Evolutionary Functions: A Genetic Approach for Constructing Predictors

Xuanting Chen[1]

[1] *Duke University, Durham, NC 27708, USA*

We demonstrate an algorithm for building predictive models using an evolutionary strategy combined with functional programming. We explain in detail the rationale of the algorithm and how a rudimentary implementation was realized. Different from the prevailing loss function - gradient descent approach, our method directly reinforces models and does not require an objective function to be optimized. This characteristic could further enable solutions to a wider spectrum of problems. Applying the method to classic classification datasets, we obtain decent performance that is on par with some widely used algorithms.

I. INTRODUCTION

From simple linear models to neural networks, the essence of predictors remains the same: functions that map inputs to outputs. A fully connected neural network with two input attributes, one hidden layer that has two neurons, a single output node could be described with the following equation:

$$\phi(w_5\phi(w_1x_1 + w_2x_2 + b_1) + w_6\phi(w_3x_1 + w_4x_2 + b_2)) + b_3$$

, where x_1, x_2 are inputs, $w_1 \sim w_6$ are weights, $b_1 \sim b_3$ are biases, ϕ is the activation function. In this regard, we can view these models as combinations of signals and operators.

Instead of manually establishing a rigid framework such as neural networks, our algorithm allows every participant, including parameters, variables, operators, and activation functions if necessary, to randomly combine with each other. In other words, the algorithm yields random mathematical formulas with the given material and selects them using a specified criteria.

A key mechanism we adopt here is to convert instances between strings and lambda expressions. When encoded as strings, the functions obtain the ability to mutate and when compiled to lambda expressions, the functions take inputs and make predictions. Fig. 1 is a rough visualization of the whole concept. Within each iteration, string representations are first compiled to lambda expressions, after which they are evaluated and selected according to their performances. Following up is the mutation phase during which new instances are reproduced, constituting the next generation.

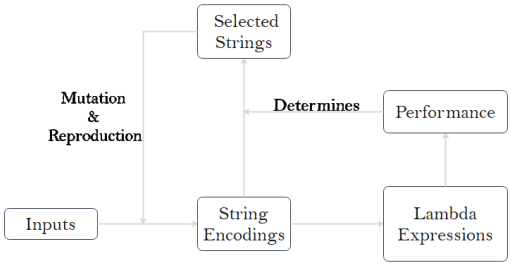

FIG. 1. A visualization of the method.

II. OBJECTS

A. String representation

The string encoding method we use for storing function information serves an important role for the whole process. A simple example can explain this process:

$$f = ReLU(x^2 + \frac{1}{y} + 0.5), \tag{1}$$

$$x(*x)(+1.0(/y))(+0.5)(R). \tag{2}$$

In this example, line (2) is a string representation of equation (1). Decomposing the string gives five shorter expressions: x,$(*x)$,$(+1.0(/y))$,$(+0.5)$,(R). Each pair of parentheses wraps an instruction that is applied to the function being formed. Tab. 1 describes the process in detail: the first column is the formula we hold, and the second column is the instruction the compiler reads. The first x without parentheses is named *kernel* as it is the starting point of the function. When the compiler holds x in its memory and reads $*x$, it will generate $x * x = x^2$ as a result. Then, in the next step, x^2 becomes what the compiler holds. The same procedure continues until all instructions are applied. Notice when $(+1(/y))$ is read, it will be recursively compiled to a lambda expression with 1 being its *kernel*.

The following are some reasons to adopt this encoding style:

- The decoding process is lossless
- Inputs to the compiler are serial
- Mutations could be controlled

Current	Read	Result
$x(kernel)$	*x	x^2
x^2	$+1.0(/y)$	$x^2 + \frac{1}{y}$
$x^2 + \frac{1}{y}$	$+0.5$	$x^2 + \frac{1}{y} + 0.5$
$x^2 + \frac{1}{y} + 0.5$	R	$ReLU(x^2 + \frac{1}{y} + 0.5)$

Current	Read	Result
$1(kernel)$	/y	$\frac{1}{y}$

TABLE I. Step by step demonstration of the decoding process.

