AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND BIOMEDICAL ENGINEERING

Diploma Project

Selected Tools for Grammatical Evolution. Analysis and Applications.

Wybrane narzędzia do ewolucji gramatycznej. Analiza i zastosowania.

Author: *Jakub Skrzyński*
Degree programme: *Computer Science*
Supervisor: *prof. dr hab. inż. Antoni Ligęza*

Kraków, 2024

SUMMARY

The aim of this thesis was to analyze selected software tools for Grammatical Evolution and present their evaluation and numerical experiments. Thesis can be conceptually divided into 6 parts covering different areas of the topic. First part, being chapters 1 - 3, is devoted for explaining basic concepts of Grammatical Evolution giving the reader broad view of the method and possible challenges that may arise while developing the model. Following part, chapters 4 and 5 contains information on needs of users and discussion on conducting a comparison between tools. Third part, chapters 6 - 9, describes in detail 3 tools that were chosen based on maturity level, continuity of maintenance and quality of documentation and sources describing them. Additionally in this part there are short descriptions of other tools that are worth of attention. Chapters 10 and 11 describe shortly possible areas of applications. Next, chapters 12 - 17 contain experiments showing example usage of tools and discussing their efficiency and accuracy as well as a general level of user experience. Thesis ends with conclusions on the conducted comparison and ideas for further work regarding topic of GE tools, presented in chapter 18.

STRESZCZENIE

Celem pracy była analiza wybranych narzędzi programowych ewolucji gramatycznej, przedstawienie ich oceny, oraz eksperymentów numerycznych. Z pracy można wydzielić 6 części, z których każda omawia poszczególne zagadnienie z tematu. Pierwsza część, złożona z rozdziałów 1 - 3, poświęcona jest wyjaśnieniu podstawowych pojęć dotyczących ewolucji gramatycznej, dając czytelnikowi szeroki pogląd na metodę i możliwe wyzwania, które mogą pojawić się podczas opracowywania modelu z jej użyciem. Kolejna część, składająca się z rozdziałów 4 i 5, zawiera dyskusję na temat metod przeprowadzenia porównania narzędzi, oraz na temat potrzeb użytkowników. Część trzecia, na którą składają się rozdziały 6 - 9, szczegółowo opisuje 3 narzędzia, które zostały wybrane ze względu na poziom rozwoju, ciągłość aktualizacji i wsparcia oraz jakość dokumentacji i źródeł je opisujących. Dodatkowo w tej części znajdują się krótkie opisy innych, wartych uwagi, narzędzi. W dalszej części, w rozdziałach 10 i 11, krótko opisano możliwe obszary zastosowań. Część piąta, składająca się z rozdziałów 12 - 17, zawiera eksperymenty pokazujące przykładowe wykorzystanie narzędzi oraz dane na temat ich efektywności i dokładności, a także ogólny poziom zadowolenia użytkownika. Pracę kończą wnioski z przeprowadzonego porównania oraz potencjalne dalsze kierunki eksploracji tematu narzędzi GE, zawarte w rozdziale 18.

Niniejszą pracę pragnę zadedykować moim wspaniałym Rodzicom Ewie i Andrzejowi Skrzyńskim, którzy wspierali mnie i motywowali w czasie przygotowywania niniejszej pracy oraz w czasie całych studiów.

Serdecznie dziękuję Panu prof. dr hab. inż. Antoniemu Ligęzie za zainspirowanie mnie do podjęcia tematu tej pracy, okazaną pomoc, udzielone cenne wskazówki, dostarczenie niezbędnych materiałów oraz informacji, stałe czuwanie nad właściwym kierunkiem pracy, duże zaangażowanie w odpowiedzi na moje liczne pytania i poświęcony czas.

Dziękuję również wszystkim osobom zaangażowanym w proces mojego kształcenia, które swoim zaangażowaniem i zapałem do przekazywania wiedzy, oraz swoją otwartością, motywowały mnie do dalszej wytężonej pracy.

Specjalne podziękowania kieruję również do mojej narzeczonej Zuzanny i mojej siostry Aleksandry, za wsparcie, częstą pomoc w ocenie jakości i czytelności objaśnień, oraz doborze najodpowiedniejszych słów.

Serdecznie dziękuję!!!

Contents

1. Artificial intelligence

1.1. Definition of artificial intelligence and its goals

To begin with, one may try to define the artificial intelligence by saying that this is a way to express the process of machine taking reasonable decisions. These decisions are based on the data that was provided to the machine from the environment. There had been many discussions how to define what the artificial intelligence is, to bring one definition one may bring article by Dimiter Dobrev [18]. A broad definition is presented in the book "Think AI" [40]. The issue is even more difficult if one tries to split the name and define it by intelligent action taken by a machine — as some books [33] mention definition of intelligence as an abstract term had yet not been fully established thought centuries. Therefore, it is reasonable to assume that there is no single precise and unique definition of AI. Nevertheless, from some definitions, one may extract some features of AI as the aspect mentioned in Encyclopedia Britannica [9] — ability to generalize and reason. This is a crucial feature for today's data processing. The family of artificial intelligence algorithms is very broad and can be used in a wide variety of applications. Literature also points out that due to modern computers' processing power, AI is often capable of detecting unseen patterns and trends that were previously not detectable, due to the complexity of data [35].

1.2. Generalization of knowledge

Generalization is a tool that allows to extract the most meaningful part of data and drop the redundant one without losing the precision. Mathematics is already in possession of powerful tools that allows us to extract the knowledge from given data. One of such methods is linear regression. Having a data set and the knowledge that the relationship should be linear, it is possible to accurately determine coefficients of this relation. An easy to spot drawback is the need of assumption of function form. Having a 4th degree polynomial and trying to use linear regression will result in inadequate result. A simple solution is to use the correct form of polynomial

or use a polynomial of much higher degree than desired, but it still does not allow discovering all functions. An example could be a real function with asymptotes, or an $abs()$ function.

1.3. Modern AI algorithms and their applications

Currently, the branch of AI is one of the most known in a whole field of Computer Science, being also subject of many research projects. Huge development of this branch resulted in lots of available AI algorithms. Rishal Hurbans's book [33] divides concepts of artificial intelligence to some intersecting categories: biology-inspired algorithms, machine learning algorithms, deep learning algorithms, search algorithms. According to this publication these concepts are used in following paradigms and to solve following problem classes: search problems, optimization problems, prediction and classification problems, clustering problems, deterministic models, stochastic/probabilistic models. Such huge variety of concepts makes AI appear in most of today's systems.

Reading through current research, one may come to conclusion that biggest efforts and hopes are utilized by concept of artificial neural networks that allows to easily adapt to many use cases without adjusting the core of algorithm. ANNs require user just to provide correct training data set and create correct architecture that allows the network to learn. Such approach moves efforts of developers from analyzing the data and creating algorithm to preparing dataset of high quality and adjusting training parameters as well as the architecture. Although it is undoubtedly a beneficial feature, it comes with a certain cost — the exact "process of thinking" is not explainable. Knowledge is being encoded into weights of synapses, and analysis of behavior of such a network poses a serious challenge due to number of variables.

1.4. Issues posed by use of artificial neural networks

Not being able to explain the whole process is a huge cost. One of the issues that emerges could be proving that the outcome will be correct in all scenarios. Critical applications require such analysis to ensure safety. The threat is not only imaginary. There were already research [59] on how to manipulate output of image classifier by changing just a single pixel. The author of the classifier is only capable of providing certain statistical measures of successfully classified images, but is not able to provide a proof that the classifier will always work in the same way under certain conditions. The reason is mentioned previously — the lack of explainability.

1.5. Explainable AI, its challenges and its role

A publication entitled “Explainable Artificial Intelligence” [29] considers XAI as the form of artificial intelligence that will be more understandable by human. Current techniques can provide explainability but on cost of precision and vi ca versa precision on cost of explainability. Best precision is provided by neural networks, but they are hardly explainable due to complex structure and amount of calculations [29].

2. Theoretical foundations of GE algorithms

2.1. Genetic Programming

2.1.1. Introduction

Genetic Programming (GP) is another branch of AI that concentrates around using concepts that humans learned from molecular biology in solving search problems. Basically, this method models how biological evolution approaches search for organism that fits best the environment. To tackle this task, programmer have to develop the measure of fitness for the problem, that will be base for deciding which entities should be reproduced in next generations.

The real advantage of the approach is that it involves defining a high level statement about the problem and criteria for the solution and evolving population of computer programs that would be able to solve it [38]. This implies that much of the time-consuming work concentrated around development of an algorithm is done by evolution.

2.1.2. Details

Given the overview of genetic programming as a method, to take full advantage of its benefits one must properly define the following 4 operations that will occur during the process: selection, replication, crossover, and mutation [63].

2.1.2.1. Operations

2.1.2.1.1 Selection Selection is an operation that, given a set of individuals from the current generation, provides a subset of it that contains individuals selected to be parents for the next population. There are currently many known methods of performing this operation, but tournament selection is currently the most common [63].

2.1.2.1.2 Replication Replication is an operation that resembles asexual reproduction in biology — for example, similarly to yeast — where the next generation is an exact copy of the parent [1]. This operation involves copying the individual into the next generation.

2.1.2.1.3 Crossover Crossover is a second type of reproduction mechanism in GP. It involves creating a child based on parents. Child should inherit part of gens from each parent, giving the possibility to further enhance features of parents. This operation has to be correctly defined to have a child that inherits features but also remains correct in terms of problem domain — for example having an algorithm running on trees, no cycle can be produced as a result of crossover.

2.1.2.1.4 Mutation Mutation serves an important role in the whole process, allowing to escape local optima of search space by introducing random modification of genome. It also has its origins in the biology.

2.1.2.2. Evolution

Once the problem had been correctly described, one may start the search process that consists of subsequent phases of selecting the best individuals from the population, reproducing them into the new population and applying mutations. The process is usually limited by the number of generations that should be created, and the best individuals are being chosen according to an adopted selection strategy. The number of generations has significant influence on the outcome of the experiment — the bigger the number, the better the solution produced by the algorithm.

2.2. Formal Languages and grammars

As written in a book by Alan Parkes [49] "a formal language is any (proper or non-proper) subset of the set of all strings which can be formed using zero or more symbols of the alphabet A. ". The same book also explains the exact meaning of the alphabet, as "An alphabet is a finite collection (or set) of symbols. The symbols in the alphabet are entities which cannot be taken apart in any meaningful way, a property which leads to them being sometimes referred to as atomic.". By the definitions for demonstration purpose let A be the alphabet consisting of atomic symbols contained in set

$$\{a, b, c\}.$$

Then we may define several languages including the language A^* which is defined as a language consisting of all strings of symbols (with repetitions) belonging to the alphabet A including the empty word [65]. Then words like

$$\{aaa, ab, aaaab, abc, bca, \epsilon\}$$

where ϵ denotes empty word, are all contained in the language, but words like following are not:

$$\{d, af, fc\}$$

A subset of language A^* is the language A^+ that does not contain the empty word. One may observe that

$$A^+ \subset A^*.$$

Having said what the language is, it becomes clear that neither of the presented languages is easy to use, as they contain infinitely many words (assuming that the alphabet is not empty). Such description of a language is too general to restrict it enough to make it useful, and defining a language as a closed set limits possibilities. There are different ways to define language, but the most general is use of grammars. Grammar is defined by the previously mentioned book as "a set of rules for generating strings". These rules can limit generated languages to such an extent that they may be used in automated processing.

Such formalization of a language allows creation of automatons that will accept and interpret the language. Examples of such application of grammars are compilers and parsers. Further formalized, PSG grammar, was proposed by Noam Chomsky and defined as a tuple (N, T, P, S) where N is set of non-terminals, T is set of terminals, P is set of production rules and S is a starting symbol.

