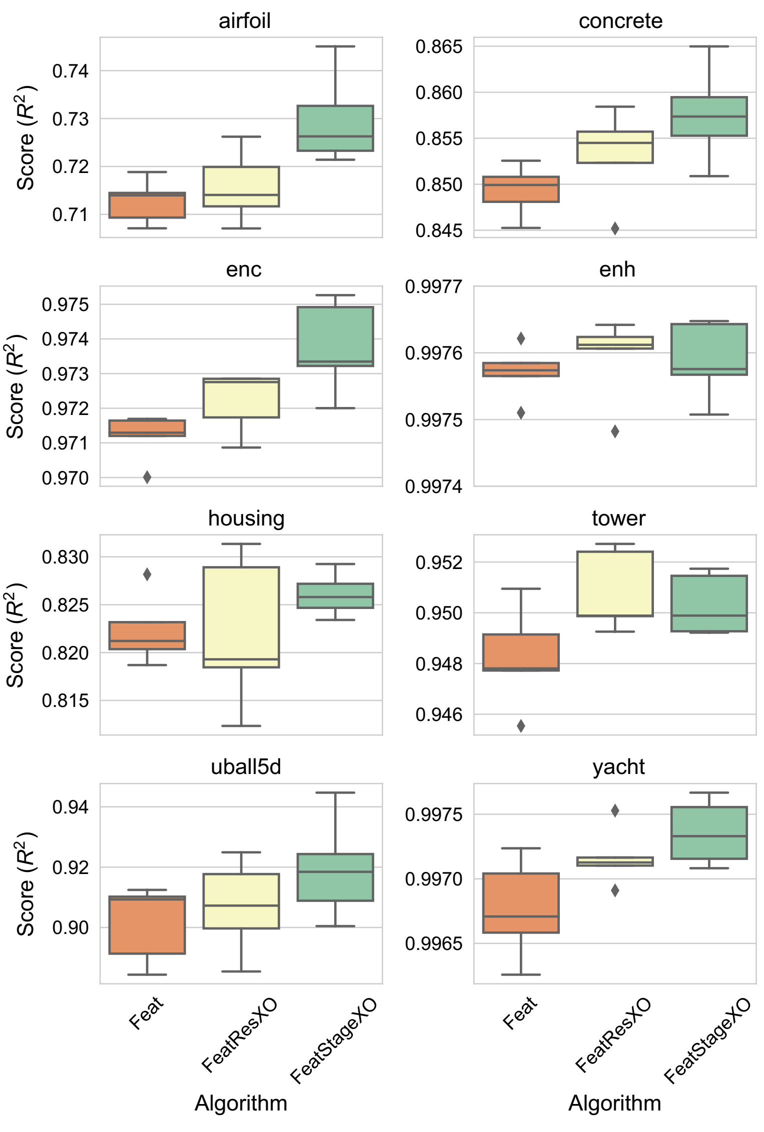


Fig. 4 Mean fivefold CV R^2 performance for different crossover operators on the eight tuning problems

Table 4 Bonferroni-adjusted p values using a Wilcoxon signed rank test of R^2 scores for the methods across all tuning problems

	Feat	FeatResXO
FeatResXO	4.6e−01	
FeatStageXO	**3.5e−02**	1.2e−01

Bold: $p < 0.05$

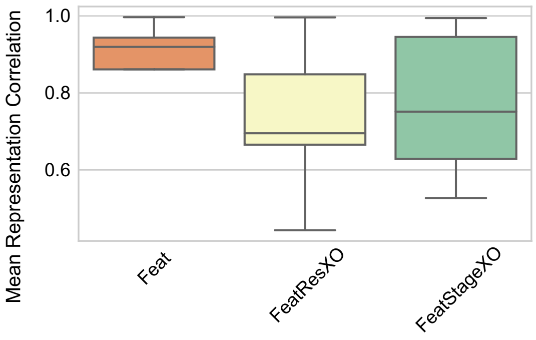


Fig. 5 Average pairwise representation correlation (Eq. 4) for different crossover operators on the eight tuning problems

Table 5 Best hyperparameter values for FEAT across the eight tuning problems

Hyperparameter	FEAT	FEAT-ResXO	FEAT-StageXO
Probability of crossover	0.75	0.75	0.75
Feedback	0.25	0.0	0.25
Probability of feature crossover	0.75	0.5	0.5
Feedback softmax normalization	Off	Off	Off

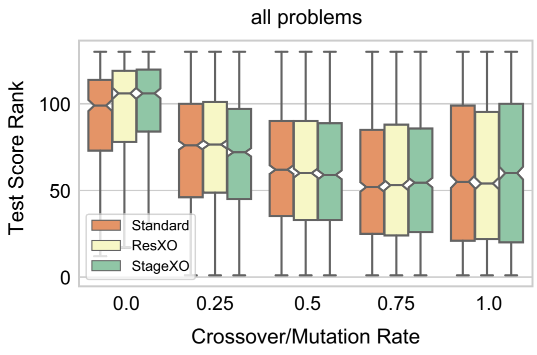


Fig. 6 Test rankings for different crossover probabilities on the eight tuning problems

feedback value was problem dependent, as shown in Fig. 7. Since the feedback essentially controls the amount of exploration versus exploitation, it stands to reason that the ideal setting of this parameter would be problem dependent. Feedback levels of 0.25 were best for FEAT and FEATStageXO, and no feedback was best for FEATResXO. For the ResXO operator, this corresponds to choosing the feature to swap out of the parent at random.

