Selective Equation Constructor: A Scalable Genetic Algorithm

Lauren Heller
Computer Science Department, Western Washington University
Bellingham, Washington 98225, USA
Email: hellerl@wwu.edu

Michail Tsikerdekis
Computer Science Department, Western Washington University
Bellingham, Washington 98225, USA
Email: Michael.Tsikerdekis@wwu.edu

***Abstract*—Efforts to improve machine learning performance begin with defining a valuable feature set. However, datasets with copious amounts of attributes can have relevant information that is obscured by its high dimensionality, which can be caused by repetitive characteristics or irrelevant qualities. Genetic algorithms provide improvements to feature sets through dimensionality reduction and feature construction. Most genetic algorithms follow the theoretical framework of evolutionary theory where a population of features randomly evolves through generations through a series of random operations such as crossover and mutation. While successful, the randomness of feature modification operations and derived constructed features may yield offsprings that under-perform compared to their ancestors, yet their properties are utilized in future generations. We developed a new genetic algorithm called Selective Equation Constructor (SEC) that evolves constructed features selectively in order to limit the shortcomings of other genetic algorithms. The algorithm leads to faster computation and better results compared to similar algorithms. Analysis of the results indicates increases in classification accuracy, decreased run time, and reduction in attribute count.**

***Index Terms*—genetic, programming, algorithm, equation, constructor**

I. Introduction

It is quite difficult to look at a feature set and determine which components will allow a machine learning task to perform well. An answer to this solution is using techniques to enhance a known feature set. Evolutionary algorithm methods can improve classification accuracy through feature selection or feature construction [1]. Feature construction, in particular, is a technique that leads to dimensionality reduction by combining several features into a single feature or series of features. The term has been popularized over the past few decades by many algorithms that are inspired from evolutionary theory (e.g., [2], [3], [4], [5], [6], [7], [8]). Genetic algorithms have demonstrated higher accuracy in classification as well as continuous predicted variable problems in a variety of machine learning problems. However, such techniques that follow a random evolution paradigm can lead to the creation of some unintelligible results if they are expansive but not accurate. A general limitation of genetic algorithms are operations that cause random modifications to "chromosomes" (i.e., feature structure) that are not necessarily moving the evolutionary process in the right direction. Consequently, many genetic algorithms have to operate longer in order to allow for the process of evolution to generate meaningful results.

978-1-5386-5150-6/18/$31.00 © 2018 IEEE

Our proposed evolutionary algorithm intends to avoid superfluous feature development through critical analysis of newly constructed features as it generates them. Higher classification accuracy in shorter computational time is achieved by:

- Ensuring that constructed features are tested for quality. We require that constructed features are "meaningful" to the evolutionary process and learning algorithm, i.e., it must be mathematical constructs that can be processed and contribute positively to the evolutionary process.
- Allowing a user to supply the modeling and fitness functions while preserving a rigorous cross-validation procedure inherently in the algorithm's structure.
- Adopting parameterization that allows for domain specific information to inform the evolutionary process (e.g., weighted symbol evolution).

The algorithm is being developed for a release as an R package. The latest development version can be accessed at: https://gitlab.cs.wwu.edu/tsikerm/selective-equation-constructor.

The rest of the paper is organized as follows. In section II, an overview of representative genetic algorithms is presented along with a discussion on their limitations. The goal is to frame this work in relation to the larger genetic algorithm domain. This also aims to highlight some of the contributions of our genetic algorithm. Section III is a comprehensive description of our proposed algorithm. Section IV presents the performance results for the proposed algorithm. Finally, Section V introduces the implications of the proposed genetic algorithm and proposes future research directions.

II. Related Works

The task of transforming feature sets for classification problems can be organized into separate categories: feature construction, selection, and reduction. Feature construction aims to generate new high-level attributes from combinations of original features in order to obtain improved outcomes for a problem [9]. Feature selection attempts to select a smaller subset from the original set of larger features [10]. While both selection and construction are capable of reducing the size of feature sets in classification problems they have a key difference. With feature selection, original features may not be descriptive enough to accomplish promising performance when used for machine learning. With feature construction, the features can be assembled in a manner that improves classification accuracy [11]. Evolutionary algorithms aim to reduce data dimensionality since having a high order feature space can mislead a machine learning algorithm [12]. The burden of examining the complete original set of features is diminished by taking away redundancies in the feature space. Two major approaches for this task are genetic programming and genetic algorithms. There are several differences between the two, one of which being the representation of the function (feature) that is constructed. Given the broad literature for these two approaches, we have identified three significant topics on these methods: dimension reduction, decision tree related approaches and grammar evolution. These are not meant to be exhaustive but representative for the context of this paper.

A. Dimension Reduction

Having a good feature space is important for achieving high machine learning performance. Typically, when faced with a complicated problem, good feature space is not always apparent. In some instances, the data may have high dimensionality (i.e., thousands of attributes). In a study by Tran, Xue, and Zhang [13], an embedded feature construction approach is used to develop new high-level features. This method improves from feature selection where the features can only be chosen from the original set of features for classification. In an embedded genetic programming (GP) for feature construction, new high-level features are introduced along with the original features, and a classifier. The methodology used by Tran et al. implements a tree based representation with a population (i.e., constructed features) where each individual is a tree. The nodes for the tree are either random constants or original feature values. With every individual, the algorithm can generate a new value from the original features, thus every individual is considered a constructed high-level feature. Following each GP iteration, the best results are used to generate unique feature subsets. The subsets are tested against training results from several different classification algorithms such as K-nearest Neighbor and Naive Bayes. The study found that even though gene expression data has a large number of features, only a small number of features are relevant to resolving problems (e.g., accurate prediction).