Then the string belongs to the language generated by CFG if and only if there exists such a finite list of applying productions to non-terminals to achieve that string. String must not have the non-terminal symbols — it has to be fully terminated.

2.2.1. Non-terminals

In the definition of PSG, non-terminal is an atomic symbol that is present in at least one production rule on the left side — being a point where another transformation of the string may occur. Then N is a set of all such symbols in the given grammar. In most of the theoretic literature these are by convention denoted with uppercase Latin letters, whereas in computer implementations, following BNF conventions, non-terminals are mostly implemented by a word enclosed in angle brackets. In case of using multiple words, spaces are replaced with underscores.

2.2.2. Terminals

The set of terminals is an alphabet of all atomic symbols that cannot be further transformed using production rules defined in the grammar. Often, authors tend to denote them with lowercase letters of the Latin alphabet.

2.2.3. Production rules

Production rules are defined as mappings from non-terminals to terminals or non-terminals. Formally one may define them as follows: the production has a form

$$\alpha \to \beta$$

where α is a left-hand side of the production, β is a right-hand side of the production and

$$\alpha \in (N \cup T)^{+}$$

$$\beta \in (N \cup T)^{*}.$$

In this definition α is a word belonging to a language generated by an alphabet being a conjunction of sets of terminals and non-terminals. Similarly, β is a word belonging to a language generated by an alphabet being a conjunction of sets of terminals and non-terminals, but this time also the empty word ϵ. Additionally, α is subjected to an additional constraint of containing at least one non-terminal symbol. Examples of such productions are:

$$A \to aaAB,$$

$$B \to bb,$$

$$aA \to ac.$$

It is worth to emphasize, that it is allowed to use terminal as a part of left component of the production rule, but as it was said previously they are not allowed to be transformed — their order in respect to other symbols must be preserved in the resulting word.

2.2.4. Start symbol

Start symbol is a symbol that always begins the process of derivation. It is a symbol from a set of non-terminals.

2.3. Context Free Grammar – CFG

On top of the CFG concept, Noam Chomsky build a hierarchy of languages based on their characteristics allowing easier computations. One of that types that is simple enough to be used in case of Grammatical Evolution is CFG — Context Free Grammar. CFG has additional requirements that apply to a set of production rules, namely in CFGs it is not allowed to use

terminal symbols on left-hand side of production and only one non-terminal may appear in the left part of production. Formally, CFG productions have a form

$$\alpha \rightarrow \beta$$

where

$$\alpha \in N, \beta \in (N \cup T)^*.$$

Example productions that are allowed are then

$$A \rightarrow aB,$$

$$B \rightarrow aa,$$

$$A \rightarrow \epsilon.$$

2.4. BNF

BNF is a notion to describe CFGs in a formalized way. This notion was originally proposed by John Warner Backus [7], later it was described as metalinguistic formulae [6]. Refereed report contains BNF in a form that differ from originally proposed notion — over the time, certain rules got modernized. There are a lot of good articles on BNF notion in the internet [46]. Each rule in BNF is described with a non-terminal, symbol ::= that, according to the mentioned article, could be understood as "may expand into" and expression describing possible replacements. Non-terminals are created by surrounding the name of the token with angle brackets. There is also the possibility of describing multiple possible choices for the new sequence — each option should be separated using | sign. An example of grammar creating a real number in BNF notion is presented below.

$$< real\_number >::=< int > . < decimal\_fraction >$$

$$< decimal\_fraction >::=< digits >$$

$$< int >::=< 1\_9 > | < 1\_9 >< digits >$$

$$< digits >::=< digits >< 0\_9 > | < 0\_9 >$$

$$< 0\_9 >::= 0|1|2|3|4|5|6|7|8|9$$

$$< 1\_9 >::= 1|2|3|4|5|6|7|8|9$$

2.5. EBNF

EBNF is an extended version of BNF that is standardized by ISO organization under standard ISO14977 [34]. As key motivations behind development of EBNF sources point out need for extendability and lack of direct representation of repetitions and optional occurrences in BNF [69]. EBNF addresses all that issues by defining new symbols, making it closer in usage to regular expressions.

2.6. Grammatical Evolution

Grammatical Evolution [47] is a concept that originates in Genetic Programming [39] taking advantage of Formal Languages Theory to provide better performance [23]. Formal Grammar provides additional domain knowledge [54] that constrains the solution and ensures it is well-structured and follows certain rules. As a result, GE provides a way to achieve a transparent and explainable model. Documentation of PonyGE2 — one of GE tools — states that GE's modularity gives huge flexibility, allowing use of alternative search strategies [23].

The main idea of Grammatical Evolution is that genotype-phenotype mapping is performed as generation of sentences according to provided grammar. Genome is used to perform selection of available production rule in a deterministic way.

This approach also makes it possible to create a syntactically correct computer program in any arbitrary language [52], or a correct and valid mathematical expression realizing the concept of so-called Symbolic Regression [68]. Furthermore, thanks to the grammars, it may be applied to wide variety of applications like for example the case mentioned by publication entitled "A Grammar-based Genetic Programming Approach to Optimize Convolutional Neural Network Architectures" [17] — evolving architecture of a Convolutional Neural Network.

3. Details of Grammatical Evolution

3.1. How GE works

Grammatical Evolution (GE) is a kind of genetic algorithm, meaning that it is based around the concept of evolution. First, the genome is established — it is usually a sequence of integer numbers initialized with random values. To reach the solution that is encoded by genome, mapping process is needed. If it is defined as a simple mathematical operation [47] — let $< s >$ be starting symbol and let $P = < a >, < b >, < c >$ be set of possible productions. Then P can be represented in BNF form as presented below.

$$< s >::=< a > \mid < b > \mid < c >$$

Let also $G[i]$ indicates the i-th codon of genome. Then a production rule is chosen according to formula stated below.

$$rule\_index = G[i] \mod |P|$$

It is important to note that production rules are indexed starting from 0. In the presented example, let $G[i]$ be 133. Then having 3 production rules, rule is chosen as presented in following calculation:

$$rule\_index = 133 \mod 3$$

$$rule\_index = 1$$

According to the calculations' gen points to second rule meaning that $< s >$ is transformed into $< b >$. For next transformations, subsequent codons are used. In case of running out of codons, one should start reading it from the beginning. This technique is heavily inspired by the natural phenomena of gene-overlapping [47]. As the authors of the original paper suggest it is important to note that each time codon is read in pair with the same non-terminal, always the same output is generated, but in case of different non-terminal, codon may produce different

output. The process of generating phenotype out of genotype may be continued till it is fully terminated, or one may specify maximal number of genotype wraps to limit possible infinite loops.

As a genetic algorithm, GE requires operation of mutation and crossover to be defined by the user.

3.2. Defining a grammar

Grammar is the additional piece of knowledge that is being provided by the user into the system, allowing efficient search that limits the search space to only entities that follow certain stated rules. As described in previous sections, it is the first step in solving a problem using GE methods. It is often mentioned as a most difficult part and indeed it has undoubtedly significant impact on correctness of solution and the speed of search. These points are clearly stated in publication "Initialization and Grammar Design in Grammar-Guided Evolutionary Computation" by words that grammar "should focus on efficient and elegant expression of language, and preserving modularity in solutions" [15].

Mentioned publication emphases also the point that grammar design is crucial for efficiency of algorithm. Another mentioned point is that grammar should not introduce bias in search by favoring some choices. There are also stated steps towards increasing quality of grammar: "Balancing, Unlinking, Eliminating non-terminals, removing grammar biases, prefix notation, Compromise grammar." [15] Similar approach is taken by other authors [41].

3.3. Defining a fitness function

Fitness is a measure of the utility of the given individual. Its purpose is to show how much value is in the individual in terms of the solution. A correct definition of the fitness calculation procedure is crucial in the process of creating a GE model.

Fitness allows selection of individuals that are going to be used in the next generations. To make it possible, the fitness function must precisely define what is valued more and what is less important.

Most of the sources point out that designing correct fitness functions may pose a challenge for users. It is very easy to mismatch the weights of features promoting not desired behavior.

There are many guidelines on how to design the best fitness functions, but there is no single rule. Looking for the guidelines, one may come across different research [8] examining statistical metrics as fitness functions for grammatical evolutions.

Some sources [2], [20] mention that to promote simplicity and explainability, it is worth binding fitness to the length of the solution.

3.4. Selection of individuals for next generation

Selection is a stage of the evolutionary process preceding the creation of a next generation, when some of the individuals from the current generation are being chosen to reproduce to create the new generation [70].

Individuals are selected using different methods, but most of them perform selection depending somehow on the value of fitness of the individual.

There are many known strategies to perform that selection [67]. Most often the tournament selection is chosen. According to sources, it involves conducting several tournaments based on fitness value among a randomly selected subset of the generation, selecting winners of tournaments.

There are lots of resources available describing possible selection strategies. Some of them use fitness measures to determine the probability of being selected and then running the random selection with this custom probability distribution.

Below, a list of a few resources describing other selection strategies is presented:

- https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm
- https://github.com/PonyGE/PonyGE2/wiki/Selection
- https://medium.datadriveninvestor.com/genetic-algorithms-selection-5634cfc45d78

There are also numerous publications handling the issue of rating the selection strategies. To mention just two of them: [56] and [72].

Choosing the right selection method may have a crucial influence on the efficiency of the final solution.

3.5. Evolution — Creating subsequent generations

Evolutionary algorithms perform, in iterations, a well-defined routine. It was clearly described by Figure 1 in the publication entitled “Comparative Study of Different Selection Techniques in Genetic Algorithm” [72]. The presented sequence is very general, and therefore it needs to be adjusted to the Grammatical Evolution technique, which is a specialized technique that falls into Genetic Programming. The modified list is presented below.

1. Initialize the population — the process when starting the genome for the population is established.
2. Mapping — process when genotype in the form of numbers is translated into phenotype that is the string described by the grammar.
3. Calculating fitness — computing the value of fitness according to the chosen strategy.
4. Checking the completion criteria — a phase when the algorithm should evaluate all of the constraints, namely desired fitness value and the maximal number of allowed generations. If the criteria are met, the algorithm should return the result.
5. Selection — process of choosing which individuals are going to be reproduced in the next generation and if they are going to be crossed over or copied in exact form.
6. Cross-over — a process when a new genome is created from two initial ones.
7. Mutation — to allow escaping local optima, random mutations are introduced into the system. In this phase, some percentage of individuals are subjected to mutations according to a chosen strategy.
8. Returning to point 2.

The presented list of steps shows the general concept of process realized by Grammatical Evolution. It is different from the normal evolutionary algorithm because it has a well-defined procedure of mapping the achieved genotype to the phenotype that could be then evaluated. The mapping process is based on grammars that were described in previous sections.

4. Requirements of users

User requirements tend to be one of the best measures in terms of pointing out the best tool. This chapter handles the topic of what user of GE tool need, providing a solid foundation for further discussion on selecting the best tool. This section is going to use concepts referring to software quality [71].

4.1. Technical aspects

4.1.1. Time efficiency

As a main technical aspect, scientific users value the efficiency of the tool, especially the time efficiency.

Precisely to measure such efficiency, similar task should be given to each tool and time measures should be taken. The task needs to be computationally hard to emphasize differences and make them distinguishable from the noises.

4.1.2. Compatibility

Another aspect that users value is compatibility of tools with the current system. Solving compatibility issues often have huge negative impact on efficiency of the whole system, therefor users value tools that are able to integrate with their environment. This aspect can be further unfolded to for example used language, its version, additional dependencies, constrains on versions of that dependencies, target platform etc. making comparison based on it quite complicated.

Additionally, the more flexible the tool is, the easier it is to port it to other platform and use in a production environment, allowing the user to take advantage of it across a wider range of applications.