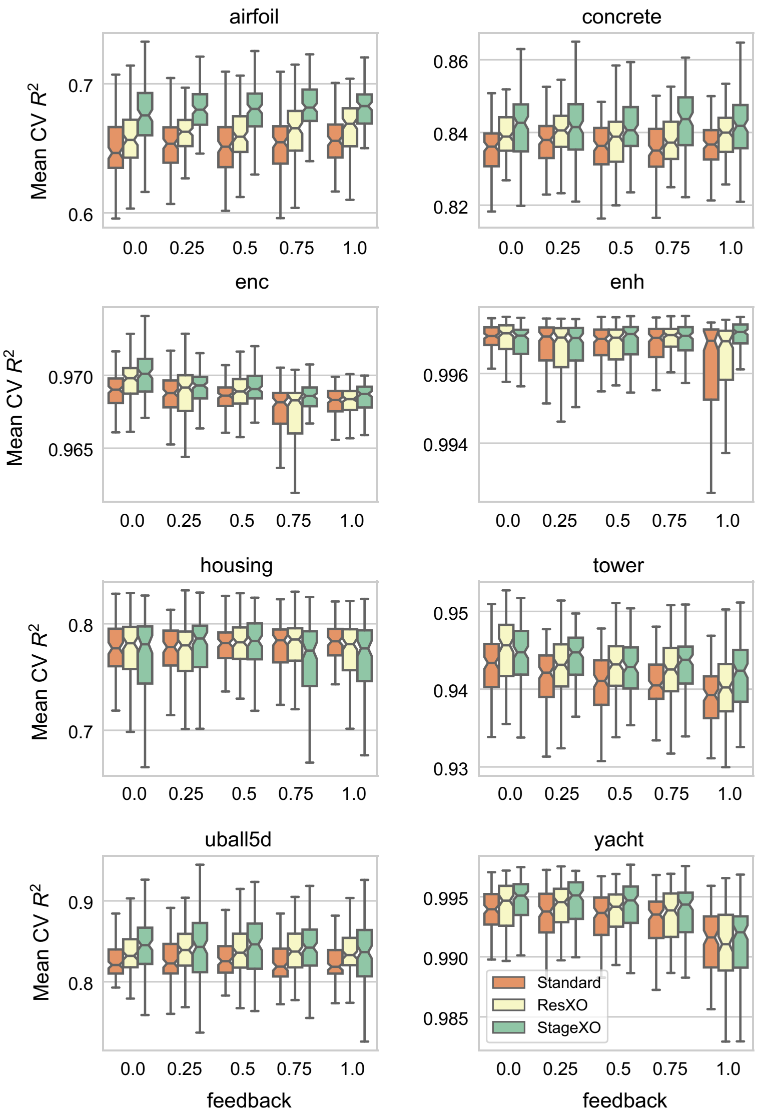


Fig. 7 Mean fivefold CV R^2 performance for different levels of feedback on the eight tuning problems

5.2 Comparison of FEAT variants

The score statistics for each of the FEAT variants and other methods are shown in Fig. 8 across 100 datasets from PMLB (Fig. 3). Full statistical comparisons are reported in "Statistical comparisons" "Appendix". Overall, FEAT and XGBoost produce the best predictive performance across datasets without significant differences between the two ($p = 1.0$). FEAT significantly outperforms MLP, RF, KernelRidge and ElasticNet ($p \leq 7.7\text{e}{-}03$), as does XGBoost ($p \leq 2.3\text{e}{-}02$). Among FEAT variants, FEATStageXO has the highest overall median R^2 value, although the differences are not significant.

As measured by the number of nodes in the final solutions, the models produced by FEAT are significantly less complex than XGBoost, RF, and MLP, as shown in Fig. 9 ($p <$ 1e−16). FEAT's final models tend to be within 1 order of magnitude of the linear models (ElasticNet), and 2–4 orders of magnitude smaller than the other non-linear methods. Among FEAT variants there are a few significant differences (see Table 9), albeit with small effect sizes as noted in Fig. 9.

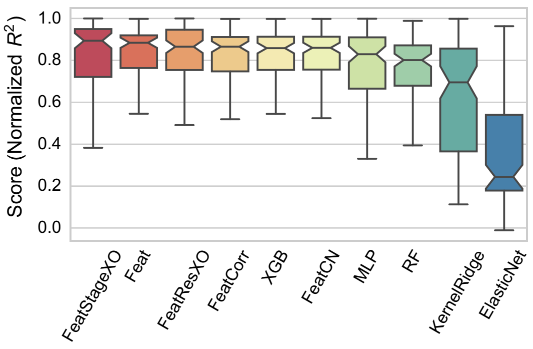


Fig. 8 Mean tenfold CV R^2 performance for various SO methods in comparison to other ML methods, across the benchmark problems

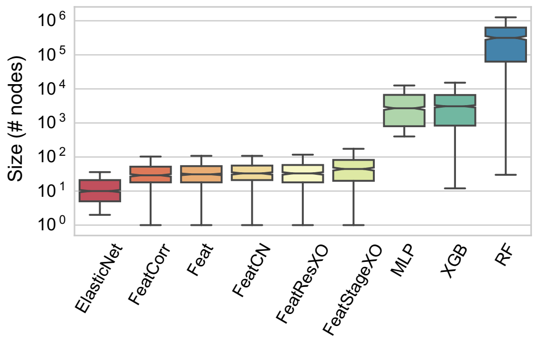


Fig. 9 Size comparisons of the final models in terms of number of nodes in the solutions

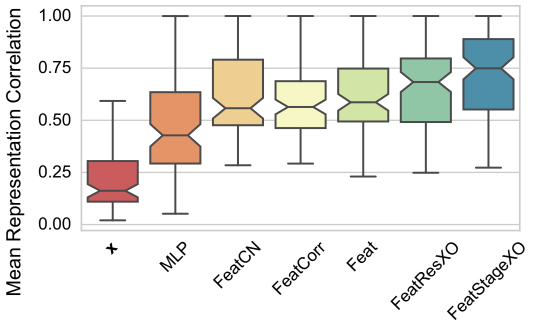


Fig. 10 Mean pairwise correlations of the representations produced by final models of different methods, in comparison to the correlations in the original attribute space (**x**)

We plot the mean pairwise correlations between features in the final representations across all problems in Fig. 10. We find that the use of objectives that minimize correlations between features (FeatCN, FeatCorr) have a small effect on minimizing the correlations in the resultant feature spaces for these problems. Interestingly, we see an opposite effect for the semantic variation operators on these problems—they tend to produce feature spaces that are more highly correlated than those produced by FEAT, unlike the results on the comparison problems in Fig. 5. Compared to the correlations in the final layer of trained MLP models, the FEAT variants tend to produce more entangled representations, indicating room for improvement.

5.3 Benchmark comparison

The comparisons of FEAT to 15 other methods is shown in Fig. 11. In this figure, each boxplot shows the distribution of rankings (in terms of test MSE) over all datasets for each method. Across problems, FEAT and FEATStageXO achieves a nearly identical ranking to EPLEX-1M, which is ϵ-lexicase selection run for 1 million evaluations. Note that FEAT achieves these similar results using 100,000 evaluations. However, the additional complexity of fitting ML models to each individual makes

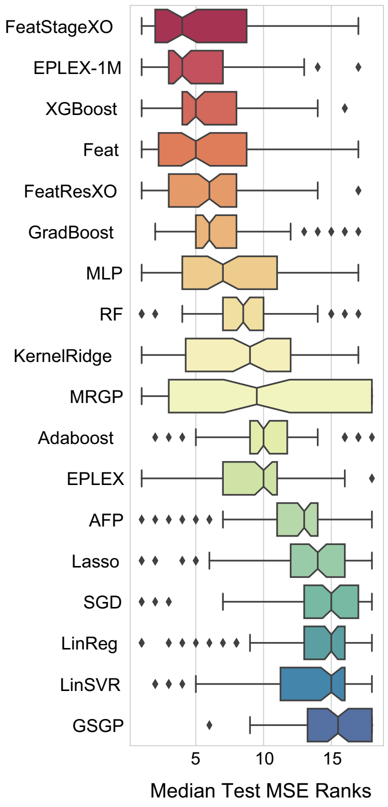


Fig. 11 Median test MSE rankings on the PMLB datasets. The box shows the quartiles of the rankings with whiskers showing the rest of the distribution excluding outliers. Algorithms are ordered top to bottom by best (lowest) to worst (highest) median ranking

