EvoDAG: A Semantic Genetic Programming Python Library

Mario Graff
CONACYT
INFOTEC Centro de Investigación e Innovación en Tecnologías de la Información y Comunicación
Email: mario.graff@infotec.mx

Eric S. Tellez
CONACYT
INFOTEC Centro de Investigación e Innovación en Tecnologías de la Información y Comunicación
Email: eric.tellez@infotec.mx

Sabino Miranda-Jiménez
CONACYT
INFOTEC Centro de Investigación e Innovación en Tecnologías de la Información y Comunicación
Email: sabino.miranda@infotec.mx

Hugo Jair Escalante
Instituto Nacional de Astrofísica, Óptica y Electrónica,
Luis Enrique Erro 1, Puebla, 72840, Mexico
Email: hugojair@inaoep.mx

***Abstract*—Genetic Programming (GP) is an evolutionary algorithm that has received a lot of attention lately due to its success in solving hard real-world problems. Lately, there has been considerable interest in GP's community to develop semantic genetic operators, i.e., operators that work on the phenotype. In this contribution, we describe EvoDAG (Evolving Directed Acyclic Graph) which is a Python library that implements a steady-state semantic Genetic Programming with tournament selection using an extension of our previous crossover operators based on orthogonal projections in the phenotype space. To show the effectiveness of EvoDAG, it is compared against state-of-the-art classifiers on different benchmark problems, experimental results indicate that EvoDAG is very competitive.**

I. Introduction

Genetic Programming (GP) is an evolutionary algorithm that has received a lot of attention lately due to its success in solving hard real-world problems [14]. Lately, there has been a growing interest in the community to develop semantic genetic operators that use information from the phenotype to create the offspring. Among these semantic operators we can find the geometric semantic crossover proposed by Moraglio *et al.* [11], the Approximately Geometric Crossover (AGX) [12], the Locally Geometric Crossover (LGX) [9], [10], the Krawiec and Lichocki Geometric Crossover (KLX), and our previous works based on the projection on the phenotype space [5], [6], just to mentioned a few.

From these semantic GP systems there are some systems that have the additional advantage of having efficient implementations; being able to evaluate a new individual in $O(n)$ where n is the size of the training set. Among the different semantic operators (see [17] for a recent survey of semantic methods in GP) the ones that seem to have the highest convergence rate were proposed by [5], [6], [3]. These techniques were inspired by the geometric semantic crossover proposed by Moraglio *et al.* [11] with the implementation of Vanneschi *et al.* [16], [2].

In this paper, we present EvoDAG[1], a Python library that implements a steady state GP system using and extending our previous crossover operator [5], [6] based on projections on the phenotype space, namely PrXO. PrXO uses the idea of creating the best offspring that can be produced by a linear combination of the parents. The manuscript contributions are firstly to implement an evolutionary system using PrXO in a library/program that can be easily installed and used by a non-programmer; secondly, to free the user from the tedious task of selecting EvoDAG parameters; thirdly to develop a supervised learning algorithm to find an appropriate model that is competitive with the state of the art in terms of performance and runtime. To show this last contribution, we decided to compare EvoDAG's performance against two state-of-the-art classifiers, namely the SVM implemented in scikit-learn package [13], and auto-sklearn [4] which is an auto-configurable classifier that recently won the AutoML Challenge [7]. These classifiers were compared on thirteen binary classification problems; the experimental results indicate that EvoDAG is very competitive, and statistically outperforms the other classifiers in nine out of thirteen problems in average performance, and in seven problems when the median performance is used instead.

The rest of this manuscript is organized as follows: Section II presents the foundations of EvoDAG whereas Section III is devoted to the implementation issues. The benchmark problems and the parameters used are presented on Section IV. The performance of EvoDAG and the comparison against other classifiers are presented on Section V. Finally, the conclusions and some possible directions for future work are given in Section VI.

II. EvoDAG Foundations

EvoDAG implements a steady state GP system with tournament selection where the only genetic operator is PrXO.