With serialization, computers can interpret the string and construct the target function. With instructions wrapped as units, mutations occur without causing compiling failures.

B. Lambda expression

The way the system compiles string representations can be aptly described as "piling up" lambda expressions. Upon initialization, the program will create fundamental functions that will be used throughout the whole simulation:

1. For each input feature, define a lambda expression that only returns its value. For example (two input features, python):

```
inputList = {
    'a': lambda x1,x2: x1,
    'b': lambda x1,x2: x2
}
```

   Here 'a' and 'b' encode the first and the second input attribute respectively.

2. Define a function that executes lambda-expression-wise operation. For example (two input features, python):

```
def compose(f,g,operator):
    if operator == '*':
        return lambda x1,x2:f(x1,x2)*
        g(x1,x2)
    elif operator == '+':
        return lambda x1,x2:f(x1,x2)+
        g(x1,x2)
```

   This function takes in two functions and outputs one concatenated function. We can add more operators as long as it is well defined.

3. For each transformation, define a corresponding method. For example (ReLU, python):

```
def ReLU(f):
    return lambda x1,x2: 0 if f(x1,x2)
    < 0 else f(x1,x2)
```

   Transformations include but are not limited to activation functions. They can be any structures that manifest mathematical significance.

During compilation, whenever a variable/constant is read, the compiler picks the corresponding lambda function; whenever an operator/transformation is read, the compiler applies it to the current lambda function pile.

III. ALGORITHM

This part will deliver a brief demonstration of mutation and selection algorithms. However, our implementation does not stand for a norm, nor is it an optimal practice.

A. Mutation

To fully explore the model space, we introduce extension, truncation, and element alternation.

1. Extension

Given a string representation, for each space between two wrapped operations, there are chances a new operation will be inserted. We use ␣ to mark spaces in the example below:

$$x␣(*x)␣(+1.0(/y))␣(+0.5)␣(R)␣,$$

and for the recursive part:

$$1.0␣(/y)␣.$$

The main idea is to insert a random cell in each space with a certain probability. We generate a random cell for insertion by choosing a random operator with a random variable/constant or by choosing a random transformation.

2. Truncation

Given a string representation, for each wrapped operation, there are chances that it will be dropped.

3. Element alternation

Given a string representation, there are chances that one or several operators will be replaced by other operators. The same rule also applies to each variable, constant, and transformation. It is also likely that the *kernel* will change.

B. Selection

In nature, an individual's ability to reproduce is likely proportional to its performance in a particular environment. However, implementing this philosophy requires building an interactive environment, which is something of the next stage. Meanwhile, the datasets we use here are designed for simple tasks that can be easily evaluated using statistical metrics. Hence, we use accuracy as the criteria to select survivors so that we demonstrate feasibility to a certain extent.

Flow:

1. Compile all strings to lambda functions, evaluate and record their accuracies
2. Obtain population median
3. Add individuals above the median to the next generation
4. For all out-performers, randomly select half and add their mutated copy to the next generation
5. For all under-performers, randomly select half and add their mutated copy to the next generation

Notice that the size of the population sometimes fluctuates. Hence, it is necessary to regulate it.

IV. EXPERIMENTS

We run the algorithm on two classic data sets: the Wisconsin Breast Cancer dataset [1] and the White Wine Quality dataset [2]. Our goal is to showcase the feasibility of the approach to a certain extent. Since our rudimentary implementation does not include many operators and transformations that could potentially strengthen the models, nor does it run a great number of generations, say 100, to fully explore the population behavior, the test results do not completely stand for the power of this method.