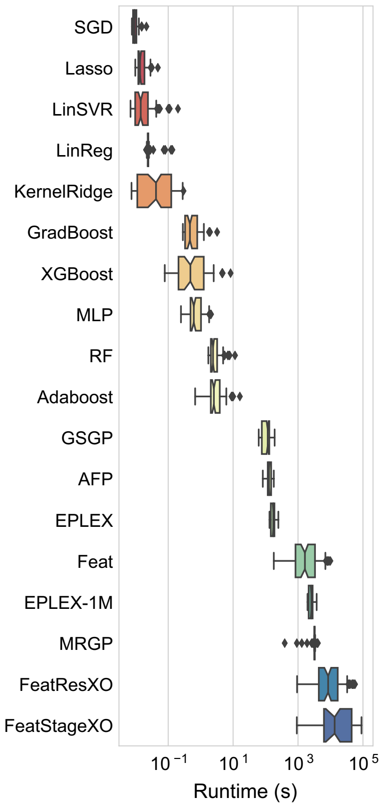


Fig. 12 Wall clock time comparisons on the PMLB datasets. Runtime is the wall clock time for a single training instance. Algorithms are ordered top to bottom by fastest to slowest

the evaluation of each individual in FEAT more costly than traditional GP. Therefore wall clock times shown in Fig. 12 reflect this, showing that the FEAT wall clock times sit somewhere between the methods that ran for 100,000 evaluations (GSGP, AFP, MRGP, EPLEX) and 1 million evaluations (EPLEX-1M). FEATResXO and FEATStageXO are slower, due to the additional complexity of semantic crossover.

A Friedman test of the MSE rankings across problems indicates significant differences. Table 10 shows post-hoc pairwise Wilcoxon signed rank tests of the results. FEAT, FEATResXO, and FEATStageXO all significantly outperform the other GP-based methods run for 100,00 evaluations. There is no significant difference found between the FEAT variants, EPLEX-1M, XGBoost, GradBoost, or MLP. The FEAT variants significantly outperform all other ML methods across the benchmark problems.

6 Discussion and conclusion

In this paper we have argued that MGP is a useful, appropriate and interesting architecture of GP for learning feature spaces for regression. Unlike other approaches within GP to tackle representation learning, MGP is exposes independent components of programs, i.e. evolving features, to the search process. This property allows for the exploitation of additional information in both the variation and evaluation steps. Here, we have proposed an MGP approach to regression, FEAT, that borrows concepts from NN learning and have used it as a test bed for new methods.

We proposed two semantic crossover methods in this work. One of these methods, based on stagewise regression, mimics forward stagewise selection to select and propagate components of parents into offspring. We find that this method, in terms of average ranking of test R^2 values, gives the best results of all tested ML methods across a set of 100 regression problems, varying in size and dimensionality. We find conflicting pieces of evidence regarding the behavior of StageXO in terms of the collinearity of representations it produces; on some sets it is able to reduce collinearity, but over a broader set of problems tends to result in more collinearity.

We also proposed additional fitness objectives during survival to encourage the orthogonality of learned feature spaces. We tested both average pairwise correlation and the condition number of the representation matrix as metrics. Neither objective had a significant effect on the performance of FEAT, and had a mild effect on reducing the collinearity of representations in the main experiment.

With these results in mind, it seems clear that further work should be dedicated to exploring methods for addressing collinearity in the MGP framework. There are many possibilities for reducing collinearity during variation. One thought is to try to prune collinear features with a mutation operator that removes the feature from the collinear pair that is less correlated with the data label. Another idea is to test different termination criteria for StageXO that may lead to less collinearity (recall that we terminate this crossover step when the offspring reaches the size of its root parent). Another approach is to produce offspring from more than two parents, as is done in recent work [18, 78]. Moving beyond variation, it may be possible to achieve lower representation entanglement through a post-processing step. In any case, the strong performance of FEAT on existing benchmarks will hopefully motivate further research within the MGP framework.

7 Supplementary material

The experiments were conducted in Python and available from http://github.com/lacava/gpem_2019. The FEAT codebase can be accessed at http://github.com/lacava/feat.

Acknowledgements This work was supported by NIH Grants K99LM012926-01A1, AI116794 and LM012601, as well as the PA CURE Grant from the Pennsylvania Department of Health. Special thanks to Tilak Raj Singh and other members of the Computational Genetics Lab at the University of Pennsylvania.

Appendix

Additional experiment information

Table 6 details the hyperparameters for each method used in the experimental results described in Sects. 4 and 5.

Table 6 Comparison methods and their hyperparameters for the comparisons in Sect. 4.2. Tuned values denoted with brackets

Method	Setting	Value
FEAT	Population size	500
	Termination criterion	200 generations, 60 min, or 50 iterations of stalled median validation loss
	Max depth	10
	Max dimensionality	50
	Objectives	{(MSE, *C*), (MSE, *C*, *Corr*),(MSE, *C*, *CN*)}
	Feedback (*f*)	{0.25, 0.5, 0.75}
	Crossover/mutation ratio	{0.25, 0.5, 0.75}
	Batch size	1000
	Learning rate (initial)	0.1
	SGD iterations/individual/generation	10
MLP	Optimizer	{LBFGS, Adam [33]}
	Hidden Layers	{1,3,6}
	Neurons	{(100), (100, 50, 10), (100, 50, 20, 10, 10, 8)}
	Learning rate	(initial) {1e−4, 1e−3, 1e−2}
	Activation	{logistic, tanh, relu}
	Regularization	$L_2, \alpha = \{1e{-5}, 1e{-4}, 1e{-3}\}$
	Max iterations	10,000
	Early stopping	True
XGBoost	Number of estimators	{10, 100, 200, 500, 1000}
	Max depth	{3, 4, 5, 6, 7}
	Min split loss (γ)	{1e−3, 1e−2, 0.1, 1, 10, 1e2, 1e3}
	Learning rate	{0, 0.01, …, 1.0}
Random Forest	Number of estimators	{10, 100, 1000}
	Min weight fraction leaf	{0.0, 0.25, 0.5}
Kernel Ridge	Kernel	Radial basis function
	Regularization (α)	{1e−3, 1e−2, 0.1, 1}
	Kernel width (γ)	{1e−2, 0.1, 1, 10, 100}
ElasticNet	l_1-l_2 ratio	{0, 0.01, …, 1.0}
	selection	{cyclic, random}

Comparison of selection algorithms

Our initial analysis sought to determine how different SO approaches performed within this framework. We tested five methods: (1) NSGA2, (2) Lex, (3) LexNSGA2, (4) Simulated annealing, and (5) random search. The simulated annealing and random search approaches are described below.