[1] https://github.com/mgraffg/EvoDAG

In order to introduce PrXO, let us start by describing the framework where EvoDAG is developed. EvoDAG is a supervised learning algorithm that learns from the instances of a training set $\mathbb{T}$ formed by $n \in \mathbb{N}$ pairs of inputs and outputs, i.e., $\mathbb{T} = \{(x_i, y_i) | i = 1 \dots n\}$ where x_i represents the i-th input[2], and y_i is the associated output. The objective is to find a function p such that $\forall_{(x,y)\in\mathbb{T}}\ p(x) = y$ and that could be evaluated in any element x of the input space. In general, it is not possible to find a function p that learns perfectly $\mathbb{T}$, consequently, one tries to find a function p that minimize an error function, e.g., sum of squared errors $\sum_{(x,y)\in\mathbb{T}}((y - p(x))^2$.

Let us consider a fixed order in $\mathbb{T}$ to define $\mathbf{t} = (y_1, \dots, y_n) \in \mathbb{R}^n$, namely the target vector, which contains all the outputs in $\mathbb{T}$. Using the order in $\mathbb{T}$, it is possible to define $\mathbf{p} = (p(x_1), \dots, p(x_n))$ that contains the evaluation of individual/function p in all the inputs x of the training set. In this scenario the fitness (using as fitness function the sum of squared error) of individual p can be computed as the square of the euclidean norm $||\mathbf{t} - \mathbf{p}||^2$.

PrXO is based on the ideas of the geometric semantic crossover proposed by Moraglio *et al.* [11]. The geometric semantic crossover is defined as follows: Let p_1 and p_2 be the first and second parent, then the offspring produced by these parent is $o = p_1 r + p_2(1 - r)$, where r is a random function or a constant in the range $[0, 1]$. The output of individual o at input x is computed as $o(x) = p_1(x)r(x) + p_2(x)(1 - r(x))$.

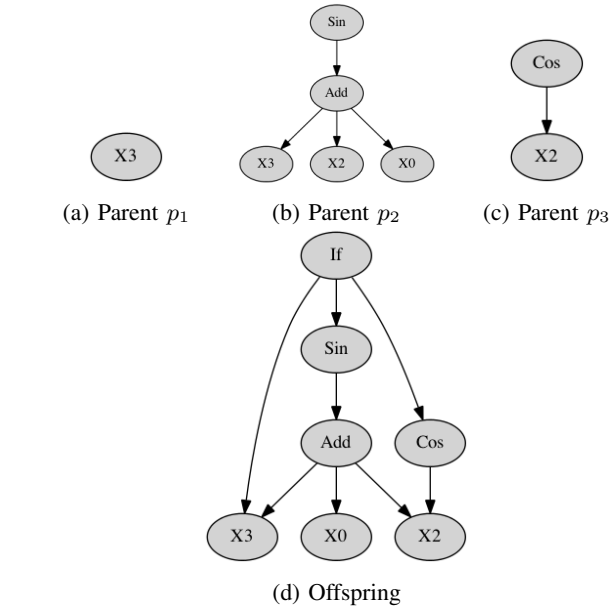
Let us assume that r in the geometric semantic crossover is a constant, then the offspring is just a linear combination of the parents where the offspring lies on the line segment intersecting the parents given the structure of the crossover and the constraint imposed, i.e., $r \in [0, 1]$.

The starting point of PrXO was to remove the constraint and make a straightforward modification to the structure of the geometric semantic crossover. That is, PrXO is defined as follows: let p_1 and p_2 be the first and second parent, then the offspring o is computed as $o = \alpha p_1 + \beta p_2$ where α and β are calculated by solving the following equation $A \cdot (\alpha, \beta)' = \mathbf{t}'$ where $A = (\mathbf{p}_1', \mathbf{p}_2')$, $\mathbf{p}_i = (p_i(x_1), \dots, p_i(x_n))$ is the evaluation of parent i in all the inputs, and $\mathbf{t}$ is the target vector. By construction, the offspring o is the projection of $\mathbf{t}$ on the plane produced by the linear combination of $\mathbf{p}_1$ and $\mathbf{p}_2$ in the case of crossover. Given that o is created to minimize $||\mathbf{t} - (o(x_1), \dots, o(x_n))||$; so, it corresponds to the orthogonal projection of $\mathbf{t}$ in that plane. Figure 1 depicts this process where A and B play the role of $\mathbf{p}_1$ and $\mathbf{p}_2$, it is observed target vector $\mathbf{t}$ which is outside the plane and the offspring is the orthogonal projection of $\mathbf{t}$ on the plane generated. Consequently, if the fitness function is the euclidean distance then the offspring has at least the fitness of the best parent.