B. Decision Tree Related

An early feature construction method, FRINGE [14], utilized exhaustive methods on decision trees to generate all possible boolean function combinations of a set of original features. In an attempt to improve on this approach, Markovitch and Rosenstein developeped FICUS [15]. This decision tree feature construction algorithm utilizes user supplied information on feature types, domain and range of basic features as well as constructor functions. From these specifications the feature space is only generated from legitimate constructed features. However, one of the method's drawbacks is that feature interaction leads to a narrow search in the feature space. Another limitation is that for any given problem the decision of proper operators is frequently ambiguous therefore there is a need for the user to supply an appropriate set of constructor functions. A similar decision tree algorithm for classification utilized the Wine[1] dataset as a benchmark to learn more about the effects of constructed features [9]. The results indicated the importance of these constructed features in relation to the success of the algorithm's performance.

C. Grammar Evolution

A more complex genetic algorithm has also been tested for feature selection and construction by Gavrilis et al. [16]. The feature construction and selection in this algorithm is based off a genetic programming method introduced by Krawiec [17]. This algorithm aims to improve classification accuracy via grammatical evolution. Grammatical evolution utilizes an evolutionary algorithm and a context-free grammar to produce a progression of terminal symbols [18]. Generally, grammatical evolution methods are represented as integers and each gene signifies a production rule from a given set as specified in the grammar. During evolution, the algorithm replaces all non-terminal symbols with selected production rules and this process continues until the end of a chromosome. If no legitimate expression is composed, the process is continued at the front of the chromosome, giving it a wrapping effect. The method utilizes the following steps. Source data are separated into two sets, one for training and the other for testing. The genetic algorithm parameters are defined. Applying this restriction reduces the formation of large expressions. A grammar that defines all possible algebraic expressions of the original feature set is created. All parts of each chromosome within the genetic pool are randomly assigned. Fitness is evaluated for each chromosome based on an evaluation function. Then, genetic operators of mutation and crossover are applied and a new generation of chromosomes are produced. Within the crossover procedure, chromosomes with the least fitness value are replaced. Finally the algorithm is terminated if a chromosome classification accuracy (fitness value) is greater than a predetermined value or the maximum number of evolutionary generations has been reached. Limitations of such methods often result from the fixed chromosome, which provides an upper boundary on the

[1]Predicting wine cultivars source based on a series of wine attributes

number of features that can be included in the constructed feature regardless of whether there are more features that could be useful if included. Another common limitation is that the use of operations such as crossover and mutations substantially change the structure of an existing feature and while the feature may remain in the population it subsequently may be selected for future evolution.

III. Selective Equation Constructor Algorithm

Our Selective Equation Constructor (SEC) algorithm consists of several steps that work together in order to ensure code modularity as well as opportunities for optimization, whether for computational performance or accuracy. Several parameters are also introduced in order to better adapt the algorithm to various domains of application. Algorithm components are shown on figure 1. In order for the genetic algorithm to be scalable, the Core Evolutionary Function (CEF) can be executed in parallel. The Evolve Function generates new constructed features that are then evaluated and passed through an x-times (not shown in the figure) n-fold cross-validation test. The Model Function (MF) is the function that a user supplies to SEC. MF is expected to utilize a particular machine learning algorithm and to output a numeric result bounded between 0 and 1 (e.g., classification error, f-measure). Results are aggregated, evaluated and stored in a commonly shared storage between CEF processes. The process then repeats until certain conditions are met (described in section III-B). The common shared storage needs to be ACID (atomicity, consistency, isolation and durability) compliant. This helps ensure that constructed features are tested only once and that each process "evolves" a different part of the tree of constructed features. It also helps outsource some of the computational cost outside of the main process (e.g., storage could be running on a secondary computer or cluster). Further, any of the components can be replaced with additional components given the modular state of the algorithm (e.g., replacing the *Evolve* function with a grammar-based evolve function).

A. Feature Evolve Function

Our feature evolution function utilizes string processing and a rule set to evolve features. Symbols have properties that are associated with them that form the complete rule set. The main two types of function symbols are those that are added to a formula and expand it (e.g., addition or subtraction) and those that are applied to a formula or parts of it (e.g., logarithm or absolute).

The process works as follows:

- Apply weighted selection from a series of symbols (e.g., $+, -, *$ etc.). Weights are either applied uniformly or updated by the main algorithm (algorithm 1).
- If symbols require a feature to be selected, e.g., $+$.
  - Select a feature or random number from a specific range.
  - If the symbol is allowed to enclose items in parentheses, decide at random on whether parentheses will be applied.

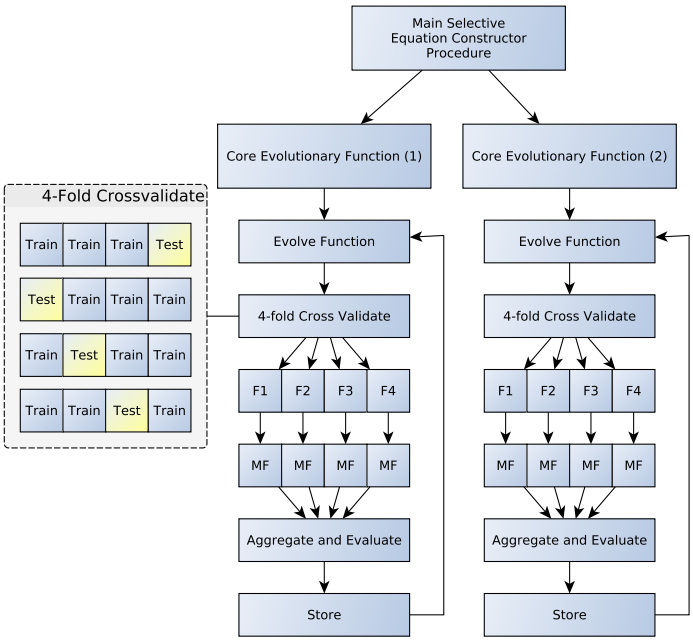

Fig. 1. Main steps of Selective Equation Constructor (SEC) algorithm.