Simulated annealing Simulated annealing (SimAnn) is a non-evolutionary technique that instead models the optimization process on the metallurgical process of annealing. In our implementation, offspring compete with their parents; in the case of multiple parents, offspring compete with the program with which they share more nodes. The probability of an offspring replacing its parent in the population is given by the equation

$$P_{sel}(n_o|n_p,t) = \exp\left(\frac{F(n_p) - F(n_o)}{t}\right) \tag{7}$$

The probability of offspring replacing its parent is a function of its fitness, F, in our case the mean squared loss of the candidate model. In Eq. 7, t is a scheduling parameter that controls the rate of "cooling", i.e. the rate at which steps in the search space that are worse are tolerated by the update rule. In accordance with [34], we use an exponential schedule for t, defined as $t_g = (0.9)^g t_0$, where g is the current generation and $t0$ is the starting temperature. $t0$ is set to 10 in our experiments.

Random search We compare the selection and survival methods to random search, in which no assumptions are made about the structure of the search space. To conduct random search, we randomly sample $\mathbb{S}$ using the initialization procedure. Since FEAT begins with a linear model of the process, random search will produce a representation at least as good as this initial model on the internal validation set.

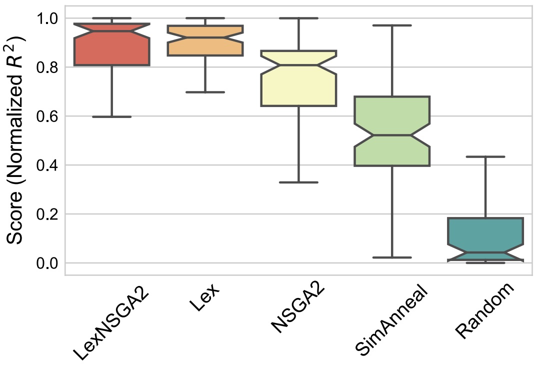


Fig. 13 Mean tenfold CV R^2 performance for various SO methods in comparison to other ML methods, across the benchmark problems

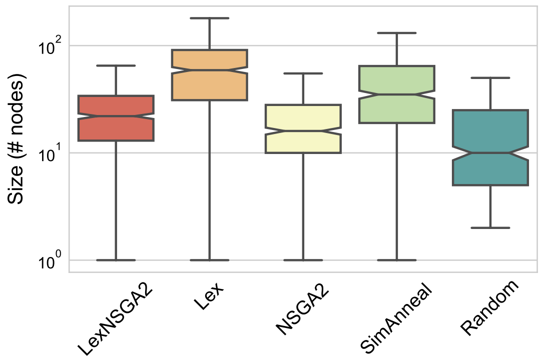


Fig. 14 Size comparisons of the final models in terms of number of parameters

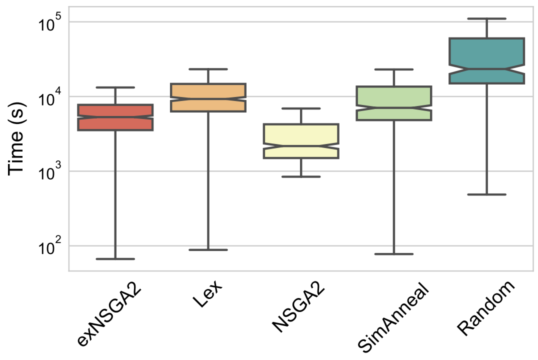


Fig. 15 Wall-clock runtime for each method in seconds

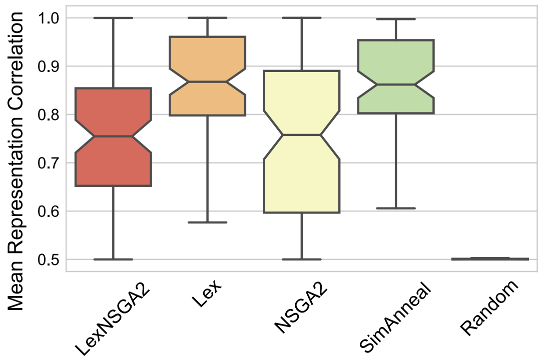
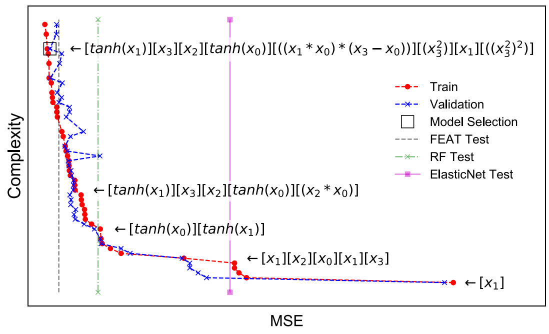


Fig. 16 Mean correlation between engineered features for different SO methods compared to the correlations in the original data (ElasticNet)

A note on archiving When FEAT is used without a complexity-aware survival method (i.e., with Lex, SimAnn, Random), a separate population is maintained that acts as an archive. The archive maintains a Pareto front according to minimum loss and complexity (Eq. 3). At the end of optimization, the archive is tested on a small hold-out validation set. The individual with the lowest validation loss is the final selected model. Maintaining this archive helps protect against overfitting resulting from overly complex/high capacity representations, and also can be interpreted directly to help understand the process being modelled.



$\phi(\mathbf{x})$	β
$\tanh(x_1)$	2953.8
x_1	1961.5
(x_3^2)	329.0
$\tanh(x_0)$	-318.5
$((x_1 \cdot x_0) \cdot (x_3 - x_0))$	303.6
$((x_3^2)^2)$	-288.7
x_3	203.3
x_2	64.8

Fig. 17 (Left) Representation archive for the visualizing galaxies dataset. (Right) Selected model and its weights. Internal weights omitted

We benchmarked these approaches in a separate experiment on 88 datasets from PMLB [60]. The results are shown in Figs. 13, 14, 15 and 16. Considering Figs. 13 and 14, we see that LexNSGA2 achieves the best average R^2 value while producing small solutions in comparison to Lex. NSGA2, SimAnneal, and Random search all produce less accurate models. The runtime comparisons of the methods in Fig. 15 show that they are mostly within an order of magnitude, with NSGA2 being the fastest (due to its maintenance of small representations) and Random search being the slowest, suggesting that it maintains large representations during search. The computational behavior of Random search suggests the variation operators tend to increase the average size of solutions over many iterations.

Illustrative example

We show an illustrative example of the final archive and model selection process from applying FEAT to a galaxy visualization dataset [8] in Fig. 17. The red and blue points correspond to training and validation scores for each archived representation with a square denoting the final model selection. Five of the representations are printed in plain text, with each feature separated by brackets. The vertical lines in the left figure denote the test scores for FEAT, RF and ElasticNet. It is interesting to note that ElasticNet performance roughly matches the performance of a linear representation, and the RF test performance corresponds to the representation [tanh(x_0)][tanh(x_1)] that is suggestive of axis-aligned splits for x_0 and x_1. The selected model is shown on the right, with the features sorted according to the magnitudes of β in the linear model. The final representation combines tanh, polynomial, linear and interacting features. This representation is a clear extension of simpler ones in the archive, and the archive thereby serves to characterize the improvement in predictive accuracy brought about by increasing complexity. Although a mechanistic interpretation requires domain expertise, the final representation is certainly concise and amenable to interpretation.