So far, it has been described that PrXO generates an offspring that is the linear combination of two parents. From this point, it is evident that PrXO has the constraint that the offspring has only two parents besides that all the offspring has as root the sum; however, these restrictions can be easily removed by following the next steps:

1) Let $\{p_1, \dots, p_k\}$ be the arguments of a function f randomly selected from the function set, where k indicates the cardinality of function f.
2) In the case, f is the sum, i.e., $\sum^k$, the offspring is $o = \sum_i^k \alpha_i p_i$.
3) On the other hand, the offspring is defined as $o = \alpha f(p_1, \dots, p_k)$ where α is computed to minimize $||\mathbf{t} - \alpha f(p_1, \dots, p_k)||$.

Figure 2 presents an example of the process used to generate an offspring. Let us assume that if is selected from the function set, then it is needed to select three arguments, e.g., p_1 (Figure 2a), p_2 (Figure 2b), and p_3 (Figure 2c). Hence the offspring generated is depicted on Figure 2d.



(a) Parent p_1 (b) Parent p_2 (c) Parent p_3

(d) Offspring

Fig. 2: Process to generate an offspring when if is selected from the function set, and the arguments are p_1, p_2, and p_3

The idea of generating an offspring using a function with

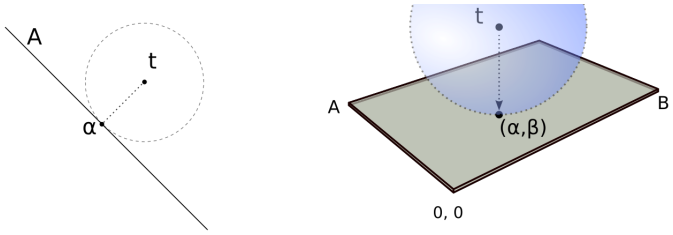


Fig. 1: The optimization process of coefficients. The orthogonal projection is finding the nearest point of target $\mathbf{t}$ into the hyperplane generated by the offspring and its coefficients. On the left, it will find α, i.e., the point being orthogonal to $\mathbf{t}$. On the right, the procedure on the plane (here, it finds two constants).

[2]Please note that x_i could be an input-vector or a scalar

more than two arguments was tested on [5] only for functions $\sum_k$ and if. We extend our previous work by including more functions of many arguments, namely $\prod_k$, $\min_k$, and $\max_k$.

III. EvoDAG Implementation and Usage

Let us start by describing the notation used and a general overview of the library. Firstly, **Population.hist** refers to the *hist* attribute of an instance of the *Population* class. Secondly, EvoDAG is composed by several classes, the ones that manage the evolutionary process are: *EvoDAG*, *Population*, and *Variable*. The search in the parameter space is performed by *RandomParameterSearch*. The model is stored in either *Model* or *Models*, and the prediction model can also take the form of an ensemble and this is handled by *Ensemble*.

EvoDAG is inspired by the geometric semantic GP developed in [16], [2]. The idea followed is that all the individuals generated are stored in a table, here all the individuals can be found in **Population.hist** which is a list of *Variable* instances.

A *Variable* is the root of a particular individual, and, one can build the complete individual from it by performing a depth-first search. The nodes (*Variable*) connected to the current one are accessed through the attribute **Variable.variable** which is a list that contains the position in *Population.hist* of the connected nodes. The fitness in the training and validation set is also stored in attributes **Variable.fitness** and **Variable.fitness_vs**, respectively. As well as, the output in the training set **Variable.hy**, the output in the test set **Variable.hy_test**, its position in **Population.hist** can be retrieved from **Variable.position**, and the coefficient(s) (i.e., α) associated to it are in **Variable.weight**.