4.1.3. Dependencies

Projects that have fewer dependencies are easier to maintain and modify, additionally the installation process as well as updates are easier to execute. Therefore, user values more the software that will have fewer dependencies.

4.2. Non-measurable features

4.2.1. Ease of learning

The typical user tends to use tools that he had already learned and is capable of fluent use. In terms of new tools, users like tools that are easy to learn and appears to them as intuitive. To satisfy this need the software needs to follow commonly used standards and conventions and provide well written documentation that no only shows parts of the tool but also explains their meaning, show possible applications, points out common issues and their solutions. Additionally, it is helpful to provide lots of examples as they present all the features in real environment and significantly influences user's understanding of documentation, resolving any possible inconsistencies.

Documentation should be also kept up to date with updates of the tool allowing user to have up-to-date information, reducing time of debugging or searching for correct solution.

4.2.2. Life cycle of tool

It is important to use tools that re still maintained. It allows users to use up-to-date solution and reduces risks of compatibility related issues as the tool still receives updates.

Additionally, users may also value frequency of releases — frequent updates ensure that new features are developed in short intervals and issues are addressed in reasonable time, not causing delays on users side.

Issue reporting and solving have also huge contribution to this aspect. Users value quick developer response to bugs, issues, and questions. As developers tend to use different methods of contact, users value more comfortable and easy accessible tools allowing interaction with developer.

4.2.3. Community around the tool

Complementary to support provided by the developer, community support is also highly valued by users. Size of community determinate ease of finding resources relevant to a wide

range of questions related to the tool. As an example, https://stackoverflow.com may be pointed, where community of developers share ideas and provide help for each other. Natural consequence of huge community is reduced time of receiving answer or finding solution to particular issue.

Tools with bigger community should be chosen over those with small or not often active community as it may have significant impact on user experience and time it takes to prepare a solution with the tool.

4.2.4. Documentation and resources

A key feature in understanding how to use the tool is comprehensive documentation and guides describing correct use of the tool with examples and advises. This enhances the experience and reduces time and effort needed to start using the tool and improves the learning process. Users would rather go for the software that they can easily understand and imagine how to use in their specific case.

Because of this, amount of the documentation, tutorials, and guides as well as their quality should be taken into account while comparing software tools.

4.2.5. Quality of code and ease of modification

Quality of code plays a key role in the maintenance process. It allows quick modifications without need to debug or refactor bigger parts of code. Also, it ensures easy understanding of underlying logic, allowing the user to accurately perform needed adjustments or to further develop the system to the own needs. Quality of code also gives the ability to perform analysis of this code quicker. Another advantage is that good quality code is more reliable and more prone to bugs.

Taking into account, the quality of the code seems to be reasonable from many points. It may have huge influence on reliability, user experience and long term life of the project.

4.2.6. License

In terms of reusing the tool in the own project, the license may be crucial as it defines the boundaries of legitimate use as well as states the conditions under which the user is allowed to use the tool. Another important thing is that it also covers rules on accessing the sources of tool. This detail is important for user experience as access to sources may significantly increase speed of development or debugging by introducing knowledge on underlying code whereas in terms of closed source system, user my only depend on the developer to receive information.

In choosing a tool that would be possible to be used, it is undoubtedly crucial to examine the license rules. Additionally, it is worth to notice if license allows accessing the sources — this may significantly help the user and prove that the system behaves in a desired way.

4.2.7. Current user knowledge and preferences

Due to currently possessed knowledge, the user may prefer one tool over another. The main factor here would be the language the tool is written in. User would prefer the one that uses language he knows best as it is easier to adopt it rather that learning first the language and then the tool itself.

Such observation has significant impact on the comparisons as this aspect may differ across the users and there is no available method to fairly judge it. The only possibility is to assess it based on preferences of developers gained via surveys and comparing by given technology's market share.

4.3. Conclusion on requirements

There are numerous criteria that need to be taken into account. Many of them are non-measurable, posing a serious challenge in terms of conducting a fair comparison. From many points it appears that due to the wide variety of possible parameters it may not be possible to point the one single winner, but rather for each particular situation the other tool may be the best choice. That choice may depend on current user knowledge, requirements, dependencies, and architecture of the current system, emphasis on certain feature, license requirements etc. being features of a single particular use case.

5. Details on analysis and comparison of tools. Selected issues.

By efficiency, one should understand the ration of work done and consumed resources. The resources in terms of algorithm are time and memory.

Comparison of efficiency of tools selected here, could not be precise due to many factors that have huge influence on potential results. A few of the identified issues causing issues with comparison were listed and described below.

5.1. Differences in architecture of tool

Among available tools there are frameworks, libraries and ready applications. From this point of view it is impossible to compare them with each other with respect to time taken to solve some problem as they serve different purposes and require different level of developer involvement. To conduct fair comparison, it would be required to isolate the common part and measure just that. Libraries that provide only subset of functionalities given in framework are likely to be faster because of fewer operations and on the other hand final execution time may get highly influenced by user written code that introduces impreciseness in measuring tool efficiency as it does not depend on the tool but on user familiarity with the tool and users general programming knowledge.

Similarly to time taken, the memory usage also highly depends on chosen architecture of the tool — libraries would use less memory as they perform less work, while application that needs from user just the configuration will use the most as it delivers more functionalities.

Having that said, consumed memory and time measures could not be a precise because amount of already prepared code is different, therefor it is impossible to treat mentioned criteria as reliable indicators of tool quality.

5.2. Differences in provided functionalities

Tools cover different functionalities that get executed during the evolution. As an example it may be pointed out that PonyGE2 prepares a full report with statistics and final output while PyNeurGen being a library sticks to providing a result in the form of a variable. Then user is responsible to take care of presentation and storing of result.

Again, such difference means that comparison based on time or used memory measure would not be fair as measures would cover different subsets of functionalities that get executed.

5.3. Different target platform

Available tools are build using different languages to target different environments. Due to such differences, comparison between some tools may be impossible. For example, it would not be reasonable to compare memory usage in c++ and python application nor the time consumed for execution. But it does not mean that one is of worse quality. Basically, higher resource usage can be treated as a trade of for using language known by most data scientists.

Again, this shows lack of ground for performing a fair comparison based only on numbers, as they are partially results of presence/lack of some advantages.

5.4. Different way of providing input data

Among available tools, the way of providing input data slightly differ. Some of them like libraries get the data passed as a function parameter, while other tools need to read them from a file. As file operations cost a lot more time than in memory operation, it includes time overhead for all tools reading data from files instead of having them ready in memory.

This difference becomes clear in terms of previously mentioned libraries vs frameworks vs applications. The convenience of ease of use — reading config from a file instead of preparing the own program that will have it ready in memory — comes with the cost of time of reading files. However, for comparison it means that this overhead should not be included in final results as one tool does take more time to execute the task but performs significantly more work.

5.5. Conclusion on possible issues

Due to presence of differences in specifications of each tool, performing exact time and memory usage measures is pointless. It is not possible to isolate a common interesting part in

all tools that could be subjected to fair numerical comparison. In each tool delivered advantages comes with certain costs and for each project different tool may be the best choice depending on user preferences and specific requirements.

As proven in earlier part of this chapter, numerical comparison based on statistics is not possible due to numerous differences in tools specifications. Therefor, another approach must be taken to present quality of tools and compare them.

5.6. Presentation of tools

According to identified requirements and issues, selected tools will be described widely to provide as many insights as possible about the experience of using the tool with results of experiments conducted with use of the tool.

Each tool has notes on its life status, size, and activity of the community around the tool to provide information on potential availability of fixes, patches, and updates as well as help and answers to specific questions.

Furthermore, description contain information about available official resources and their quality. It includes official publications, wiki pages, tutorials, and documentation.

Besides of that, descriptions contain information on available literature that provides information related to the tool as well as reviews of them.

Each description is accompanied by description of usage examples with detailed instructions on recreating results.

5.7. Method of comparison

Due to already mentioned factors, exact comparison is not possible to be done. Instead, for each tool, advantages and disadvantages are presented without pointing out a winner. Presented details are provided to make the reader able to conduct the comparison on her/his own, having in mind project specific needs and constraints. As it was mentioned, these may have significant impact on the selection.

6. PonyGE2

PonyGE2 [25] is a tool developed in Python. It does come with all necessary features to start model development. The tool is very mature and advanced. It is designed as a standalone, modifiable application.

6.1. Literature

PonyGE2 is pointed out as a tool for people who start work with Grammatical Evolution. Some publications classify this tool as a state of the art [61]. In the article "Software review: Pony GE2" it is described as a well written tool that has a lot of possible customization options [60]. The tool is said to be very efficient and lightweight. The paper also mentions some drawbacks, including old multi-objective optimization algorithms or lack of good issue reporting tool for community. As another issue, the author of the article points out possible inconvenience for not experienced users — lack of easy installation, for example by python package manager pip.

The mentioned article brings up a lot of potential use cases, pointing out that the package comes with ready to use examples.

Potential use of the tool was also mentioned in the description of a workshop that took place during PyCon 2022 conference [28]. Authors of the workshop suggest that PonyGE2 may be used to create high entropy seeds for cryptography operations. This is an important use case, as high entropy seeds are one of the crucial parts of cryptographic data protection that is resistant to crypto analysis. The high level of entropy guarantees that the seed should be close to random.

The article presenting another tool — GRAPE [11] — brings up another important issue that it is important to construct software around well known and adopted frameworks, which PonyGE2 does not — it is build around its own architecture. The mentioned paper also compares GRAPE to PonyGE2, presenting that there are fewer methods available in PonyGE2. Although drawbacks of PonyGE2 are brought up, both tools are presented as comparable when it comes

to achieved results. Such altitude of author suggests that PonyGE2 is one of the current leaders in GE implementation.

Some of currently available tools also tends to build around PonyGE2. For example, “evoltree: Evolutionary Decision Trees” [36] is said to be using PonyGE2 as its engine for some cases.

PonyGE2 has also been used as a tool in improving architecture of convolutional neural network [16] proving that it achieves probably best results for nowadays use cases.

The main source of information about this tool is its own wiki https://github.com/PonyGE/PonyGE2/wiki, the original paper, and well-prepared examples. It is also possible to easily reverse engineer the mechanics of the tool, as the code does contain useful comments. Additionally, there is a tutorial, written by the tool user, available via URL https://towardsdatascience.com/introduction-to-ponyge2-for-grammatical-evolution-d51c29f2315a. It provides clear guidelines on designing the own models.

6.2. Documentation

Documentation of the tool does contain a lot of useful information, it is written in such a fashion that users not familiar with programming should also be able to take advantage of it. It does mostly describe concepts used in the tool and parameters that could be changed to fine-tune the process. It does not contain a very detailed description of technical aspects, but in conjunction with comments present in the code, it does answer all the questions that may emerge while preparing the own model.

Some of the most important topics emphasized by the documentation are presented below.

6.2.1. Requirements

From the documentation, one may read that the tool has very little dependencies [24]. The requirements are presented as follows:

> PonyGE2 requires Python 3.5 or higher. Using matplotlib, numpy, scipy, scikit-learn (sklearn), pandas.

6.2.2. Evolutionary parameters

PonyGE2 gives the user direct access to evolutionary parameters in a couple of different ways. First of all, these data are stored in a single dictionary and accessed through it to maintain consistency.

User may modify these parameters by use of special file by putting in each line parameter name, colon and its value. Then path to file is passed to the main script as a console argument, allowing the program to read all parameters in a simple way.

Another method to set the parameters is by use of command line arguments. This way allows quicker changes, but in case of batch jobs may not be ideal.

The order of sources of parameters is set as follows: default values are taken from the dictionary, then they are updated with file contents, last is updating them by values of command line arguments.