Table 7 Algorithms from Orzechowski et. al. [61] with their parameter settings

Algorithm name	Parameter	Values
eplex, afp, mrgp	Pop size/generations	{100/1000,1000/100}
	Max program length/max depth	{64/6}
	Crossover rate	{0.2,0.5,0.8}
	Mutation rate	1-crossover rate
gsgp	Pop size/generations	{100/1000,200/500,1000/100}
	Initial depth	{6}
	Crossover rate	{0.0,0.1,0.2}
	Mutation rate	1-crossover rate
eplex_1M	Pop size/generations	{500/2000,1000/1000,2000/500}
	Max program length	{100}
	Crossover rate	{0.2,0.5,0.8}
	Mutation rate	1-crossover rate
AdaBoostRegressor	'n_estimators'	{10, 100, 1000}
	'learning_rate'	{0.01, 0.1, 1, 10}
GradientBoostingRegressor	'n_estimators'	{10, 100, 1000}
	'min_weight_fraction_leaf'	{0.0, 0.25, 0.5}
	'max_features'	{'sqrt','log2', None}
KernelRidge	'kernel'	{'linear', 'poly', 'rbf', 'sigmoid'}
	'alpha'	{1e−4, 1e−2, 0.1, 1}
	'gamma'	{0.01, 0.1, 1, 10 }
LassoLARS	'alpha'	{ 1e−04, 0.001, 0.01, 0.1, 1 }
LinearRegression	default	default
MLPRegressor	'activation'	{'logistic', 'tanh', 'relu'}
	'solver'	{'lbfgs','adam','sgd'}
	'learning_rate'	{'constant', 'invscaling', 'adaptive'}
RandomForestRegressor	'n_estimators'	{10, 100, 1000}
	'min_weight_fraction_leaf'	{0.0, 0.25, 0.5}
	'max_features'	{''sqrt','log2', None}
SGDRegressor	'alpha'	{1e−06, 1e−04, 0.01, 1 }
	'penalty'	{'l2', 'l1', 'elasticnet'}
LinearSVR	'C'	{1e−06, 1e−04, 0.1, 1 }
	'loss'	{'epsilon_insensitive', 'squared_ epsilon_insensitive'}
XGBoost	'n_estimators'	{10, 50, 100, 250, 500, 1000}
	'learning_rate'	{1e−4, 0.01, 0.05, 0.1, 0.2}
	'gamma'	{0, 0.1, 0.2, 0.3, 0.4}
	'max_depth'	{6}
	'subsample'	{0.5, 0.75, 1}

The parameters in quotations refer to their names in the scikit-learn implementations

Table 8 Bonferroni-adjusted p values using a Wilcoxon signed rank test of R^2 scores for the FEAT variants across all benchmarks

	ElasticNet	Feat	FeatCN	FeatCorr	FeatResXO	FeatStageXO	KernelRidge	MLP	RF
Feat	**4.2e−14**								
FeatCN	**1.6e−12**	**1.2e−02**							
FeatCorr	**2.7e−12**	**1.1e−02**	1.0e+00						
FeatResXO	**1.4e−14**	1.0e+00	**3.1e−02**	7.2e−02					
FeatStageXO	**5.2e−13**	1.0e+00	2.0e−01	9.4e−01	1.0e+00				
KernelRidge	**2.1e−12**	**7.3e−05**	**3.0e−02**	**6.6e−03**	**1.5e−05**	**1.3e−03**			
MLP	**1.9e−11**	**7.7e−03**	1.0e+00	1.0e+00	**1.4e−03**	**9.1e−03**	1.0e+00		
RF	**2.4e−09**	**2.5e−09**	**1.5e−05**	**7.3e−07**	**1.3e−08**	**1.8e−06**	1.0e+00	9.9e−02	
XGB	**2.8e−14**	1.0e+00	1.0e+00	1.0e+00	1.0e+00	1.0e+00	**2.0e−04**	**2.3e−02**	**6.5e−13**

Bold: $p < 0.05$

Table 9 Bonferroni-adjusted p values using a Wilcoxon signed rank test of sizes for the FEAT variants across all benchmarks

	ElasticNet	Feat	FeatCN	FeatCorr	FeatResXO	FeatStageXO	MLP	RF
Feat	**1.4e−13**							
FeatCN	**4.1e−16**	**9.5e−08**						
FeatCorr	**2.0e−12**	1.0e+00	**1.2e−07**					
FeatResXO	**1.1e−12**	7.5e−01	**4.9e−03**	1.0e+00				
FeatStageXO	**3.9e−16**	**9.3e−09**	1.0e+00	**3.6e−07**	**1.0e−04**			
MLP	**2.1e−17**	**8.9e−17**	**7.0e−17**	**1.0e−16**	**1.1e−16**	**9.4e−17**		
RF	**4.7e−20**	**7.6e−17**	**7.3e−17**	**1.0e−16**	**1.1e−16**	**4.3e−17**	**6.4e−17**	
XGB	**2.3e−17**	**8.4e−17**	**9.7e−17**	**1.1e−16**	**1.1e−16**	**8.5e−17**	1.0e+00	**1.1e−17**

Bold: $p < 0.05$

Table 10 Bonferroni-adjusted p values using a Wilcoxon signed rank test of MSE scores for the methods across all benchmarks