    - $*$ *true*, decide at random whether the formula will be encircled in parentheses as a whole or partially.
      - · *true*, encircle the whole formula in parentheses before adding a new symbol, e.g., $a*b+c \implies (a*b+c)+d$.
      - · *false*, find symbols that exist in the formula and are good candidates for placing parentheses after them. If they exist, pick one of them at random then apply parentheses, e.g., $a*b+c \implies a*(b+c)+d$, otherwise, just add the new symbol without parentheses, e.g., $a*b+c \implies a*b+c+d$.
    - $*$ *false*, add symbol without parentheses, e.g., $a*b+c \implies a*b+c+d$.
- If the symbol does not require a feature to be selected, e.g., log.
  - Decide at random whether the symbol will apply parentheses to the formula as a whole or partially
    - $*$ *true*, apply symbol to the whole formula, e.g., $a*b+c \implies log(a*b+c)$.
    - $*$ *false*, find symbols that exist in the formula and are good candidates for placing parentheses after them. If they exist, pick one at random and apply the symbol with parentheses, e.g., $a*b+c \implies a*log(b+c)$, otherwise, enclose the whole formula, e.g., $a*b+c \implies log(a*b+c)$.

The main premise behind our feature evolution function is that it can gradually expand an existing feature by adding more parts to it. In other words, constructed features keep being expanded and never contracted. This is a major departure from chromosome-based genetic algorithms that often have fixed

size apriori. However, in our algorithm, we want to expand constructed features as long as we keep yielding statistically positive improvements from their "geneological" tree.

B. Core Evolutionary Function

The core process of the algorithm that can run in parallel is presented in algorithm 1. Parameter id identifies each unique process and helps set a different seed for the programming language's pseudorandom generator. Communication between multiple CEF processes occurs through a common shared storage (e.g., database) and hence a series of storage parameters are passed through $strg\_params$. In lines 2 to 5, the algorithm initializes variables, connects to a common storage and selects a $parent$ constructed feature from $db.candidates$ that may already exist in the storage (e.g., if another CEF process has already stored results or the algorithm resumes from a previous result set). db for presentation purposes represents a tuple of tuples and access to subtuples (that may contain subsequent tuples) is highlighted using the *tuple.subtuple* notation. When $db.all$ (that contains all tested formulas) is initialized a default feature is constructed as $y = 1$, which becomes the initial root node ($parent$) for feature evolution. This can be seen on fig. 2. The function *LockPickRelease* locks the shared storage (so that other processes have to be placed in a queue to access it), picks a new $parent$ node from $db.candidates$ and if another $parent$ is provided as a parameter, it releases that record in $db.candidates$ so that other CEF processes can reserve it. The selection of a new parent is performed using weighted sampling where the corresponding weights are based on the $\overline{results}$ (5th item) in the 5-tuple stored in $db.candidates$. The weighted selection of a parent helps ensure that while all candidate constructed features (CFs) could potentially be selected to be evolved, the more successful ones have a higher probability. This results in the algorithm reaching better results faster through selecting more successful CFs but still maintaining an escape route from any local minima that may exist in the dataset.

CEF's main loop starts at line 8. The loop persists until the number of uniquely tested constructed features reaches the $generations$ parameter. The name of the parameter loosely relates to how it is used in other genetic algorithms, however, as per our definition it does not imply geneological depth but a combination of breadth and depth of total tested constructed features. $tolerance$ is the control variable that limits the depth for repetitive unsuccessful constructed features. The loop attempts to evolve the existing $parent.CF$ to a new CF based on the *Evolve* function described in section III-A. After a new CF is generated it is calculated based on the source dataset (not shown in the algorithm to reduce clutter) and is further sanitized (function *CalcSan*). By sanitation we mean ensuring that the results contained in $dataCF$ can be used in further calculations. For our implementation we decided to use the following: *Not a Number* (NaN) = 0, *Not Available* (NA) = $\overline{dataCF}$, $-\infty = min(dataCF)$, and, $+\infty = max(dataCF)$. Such abnormalities may occur due to problematic features (division by 0) or features that lead

Algorithm 1 SEC's Core Evolutionary Function

1: **procedure** CEF(id, $strg\_params$, $generations$, $tolerance$, $kill\_limit$, $modfun$, $bthres$, $dthres$, $cand\_limit$)
2: $seed \leftarrow id$
3: $tolerance\_count \leftarrow 0$
4: Connect to db using $strg\_params$
5: $parent \leftarrow$LockPickRelease()
6: $oldresults \leftarrow parent.results$
7: $weights \leftarrow$ uniform distribution
8: **while** len($db.all$) $\leq generations$ **do**
9: **if** $tolerance\_count \neq tolerance$ **then**
10: $CF \leftarrow$ Evolve($parent.CF$, $weights$)
11: $dataCF \leftarrow$ CalcSan(new_CF)
12: $attempts \leftarrow 0$
13: $kill\_counter = 0$
14: **while** $CF \notin db$ **or** $SD(dataCF) = 0$ **or** $SD(dataCF) = parent.SD(dataCF)$ **do**
15: $CF \leftarrow$ Evolve($parent$ CF)
16: $dataCF \leftarrow$ CalcSan(new_CF, $weights$)
17: $attempts \leftarrow attempts + 1$
18: **if** $attempts = 10$ **then**
19: $parent \leftarrow$LockPickRelease($parent$)
20: **end if**
21: **if** $kill\_counter = kill\_limit$ **then**
22: Release($parent$)
23: Exit
24: **end if**
25: **end while**
26: $results \leftarrow$ Crossvalidate($modfun, dataCF$)
27: $bf \leftarrow$ CalculateBF($results, oldresults$)
28: $diff \leftarrow \overline{results} - \overline{oldresults}$
29: Lock db
30: Store $CF, results, bf, diff, \overline{results}$ in $db.all$
31: Release($parent$)
32: **if** $diff \geq dthres$ **and** $bf \geq bthres$ **then**
33: Store $CF, results, bf, diff, \overline{results}$ in $db.candidates$
34: Reserve CF node in $db.candidates$
35: Retain highest $cand\_limit$ in $db.candidates$
36: $tolerance\_count = 0$
37: $weights \leftarrow$ update using CF and store/update $db.weights$
38: $oldresults \leftarrow results$
39: **else**
40: $tolerance\_count \leftarrow tolerance\_count + 1$
41: **end if**
42: $parent \leftarrow$ latest tuple stored in $db.all$
43: Unlock db
44: **else**
45: $tolerance\_count = 0$
46: $parent \leftarrow$LockPickRelease($parent$)
47: $oldresults \leftarrow parent.results$
48: **end if**
49: **end while**
50: **end procedure**

to extremely large numbers to be held in memory (leading to infinity in some programming languages).