A. Cancer diagnosis

Data source: Breast Cancer Wisconsin Data Set
Normalization: min-max
Training set size: 483
Test set size: 86
Candidate operators: +, *, /
Candidate parameters: $\mathcal{G}(0, 1.0)$
Candidate transformations: ReLU, Sigmoid
Initial population: 10,000

Three trials are conducted, and all simulations terminate at the 25th generation. Fig.2 shows how training accuracy changes during these trials. Within the first five generations, all best-performers of the three trials are able to obtain an accuracy higher than 96%. This indicates that the algorithm can yield good models in a short period on this dataset. It is worth noting that the converging time and the final results vary in the three trials. This is because the method is essentially a random process. However, we can observe an increasing trend in these trials. This is because every generation is theoretically at least as good as its previous generation due to the nature of our algorithm. Meanwhile, although trial 1 seems to be struggling during generations 5 to 20, it eventually obtains a decent level of accuracy. In my opinion, the performance can continue to increase if more generations are simulated. The following sections demonstrate details of each trial. There is an interesting fact that all best-performers use the sigmoid function, which is completely a result of the selection process.

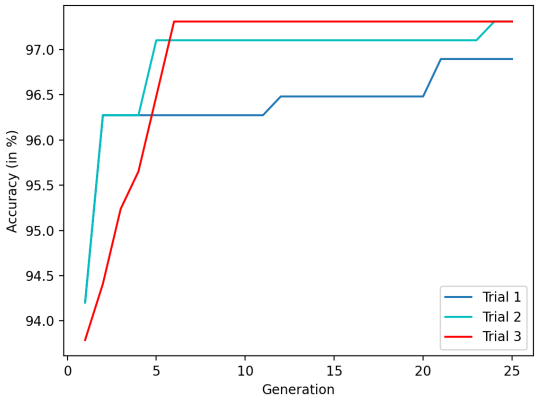

FIG. 2. Training accuracy as a function of generations.

1. *Trial 1*

Best performer:

d(+-0.51912194)(/b(*v)(*0.68898687)
(+0.27690040))(/1.17815124)(+γ (+y))(S)

Training accuracy: 96.9% Test accuracy: 96.5%

Rearranged as:

$$f = Sigmoid(\frac{d - 0.52}{0.81 * bv + 0.33} + \gamma + y)$$

Meanings of the attributes:

$d \Rightarrow$ **mean area for nucleus**
$b \Rightarrow$ **texture (std of gray-scale values)**

$v \Rightarrow$ **largest texture value**

$\gamma \Rightarrow$ **largest symmetry value**

$y \Rightarrow$ **largest smoothness (local variation in radius)**

Insights:

The result suggests the possibility of being a malignant case is positively related to the mean nucleus area, the largest symmetry value, and the largest smoothness, but inversely proportional to textures.

2. Trial 2

Best performer:

w(*y(*w))(/c)(*u)(+-0.03231381)(S)

Training accuracy: 97.3% Test accuracy: 90.7%

Insights:

This best-performer is an over-fitted model as the test accuracy is considerably low. It could be resulted from an imbalanced train-test split.

3. Trial 3

Best performer:

0.85585965(*-1.95243541)(/0.75483675)
(*0.54581506)(+o)(/x)(+1.32684316
(+-1.61840215(/w))(*β)(*-1.77725415))
(+2.30081003)(*c)(+v)(S)

Training accuracy: 97.3% Test accuracy: 96.5%

Rearranged as:

$$f = Sigmoid(c*(\frac{o-1.21}{x}+\beta*(\frac{2.88}{w}-2.36)+2.30)+v)$$

Meanings of the attributes:

$o \Rightarrow$ **standard error for smoothness**

$x \Rightarrow$ **largest area**

$w \Rightarrow$ **largest perimeter**

$\beta \Rightarrow$ **largest # of concave portions of the contour**

$c \Rightarrow$ **mean size of the core tumor**

$v \Rightarrow$ **largest texture value**

Insights:

The model suggests the possibility of being a malignant case is positively related to the standard error for smoothness, the largest number of concave portions of the contour among all cells, the largest texture value, and the mean size of the core tumor, but inversely related to the largest nuclei area and the largest perimeter. The model uses a different set of parameters compared to the model in trial 1, which indicates the underlying principles of this task can be represented from different perspectives.