	Ada-boost	AFP	EPLEX	EPLEX-1M	FEAT	FEAT-ResXO	FEAT-StageXO	Grad-Boost	GSGP	Kernel-Ridge	Lasso	LinReg	LinSVR	MLP	MRGP	RF	SGD
AFP	**1.2e−02**																
EPLEX	1.0e+00	**7.3e−08**															
EPLEX-1M	**8.2e−09**	**9.4e−12**	**3.4e−09**														
FEAT	**2.6e−08**	**3.8e−10**	**1.9e−06**	1.0e+00													
FEA-TResXO	**1.8e−04**	**2.2e−06**	**5.8e−04**	**5.0e−02**	**1.7e−02**												
FEAT-StageXO	**9.1e−08**	**8.2e−10**	**4.1e−06**	1.0e+00	1.0e+00	**5.8e−03**											
GradBoost	**1.5e−08**	**2.3e−08**	**5.6e−03**	**3.7e−02**	1.0e+00	1.0e+00	1.0e+00										
GSGP	**8.4e−11**	**2.6e−07**	**4.1e−12**	**1.5e−14**	**8.1e−14**	**5.1e−13**	**2.3e−14**	**2.6e−14**									
Kernel-Ridge	1.0e+00	**1.5e−03**	1.0e+00	**8.1e−04**	**2.1e−02**	1.0e+00	**1.1e−02**	9.9e−01	**1.4e−14**								
Lasso	**8.3e−04**	1.0e+00	**1.9e−06**	**4.4e−12**	**4.5e−10**	**3.5e−07**	**4.8e−10**	**1.8e−08**	**2.1e−02**	**8.6e−08**							
LinReg	**1.1e−04**	**7.0e−03**	**2.3e−07**	**3.7e−12**	**4.5e−11**	**4.5e−09**	**6.1e−11**	**1.7e−08**	1.0e+00	**3.0e−09**	1.0e+00						
LinSVR	**1.1e−03**	8.6e−02	**2.3e−06**	**4.6e−12**	**1.3e−09**	**5.7e−07**	**9.0e−10**	**2.6e−08**	3.9e−01	**7.9e−09**	1.0e+00	1.0e+00					
MLP	**2.6e−02**	**2.6e−06**	1.0e+00	1.0e+00	1.0e+00	1.0e+00	1.0e+00	1.0e+00	**9.6e−15**	7.4e−01	**6.8e−07**	**2.0e−08**	**1.3e−07**				
MRGP	1.0e+00	1.0e+00	1.0e+00	**1.1e−04**	**7.0e−04**	2.8e−01	**3.6e−04**	2.1e−01	**2.1e−05**	1.0e+00	2.3e−01	**2.5e−02**	1.5e−01	1.0e+00			
RF	**7.9e−05**	**1.2e−04**	1.0e+00	**7.4e−06**	**1.0e−04**	2.0e−01	**7.6e−05**	**2.3e−06**	**2.1e−13**	1.0e+00	**4.7e−06**	**4.4e−06**	**1.3e−06**	1.0e+00	1.0e+00		
SGD	**2.0e−06**	**6.4e−05**	**1.9e−09**	**9.6e−13**	**2.7e−12**	**1.5e−09**	**1.3e−11**	**4.1e−11**	1.0e+00	**2.6e−09**	5.5e−02	1.0e+00	1.0e+00	**1.8e−09**	**4.5e−03**	**2.8e−09**	
XGBoost	**2.1e−08**	**6.9e−11**	**5.3e−05**	1.0e+00	1.0e+00	1.0e+00	1.0e+00	**8.9e−03**	**6.0e−15**	**6.9e−03**	**1.3e−10**	**6.4e−10**	**2.2e−10**	1.0e+00	**2.8e−03**	**3.6e−08**	**5.8e−12**

Bold: $p < 0.05$

Statistical comparisons

We perform pairwise comparisons of methods according to the procedure recommended by Demšar [14] for comparing multiple estimators (Table 7). In Table 8, the CV R^2 rankings are compared. In Table 9, the best model size rankings are compared. Note that KernelRidge is omitted from the size comparisons since we don't have a comparable way of measuring the model size.

References

1. I. Arnaldo, K. Krawiec, U.M. O'Reilly, Multiple regression genetic programming, in *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation* (ACM Press, 2014), pp. 879–886. https://doi.org/10.1145/2576768.2598291. http://dl.acm.org/citation.cfm?doid=2576768.2598291. Accessed 15 Oct 2019
2. I. Arnaldo, U.M. O'Reilly, K. Veeramachaneni, Building predictive models via feature synthesis, in *GECCO* (ACM Press, 2015), pp. 983–990. https://doi.org/10.1145/2739480.2754693. http://dl.acm.org/citation.cfm?doid=2739480.2754693. Accessed 15 Oct 2019
3. D.A. Belsley, A guide to using the collinearity diagnostics. Comput. Sci. Econ. Manag. **4**(1), 33–50 (1991). https://doi.org/10.1007/BF00426854
4. Y. Bengio, A. Courville, P. Vincent, Representation learning: a review and new perspectives. IEEE Trans. Pattern Anal. Mach. Intell. **35**(8), 1798–1828 (2013)
5. P.P. Brahma, D. Wu, Y. She, Why deep learning works: a manifold disentanglement perspective. IEEE Trans. Neural Netw. Learn. Syst. **27**(10), 1997–2008 (2016)
6. M. Castelli, S. Silva, L. Vanneschi, A C++ framework for geometric semantic genetic programming. Genet. Program. Evol. Mach. **16**(1), 73–81 (2015). https://doi.org/10.1007/s10710-014-9218-0
7. T. Chen, C. Guestrin, XGBoost: a scalable tree boosting system, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16* (ACM, New York, NY, USA, 2016), pp. 785–794. https://doi.org/10.1145/2939672.2939785
8. W.S. Cleveland, *Visualizing Data* (Hobart Press, New Jersey, 1993)
9. A. Cline, C. Moler, G. Stewart, J. Wilkinson, An estimate for the condition number of a matrix. SIAM J. Numer. Anal. **16**(2), 368–375 (1979). https://doi.org/10.1137/0716029
10. E. Conti, V. Madhavan, F.P. Such, J. Lehman, K.O. Stanley, J. Clune, Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. arXiv:1712.06560 [cs] (2017)
11. C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, S. Yang, Adanet: adaptive structural learning of artificial neural networks. arXiv preprint arXiv:1607.01097 (2016)
12. V.V. De Melo, Kaizen Programming, in *GECCO* (ACM Press, New York, 2014), pp. 895–902. https://doi.org/10.1145/2576768.2598264. http://dl.acm.org/citation.cfm?doid=2576768.2598264
13. K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II, in *Parallel Problem Solving from Nature PPSN VI*, vol. 1917, ed. by M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J.J. Merelo, H.P. Schwefel (Springer, Berlin, 2000), pp. 849–858. http://repository.ias.ac.in/83498/. Accessed 15 Oct 2019
14. J. Demšar, Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. **7**(Jan), 1–30 (2006)
15. C. Eastwood, C.K.I. Williams, A framework for the quantitative evaluation of disentangled representations, in *ICLR* (2018). https://openreview.net/forum?id=By-7dz-AZ. Accessed 15 Oct 2019
16. C. Fernando, D. Banarse, M. Reynolds, F. Besse, D. Pfau, M. Jaderberg, M. Lanctot, D. Wierstra, Convolution by evolution: differentiable pattern producing networks. arXiv:1606.02580 [cs] (2016)
17. R. Ffrancon, M. Schoenauer, Memetic Semantic Genetic Programming (ACM Press, 2015), pp. 1023–1030. https://doi.org/10.1145/2739480.2754697. http://dl.acm.org/citation.cfm?doid=2739480.2754697