A. Initial Population

EvoDAG uses an unusual procedure to initialise the population. It is worth to mention that in our previous research work [5], EvoDAG initialisation mechanism does not present any difference in performance with the traditional ramped half-and-half method; nonetheless, it is considerable simpler to implement. EvoDAG initialisation method fills the initial population with individuals that only have inputs without any duplication; in the case the population size is greater than the number of inputs, then, the remaining individuals are generated as follows. Let us assume there are ℓ inputs, then, the $\ell + 1$ individual is generated by randomly selecting a function from the function set. Assume f_k is selected, where k is the number of arguments of f, the next step is to select k individuals from the ℓ individuals to serve as the k arguments to function f. At this point, the individual $\ell+1$ is ready to be in the population and it could be selected as an argument for the individual $\ell+2$. This process continues until the number of individuals reaches the population size.

The method that creates the initial population is **EvoDAG.create_population**, and the method that inserts an individual to the population is **EvoDAG.add** which calls **Population.add**. It is worth to mention that before an individual is added to the population the α coefficients are computed as well as its fitness in the training and validation set. In the case the fitness is not a number or α could not be computed then the individual is discarded a new individual is generated.

B. Evolution

Once the initial population has been created, the evolutionary process starts following either a steady-state or generational approach. The evolution stops until an stopping criteria (**EvoDAG.stopping_criteria**) is achieve. Then at each step an offspring is created (**EvoDAG.random_offspring**) using the following procedure. A function f_k is randomly drawn from the function set, and, then the k arguments are selected from the population using tournament selection (**Population.tournament**). Then the offspring replace an individual of the current population using a negative tournament selection, and the process continues. The method **EvoDAG.replace** that calls **Population.replace** is responsible of replacing an individual of the current population with the offspring.

In previous research work (see [6]) it has been evident that a GP system using PrXO suffers from overfitting. In fact, in order to avoid overfitting in [5] was decided to use a very small population size (i.e., 10); however, EvoDAG uses a different strategy borrow from the machine learning community. EvoDAG stops the evolution using an early stopping criteria. That is, each evolved individual is tested on a validation set, and, the evolution stops when it is impossible to improve the fitness in the validation set for a number of rounds. The validation set is taken from the training set by randomly selecting a 20% of the training data. Clearly, this approach is debatable and further research is needed to understand the effects of this decision.

At this point, it is convenient to exemplify and individual evolved by EvoDAG. Figure 3 depicts an individual evolved by EvoDAG doing the following. Let $p = [x_1, x_2, x_3, (\exp x_2)]$ be the initial population and p_i refers to the i-th individual in population p. The first offspring was created by selecting function if from the function set and as arguments p_1, p_4, and p_3 (i.e., x_1, $(\exp x_2)$, and x_3) resulting in $o = (\text{if } x_1\ (\exp x_2)\ x_3)$. Offspring o was set to p_2 thus the population is $p = [x_1, (\text{if } x_1\ (\exp x_2)\ x_3), x_3, (\exp x_2)]$. The second offspring was created by selecting $\ln$ from the function set and p_3 as argument resulting in $(\ln x_3)$ this offspring was set in p_4 leaving the population as: $p = [x_1, (\text{if } x_1\ (\exp x_2)\ x_3), x_3, (\ln x_3)]$. Finally, $\sum^k$ was randomly selected from the function set where $k = 4$, and, the arguments were p_1, p_2, p_3 and p_4 this produces as offspring the individual depicted on Figure 3.

Figure 3 shows, in a simple example, the reason behind the name of EvoDAG. That is, it can be observed that the function evolved is represented as a directed acyclic graph, this is a natural result of the implementation, as well as the modifications performed to the geometric crossover. Normally, one would expect that a GP system having its origins in the geometric semantic operator would evolve trees; however, as can be seen, this is not the case of EvoDAG.