Additionally, the package allows users to add own parameters that may be used in algorithm

6.2.3. Grammars

According to documentation, the most important part of GE algorithm execution is to specify correct grammar that will correctly describe the solution of the problem. Authors explicitly say that quality of grammar does have direct impact on performance of the whole system

6.2.4. Details on genome

Genome is represented as a set of codons that are limited by codon size parameter.

Documentation provides the information that in case the genome is too short to fully terminate, the sequence wrapping may occur within limits specified by the user. Namely, a wrapping operator gets applied to the genome to allow re-reading the genome from the beginning.

6.2.5. Fitness function

Package comes with ready fitness functions that are prepared as examples. User is allowed to define own functions as classes derived from `fitness.base_ff_classes.base_ff`. Worth to notice is that by default package is minimizing value of fitness. To cause maximization, one should set parameter `maximize` to `True`.

6.3. Maintenance

The tool is available via GitHub repository that has the last commit done in 2022. Additionally, the repository contains the issues that are dated also in 2022 that were addressed by tool maintainers. The tool is clearly used by an active community. The repository contains information about over 80 forks. Most of public forks were updated within last year. Additionally, by querying google search engine with the phrase “PonyGE2” one may get numerous results that describe the tool.

Additionally, there exist third party modified versions of the tool, proving the presence of community around the tool. Some of the repositories can be seen in forks section of repository and on websites like https://paperswithcode.com/paper/ponyge2-grammatical-evolution-in-python.

Besides, there is data related to the tool easily available to users, it is still non-comparable with leading software tools in other fields. There are not many guides and tutorials for new users, and the main source of knowledge is the documentation.

Having all of that information, it may be concluded that PonyGE2 is a tool that is still maintained by authors and community. As a result of narrow specialization of the tool the size of community is non-comparable to leading software, but in comparison to other GE tools, the community around PonyGE is well established and of considerable size. Additionally, information provided by developers cover most of the use cases and potential questions that may arise during exploitation.

6.4. Installation

Installation is straight forward. It involves downloading the sources from repository https://github.com/PonyGE/PonyGE2 and installing dependencies using pip, via provided `requirements.txt`. It is compatible with latest versions of python. During experiments, it was tested using python 3.7, but the documentation mentions that every version >=3.5 is allowed.

The procedure described in the documentation worked as intended without any problems. The installation was deployed under venv.

6.5. Usage

PonyGE2 is written in a fashion that it can be used as a ready application called by a command line. In terms of programming own experiments it resembles fashion similar to frameworks providing strict limitations and clear guidelines along with all necessary scaffolding to place own code.

Data sets and parameters are passed to the PonyGE2 by use of files. It is also possible to pass parameters as a command line arguments. The whole list of available command line arguments with their descriptions can be obtained by use of command presented below:

```
python ponyge.py --help
```

To run the tool, user should set working directory to `src` folder of the tool and call python to execute its main file — `ponyge.py`. Parameters of evolution are specified by command line arguments or contained in parameters file — then just file need to be passed as an argument using `-parameters` key. Parameters include values that determine features like BNF grammar to be used during the mapping process; size of population; number of generations; probability of crossover and mutations; maximal length of genome; selection strategy; used fitness function. Detailed description of all parameters meaning can be accessed by calling PonyGE2 with argument `-help`.

It is worth to mention that during evolution, user is informed about progress of execution as a percentage of work done. Additionally, by use of `-verbose` parameter, the user may enable printing additional information on current work.

The tool comes with predefined safe mathematical functions that can be used in grammars to limit runtime errors like illegal zero division. Predefined functions define special values for arguments that would normally cause issue, allowing easier design of grammar and more efficient evolution. Mentioned functions are contained in file located under following path `src/utilities/fitness/math_functions.py`

The repository does contain many predefined fitness functions allowing the user to quickly prepare an experiment without need to prepare the own function, but not preventing the user from doing so.

6.5.1. Grammar design

PonyGe2 does provide a well-designed mechanism for handling grammar. First of all, grammar is placed in an external file that may be shared by different models. It is defined using the usual BNF notion. Examples of grammars are delivered with the package, making it even easier to learn the principles of BNF syntax.

As the original publication [25] says that the first non-terminal in the definition of BNF grammar is treated as a start symbol, reducing the amount of needed configuration. It could be considered as a good practice as the starting point is a specific feature of the grammar, not the model — this approach allows storing the starting symbol and grammar in a single place.

It is also worth mentioning that PonyGE2 comes with a mechanism able to handle indentations and new line characters in the generated code, making it possible to quickly design grammars describing syntactically correct Python code without performing additional transformations on phenotype before evaluation. Indentation blocks are enclosed by special sequences of characters: `{:` and `:}`. Additionally, to insert a new line, one may use sequence `{::}`.

6.5.2. Fitness function

To define the own fitness function, the user should create a class that inherits from `base_ff`. To define an evaluation routine, one should override function `evaluate` that is declared as follows.

```
def evaluate(self, ind, **kwargs):
    pass
```

An example class is presented below. It assumes that the fitness calculation formula is always contained inside an evolved function and stores the value in a variable named `FITNESS`. This can be achieved by specifying a correct starting point in BNF grammar.

```
from fitness.base_ff_classes.base_ff import base_ff


class own_fit(base_ff):
    # Parameter controling if fitness should be
    # maximised or minimised
    maximise = True

    def __init__(self):
        # Initialise base fitness function class.
        super().__init__()

    def evaluate(self, ind, **kwargs):
        # d is dictionary that contains variables to be
        # used or set by exec()
        d = {}
        # p contains string that is result of genotype to
        # phenotype mapping
        p = ind.phenotype
```

```
20          # Executing phenotype with specified dictionary d
21          exec(p, d)
22
23          # Output can be directly retrived from dictionary
24          # passed to exec(). In this example it is assumed
25          # that value is calculated inside program passed
26          # to exec function and its value is stored in
27          # variable named 'FITNESS'
28          s = d['FITNESS']
29
30          return s
```

PonyGE2 sources does contain a lot of examples on writing own fitness classes. Mentioned examples are contained in directory `src/fitness`. Each example does contain comments that explain clearly the purpose of the code and enabling the user to quickly get an idea on creating the own class.

It is worth to mention that predefined classes do cover most of the use cases.

6.6. Retrieving evolved results

PonyGE2 comes with all necessary tools. Produced results are interpreted and sent to console and in parallel report is being saved in results directory. It contains a list of used parameters; file containing description of the best entity — number of generation, phenotype, phenotype, and fitness values on training and test data sets; PDF file containing graph of fitness thought the generations; file containing numeric statistics about the process.

Such approach is a significant advantage, moving the users' attention to developing best experiment instead of taking care of preserving data and preparing report.

6.7. Examples

The package does contain several examples that may be used or transformed to receive an easily working solution. Examples cover different areas of application

6.7.1. Regression

In this particular example, it is shown how to evolve function formulae that describes the provided data set. For the purpose of testing the software, I have modified it to perform regression over a linear function. The given function was

$$f(x) = 2x + 5$$

The data set contained two variables: first one $x_1 \in < 1; 38 > \wedge \ x_1 \in N$ and second one being random integer value. The following parameters were used:

```
CACHE:                  True
CODON_SIZE:             100000
CROSSOVER:              variable_onepoint
CROSSOVER_PROBABILITY:  0.75
DATASET_TRAIN:          2xp5/dataset.txt
DATASET_TEST:           2xp5/dataset.txt
DEBUG:                  False
ERROR_METRIC:           mse
GENERATIONS:            50
MAX_GENOME_LENGTH:      500
GRAMMAR_FILE:           supervised_learning/regression2.bnf
INITIALISATION:         PI_grow
INVALID_SELECTION:      False
MAX_INIT_TREE_DEPTH:    10
MAX_TREE_DEPTH:         17
MUTATION:               int_flip_per_codon
POPULATION_SIZE:        500
FITNESS_FUNCTION:       supervised_learning.regression
REPLACEMENT:            generational
SELECTION:              tournament
TOURNAMENT_SIZE:        2
VERBOSE:                False
```

The file `regression2.bnf` had just been adjusted not to include non-existing variables. Used BNF grammar is presented below:

```
<e>  ::=  <e>+<e>|
          <e>-<e>|
          <e>*<e>|
          pdiv(<e>,<e>)|
          psqrt(<e>)|
          np.sin(<e>)|
          np.tanh(<e>)|
```

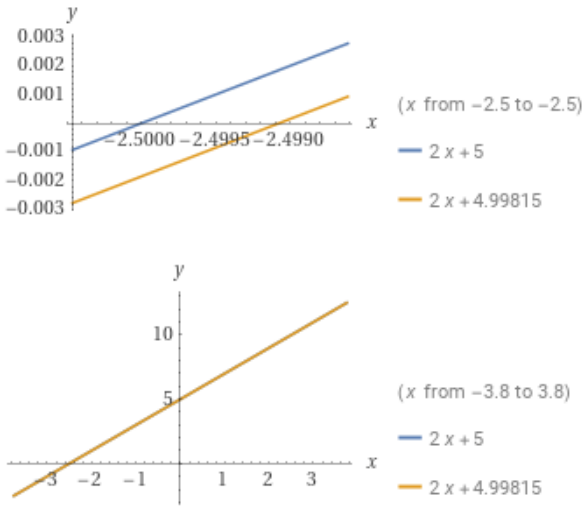

Figure 6.1. Comparison of simplified evolved function to the desired function $f(x) = 2x + 5$. In the second plot both lines are so close to each other that the blue is almost visually covered with the orange line, showing a high precision of the gathered result. Plot has been prepared using https://www.wolframalpha.com/

```
            np.exp(<e>)|
            plog(<e>)|
            x[:, 0]|x[:, 1]|
            <c><c>.<c><c>
<c>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

As a result, PonyGE2 returned the evolved expression that is presented below. It does contain some issues like using functions instead of constants, but after calculating the numeric values of those functions and simplifying the formula the approximation is very close.

```
x[:, 0]-np.sin(66.59)+plog(83.09)+x[:, 0]
```

After applying the mentioned numerical approximation and simplifications, the evolved formula may be stated as follows:

$$2x + 4.99815$$

6.7.2. Program synthesis

The next example covers generation of valid Python code that should solve the problem specified in the data set. To run the example, one need to execute the following command, having current working directory set to src directory of PonyGE2 repository content.

```
python ponyge.py --parameters progsys.txt
```

Parameters file specify one of variants of the example presenting the general idea, but there are also predefined other data sets located in folder `PonyGE2/datasets/progsys`. Complementary grammars for data sets are saved in location `PonyGE2/grammars/progsys`.

This example shows a more complex approach, as inside its fitness function file there is a mechanism that allows inclusion of additional supplementary code that is not directly related to the solution but provides fitness calculating functionalities. For each data set complementary file with evaluation code is prepared that set variable named `quality`.

6.7.2.1. Compare String Lengths

This instance of Program Synthesis example is the default one. The challenge here is to evolve a program that will test if the length of provided strings meet the following condition $\text{LEN}(str_1) < \text{LEN}(str_1) < \text{LEN}(str_1)$ [32]. Strings are provided to program inside variables named `in0`, `in1`, `in2`.

Evolved code is being evaluated by previously mentioned helper code contained in `Compare String Lengths-Embed.txt`. To be precise, thanks to function `get_data` and `format_program` from `progsys.py` fitness function, the evolved code is placed inside `evolve` function defined in 92 line of helper code exactly in place of `<insertCodeHere>` in line 94. Knowing that, it is easy to understand the way the fitness is calculated.

Experiment had been executed using customized parameters to acquire better quality results. Actual values of parameters are presented below.