After sanitation is applied, the new CF is checked for "meaningfulness". In essence the feature needs to be calculable as well as have a standard deviation that is both non-zero and different from the parent's standard deviation. If any of these fail, CF is discarded and a new CF is generated from the $parent.CF$. If several attempts (10 in our example) fail to yield a "meaningful" result, a new $parent$ is selected (line 19) and the process repeats. $kill\_limit$ serves as a failsafe parameter that will ensure that CEF will terminate if it is unable to generate a unique "meaningful" constructed feature that does not exist in the storage after several attempts (lines 14 to 25). This often occurs when a small number of attributes exist in a dataset.

The testing sequence (lines 26 to 31) applies an x-times n-fold cross-validation on the dataset using the model function $modfun$ and $dataCF$ (derived from CF) as the sole predictor feature. $modfun$ is provided by the user in order to ensure that domain knowledge is incorporated in the process. For example, for low-dimensional noisy datasets one may opt to use a random forest algorithm whereas for high-dimensional datasets one may utilize a neural network. The output of $modfun$ is a performance metric (e.g., classification error for multivariate problems or f-measure for continuous predicted variables). $results$ is an array that contains $x * n$ results as they are derived from $modfun$.

The newly tested CF along with all results are then stored in $db.all$ and the current $parent$ is "released" so that other processes may select it. To check whether the new CF is a substantial improvement over the previous one, we utilize two metrics: Bayes Factor (BF) and mean difference between current $results$ and $oldresults$. The mean difference demonstrates the increase (if any) in predictive accuracy, while BF shows how significant the result is relative to our $x * n$ sample size of results. The main difference between the typical statistical significance value (e.g., p in frequentist t.test) and BF, is that BF represents the probability odds of H_1 against H_0, with H_1 being the hypothesis supporting that the difference is significant between two samples and H_0 supporting the opposite [19]. This is often formally described as $BF_{10} = \frac{P(H_1)}{P(H_0)}$. BF has been shown to be a superior function for determining probability odds [20], [21]. The two parameters that set the thresholds of what is deemed as an acceptable result are $bthres$ and $dthres$.

If CF passes these fitness checks (line 32), then it is stored in the $db.candidates$ array of tuples and it is marked as "in use" so that other processes will not select it. Further, the $tolerance\_count$ is set to 0 since a "good" CF has been found, and, $db.candidates$ is adjusted to $cand\_limit$ discarding any older CF with lower performance. Finally, $db.weights$ for symbols are updated based on the symbols that are found in the successful CF. $oldresults$ is updated based on the result array of the newly successful CF. On the other hand, if fitness checks do not succeed (line 39) then the $tolerance\_count$ increments. Finally, the new CF record in $db.all$ is set as the $parent$ and the db is unlocked so that other processes can use it. Notice that while a new CF always evolves from its immediate parent, fitness comparisons are made with the previous successful ancestor's $oldresults$, which is not necessarily the parent (observe lines 6, 27, 28, 38, 47). This approach helps ensure that rudimentary small improvements between consecutive "failed" CFs are not accounted as successes.

Some further parameterization has been removed from the algorithm presented in order to ensure clarity. For example, if line 37 is removed, then the symbol selection used in the *Evolve* function is applied uniformly. In the presented form, $weights$ serves as a domain information factor that is incorporated in the genetic algorithm and aims to improve how constructed features evolve. This is also highlighted as an important problem that needs to be addressed by genetic algorithms [13] and hence why we have provided that option.

The results produced by one or more CEF processes as stored in $db.all$ and $db.candidates$ can be visualized as a tree. Figure 2 shows two fictitious evolutionary trees with tolerance set to 1 and 3 respectively. When tolerance is set to 1, CEF expects that every single step of evolution for CF will yield a statistically significant and positive result passing the necessary thresholds. If a step fails, then the algorithm will revisit a previous successful CF from $db.candidates$ and attempt to evolve a unique constructed feature once again. On the other hand, when tolerance is set to 3, repeated constructed features that failed to pass the test are "tolerated" as long as the path does not exceed 3 failed attempts. In the event that a successful CF is produced before then, then the $tolerance\_count$ resets and the algorithm can produce another 3 failed attempts until it needs to pick another parent again. While the process ensures that the same constructed feature will never have to be tested more than once, the algorithm is not considered exhaustive since it cannot start or expand an evolving step from any of the failed CF nodes. The process of jumping between different parents every y failed attempts helps the algorithm escape any local minima that may exist in relation to maximizing accuracy.

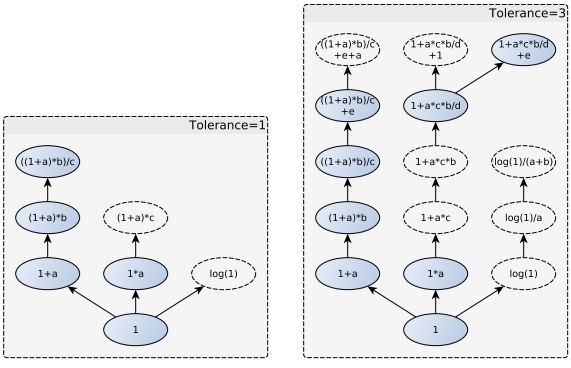

Fig. 2. Example evolutionary trees generated by SEC based on tolerance 1 and 3 respectively. Successful nodes that become candidates for future evolutions are depicted by solid circles whereas nodes that fail to pass fitness tests are shown as dashed line circles.