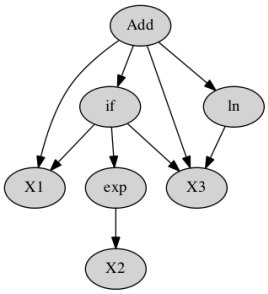


Fig. 3: An individual generated by EvoDAG

C. Model

At the end of the evolutionary process EvoDAG delivers a model that can be a classifier or a regressor. In the case the problem is a symbolic regression or a binary classification, EvoDAG returns an instance of *Model*. In multi-class problems, EvoDAG uses the one-vs-all strategy so it returns a list of *Model* one for each class; however, in order to facilitate its use, a wrapper was developed (*Models*) that takes care of all the details of one-vs-all strategy.

In our previous research work [5], it was observed that the performance of PrXO improves considerable by creating an ensemble. Based on this, EvoDAG is able to evolve a simple ensemble (*Ensemble*) by automatically selecting different training and validation data, and the predictions of the ensemble is the median of the individuals predictions. In the case of classification, the decision function, of each classifier, is used (bounded to $[-1, 1]$), and, the label is taken from the median of the decision function values.

D. Search in the Parameter Space

EvoDAG as any other evolutionary algorithm has many parameters that need to be set in order to obtain an acceptable performance. The process of selecting the parameters is tedious and for a non-expert it could be very time consuming. In order to facilitate this process, it was decided to implement a random search strategy in the parameter space. Clearly, there is a huge amount of work in this topic an further research is needed; however, we decided to use this simple technique because it has been proved that (see [1]) random search is better, most of the time, that manually tuning the parameters. *RandomParameterSearch* deals with the search process and the best configuration is the one that obtained the best median fitness of three independent trails using as performance the balance error rate or root square error, in case of classification or regression problems, respectively. Table I presents the possible values of EvoDAG parameters. The unique individuals parameter indicates whether the evolution should produce unique individuals or not, this is performed at the genotype so it could be still possible to evolve equivalent individuals in the phenotype.

E. Installation

The first step in order to use EvoDAG is to install it. Let us assume there is a working Python environment in your system[3]. If it is not the case, one could easily install Anaconda Python distribution. Then EvoDAG dependencies can be installed as follows:

TABLE I: EvoDAG possible parameters values. Each parameters is a subset randomly selected from the set of possible values, the only constrain is that $\mathcal{F}$ always contains $\sum$.

Parameter	Set of possible values
Function Set ($\mathcal{F}$)	$\{\sum_k, \prod_k, /, \exp, \sqrt{}, \sin, \cos, \ln, (\cdot)^2, \text{sigmoid}, \min_k, \max_k, \text{if}\}$
k	$\{0, 2, 5, 10, \ldots, 30\}$
Unique individuals	$\{\text{true}, \text{false}\}$
population size	$\{500, 1000, 2000, 3000\}$
Early stopping rounds	$\{125, 250, 500, 1000, 2000\}$

```
$ pip install numpy
$ pip install cython
```

After the dependencies have been correctly installed on the system, it is time to install EvoDAG using either:

```
$ pip install EvoDAG
$ conda install -c mgraffg evodag
```

F. Usage

At this point EvoDAG is ready to be used. EvoDAG can read the training set from a CSV (comma separated values) file, where each row represents a pattern and each column is an input, the last column correspond to the class. Let us assume that the training set is *t.csv*, it is required to perform a sampling (five samples) in the parameter space, and, to store the fitness in (*p.evo*):

```
$ EvoDAG-params -C -r 5 -P p.evo t.csv
```

Once the configuration has been found, it is time to train a model, let us train an ensemble of 7 classifiers, and, store it in *m.evo*:

```
$ EvoDAG-train -n 7 -P p.evo -m m.evo t.csv
```

Finally, let us assume one would like to predict some inputs values that are stored in *test.csv*, and, store these predictions in *p.evo*:

```
$ EvoDAG-predict -m m.evo -o pr.evo test.csv
```

IV. Problems and Parameters Settings

It was decided to compare the EvoDAG performance in terms of the balance error rate (BER) against two different classifiers that can be used in Python. The first one is the well known Support Vector Machine (SVM), the SVM used is the one implemented in the scikit-learn package [13], and the second classifier[4] is auto-sklearn [4] which is an auto-configurable classifier that recently won the AutoML Challenge [7].