18. S.B. Fine, E. Hemberg, K. Krawiec, U.M. O'Reilly, Exploiting subprograms in genetic programming, in *Genetic Programming Theory and Practice XV, Genetic and Evolutionary Computation*, ed. by W. Banzhaf, R.S. Olson, W. Tozier, R. Riolo (Springer, Berlin, 2018), pp. 1–16
19. D. Floreano, P. Dürr, C. Mattiussi, Neuroevolution: from architectures to learning. Evol. Intell. **1**(1), 47–62 (2008). https://doi.org/10.1007/s12065-007-0002-4
20. Y. Freund, R.E. Schapire, A desicion-theoretic generalization of on-line learning and an application to boosting, in *Computational Learning Theory*, ed. by P. Vitanyi (Springer, Berlin, 1995), pp. 23–37. https://doi.org/10.1007/3-540-59119-2_166
21. J. Friedman, T. Hastie, R. Tibshirani, The elements of statistical learning. Springer series in statistics, vol. 1 (Springer, Berlin, 2001). http://statweb.stanford.edu/tibs/book/preface.ps. Accessed 15 Oct 2019
22. A.H. Gandomi, A.H. Alavi, A new multi-gene genetic programming approach to nonlinear system modeling. Part I: materials and structural engineering problems. Neural Comput. Appl. **21**(1), 171–187 (2012). https://doi.org/10.1007/s00521-011-0734-z
23. F. Gomez, J. Schmidhuber, R. Miikkulainen, Efficient non-linear control through neuroevolution, in *ECML*, vol. 4212 (Springer, 2006), pp. 654–662. http://link.springer.com/content/pdf/10.1007/11871842.pdf#page=676
24. A. Gonzalez-Garcia, J. van de Weijer, Y. Bengio, Image-to-image translation for cross-domain disentanglement. arXiv preprint arXiv:1805.09730 (2018)
25. Goodfellow, I., H. Lee, Q.V. Le, A. Saxe, A.Y. Ng, Measuring invariances in deep networks, in *Advances in Neural Information Processing Systems*, pp. 646–654 (2009)
26. M. Graff, E.S. Tellez, E. Villaseñor, S. Miranda, Semantic genetic programming operators based on projections in the phenotype space. Res. Comput. Sci. **94**, 73–85 (2015)
27. N. Hadad, L. Wolf, M. Shahar, A two-step disentanglement method, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 772–780 (2018)
28. I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, A. Lerchner, β-VAE: Learning basic visual concepts with a constrained variational framework, in *ICLR* (2017)
29. A.E. Hoerl, R.W. Kennard, Ridge regression: biased estimation for nonorthogonal problems. Technometrics **12**(1), 55–67 (1970)
30. C. Igel, Neuroevolution for reinforcement learning using evolution strategies, in *The 2003 Congress on Evolutionary Computation, 2003. CEC'03*, vol. 4 (IEEE, 2003), pp. 2588–2595. http://ieeexplore.ieee.org/abstract/document/1299414/. Accessed 15 Oct 2019
31. V. Ingalalli, S. Silva, M. Castelli, L. Vanneschi, A multi-dimensional genetic programming approach for multi-class classification problems, in *Genetic Programming*, ed. by M. Nicolau (Springer, Berlin, 2014), pp. 48–60. https://doi.org/10.1007/978-3-662-44303-3_5
32. G. James, D. Witten, T. Hastie, R. Tibshirani, An introduction to statistical learning, in *Springer Texts in Statistics*, vol. 103, ed. by N.H. Timm (Springer, New York, 2013). https://doi.org/10.1007/978-1-4614-7138-7
33. D.P. Kingma, J. Ba, Adam: a method for stochastic optimization. arXiv:1412.6980 [cs] (2014).
34. S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)
35. M. Kommenda, G. Kronberger, M. Affenzeller, S.M. Winkler, B. Burlacu, Evolving simple symbolic regression models by multi-objective genetic programming, in *Genetic Programming Theory and Practice, vol. XIV. Genetic and Evolutionary Computation* (Springer, Ann Arbor, MI, 2015)
36. K. Krawiec, Genetic programming-based construction of features for machine learning and knowledge discovery tasks. Genet. Program. Evol. Mach. **3**(4), 329–343 (2002). https://doi.org/10.1023/A:1020984725014
37. K. Krawiec, On relationships between semantic diversity, complexity and modularity of programming tasks, in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference* (ACM, 2012), pp. 783–790. http://dl.acm.org/citation.cfm?id=2330272. Accessed 15 Oct 2019
38. K. Krawiec, *Behavioral Program Synthesis with Genetic Programming*, vol. 618 (Springer, Berlin, 2016)
39. K. Krawiec, U.M. O'Reilly, Behavioral programming: a broader and more detailed take on semantic GP, in *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation* (ACM Press, 2014), pp. 935–942. https://doi.org/10.1145/2576768.2598288. http://dl.acm.org/citation.cfm?doid=2576768.2598288. Accessed 15 Oct 2019

40. A. Kumar, P. Sattigeri, A. Balakrishnan, Variational inference of disentangled latent concepts from unlabeled observations, in *ICLR* (2018). https://openreview.net/forum?id=H1kG7GZAW. Accessed 15 Oct 2019
41. W. La Cava, T. Helmuth, L. Spector, J.H. Moore, A probabilistic and multi-objective analysis of lexicase selection and ϵ-lexicase selection. Evolut. Comput. (2018). https://doi.org/10.1162/evco_a_00224
42. W. La Cava, J. Moore, A general feature engineering wrapper for machine learning using \epsilon-lexicase survival, in *Genetic Programming* (Springer, Cham, 2017), pp. 80–95. https://doi.org/10.1007/978-3-319-55696-3_6
43. W. La Cava, J.H. Moore, Ensemble representation learning: an analysis of fitness and survival for wrapper-based genetic programming methods, in *GECCO '17: Proceedings of the 2017 Genetic and Evolutionary Computation Conference* (ACM, Berlin, Germany), pp. 961–968 (2017). https://doi.org/10.1145/3071178.3071215. arxiv:1703.06934
44. W. La Cava, J.H. Moore, Semantic variation operators for multidimensional genetic programming, in *Proceedings of the 2019 Genetic and Evolutionary Computation Conference, GECCO '19* (ACM, Prague, Czech Republic, 2019). https://doi.org/10.1145/3321707.3321776. arXiv:1904.08577
45. W. La Cava, S. Silva, K. Danai, L. Spector, L. Vanneschi, J.H. Moore, Multidimensional genetic programming for multiclass classification. Swarm Evolut. Comput. (2018). https://doi.org/10.1016/j.swevo.2018.03.015
46. W. La Cava, T.R. Singh, J. Taggart, S. Suri, J.H. Moore, Learning concise representations for regression by evolving networks of trees, in *International Conference on Learning Representations, ICLR* (2019). arxiv:1807.00981 (in press)
47. W. La Cava, L. Spector, K. Danai, Epsilon-lexicase selection for regression, in *Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16* (ACM, New York, NY, USA, 2016), pp. 741–748. https://doi.org/10.1145/2908812.2908898
48. Q. Le, B. Zoph, Using machine learning to explore neural network architecture (2017). https://ai.googleblog.com/2017/05/using-machine-learning-to-explore.html. Accessed 15 Oct 2019
49. C. Liu, B. Zoph, J. Shlens, W. Hua, L.J. Li, L. Fei-Fei, A. Yuille, J. Huang, K. Murphy, Progressive neural architecture search. arXiv preprint arXiv:1712.00559 (2017)
50. T. McConaghy, FFX: Fast, scalable, deterministic symbolic regression technology, in *Genetic Programming Theory and Practice IX*, ed. by R. Riolo, E. Vladislavleva, J.H. Moore (Springer, Berlin, 2011), pp. 235–260. https://doi.org/10.1007/978-1-4614-1770-5_13
51. D. Medernach, J. Fitzgerald, R.M.A. Azad, C. Ryan, A new wave: a dynamic approach to genetic programming, in *Proceedings of the Genetic and Evolutionary Computation Conference 2016, GECCO '16* (ACM, New York, NY, USA, 2016), pp. 757–764. https://doi.org/10.1145/2908812.2908857
52. V.V. de Melo, W. Banzhaf, Automatic feature engineering for regression models with machine learning: an evolutionary computation and statistics hybrid. Inf. Sci. (2017). https://doi.org/10.1016/j.ins.2017.11.041
53. G. Montavon, K.R. Müller, Better representations: invariant, disentangled and reusable, in *Neural Networks: Tricks of the Trade, Lecture Notes in Computer Science*, ed. by G. Montavon, K.R. Müller (Springer, Berlin, 2012), pp. 559–560
54. A. Moraglio, K. Krawiec, C.G. Johnson, Geometric semantic genetic programming, in *Parallel Problem Solving from Nature-PPSN XII* (Springer, 2012), pp. 21–31. http://link.springer.com/chapter/10.1007/978-3-642-32937-1_3. Accessed 15 Oct 2019
55. M. Muharram, G.D. Smith, Evolutionary constructive induction. IEEE Trans. Knowl. Data Eng. **17**(11), 1518–1528 (2005)
56. L. Muñoz, S. Silva, L. Trujillo, M3gp—multiclass classification with GP, in *Genetic Programming* (Springer, 2015), pp. 78–91. http://link.springer.com/chapter/10.1007/978-3-319-16501-1_7. Accessed 15 Oct 2019
57. L. Muñoz, L. Trujillo, S. Silva, M. Castelli, L. Vanneschi, Evolving multidimensional transformations for symbolic regression with M3gp. Memet. Comput. (2018). https://doi.org/10.1007/s12293-018-0274-5
58. K. Neshatian, M. Zhang, P. Andreae, A filter approach to multiple feature construction for symbolic learning classifiers using genetic programming. IEEE Trans. Evolut. Comput. **16**(5), 645–661 (2012). (ZSCC: 0000081)
59. R.M. O'brien, A caution regarding rules of thumb for variance inflation factors. Qual. Quant. **41**(5), 673–690 (2007). https://doi.org/10.1007/s11135-006-9018-6. (ZSCC: 0005201)