IV. Results

A. Comparison With Other Genetic Algorithms

We evaluated our genetic algorithm based on several datasets that have already been tested by other related works that we covered in section II. The datasets that we tested were the following: Wine[2], Breast Cancer Wisconsin (Diagnostic) Data (WDBC)[3], Prostate Tumor[4], Pima Indians Diabetes dataset[5], Liver Disorders dataset[6], Diffuse large b-cell lymphomas (DLBCL) and follicular lymphomas[7], Ionosphere dataset[8], and Glass Identification dataset[9].

The dataset selection was made so that we can test the algorithm's performance in a diverse set of classification problems. These problems vary on the levels of classification (binary or multivariate), volume of attributes (also often described as features) and amount of instances (number of rows). Details about the datasets are shown on Table I. DLBCL and Prostate are considered high dimensional classification problems with few observations whereas the rest of the datasets are binary and multivariate classification datasets containing a more reasonable amount of observations. These datasets have also been used in past genetic algorithm studies and therefore serve as good evaluation data points for the performance of our genetic algorithm.

TABLE I
Dataset Characteristics

Dataset	Attributes	Instances	Classes
Wine	13	178	3
DLBCL	5469	77	2
Glass	10	214	6
Ionosphere	35	351	2
Liver	7	345	2
Pima	9	768	2
Prostate	10510	102	2
WDBC	32	569	2

We utilized the following two studies as a comparison for the results of our genetic algorithm: [16] and [13]. Both were described in section II and involve genetic algorithms that are comparable and aim to achieve increased performance and dimensionality reduction by means of feature construction. As such, results that we presented for our algorithm in subsequent tables also contain comparable results from these two studies. For performance comparison, we have restricted our testing settings to the same settings utilized by each of the comparable studies. A summary of parameters is shown on table II. Most genetic algorithms utilize parameters that we have purposeful omitted from our SEC algorithm. For example there is no use of a tournament size, random selection of individuals or crossover operations. Additionally, the mutation operation that is the closest operation from a genetic algorithm that can be found on our algorithm does not utilize a random selection of a feature that is then randomly mutated. Instead, we select based on weighted probability of a successful CF (compared to its last successful ancestor) and then evolve it by means of expansion such that the number of symbols in a parent formula are always less than the symbols found on the new child formula.

These differences between our approach and other genetic algorithms resulted in the need to adjust our parameters so that our results can be realistically comparable with other studies. Henceforth, when we estimated our parameter values we approximated the relative amount of unique formulas that were tested by the other two studies. For example, in [16], a population of 500 that is to be evolved 200 times with a mutation rate of 5% will theoretically result in 5,000 unique formulas. That does not take into account the tournament selection for crossover operations, which would increase the amount of theoretically unique formulas that could be tested. Similarly, in [13], a population of 15,000 that is to be evolved 50 times with a mutation rate of 20% can theoretically result in 150,000 unique formulas. This does not account for crossover operations that may also happen on the population from generation to generation. In order to produce a conservative estimate that is still comparable to these studies, we decided that our algorithm will generate and test a total of 5,000 unique formulas. That became its upper limit ($generations$ parameter).

TABLE II
Genetic algorithm parameters as well as parameters from comparable papers

Parameter	Parameter Value		
	SEC	***GP-Grammar[16]***	***GP-Tree[13]***
Generations	Varied until 5000 mutated formulas	200	50
Population	Init. at 1, max. 1000	500	$features * \beta$
Mutation Rate	Weighted based on performance	0.05	0.2
Selection Rate	N/A	0.05	Weighted based on performance
Tournam. Size	N/A	10	7
Function Set	+,-,*,/,x^y,$\sqrt{}$ log, abs, exp	+,-,*,/,x^y sin, cos, log, exp	+,-,*,% $\sqrt{}$, max, if

$\beta = \{3, 2, or 1\}$ for features 5,000, 5,000-20,000 and larger than 20,000.

We utilized K-Nearest Neighbor (KNN) [22] as our benchmark function for evaluating the predictive potential of each constructed feature. It is a non-parametric method that can be applied for classification as well as regression problems. KNN is lazy learning in the sense that the generalization of the training data is made when the system is queried. For numeric (continuous) predictor variables, KNN utilizes Euclidean distance to identify the K nearest numbers, where

[2]https://archive.ics.uci.edu/ml/datasets/wine

[3]https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

[4]http://www.gems-system.org/

[5]https://archive.ics.uci.edu/ml/datasets/pima+indians+diabetes

[6]https://archive.ics.uci.edu/ml/datasets/liver+disorders

[7]http://www.gems-system.org/

[8]https://archive.ics.uci.edu/ml/datasets/ionosphere

[9]https://archive.ics.uci.edu/ml/datasets/glass+identification

$K \in \mathbb{Z}+$. For classification problems (the datasets that are included in this study), the majority vote of K neighbors to an object is utilized. Both comparative studies have utilized KNN among other algorithms and KNN yielded the highest performance accuracy. The K selection was adjusted to reflect the selection made by each study for a corresponding dataset. In addition, the algorithm is computationally cheaper than other comparable algorithms (e.g., C4.5 [23]). KNN is also a problematic algorithm to use for high dimensional problems [24], which in turn can better demonstrate the benefits of dimensionality reduction that is the result of feature construction by SEC.

The fitness function that we utilized to evaluate performance was classification error, $E = \frac{f}{N}$, where f is the number of falsely classified cases (false negatives) and N is the total sample size. In practice, since $modfun$ is passed through an n-fold cross-validation via *CEF*, the classification error internally becomes $E = \frac{f}{Nn}$. In other words, the classification error of just one fold, which is our testing fold. Since, higher results for performance metric are considered positive (based on how *CEF* is constructed), we utilized the classification accuracy to measure the success of each formula, $E' = 1 - E$. We utilized 3-time 10-fold cross-validation procedure for all tests in order to avoid overfitting and to better estimate the accuracy of predictions. The procedure is built into SEC and other genetic algorithm studies have widely implemented an n-fold cross-validation procedure as well.