In order to make this comparison as fair as possible, different SVM configurations were tested making an uniform

[3]EvoDAG has only been tested on Linux and OSX

[4]Please note that auto-sklearn is in fact a meta-classifier: this method searches in the space of models that can be build with the sklearn library.

sampling from the parameters space (see [1]). Table II presents the different parameter values of the SVM that were tested. Specifically, three different tasks were created: the first one trained a SVM with a linear kernel, where 734 different configurations of the parameters C and whether the intercept coefficient were used; the second task corresponds to the polynomial kernel (see the table for the parameters sampled); and the third task corresponds to the RBF and sigmoid kernels. From these 2202 different SVM configurations, it was selected the configuration that obtained the lowest median BER on a validation set (20% of the training data) in three independent trails each one using a different validation set. Lastly, an ensemble of 100 SVM was trained to perform the prediction. This ensemble is equivalent to the one used in EvoDAG, that is, each SVM was trained by selecting 80% of the training set instance, then the final prediction is the median of the decision function output.

On the other hand, auto-sklearn was used with its default parameters given that auto-sklearn search in the parameter space of its classifiers as well as in the parameter space of its implemented pre-processing techniques.

TABLE II: SVM possible parameters values according to the kernel.

Parameter	Set of possible values
All kernels	
C	Exponential random variable (scale equals 100)
Linear kernel	
fit intercept	$\{\mathrm{true}, \mathrm{false}\}$
Polynomial kernel	
degree	$\{2, 3, 4, 5\}$
γ	Exponential random variable (scale equals 0.1)
RBF and sigmoid kernels	
γ	Exponential random variable (scale equals 0.1)

The problems used in the comparison are the Gunnar Raetsch's Benchmark Datasets [15][5]. This benchmark was selected because it has been used traditionally to measure the performance of classification methods, in addition to this, it presents varied characteristics in terms of number of input features, number of training objects, among others, resulting in a rich suite of classification problems (see Table IV). Each dataset consists in 100 training and test pairs, the exceptions are *image* and *splice* which contain only 20 trails.

V. Results

Table IV presents the average performance with 95% confidence intervals in terms of the balance error rate (BER) of auto-sklearn, SVM, and EvoDAG on thirteen classification problems. The best performance among each classification dataset is in bold face to facilitate the reading; it was compared against the others in order to know whether the difference in performance was statistically significant. The superscript * indicates that the difference between the best performance and the one having the superscript is statistically significant with a 95% confidence. This statistical test was performed using the Wilcoxon signed-rank test [18] and the p-values were adjusted using the Holm-Bonferroni method [8] in order to consider the multiple comparisons performed.

TABLE III: Data sets used in the comparison.

Dataset	Input Features	Training Patterns	Test Patterns
Banana	2	400	4900
Titanic	3	150	2051
Thyroid	5	140	75
Diabetes	8	468	300
Breast-Cancer	9	200	77
Flare-Solar	9	666	400
Heart	13	170	100
Ringnorm	20	400	7000
Twonorm	20	400	7000
German	20	700	300
Image	20	1300	1010
Waveform	21	400	4600
Splice	60	1000	2175

It can be observed from Table IV that SVM obtained the best performance in four of the datasets, and EvoDAG obtained the best performance in the rest of the datasets (nine). In all the cases, the difference in performance was statistically significant. One characteristic that caught our attention is the high confidence intervals of auto-sklearn, it is one order of magnitude higher than the other systems. Analyzing the predictions performed by auto-sklearn, it is found that in some of the trails the algorithm predicts only one class, obtaining, consequently, the worst possible performance, i.e., BER equals 50. This behaviour, clearly, can be automatically spotted, and, one possible solution could be as simple as execute auto-sklearn again on that particular case. Nonetheless, we decided to keep auto-sklearn without modification, and, instead, it is decided to include another table that presents the performance using the median.