```
1  CACHE:                  False
2  CODON_SIZE:             100000
3  CROSSOVER:              variable_onepoint
4  CROSSOVER_PROBABILITY:  0.9
5  DATASET_TRAIN:          Compare String Lengths/Train.txt
6  DATASET_TEST:           Compare String Lengths/Test.txt
7  DEBUG:                  False
8  ERROR_METRIC:           mse
9  GENERATIONS:            300
10 MAX_GENOME_LENGTH:      500
11 GRAMMAR_FILE:           progsys/Compare String Lengths.bnf
12 INITIALISATION:         PI_grow
13 INVALID_SELECTION:      False
14 MAX_INIT_TREE_DEPTH:    10
15 MAX_TREE_DEPTH:         17
16 MUTATION:               int_flip_per_codon
17 POPULATION_SIZE:        800
18 FITNESS_FUNCTION:       progsys
19 REPLACEMENT:            generational
```

```
SELECTION:              tournament
TOURNAMENT_SIZE:        5
VERBOSE:                False
```

The evolved result is presented below. Fitness shows that the result is very good but judging from humans point of view, the result is rather poor due to lack of explainability. There are many possible causes of it.

```
i0 = int(); i1 = int(); i2 = int()
b0 = bool(); b1 = bool(); b2 = bool()
s0 = str(); s1 = str(); s2 = str()
res0 = bool()
res0 = getCharFromString('eF'.lstrip(), abs(i1)) in in2.capitalize()
.rstrip().rstrip().rstrip(in2.lstrip().lstrip().capitalize().rstrip(
in2.capitalize().capitalize().rstrip().capitalize().rstrip().rstrip(
(in0 + 'dp').capitalize().capitalize())))
if ('Z6'.lstrip(('ee>' + in0)) + 'p;').endswith('y[e'):{:
i2 -= i2
:}
```

Created code seems to be very complicated and hard to read, but despite it, the function seems to work with the provided data set. To solve the issue of readability, most of the publications suggest incorporating simplicity measure into fitness calculation. One of the simplest methods is to measure length and assume that simplicity is inverse proportional to length. Then one should change fitness accordingly to calculated simplicity of formula to maximize simplicity.

Another possible approach is to allow more generations to be evaluated or adjust other evolutionary parameters, allowing a bigger search space to be explored.

6.8. General remarks

The tool is very mature and up to date, compatible with latest versions of python and not causing any issues with environment configuration. Documentation describes clearly the functionalities and is accompanied by well-prepared examples. Code quality is very good — it is clear and easy to read. Code is equipped with well written comments that give the user insights on how the system works.

On the other hand, the tool is lacking tutorials dedicated for new users that would cover first steps in using the software. The user then has to explore examples and carefully read a significant part of documentation to reach the starting point and gain control over first experiments. However, due to narrow specialization of the tool, level of the documentation should

not be considered as a disadvantage as development of tutorials could be very time-consuming for developers and available resources contain most of needed information, needing just a little more time from user.

Taking all facts mentioned in this chapter into consideration, this tool can be pointed as worth of attention and learning.

7. PyNeurGen

PyNeurGen [57] is a Python library, developed to provide end user with all functionalities needed to develop hybrids of GE and neural networks. It comes with huge amount of features but is not maintained properly. The code base is outdated.

7.1. Literature and sources of information

This tool is not very widely described among publications. Most of its description is contained in its own documentation [57]. It is described as a tool for the creation of genetic-neural hybrids that uses the concept of Grammatical Evolution to accomplish tasks related to the genetic part of the hybrid solution.

Despite its main purpose of wrapping neural networks with grammatical evolution, according to found materials [57], [64] it is capable of solving pure grammatical evolution problems without engaging neural networks.

The tutorial presented in the documentation provides a comprehensive guide on how to construct programs solving GE problems; however, one should still keep in mind extended capabilities of the tool.

The tool has been added to `https://pypi.org/` but the description does not contain much information about the software. Additionally, the last update is written to be done in 2012. The project site is located under the following URL address: `https://pypi.org/project/pyneurgen/`.

7.2. Documentation

Tool comes with own official website hosted on `sourceforge.net` under address `https://pyneurgen.sourceforge.net/tutorial_ge.html`. It does contain an introductory tutorial on how to develop programs using PyNeurGen. The website has also API

section that contains a brief description of all public functions, but is lacking detailed descriptions of parameters and return types. Descriptions only grasp the main purpose of endpoint, leaving the user without knowledge of exact use.

Code itself can be considered as another source of knowledge as it does contain a lot of comments and is written in such a manner that having knowledge from documentation and tutorial it is possible to gain lacking information; however, it does consume a lot of time and degrades user experience.

7.3. Maintenance

Official repositories of the project seems not to have been updated since 2012 year, however it is possible to find third parties who developed some additional features. For example repository https://github.com/jacksonpradolima/PyNeurGen does contain a version with last commit done in 2021, however, the author explicitly emphasizes no affiliation with official project.

Searches performed using google search engine show that there are not many resources handling the topic of PyNeurGen. Most of the results are connected with the official website or the mentioned fork on GitHub that tries to provide some updates to the software. Unfortunately, it shows that the community gathered around this tool is very small and inactive.

Additionally, the tool is lacking compatibility with latest versions of Python — it does require version 2 of Python, creating potential integration issues like using it in conjunction with other libraries that require newer versions.

7.4. Installation

Firstly, as stated in the above section, it is worth mentioning that the library is not compatible with new versions of Python. To use it, it is needed to have python 2 installed.

The package is available via pip package manager by use of the following command.

```
pip install pyneurgen
```

During tests, it went out that this method did not work smoothly. Authors' documentation suggests using another tool — easy_install

```
easy_install pyneurgen
```

This method also posed some difficulties. In order to install the library a clean virtual machine under Linux Ubuntu 20.04 was used and following steps were executed

```
sudo apt-add-repository universe
sudo apt update
sudo apt install python2
curl https://bootstrap.pypa.io/pip/2.7/get-pip.py --output get-pip.py
python2 get-pip.py
PATH=$PATH:/home/pyneurgen/.local/bin
pip --version
sudo apt-get install unzip
sudo apt-get install python-setuptools
```

Then the sources of pyneurgen should be downloaded from the repository http://sourceforge.net/projects/pyneurgen/files/ and then unzipped using the unzip tool that was installed by the commands provided above.

After following all the specified steps, library is ready to be used. The installation folder contains examples that are located in the demo folder and can be run just like any other ordinary Python program.

```
~/pyneurgen-0.2/pyneurgen/demo$ python2 sample_grammatical_evolution.py
```

7.5. Usage

The tool is prepared as an external library for Python programming. To take advantage of its features, user need to develop an own python application using provided functionalities. The tool's website https://pyneurgen.sourceforge.net/tutorial_ge.html contains a tutorial on developing the application that is accompanied by an example in the downloaded sources (`pyneurgen-0.2/lib-files/pyneurgen/demo`).

First step is to define the grammar in BNF notion. Documentation is lacking information about defining the staring point for the grammar, but reverse engineering of the code shows that the start symbol is hard coded in file `genotypes.py` in function `get_preprogram()` to be "`<S>`". Additionally, documentation [19] mentions that each non-terminal starting with characters `<S` will preserve the indentation spaces to ensure Python syntax validity.

BNF grammar should ensure, for example by providing it in the starting point rule, that code after evaluation provides value of fitness to the library. All code responsible for that calculation is to be written by user inside grammar. To return fitness to the library, one would need to use the following syntax.

```
    self.set_bnf_variable('<fitness>', fitness)
```

Where `fitness` is a variable containing the final value of fitness. The delivered example shows clearly how to use this functionality.

Having that done, the user should generate `GrammaticalEvolution` object from `grammatical_evolution` module. Then all settings are done using this object's methods. It is worth to mention that this tool allows to specify different completion criteria than just defined number of generations, namely tool allows specifying desired fitness that one want to achieve — after reaching that value or better evolution stops.

A complete list of possible settings is presented in the documentation and the tutorial. Special attention should be paid to the number of generations, the size of population, completion criteria, the type of fitness optimization, and the size of the program. The last one is especially important, as the length of the starting point is also considered as a component of program size. In case of providing a maximal value that is less or equal to the length of the start symbol, evolution may end with not fully terminated string, leading to execution errors.

After providing all of the settings, the user has to invoke `create_genotypes()` method on a created `GrammaticalEvolution` object and then follow it by calling `run` on that object. The last mentioned function provide evolution result as the return value that can be directly printed into the standard output using print function.

It is worth to mention that PyNeurGen provide functionality of preserving variables that were calculated during phenotype evaluation. Namely, it is possible to use `set_bnf_variable` function in BNF grammar with other variable name than `<fitness>` to save additional data. Additionally, there also exists a mechanism for accessing that value to include it in calculations. Following code presents mentioned functionalities

```
    value = float(i) / float(100)
    self.set_bnf_variable('<value>', value)

    self.runtime_resolve('<value>', 'float')
```

Presented code is taken directly from the example and shows how a value calculated inside python code can be then passed into BNF rules, for example similarly to a constant value.

7.6. Retrieving evolved results

In PyNeurGen, as stated in the section above, results are retrieved as a return value of the library function that is responsible for starting the evolution routine. There is no such mechanism as in PonyGE2 that creates report for user.

Below there is an example code snippet presenting running the model and retrieving the evolved result.

```
# Be aware that it is Python 2.*!
# [...]
```

```
ges = GrammaticalEvolution()
ges.create_genotypes()
print ges.run()
```

Documentation does not state it clearly, but reverse engineering of library code shows that `ges.run()` returns the best individual from all of the evolved ones.

It is also possible to access a list of fitness values of all individuals sorted according to the chosen strategy, using the following code.

```
print ges.fitness_list.sorted()
```

Additionally, to receive pure evolved program from the best individual's genotype, one may use the following piece of code

```
gene = ges.population[ges.fitness_list.best_member()]
print gene.get_program()
```

It is worth to mention that `population` is a list of `Genotype` objects. Then `fitness_list` is an object of class derived from build in `list`. Each element of that list is another list containing fitness value and member number in order as specified. Function `best_member` does return number of individual being best in terms of chosen fitness strategy. It allows retrieving its genome from population.

7.7. Example

As it was mentioned in section 7.4 package contains single example of usage that presents its abilities in field of GE. Example solves a problem described in documentation as follows:

> *"For values 0 to 99, what expression could be used to minimize:*
>
> *`abs(expression - pow(x, 3))`"*
>
> Source: https://pyneurgen.sourceforge.net/tutorial_ge.html [57]

The code comes with defined BNF grammar and starting point that contains necessary instructions that allow calculating and setting the fitness of the solution.

To run the example, one needs to navigate to `/pyneurgen/demo` and then call python using the command presented below.