60. R.S. Olson, W. La Cava, P. Orzechowski, R.J. Urbanowicz, J.H. Moore, PMLB: A large benchmark suite for machine learning evaluation and comparison. BioData Mining (2017). ArXiv preprint arXiv:1703.00512
61. P. Orzechowski, W. La Cava, J.H. Moore, Where are we now? A large benchmark study of recent symbolic regression methods. arXiv:1804.09331 [cs] (2018). https://doi.org/10.1145/3205455.3205539.
62. T.P. Pawlak, B. Wieloch, K. Krawiec, Semantic backpropagation for designing search operators in genetic programming. IEEE Trans. Evol. Comput. **19**(3), 326–340 (2015)
63. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**(Oct), 2825–2830 (2011)
64. H. Pham, M.Y. Guan, B. Zoph, Q.V. Le, J. Dean, Efficient neural architecture search via parameter sharing. ArXiv preprint arXiv:1802.03268 (2018)
65. E. Real, Using evolutionary AutoML to discover neural network architectures (2018). https://ai.googleblog.com/2018/03/using-evolutionary-automl-to-discover.html. Accessed 15 Oct 2019
66. E. Real, S. Moore, A. Selle, S. Saxena, Y.L. Suematsu, J. Tan, Q. Le, A. Kurakin, Large-scale evolution of image classifiers. arXiv:1703.01041 [cs] (2017)
67. M. Schmidt, H. Lipson, Age-fitness pareto optimization, in *Genetic Programming Theory and Practice VIII* (Springer, 2011), pp. 129–146. http://link.springer.com/chapter/10.1007/978-1-4419-7747-2_8. Accessed 15 Oct 2019
68. D. Searson, M. Willis, G. Montague, Co-evolution of non-linear PLS model components. J. Chemom. **21**(12), 592–603 (2007). https://doi.org/10.1002/cem.1084
69. D.P. Searson, D.E. Leahy, M.J. Willis, GPTIPS: an open source genetic programming toolbox for multigene symbolic regression, in *Proceedings of the International Multiconference of Engineers and Computer Scientists*, vol. 1 (IMECS, Hong Kong, 2010), pp. 77–80
70. S. Silva, L. Munoz, L. Trujillo, V. Ingalalli, M. Castelli, L. Vanneschi, Multiclass classification through multidimensional clustering, in *Genetic Programming Theory and Practice XIII*, vol. 13 (Springer, Ann Arbor, MI, 2015)
71. L. Spector, Assessment of problem modality by differential performance of lexicase selection in genetic programming: a preliminary report, in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion* (2012), pp. 401–408. http://dl.acm.org/citation.cfm?id=2330846. Accessed 15 Oct 2019
72. K.O. Stanley, Compositional pattern producing networks: a novel abstraction of development. Genet. Program. Evolvable Mach. **8**(2), 131–162 (2007). https://doi.org/10.1007/s10710-007-9028-8
73. K.O. Stanley, J. Clune, J. Lehman, R. Miikkulainen, Designing neural networks through neuroevolution. Nat. Mach. Intell. **1**(1), 24 (2019). https://doi.org/10.1038/s42256-018-0006-z
74. K.O. Stanley, D.B. D'Ambrosio, J. Gauci, A hypercube-based encoding for evolving large-scale neural networks. Artif. Life **15**(2), 185–212 (2009). https://doi.org/10.1162/artl.2009.15.2.15202
75. K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies. Evolut. Comput. **10**(2), 99–127 (2002). https://doi.org/10.1162/106365602320169811
76. R. Tibshirani, Regression shrinkage and selection via the lasso. J. R. Stat. Soc. Ser. B Methodol. **58**, 267–288 (1996)
77. R. Tibshirani, T. Hastie, B. Narasimhan, G. Chu, Diagnosis of multiple cancer types by shrunken centroids of gene expression. Proc. Natl. Acad. Sci. **99**(10), 6567–6572 (2002). https://doi.org/10.1073/pnas.082099299
78. L. Vanneschi, M. Castelli, L. Manzoni, K. Krawiec, A. Moraglio, S. Silva, I. Gonçalves, PSXO: population-wide semantic crossover, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (ACM, 2017), pp. 257–258
79. E. Vladislavleva, G. Smits, D. den Hertog, Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. IEEE Trans. Evol. Comput. **13**(2), 333–349 (2009). https://doi.org/10.1109/TEVC.2008.926486
80. W. Whitney, Disentangled representations in neural models. arXiv:1602.02383 [cs] (2016).
81. B. Zoph, Q.V. Le, Neural architecture search with reinforcement learning (2016). https://arxiv.org/abs/1611.01578