TABLE IV: Average performance in term of the balance error rate with 95% confidence intervals. Superscript * indicates that the difference in performance between the best (bold face) and the system with the superscript is statistically significant with a confidence of 95%.

Dataset	auto-sklearn [4]	SVM	EvoDAG (0.3.6)
banana	$28.00 \pm 3.69^*$	$\mathbf{11.27 \pm 0.18}$	$12.14 \pm 0.16^*$
titanic	$37.18 \pm 1.64^*$	$30.27 \pm 0.36^*$	$\mathbf{30.02 \pm 0.27}$
thyroid	$23.38 \pm 3.99^*$	$\mathbf{6.13 \pm 0.76}$	$8.22 \pm 0.75^*$
diabetis	$37.65 \pm 2.01^*$	$26.65 \pm 0.44^*$	$\mathbf{24.70 \pm 0.39}$
breast-cancer	$42.36 \pm 1.38^*$	$36.25 \pm 1.04^*$	$\mathbf{34.42 \pm 1.06}$
flare-solar	$39.05 \pm 1.49^*$	$33.41 \pm 0.38^*$	$\mathbf{32.93 \pm 0.33}$
heart	$27.69 \pm 2.85^*$	$18.12 \pm 0.63^*$	$\mathbf{16.49 \pm 0.67}$
ringnorm	$15.49 \pm 4.24^*$	$\mathbf{1.96 \pm 0.10}$	$2.47 \pm 0.07^*$
twonorm	$20.87 \pm 4.49^*$	$2.90 \pm 0.09^*$	$\mathbf{2.68 \pm 0.04}$
german	$39.45 \pm 1.62^*$	$29.00 \pm 0.50^*$	$\mathbf{28.86 \pm 0.52}$
image	$21.29 \pm 10.54^*$	$\mathbf{3.32 \pm 0.29}$	$4.32 \pm 0.48^*$
waveform	$22.67 \pm 3.53^*$	$10.62 \pm 0.21^*$	$\mathbf{10.37 \pm 0.09}$
splice	$10.79 \pm 7.43^*$	$11.23 \pm 0.37^*$	$\mathbf{9.88 \pm 0.53}$

Table V presents the median performance (BER) of the different classifiers, it is observed from the table, that auto-sklearn obtained the best performance on two datasets, SVM

[5]These datasets are available in http://theoval.cmp.uea.ac.uk/matlab/benchmarks or http://ws.ingeotec.mx/∼mgraffg/classification

obtained the best performance on three datasets, and EvoDAG had the best performance in eight datasets. Comparing the average and median performance, it can be observed that EvoDAG had the best average performance in *titanic*, and on median auto-sklearn has the best performance on this dataset; whereas, SVM had the best average performance in *image* dataset, and on median performance auto-skearn has the best performance on this dataset.

TABLE V: Median performance in terms of the balance error rate of the different classifiers

Dataset	auto-sklearn [4]	SVM	EvoDAG (0.3.6)
banana	13.39	**11.02**	11.97
titanic	33.10	29.63	**29.57**
thyroid	11.96	**5.95**	8.07
diabetis	31.17	26.58	**24.53**
breast-cancer	41.64	35.38	**33.93**
flare-solar	34.90	33.31	**32.88**
heart	20.68	18.28	**16.43**
ringnorm	2.07	**1.83**	2.44
twonorm	3.29	2.83	**2.66**
german	36.25	28.88	**28.75**
image	**3.06**	3.41	4.36
waveform	11.41	10.45	**10.39**
splice	**3.39**	11.07	10.20

Experimental results presented so far demonstrate the competitiveness of EvoDAG when facing classification problems and when compared to state of the art methodologies.