Results for all datasets that were utilized for this study along with the respective parameters used for our algorithm are displayed on table III. We experimented with several tolerance values and found that the range 2 and 3 performed well for most datasets (although longer runs with higher tolerance values yielded higher accuracy). Weighted symbol evolution had a varied effect on accuracy. This is likely due to some of the structures in the datasets "favoring" some mathematical functions than others. In other words, while the algorithm has no prior domain knowledge for a dataset, repeated successes and failures can inform some of the evolution through the weighted symbol selection in the *Evolve* function. Further parameters that are not displayed in the table were the Bayes Factor threshold ($bthres$) that was set to 10 as well as the necessary difference for an evolution to be considered substantial to be anything larger than 0 ($dthres$). Bayes factors above 10 are generally considered to be strong evidence in support that a difference between two distributions is significant [25].

Results were measured against the aforementioned comparable studies and are shown on tables IV and V. We report for SEC the mean classification accuracy from the 3-time 10-fold cross-validation. Overall, our algorithm achieved higher performance when testing an equal or less amount of unique formulas than the other two approaches. In other words, it yielded better results faster.

B. SEC Parameters and Long Term Evolution

There are several parameters for the SEC algorithm that can influence outcomes. Since the algorithm is not exhaustive in its current implementation, identifying the ideal parameters can help discover the best possible constructed feature in the shortest amount of time. The parameters that hold the most weight in SEC's evolutionary process are: tolerance, weighted symbol evolution and generations.

Tolerance can allow for larger more complex constructed features that may however not yield positive outcomes. Therefore, the higher the tolerance, the more time the algorithm has to explore paths and as such other paths may be deprived of that opportunity. In general, a high tolerance needs to be accommodated by a high number of generations (the total amount of unique formulas that the algorithm can evaluate before it terminates). For demonstration purposes, we selected the WDBC dataset that sits in the middle range between number of attributes and instances from our tested datasets. We tested a range of different tolerance levels between 1 to 5.

In addition to the tolerance range that we tested, we also applied these settings with weighted symbol evolution active as well as inactive. Although, weight symbol evolution can be domain specific and not every dataset may positively respond to such a parameter, we believe there is merit in indicating these results as it is part of future exploratory work for our

TABLE III
KNN RESULTS FOR DATASETS SEC WAS APPLIED ON

Dataset	**SEC-KNN Parameters**	**Best CF Mean Class. Accuracy**	**All Dataset Features Mean Class. Accuracy**
DLBCL	Tol:2, WSE:T, K:1	94.3%	87.1%
Prostate	Tol:2, WSE:T, K:1	92.3%	79.8%
Wine	Tol.:2, WSE:F, K:10	96.8%	72.1%
Glass	Tol:3, WSE:F, K:10	72.2%	66.2%
Ionosphere	Tol:3, WSE:T, K:10	91.9%	83%
Liver	Tol:3, WSE:T, K:10	73.2%	68.7%
Pima	Tol:3, WSE:F, K:10	76.8%	73.5%
WDBC	Tol:3, WSE:T, K:10	97.5%	93.1%

Note: WSE: Weighted Symbol Evolution, Tol.: Tolerance, K: KNN k value, CF: Constructed Feature.

TABLE IV
RESULTS FOR HIGH DIMENSIONAL DATASETS

Dataset	**Classification Error**	
	SEC	***GP-Tree[13]***
DLBCL	94.3%	86.65%
Prostate	92.3%	83.72%

TABLE V
RESULTS FOR LOW DIMENSIONAL DATASETS.

Dataset	**Classification Error**	
	SEC	***GP-Grammar[16]***
Wine	96.8%	94.44%
Glass	72.8%	71.03%
Liver	73.2%	69.94%
Ionosphere	91.9%	90.34%
Pima	77.4%	76.82%
WDBC	97.5%	96.14%

algorithm. Results of the best calculated constructed feature for each experimental set of parameters are shown on table VI.

TABLE VI
RESULTS ON EXPERIMENTS FOR WDBC DATASET WITH VARIOUS PARAMETERS FOR 5,000 GENERATIONS, KNN=10 AND 3-TIME 10-FOLD CROSS-VALIDATION.

Tolerance	Weighted Symbol Evolution	
	Active	***Inactive***
1	97.2%	97.1%
2	96.1%	96.1%
3	96.4%	95.8%
4	96.1%	97.1%
5	96.8%	97.4%

Overall, most of these parameters have a minor influence on the output. Noticeably, even when tolerance is set to 5, results are still positive even though more failed constructed features are bound to exist. Fig. 3 shows the constructed feature evolutionary tree. Noticeably, good constructed features (large circles) can be found even after several failed constructed features.

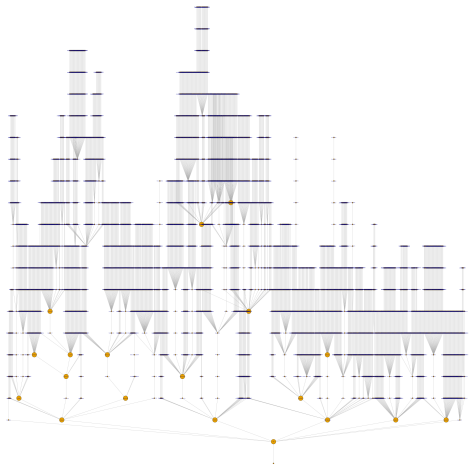

Fig. 3. Constructed feature evolutionary tree for WDBC dataset using 5,000 generations, KNN=10, 3-time 10-fold cross-validation and tolerance=5. Each node (circle) represents a feature. Larger nodes represent successful constructed features that are a statistical improvement over their previous successful predecessor in their tree branch. The longer the graph distance from the root node (found at the bottom of the graph), the larger the features are.