```
python2 sample_grammatical_evolution.py
```

The examples output contains a lot of information, including the fitness list described earlier, individuals with their fitness value, and finally the best individual that is the result of an evolutionary process.

7.8. General remarks

It appears that the tool is very outdated and does need a lot of work to prepare the environment. Last update of its repository on Sourceforge was done in 2012 year. The same applies to its PyPI repository https://pypi.org/project/pyneurgen/#history. Amount of examples is rather low in comparison to other tools. To be precise, there is only single example covering Grammatical Evolution and two covering other part of library.

The process of preparing the experiment seems to be straightforward, but a bit limited at the same time. User must prepare a grammar that is capable of generating python code that would set <fitness> variable. There is also a mechanism allowing user to add variables that will be added into BNF during evaluation. Additionally, one may observe that every experiment will have a very similar structure that was presented in the code of the example. The user is then only responsible for designing the BNF and adjusting the parameters of the evolution.

During experiments, it was discovered that, unfortunately, architectural choices made the tool unsuitable for some purposes and therefore limited its potential uses. It appears that processing large populations or many generations, the library often gets stuck. Additionally, defining grammar that prepares fitness calculation poses another challenge. Moreover, creating grammar for complex solutions also tends to cause problems with correct parsing.

8. gramEvol

gramEvol [44] is a package for R language that provides all necessary functionality to build a GE model. It does require very little configuration and is very well documented. It is kept up to date.

8.1. Literature

A primary source of knowledge for gramEvol is the article developed by the authors of the tool [44]. Additionally, there exists another paper developed by the authors entitled "Grammatical Evolution: A Tutorial using gramEvol" [45]. Technical aspects along with documentation are described in a PDF file served under the following URL `https://cran.r-project.org/web/packages/gramEvol/gramEvol.pdf` [21]. There also exists a website hosted on GitHub that contains the introductory tutorial [22]. Details about the tool are available via its official package website `https://cran.r-project.org/web/packages/gramEvol/index.html`.

The tool is also mentioned by a book on Grammatical Evolution entitled "Grammar-Based Feature Generation for Time-Series Prediction" [12]. The tool is being referred to, by some authors, as a state-of-the-art [53].

In terms of applications, there are a lot of examples of use cases involving Grammatical Evolution implemented in gramEvol. To grasp just a bare idea of a huge variety of applications, a few of the publications from different fields where it got applied, are listed here:

- Grammatical Evolution for Detecting Cyberattacks in Internet of Things Environments [3],
- Multi-Objective Allocation of COVID-19 Testing Centers: Improving Coverage and Equity in Access [73],
- Solution of Mixed-Integer Optimization Problems in Bioinformatics with Differential Evolution Method [53].

When it comes to internet resources handling the topic of the tool, there are not many available sources. One may encounter a couple of questions on stackoverflow.com, that remained unanswered for 8 and 6 months (Last access 08-01-2023): https://stackoverflow.com/questions/76201565/ https://stackoverflow.com/questions/76556657/. There also seems to exist some discussions on other portals that were more active, one example could be found here: https://community.rstudio.com/t/variations-of-genetic-algorithms/116110.

8.2. Documentation

As stated in the previous section there are many sources of knowledge for this tool, most of them are listed below:

- https://cran.r-project.org/web/packages/gramEvol/index.html
- https://github.com/fnoorian/gramEvol?tab=readme-ov-file
- https://cran.r-project.org/web/packages/gramEvol/gramEvol.pdf
- https://fnoorian.github.io/gramEvol/inst/doc/ge-intro.html
- https://www.jstatsoft.org/article/download/v071i01/1013
- https://rdrr.io/cran/gramEvol/

Documentation provides clear guides and tutorials that allow new users to quickly familiarize themselves with functions delivered by the tool.

The tutorial provides additionally the theoretical foundations for beginners to understand the mechanics behind the Grammatical Evolution.

8.3. Maintenance

The main code repository of the tool is located on GitHub. Statistics of the repository show that the last commit was done in June 2023. Updates are not frequent, but are performed systematically.

There are 2 reported forks of the repository, but they seem to be inactive for a long period of time.

As it was described above, there are some questions on stackoverflow.com but no responses were provided.

Available sources prove that the community of users concentrated around that tool are rather scientists than usual developers. Most of the sources that mention using gramEvol are scientific publications. Despite the closed character of the community, it appears that information on the tool is easily available to users.

8.4. Installation

Installation is performed using the command given in the README file in the repository and the getting started guides. The command is presented below.

```
install.packages("gramEvol")
```

After submitting it into the R console, the package is installed automatically. It is also possible to use the latest version of the GitHub repository. This can be accomplished by the use of the following code listed also in the project repository.

```
if (!require("devtools")) install.packages("devtools")
devtools::install_github("fnoorian/gramEvol")
```

A test of the installation procedure was performed using version 4.3.2 of R for the Windows operating system.

8.5. Usage

To begin with, the user should define a grammar using `CreateGrammar` function that takes as parameters a list of rules and the starting symbol. If no starting symbol is set, then the first rule in the list is treated as the starting point.

The next step is defining the fitness function that will be used to rate the individuals. To do so, the data set should also be declared. The function should take evolved expression as a parameter and return the value of fitness.

Having prepared the fitness function and the grammar, it is time to code the settings. It is done as parameters of `GrammaticalEvolution(...)`. Few options are presented in the following example taken from the introductory tutorial.

```
GrammaticalEvolution(grammarDef, evalFunc,
    numExpr = 1,
    max.depth = GrammarGetDepth(grammarDef),
    startSymb = GrammarStartSymbol(grammarDef),
    seqLen = GrammarMaxSequenceLen(grammarDef, max.depth, startSymb),
    wrappings = 3,
    suggestions = NULL,
```

```
    optimizer = c("auto", "es", "ga"),
    popSize = 8, newPerGen = "auto", elitism = 2,
    mutationChance = NA,
    iterations = 1000, terminationCost = NA,
    monitorFunc = NULL,
    plapply = lapply, ...)
```

As a great advantage of the tutorial, it can be pointed out that names are self-explanatory. It can be seen that the user has full control of most of the important parameters of the evolutionary process.

To begin with, `popSize` does describe the size of the population, `iterations` sets the number of generations, `max.depth` sets the maximal depth of search, it is important to know that by default it is set to the number of rules in the grammar. To find a full description, one may refer to the documentation [21].

Results are then stored in a created variable and can be retrieved using the code presented in the example. There is no mechanism for auto-saving the achieved result.

8.6. Retrieving evolved results

As was mentioned in the section above, the package had not been equipped with automatic generation of reports. The user is the one who is responsible for designing data saving routine.

The value received as a result of calling the `GrammaticalEvolution(...)` may be assigned to a variable, that can be directly printed into the console, as presented below.

```
ge <- GrammaticalEvolution(...)
print(ge)
```

Such a piece of code prints information on the result, including the expression, fitness value, and the number of generation. However, it is not the only way to get the result. It is possible to access the best expression directly using the following code.

```
print(ge$best$expressions)
```

Its value can be assigned to another variable and reused in following instructions to prepare other presentations or conduct additional calculations.

8.7. Example

The repository does contain an example of an evolving formula for Kepler's law based on a very small set of records. For the reader's convenience, the code is presented below.

```
library("gramEvol")

grammarDef <- CreateGrammar(list(
  expr  = grule(op(expr, expr), func(expr), var),
  func  = grule(sin, cos, log, sqrt),
  op    = grule(`+`, `-`, `*`),=
  var   = grule(distance, distance^n, n),
  n     = gvrule(1:4)
))

planets <- c("Venus", "Earth", "Mars", "Jupiter", "Saturn","Uranus")
distance <- c(0.72, 1.00, 1.52, 5.20, 9.53, 19.10)
period <- c(0.61, 1.00, 1.84, 11.90, 29.40, 83.50)

SymRegCostFunc <- function(expr) {
     result <- eval(expr)

       if (any(is.nan(result)))
           return(Inf)

       return (mean(log(1 + abs(period - result))))
}
ge <- GrammaticalEvolution(grammarDef, SymRegCostFunc,iterations = 50)
print(ge)

best.expression <- ge$best$expression
print(ge$best$expressions)
print(data.frame(distance, period, Kepler = sqrt(distance^3),
GE = eval(best.expression)))
```

Running the given code goes smoothly and results in printing the resulting data in the console. The result is astonishing, as it predicts exactly the real formula.

```
  distance period     Kepler         GE
1     0.72   0.61  0.6109403  0.6109403
2     1.00   1.00  1.0000000  1.0000000
3     1.52   1.84  1.8739819  1.8739819
4     5.20  11.90 11.8578244 11.8578244
5     9.53  29.40 29.4197753 29.4197753
6    19.10  83.50 83.4737743 83.4737743
```

Results are also returned in a surprisingly short time in comparison to other tested tools. It seems to be the best of the predefined examples among all the presented tools, achieving the best time and accuracy. On the other hand, this example does contain a grammar that provides

more knowledge than just a general description of the shape of every mathematical equation, meaning that more expert knowledge is introduced into the model than in cases presented by other tools. Nevertheless, it is a well-prepared example showing the potential of the tool.

Additionally, the execution was stable and did not require any additional adjustments or fixes.

Tutorials provide also information that it is possible to equip the program with a verbose output. To do so, a monitoring function needs to be defined and provided to the final call. The code below presents the mentioned modification.