VI. Conclusions

In this contribution, we have presented EvoDAG which is a GP system with tournament selection and projection crossover. EvoDAG implements and extends our previous work [5] in a Python library that can be easily installed and used from the command line. In order to free the user from the tedious process of selecting EvoDAG's parameters, it was decided to make an uniform sampling of the parameter space using the idea described by [1], and for each different set of parameters three instances of EvoDAG are executed and the performance is the median of balance error rate and root squared errors in classification and regression, respectively. Finally, it is recommended to use EvoDAG as an ensemble, in particular, using the median of each individual classifier as final prediction. In the case of classification, all the single predictions are bounded to $[-1, 1]$.

The EvoDAG performance (using BER) was compared against two state-of-the-art classifiers both are available from Python packages, namely, auto-sklearn [4] and SVM [13]. The results show that EvoDAG is very competitive outperforming in nine out of thirteen of the classification problems tested when the comparison is performed using the average. On the other hand, when the performance is measured as the median, EvoDAG outperformed the other classifiers in eight out of the thirteen problems tested.

As future work, we would like to evolve classifiers that can deal with multiple classes and labels, as well as to evolve a model that produces multiple continuous outputs.

References

[1] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[2] Mauro Castelli, Sara Silva, and Leonardo Vanneschi. A C++ framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 16(1):73–81, April 2014. 00004.

[3] Mauro Castelli, Leonardo Trujillo, Leonardo Vanneschi, Sara Silva, Emigdio Z-Flores, and Pierrick Legrand. Geometric Semantic Genetic Programming with Local Search. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, GECCO '15, pages 999–1006, New York, NY, USA, 2015. ACM. 00000.

[4] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[5] Mario Graff, Eric S. Tellez, Hugo Jair Escalante, and Sabino Miranda-Jimenez. Semantic Genetic Programming for Sentiment Analysis. In *NEO 2015*, number 663 in Studies in Computational Intelligence, pages 43–65. Springer International Publishing, 2016.

[6] Mario Graff, Eric Sadit Tellez, Elio Villasenor, and Sabino Miranda-Jiménez. Semantic genetic programming operators based on projections in the phenotype space. *Research in Computing Science*, 94:73–85, 2015.

[7] I. Guyon, I. Chaabane, H. J. Escalante, S. Escalera, D. Jajetic, J. R. Lloyd, N. Macía, B. Ray, L. Romaszko, M. Sebag, A. Statnikov, S. Treguer, and E. Viegas. A brief review of the ChaLearn AutoML challenge. In *Proc. of AutoML 2016@ICML*, 2016.

[8] Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, January 1979. 10011.

[9] Krzysztof Krawiec and Tomasz Pawlak. Locally Geometric Semantic Crossover. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 1487–1488, New York, NY, USA, 2012. ACM. 00014.

[10] Krzysztof Krawiec and Tomasz Pawlak. Quantitative Analysis of Locally Geometric Semantic Crossover. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, number 7491 in Lecture Notes in Computer Science, pages 397–406. Springer Berlin Heidelberg, 2012. 00002.

[11] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *Parallel Problem Solving from Nature - PPSN XII*, number 7491 in Lecture Notes in Computer Science, pages 21–31. Springer Berlin Heidelberg, January 2012.

[12] T. Pawlak, B. Wieloch, and K. Krawiec. Semantic Backpropagation for Designing Search Operators in Genetic Programming. *IEEE Transactions on Evolutionary Computation*, Early Access Online, 2014.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, March 2008. 00724.

[15] G. Rätsch, T. Onoda, and K.-R. Müller. Soft margins for adaboost. *Machine Learning*, 42(3):287–320, 2001.

[16] Leonardo Vanneschi, Mauro Castelli, Luca Manzoni, and Sara Silva. A new implementation of geometric semantic GP and its application to problems in pharmacokinetics. In Krzysztof Krawiec, Alberto Moraglio, Ting Hu, A. Şima Etaner-Uyar, and Bin Hu, editors, *Genetic Programming*, number 7831 in Lecture Notes in Computer Science, pages 205–216. Springer Berlin Heidelberg, January 2013.

[17] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. A survey of semantic methods in genetic programming. *Genetic Programming and Evolvable Machines*, 15(2):195–214, June 2014.

[18] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80, December 1945.