The overall evolutionary trend for long runs will initially resolve in a positive performance trend that is statistically significant (e.g., using linear regression) but will eventually plateau. Fig. 4 shows the overall trend for 100,000 generation execution for the WDBC dataset. Users that wish to obtain the peak accuracy through a genetic algorithm's execution can utilize a metric that identifies the plateau (e.g., the point where the coefficient in a linear regression will become 0 or the point where the curve flattens). The best constructed featured occurred at the 92,999th attempt and had a mean accuracy of 98.37% with a standard deviation of 0.012 and its structure is:

$$\frac{(1 * c_1 - log(t_1) - a_3) * s_3 * \sqrt{t_2 - C_1 + t_1} + \frac{s_3}{C_2}}{t_2} + {C_2}^3 - a_2 \quad (1)$$

where c is concavity, t is texture, a is area, s is smoothness, C is compactness, and indexes correspond to measurements as they are derived from three different cell nuclei that were sampled in the WDBC dataset. The dataset contains a total of ten different measurements for each of the three nuclei. One can observe that the most successful dimensionality reduction via a genetic algorithm utilized only five. While many of the constructed features produced by genetic algorithms may not be theoretically informative, their visualization may inform a researcher to establish a new research direction and as such there is merit in observing the complexity of such constructed features.

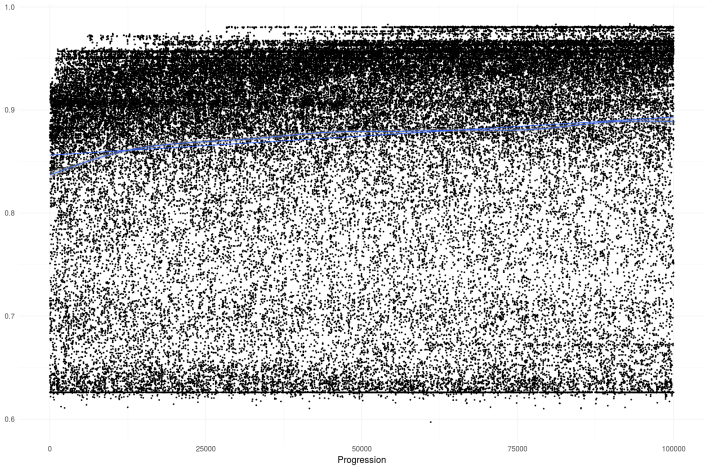

Fig. 4. Constructed feature evolutionary progression for WDBC dataset using 100,000 generations, KNN=10, 3-time 10-fold cross-validation and tolerance=5. The Y axis represents feature accuracy performance while the X axis represents the auto-increment numeric identifier for each feature. Features with higher numeric identifiers are not necessarily in deeper branches of the tree. Linear and locally weighted scatterplot smoothing (LOESS) lines are plotted on the graph. Noticeable increase in performance over time as well as a gradual plateau in higher x-values (generations)

C. Performance

The algorithm is constructed in such a way that it can run in parallel which leverages multiple cores yielding faster results at the expense of the process becoming non-deterministic. Compared to other genetic algorithms, SEC will not test constructed features that are not "meaningful" and as such it utilized CPU time more efficiently. The most expensive aspect of the genetic algorithm is the cross-validation procedure as well as the machine learning algorithm that is utilized by the user provided model function ($modfun$). As such, it is advisable to benchmark the model function before passing it to SEC.

Our 3-time 10 fold cross-validation tests for KNN=10, 5,000 generations and 16 cores for the WDBC dataset (table VI)

resulted in a CPU time with a mean of 1,575 and a standard deviation of 335. Notably, lower tolerance values resulted in higher CPU time since re-selection of a new parent node to be evolved happened more often. The actual mean time was 6.98 minutes with a standard deviation of 1.98. Based on related literature the results are comparable with other genetic algorithms except those that do not utilize cross-validation as the main evaluation procedure (e.g., information gain ratio [26]).

V. Future Work

The major bottleneck can be found in the ACID compliant data storage that is used for information sharing for constructed features between different CEF (algorithm 1) processes that run in parallel. In our experiments, we utilized a SQLite database, which resulted in an upper limit in the number of processes that could connect and write to it. Furthermore, writing to disk is a more expensive operation. The effect is not so apparent for model functions that have slower machine learning algorithms (e.g., random forests) but given the lazy learning nature of KNN, even a 10-time 10-fold cross-validation was executed fast enough to create queues between the different CEF processes attempting to write to SQLite. Experimenting with different shared storages is bound to yield faster results. Further, larger datasets need to be utilized in order to evaluate the how the algorithm scales along with subsequently different machine learning algorithms.

Our algorithm was tested only for classification problems, however, as presented, the algorithm can be directly applied for continuous variable prediction problems (e.g., temperature prediction for climatic models). Future work, needs to evaluate the performance of this method over such prediction problems.

There is also a need to evaluate the effect of alternative evolve functions for feature construction. While for our purposes, an open-ended length for our features allowed the algorithm to keep developing more complex structures, alternative evolve functions such as grammar-based or tree-based may also result in positive results. Continued exploration should also address more complex mathematical symbols as well as how they can be efficiently solved (e.g. integrals).

Another aspect that needs to be evaluated is whether failed branches should also be given the opportunity to be re-explored by generating secondary branches in the tree of constructed features. The avoidance of exploring such branches could generate non-optimal solutions and trap the process in local minima. Additionally, weighted symbol evolution also forces the evolutionary process to operate on non-random operators, which may also lead to local minima. Finally, weighted symbol evolution as a weighted sampling function does not leverage the potential of machine learning algorithms that can be applied to further optimize the problem. As such, future work should evaluate symbol selection using more complex supervised learning algorithms such as decision trees or even neural networks.