```
customMonitorFunc <- function(results){
    cat("------------------\n")
    print(results)
}
ge <- GrammaticalEvolution(grammarDef, SymRegCostFunc, iterations = 50,
    ↪ monitorFunc = customMonitorFunc)
print(ge)
```

It causes the information about each generation, to be printed out in the console window. Such behavior may be useful when conducting experiments with a larger number of generations, to examine if the process is following the desired path.

8.8. General remarks

The use of the tool is very straightforward and intuitive. It comes with well-written tutorials describing foundations and basic functionalities. Performing the experiment does not require much work to be done. The final user is only requested to provide really necessary information describing the experiment without the boilerplate code.

Documentation is detailed enough to provide all necessary information to deliver answers to all integration-related questions, not forcing user to do the reverse engineering of the tool.

Resources do contain the theory supporting the description of the tool and ensuring the right understanding of used terms.

Prepared programs are executed stably, achieving also astonishing time consumption. Additionally, modifications are made easy, allowing users to define their own fitness function, without a need to know the exact mechanics of the package.

As another advantage, the simplicity of the installation process could be pointed out. It does require a single command that is clearly described and provided in the installation instructions.

The tool's quality is proven by a number of projects that do use it for real-life applications. Additionally, it ensures the presence of easily available implementation examples.

On the other hand, chosen language, is not currently the leading one in terms of popularity, as shown in the survey by Stackoverflow.com [48]. The presented data clearly show that Python does more than 10 times better in terms of popularity than R. However, despite the low overall popularity, it is one of most popular languages for statistical calculations, used by field experts [50]. It is claimed by some resources [31] that it does pose issues for new users, coming with "steep at beginning learning curve" but it does pay off in the future use. Additionally, currently the popularity of R is rising [51] and [50].

To conclude, one may say that gramEvol is a very mature tool that comes with a production-ready set of functionalities and user-friendly documentation and guides. Usage could pose some issues for new users who are not yet familiar with R language, but provided examples and documentations allow even non R users to grasp idea about usage of the tool.

9. Other available tools

This section does present a brief overview of other available tools, with some comments regarding their quality.

9.1. GRAPE

Tool, published in 2022 in an article entitled "GRAPE: Grammatical Algorithms in Python for Evolution" [11]. It is developed on top of the DEAP package [26] which is a Python framework for evolutionary computations.

The source code of the tool is available on GitHub https://github.com/bdsul/grape. The tool seems to be still maintained, having the last commit done 3 weeks ago. There are also 7 forks of its repository, proving that it has some community working around it.

The repository does contain also examples of usage, however, a quick examination shows that these examples are unfortunately quite complicated and lengthy, therefore time-consuming to fully analyze.

Additionally, there is no extensive documentation about the tool easily available, making it hard to get started using the tool.

To conclude, GRAPE is worth attention in the coming years. It is written in one of the leading languages — Python, it uses a framework developed for Genetic Programming that provides clear documentation and seems to be a mature and maintained library used by a huge community. Additionally, GRAPE is said to be comparable with PonyGE2 [11].

9.2. GELab

GELab [30] is a grammatical Evolution tool designed for Matlab. The paper was published in 2021, however, other sources claim that the tool itself has been available since 2018. Its sources can be found in GitHub repository https://github.com/adilraja/GELAB.

The mentioned publication does contain an extensive explanation of the tool, accompanied by theory standing behind Grammatical Evolution.

The author claims that originally it was designed as a tool for Java, but then transformed into a Matlab tool.

Activity on GitHub shows that the tool is probably no longer maintained, not having any commits for the last 4 years. Additionally, there are not many available resources that describe the tool, other than the mentioned publication.

To conclude the tool is quite modern however lacks developer support, but still, it is worth of attention of Matlab users.

9.3. PonyGE

Predecessor of PonyGE2. Tool intended to be kept simple and contained in a single file. Today it is available via Google Code Archive https://code.google.com/archive/p/ponyge/. It comes with 3 wiki pages that describe the installation process and one use case.

In a README file, the author claims compatibility with Python 2 and 3. Additionally, the file contains a brief description of tool usage.

Sources do contain comments suggesting that there are some improvements needed. Additionally, the tool is no longer maintained and there are no easily available sources describing it. Wikipedia claims the tool was first published in 2010 [64].

The tool is a simple example of Grammatical Evolution implementation, good for reverse engineering such a tool, but it is not the best choice for large-size projects.

9.4. AGE

AGE is yet another Grammatical Evolution implementation. This one was prepared in C and Lua, however, the current version of the author's website claims switching C to C++.

Documentation for this tool is contained mainly in the author's bachelor thesis [42] and newer features are described in an additional PDF file. Moreover, the author's master's thesis [43] also references the project. All official sources are available via the author's website http://nohejl.name/age.

A quick look at the sources shows that the library could be said to be complex. However, the mentioned documentation contains a simple description of the way the user should proceed.

AGE is an interesting example of GE software, but it lacks community support to become a popular choice among the users.

Additionally the tool is lacking updates. The last described one is dated to have been uploaded on 23 Novermber 2011. No further activity around the tool is described on author's website.

9.5. GenClass

GenClass [4] is a simple tool written in C++, that's goal is to create simple data classification scripts, using Grammatical Evolution. Its source is freely distributed via GitHub repository https://github.com/itsoulos/GenClass.

The project repository does have the wiki containing all the necessary information to begin preparing the own experiments. The repository also contains examples of use and example output accompanied by a brief description.

The tool received the last update in September 2021. There are a few online sources that mention the tool, mostly directly connected with the GenClass. It means that the tool is probably no longer maintained and does not have a community that supports it.

GenClass is an interesting tool, that is worth remembering. It could pose some issues due to lack of maintenance, but it may be useful in data classification problems. There is documentation covering enough details to get familiar with basic use cases.

10. Applications described from theoretical point of view

This chapter is going to present a few of the theoretical fields that may benefit from using the GE approach of searching for the solution, to familiarize the reader with the potential of the GE.

10.1. Symbolic regression

Symbolic regression [5], [68] could be pointed out as an obvious example of Grammatical Evolution's application in theoretical research.

Symbolic regression is described as constructing a mathematical expression that fits best the provided dataset.

A grammatical evolution model can be prepared to realize that task, as it was shown by some examples provided with GE tools. It does require specifying a correct grammar, that would describe the allowed expressions, and a fitness calculation method that would rate the received expression.

The efficiency of using GE in this application was clearly visible while evolving the Kepler's Law from a little dataset by use of gramEvol.

10.2. Extracting rules for a classifier from delivered dataset

This section describes use of GE to produce the best rule set that would accomplish the classification task.

One of the described tools is designed especially to accomplish this single task [4] — GenClass (https://github.com/itsoulos/GenClass). It provides the functionality of creating C++ classifier as a result of operations on the dataset.

10.3. Generating the architecture of neural networks

Right architecture is crucial for neural network performance and is often a subject of research. There are many established guidelines on defining models by hand, but GE may help to automatize the whole procedure.

Defining a correct grammar that would describe the potential structure of the neural network and the fitness function that takes into consideration time efficiency and accuracy metrics would allow GE to automatically search for the best possible neural network architecture.

There are available sources presenting such an approach [10], [37].

10.4. Creating a syntactically valid program

GE tools are often prepared mostly for the creation of code that gets executed and evaluated. It allows huge flexibility in terms of applications. The user can provide input data in literary any shape and generate a fully functional computer program that performs a given task identified by the fitness function.

10.5. Conclusion on theoretical applications

GE provides enormous flexibility in terms of its applications, as most of the problems would be able to be expressed as the GE models. In this chapter, there were presented a few examples to prove that idea.

11. Real life applications

This chapter is meant to describe a few cases where GE models were applied to a specific case.

11.1. Discovery of relations between data and retrieving the original formula

There are known cases of discovering the relation between data, without prior knowledge or theoretical foundations, based purely on empirical data. This is very similar to the theoretical concept of symbolic regression.

As an example of a formula that was achieved based only on the data was the discovery of Ohm's Law [66].

Today, thanks to the wide availability of computing power, algorithms like GE could reduce the time and effort needed for such discoveries and further optimize results. The example presented by documentation of gramEvol seems to prove also the robustness of this approach even if the size of the dataset is limited.

11.2. Solving models describing the placement of facilities

The problem of finding the best location of facilities can be often reduced to multi-objective optimization that can be then subjected to GE approach.

There are examples of usage of presented tools in such cases [73].

11.3. Detecting cybersecurity threats

One of the publications suggests that the ability of GE to evolve a program in any programming language can be utilized to create software capable of detecting cyberattacks on IoT devices [3].

The paper shows a study on the case, development of solutions and finally proves the ability of Ge to evolve a program capable of identifying the attack that it was not trained on.

12. Estimation of the function counting primes using PonyGE2

Function $\pi(x)$ is defined as the number of prime numbers less or equal to x [55] [62]. Due to it nature, there is currently no way to precisely calculate its value.

Grammatical Evolution may be used then to evolve a formula that will try to approximate the function π.

12.1. Used grammar

For the purpose of evolving formulae, approximating $\pi(x)$ a grammar describing mathematical expressions is needed. For the purpose of demonstration, grammar that comes with PonyGE2 will be used with slight modification. For demonstration of this application, PonyGE2 will be used.

```
<e>  ::=  <e>+<e>|
          <e>-<e>|
          <e>*<e>|
          pdiv(<e>,<e>)|
          psqrt(<e>)|
          np.sin(<e>)|
          np.tanh(<e>)|
          np.exp(<e>)|
          plog(<e>)|
          x[:, 0]|
          <c><c>.<c><c>
<c>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

12.2. Data set

The following features of PonyGE2, a mechanism of data set loaded from tabulation separated text file, will be used. To generate the dataset, a c++ program is provided. The program has a hard-coded array of prime numbers that it can refer during operation to optimize time consumption.

```
#include <iostream>

int list_of_prime[] = {/* Here goes hard coded primes in ascending order */ };

const int num_of_primes = sizeof(list_of_prime)/sizeof(int);
const int max_prime = list_of_prime[num_of_primes-1];

int main(){

  std::cout<<"x0\tresponse"<<std::endl;

  for(int counter = 0, it=0, i = 0; i<=max_prime && it < num_of_primes; i ++){
    int next_prime = list_of_prime[it];
    if(i>=next_prime){
      counter++;
      it++;
    }
    std::cout<<i<<"\t"<<counter<<std::endl;
  }

  return 0;
}
```

Prime numbers may be taken from one of the internet sources. Generated data set quality is dependent on number of prime numbers. Data generated by the program should be stored in a text file that will be later refereed in the parameters file.

12.3. PonyGE2 parameters

Below, all parameters of the experiment are listed. Most of them are taken directly from the regression example provided in the repository of PonyGE2.

```
CACHE:                  True
```

```
CODON_SIZE:              100000
CROSSOVER:               variable_onepoint
CROSSOVER_PROBABILITY:   0.75
DATASET_TRAIN:           prime_counting/Train.txt
DATASET_TEST:            prime_counting/Test.txt
DEBUG:                   False
ERROR_METRIC:            mse
GENERATIONS:             400
MAX_GENOME_LENGTH:       500
GRAMMAR_FILE:            supervised_learning/regression-prime.bnf
INITIALISATION:          PI_grow
INVALID_SELECTION:       False
MAX_INIT_TREE_DEPTH:     10
MAX_TREE_DEPTH:          17
MUTATION:                int_flip_per_codon
POPULATION_SIZE:         500
FITNESS_FUNCTION:        supervised_learning.regression
REPLACEMENT:             generational
SELECTION:               tournament
TOURNAMENT_SIZE:         2
VERBOSE:                 False
```

12.4. Results

During execution of the experiment, multiple functions that achieved shape similar to $\pi(x)$ were evolved. An issue of formulae evolved using this method is their complexity. Formulas have lots of nested operations. one of the achieved results is presented below:

```
 2 sqrt(x) + x/(tanh((x + sqrt(tanh(78.45) sin(51.98)) x - log(sqrt
↪ (84.76) + 47.5))/exp(log(log(69.92) + 7.51)) x) + sqrt(38.86) + log(
↪ log(x - log(sin(x) + 15.6)  tanh(tanh(sqrt(x))) tanh(sin(log(x)) +
↪ 69.37) x)))
```

As it can be seen in the figure 12.1 generated solution does have a similar shape to the desired function, however unfortunately the approximation is not perfect and there are differences between predicted values and the actual. Therefor further executions of experiment with higher number of generations were performed and following results were obtained:

```
 x/(ln(x/(ln(ln(92.89-sin(x)+x*x+sin(x)-64.03*sqrt(x)*ln(exp(sin(89.77))
↪ ))*sqrt(sin(19.94))))))
```

The results as for a simple experiment are rather satisfying, but the differences are still present. Table 12.1 shows comparison of function values for two values of x showing that the