B. Wine quality prediction

This is a multi-class classification problem. Our initial design is to construct models with 7 (the number of classes) binary classifiers and let good predictors exchange "genes" to accelerate the gathering of advantageous structures. However, the mechanism of "sexual reproduction" is still under investigation. Hence, we adopt a compromised, "asexual" strategy.

The strategy we use is to develop binary classifiers for each class independently and put a softmax layer at the end. One of the results is demonstrated below (population size: 1,0000, trained for 15 generations):

Best performer:

[-2.29044175(/0.03372067(/a)(+a))(S),

f(/-0.07121823)(/b(R)(+g))
(+-1.30766254(+0.23611516))(+c)(S),

1.05588167(*k)(/0.21587891(+b))
(+-0.51694618)(/-0.50769838)(S),

-0.15525971(+b)(+b)(*-0.40510551)
(/g(/b))(/k)(S),

2.46027132(*k)(+e(*-1.71201650
(/-1.26789814)(*2.74142959(+i)
(/-0.08007508(/h))(/-0.57108328
(+-0.04808928))(*2.09814987(*b))
(*0.53491796(+-2.61767458)))))(+i)
(+-1.74431657(+-0.56557012))(S),

k(+f)(+-1.82682471)(/0.26178213)
(+1.43949461)(S),

0.43182767(/k)(/i(*a))(*-1.46162216)(S)]

Accuracy: 53.6%
OvR ROC AUC score (macro): 0.763

Tab 2. lists the accuracies and AUC scores obtained from each algorithm [3]. The ROC AUC score

Algorithm	Accuracy	ROC AUC
Ours	53.6%	0.763
KNN	52.1%	0.791
LR	52.4%	0.754
Decision Tree	58.4%	0.715
Random Forest	65.1%	0.897

TABLE II. Step by step demonstration of the decoding process.

measures the level of the practical value of the predictions, the higher the better. Compared to other algorithms, the performance of our algorithm is slightly better than KNN and LR in terms of accuracy and is better than LR and decision tree in terms of AUC score. It is worth noting that random forest outperforms other algorithms by a wide margin. This could result from the voting mechanism it uses. However, our algorithm could theoretically evolve the voting mechanism independently with enough generations and necessary operators, but we cannot test this hypothesis at this stage due to a lack of computational resources.

V. COMPUTATION TIME

Most of the experiments were done on Google Colab without GPU acceleration. Hence, it usually takes hours to yield a decent result. However, as our algorithm is highly suitable for parallel computing, with GPU turned on, we believe the computing time will be dramatically shortened. Meanwhile, hyperparameters such as mutation rates are still poorly researched. We believe from both the hardware and the algorithmic perspectives, there exists huge space for improvement.

VI. CONCLUSIONS

This paper proposed a genetic approach for constructing predictive models. The approach is based on an encoding method that represents lambda expressions as strings, which makes function-level mutation possible.

We successfully realized a rudimentary implementation and obtained decent performance. The performance of our implementation appears to be close to logistic regression. A possible reason for this could be our element selection: we choose addition, multiplication, and division as our operations, Sigmoid and ReLU as our transformation functions, which highly resembles logistic regression. A wider selection of mathematical components could make a considerable difference.

Our expectation of this approach is high as this approach does not have a theoretical upper bound. At the same time, this approach has the potential to provide solutions to problems that currently cannot be solved. For example, it is hard to train robots to cooperate using massive mathematical models as it is hard to formulate loss functions that fully reflect the level of cooperation. However, our algorithm can optimize models directly using results such as the degree of completion of tasks based on collaborative talent. We are finding other great ideas to better implement this approach and we will keep exploring with future studies.

[1] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science. The TensorNetwork library and TTN code can be downloaded from https://github.com/google/TensorNetwork.

[2] P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis, *Modeling wine preferences by data mining from physicochemical properties.*In Decision Support Systems, Elsevier, 47(4):547-553, (2009).

[3] G. Turgay, *Comparing Classification Models for Wine Quality Prediction*, (2020). Article accessible at: https://towardsdatascience.com/comparing-classification-models-for-wine-quality-prediction-6c5f26669a4f