VI. Conclusion

The SEC algorithm improves evolutionary algorithm efforts by integrating opportunity for performance enhancement. An analysis of similar methods shows how the algorithm optimizes its resources by avoiding weaknesses in potential constructed features. The selective evolution approach ensures that results which lack quality are disregarded, pushing the outcome towards results that yield higher classification accuracy while doing so at a faster rate than comparable studies. As such, SEC offers an alternative genetic algorithm that aims to resolve some of the issues presented in other genetic algorithms. We hold that as problems in every domain become more high dimensional, the need for advancing genetic algorithms becomes even more important and this study aims to produce a step forward in this direction.

References

[1] K. Neshatian, M. Zhang, and P. Andreae, "A Filter Approach to Multiple Feature Construction for Symbolic Learning Classifiers Using Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 16, no. 5, pp. 645–661, 2012.

[2] R. Alfred, "Feature Transformation: A Genetic-Based Feature Construction Method for Data Summarization," *Computational Intelligence*, vol. 26, no. 3, pp. 337–357, jul 2010. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-8640.2010.00362.x

[3] K. Lillywhite, D.-J. Lee, B. Tippetts, and J. Archibald, "A feature construction method for general object recognition," *Pattern Recognition*, vol. 46, no. 12, pp. 3300–3314, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320313002549

[4] S. Ahmed, M. Zhang, L. Peng, and B. Xue, "Multiple Feature Construction for Effective Biomarker Identification and Classification Using Genetic Programming," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14. New York, NY, USA: ACM, 2014, pp. 249–256. [Online]. Available: http://doi.acm.org/10.1145/2576768.2598292

[5] D. García, A. González, and R. Pérez, "A two-step approach of feature construction for a genetic learning algorithm," in *2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011)*, 2011, pp. 1255–1262.

[6] A. Purohit, N. S. Chaudhari, and A. Tiwari, "Construction of classifier with feature selection based on genetic programming," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–5.

[7] P. G. Espejo, S. Ventura, and F. Herrera, "A Survey on the Application of Genetic Programming to Classification," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 40, no. 2, pp. 121–144, 2010.

[8] C. T. Tran, P. Andreae, and M. Zhang, "Impact of imputation of missing values on genetic programming based multiple feature construction for classification," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, 2015, pp. 2398–2405.

[9] M. Muharram and G. D. Smith, "Evolutionary constructive induction," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, pp. 1518–1528, 2005.

[10] Y. Liu, F. Tang, and Z. Zeng, "Feature Selection Based on Dependency Margin," *IEEE Transactions on Cybernetics*, vol. 45, no. 6, pp. 1209–1221, 2015.

[11] U. Kamath, K. De Jong, and A. Shehu, "Effective Automated Feature Construction and Selection for Classification of Biological Sequences," *PLoS ONE*, vol. 9, no. 7, p. e99982, jul 2014. [Online]. Available: http://dx.plos.org/10.1371/journal.pone.0099982

[12] M. L. Raymer, W. F. Punch, E. D. Goodman, L. A. Kuhn, and A. K. Jain, "Dimensionality reduction using genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 2, pp. 164–171, 2000.

[13] B. Tran, B. Xue, and M. Zhang, "Genetic programming for feature construction and selection in classification on high-dimensional data," *Memetic Computing*, vol. 8, no. 1, pp. 3–15, 2016. [Online]. Available: https://doi.org/10.1007/s12293-015-0173-y

[14] G. Pagallo, "Learning DNF by Decision Trees," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 639–644. [Online]. Available: http://dl.acm.org/citation.cfm?id=1623755.1623856

[15] S. Markovitch and D. Rosenstein, "Feature Generation Using General Constructor Functions," *Machine Learning*, vol. 49, no. 1, pp. 59–98, 2002. [Online]. Available: https://doi.org/10.1023/A:1014046307775

[16] D. Gavrilis, I. G. Tsoulos, and E. Dermatas, "Selecting and Constructing Features Using Grammatical Evolution," *Pattern Recogn. Lett.*, vol. 29, no. 9, pp. 1358–1365, jul 2008. [Online]. Available: http://dx.doi.org/10.1016/j.patrec.2008.02.007

[17] K. Krawiec, "Genetic Programming-based Construction of Features for Machine Learning and Knowledge Discovery Tasks," *Genetic Programming and Evolvable Machines*, vol. 3, no. 4, pp. 329–343, 2002. [Online]. Available: https://doi.org/10.1023/A:1020984725014

[18] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, 2001.

[19] M. Tsikerdekis, "Bayesian Inference," in *Modern Statistical Methods for HCI*, J. Robertson and M. C. Kaptein, Eds. London, U.K.: Springer, 2016, ch. 7, pp. 173–197.

[20] J. K. Kruschke, "Bayesian estimation supersedes the t test." *Journal of Experimental Psychology: General*, vol. 142, no. 2, pp. 573–603, 2013.

[21] E.-J. Wagenmakers, M. D. Lee, T. Lodewyckx, and G. Iverson, "Bayesian versus frequentist inference," in *Bayesian evaluation of informative hypotheses in psychology*, H. Hoijtink, I. Klugkist, and P. A. Boelen, Eds. New York: Springer, 2008, pp. 181–207.

[22] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, aug 1992. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879

[23] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[24] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When Is "Nearest Neighbor" Meaningful?" in *Proceedings of the 7th International Conference on Database Theory*, ser. ICDT '99. London, UK, UK: Springer-Verlag, 1999, pp. 217–235. [Online]. Available: http://dl.acm.org/citation.cfm?id=645503.656271

[25] R. E. Kass and A. E. Raftery, "Bayes Factors," *Journal of the American Statistical Association*, vol. 90, no. 430, pp. 773–795, 1995. [Online]. Available: http://www.jstor.org/stable/2291091

[26] F. E. B. Otero, M. M. S. Silva, A. A. Freitas, and J. C. Nievola, "Genetic Programming for Attribute Construction in Data Mining BT - Genetic Programming," C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 384–393.