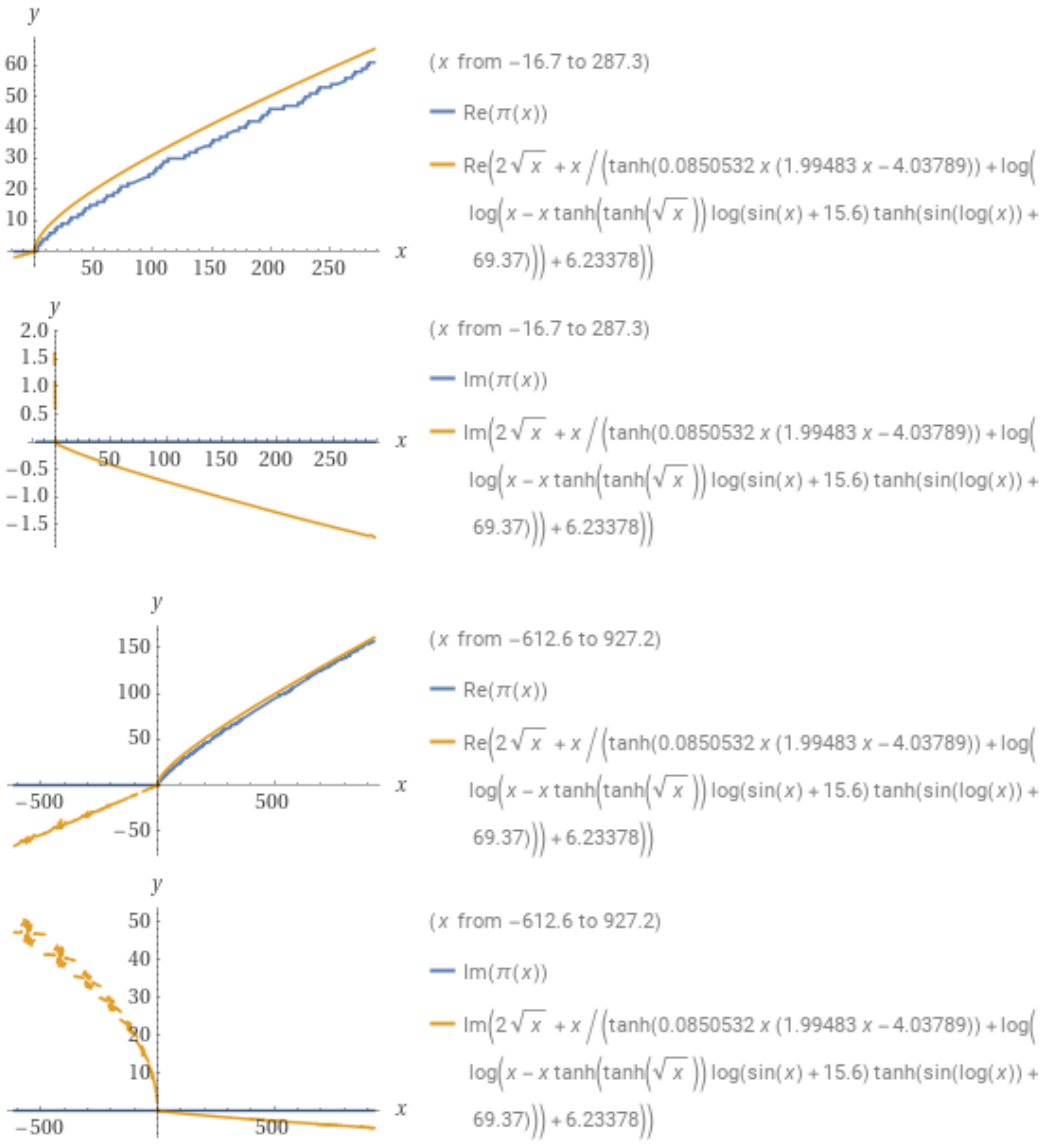

Figure 12.1. First generated solution in comparison to actual function prepared using https://www.wolframalpha.com/. In the second pair of plots blue and orange lines are almost indistinguishible. To examine closely the plot one may use url `https://www.wolframalpha.com/input?i=pi%28x%29+vs+2sqrt%28x%29%2Bx%2F%28tanh%28%28x%2Bsqrt%28tanh%2878.45%29sin%2851.98%29%29x-log%28sqrt%2884.76%29%2B47.5%29%29%2Fexp%28log%28log%2869.92%29%2B7.51%29%29x%29%2Bsqrt%2838.86%29%2Blog%28log%28x-log%28sin%28x%29%2B15.6%29tanh%28tanh%28sqrt%28x%29%29%29tanh%28sin%28log%28x%29%29%2B69.37%29x%29%29%29`.

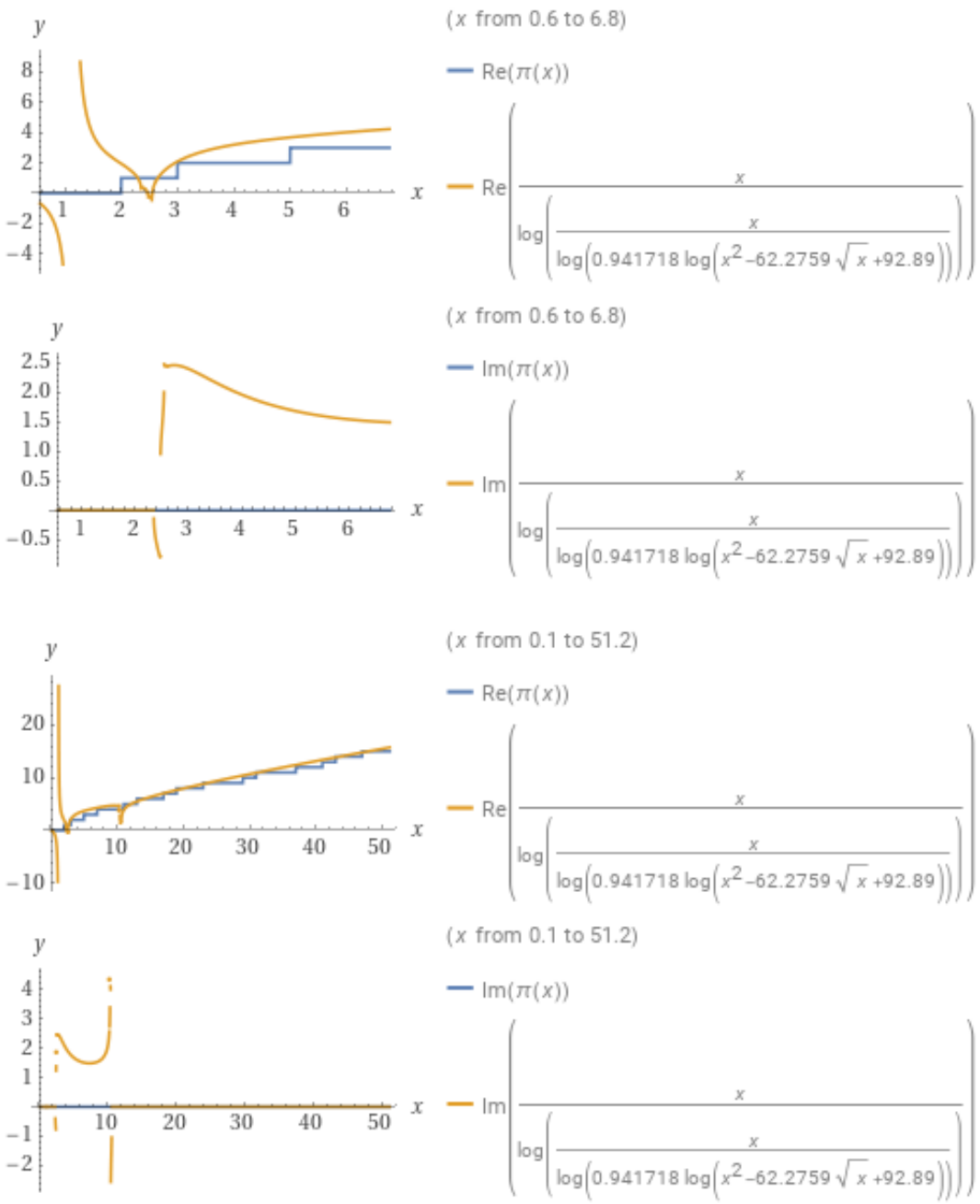

Figure 12.2. Second generated solution in comparison to actual function prepared using https://www.wolframalpha.com/. Closer examination is possible by use of this url: https://www.wolframalpha.com/input?i=pi%28x%29+vs+x%2F%28ln%28x%2F%28ln%28ln%2892.89-sin%28x%29%2Bx*x%2Bsin%28x%29-64.03*sqrt%28x%29*ln%28exp%28sin%2889.77%29%29%29%29*sqrt%28sin%2819.94%29%29%29%29%29%29.

x	$f_2(x)$	$\pi(x)$
100	26.0574	25
1400	222.801	222

Table 12.1. Values of $\pi(x)$ in comparison to values of second evolved function

evolved function is close but does not describe the function exactly. To gain a visual overview, one may refer to figure 12.2.

Results of the experiment are even more promising when it comes to execution time. Evolving second formula took only 143,7s. Judging by achieved effect in reference to consumed resources, it may be worth to further increase precision by allowing exploration of a larger search space. This may result in creating even better approximation of the given function. Additionally, one may provide a larger dataset to maintain the shape of the function throughout a wider range of values.

13. Estimation of the standard acceleration gravity value from simple pendulum measurements

13.1. Introduction and data set

By transforming physics formulas, one may derive the following formula describing gravitational acceleration g in terms of values that can be measured using simple pendulum.

$$g = \frac{4\pi^2 l}{T^2}$$

The goal of this experiment is evolving such formula with small data set containing value of l being the length of pendulum and its period T accompanied by value of g taken from constants table.

The dataset had been prepared using measurements performed on a simple pendulum moved from the equilibrium point by a small angle, allowing $sin(\alpha)$ approximation to make the above formula valid. Time measurement were manually triggered. Each measurement was taken using 10 full movements to limit possible error.

Prepared data set is presented in table 13.1. As data set does not provide enough data — it does contain only one pendulum length — it may be required to enrich it and retry the experiment

13.2. Used grammar

A similar grammar to the one used in the experiment presented in chapter 12, that was adopted from PonyGE2 example code, was used here, with slight modification, allowing constant pi.

```
<e>  ::=  <e>+<e>|
      <e>-<e>|
      <e>*<e>|
      pdiv(<e>,<e>)|
```

$T[s]$	$l[m]$	$g[ms^{-2}]$
1.447	0.522	9.80665
1.459	0.522	9.80665
1.443	0.522	9.80665
1.447	0.522	9.80665
1.441	0.522	9.80665
1.444	0.522	9.80665
1.434	0.522	9.80665
1.453	0.522	9.80665

Table 13.1. Data set prepared based on real measurements and real value of g taken from https://physics.nist.gov/cgi-bin/cuu/Value?gn

```
      psqrt(<e>)|
      np.sin(<e>)|
      np.tanh(<e>)|
      np.exp(<e>)|
      plog(<e>)|
      x[:, 0]|x[:, 1]|
      <c><c>.<c><c> |
      np.pi
<c>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Then let x_0 be the period T given in seconds and let x_1 be the length of pendulum l expressed as meters.

13.3. Used parameters

Below, parameters used for the evolution are presented. For the purpose of the test population of size, 500 is going to be evolved for 400 generations.

```
CACHE:                  True
CODON_SIZE:             100000
CROSSOVER:              variable_onepoint
CROSSOVER_PROBABILITY:  0.75
DATASET_TRAIN:          pendulum/Train.txt
DATASET_TEST:           pendulum/Test.txt
```

```
DEBUG:                  False
ERROR_METRIC:           mse
GENERATIONS:            400
MAX_GENOME_LENGTH:      500
GRAMMAR_FILE:           supervised_learning/regression-pendulum.bnf
INITIALISATION:         PI_grow
INVALID_SELECTION:      False
MAX_INIT_TREE_DEPTH:    10
MAX_TREE_DEPTH:         17
MUTATION:               int_flip_per_codon
POPULATION_SIZE:        500
FITNESS_FUNCTION:       supervised_learning.regression
REPLACEMENT:            generational
SELECTION:              tournament
TOURNAMENT_SIZE:        2
VERBOSE:                False
```

13.4. Achieved results

After running the program, the formula provided below had been returned. Execution tool 236 seconds. The first remark could be that the formula is much more complicated than it is desired to be.

```
psqrt(97.17-np.tanh(plog(71.76-np.sin(np.tanh(00.87))+plog(x[:, 0])*np.tanh
    ↪ (psqrt(np.pi68.67*np.pi)+x[:, 1])*pdiv(np.pi,87.89-plog(40.97)+np.sin
    ↪ (40.97)+psqrt(np.sin(np.pi))+np.exp(np.pi)))))
```

To be able to quickly transform provided results into more readable form wolframalpha.com will be used. However, to use it, it is required to transform the formula. There are functions like `pdiv` or `plog` that are defined inside PonyGE2 `src/utilities/fitness/math_functions.py`. Additionally, functions from NumPy should be described in a way the WolframAlpha tool is able to understand. The resulting prompt is presented below.

```
sqrt(97.17-tanh(log(71.76-sin(tanh(00.87))+log(T)*tanh(sqrt(pi-68.67*pi)+l
    ↪ ])*
(pi/(87.89-log(40.97)+sin(40.97)+sqrt(sin(pi))+exp(pi))))))
```

Having that transformed with WolframAlpha, one receives the formula presented below. It appears that it is still very complicated and does not resemble the desired one.

$$\sqrt{97.17 - \frac{-1 + (71.1147 + (0.0293089i)\log(T)\tan((14.5805 + 0i) - il))^2}{1 + (71.1147 + (0.0293089i)\log(T)\tan((14.5805 + 0i) - il))^2}}$$

It appears that as anticipated data set does not contain enough data to produce an accurate result. This may be the case, as really there is just a single value l, g and T with noise To solve the issue, artificial measurements will be introduced.

13.5. Enriched data set

To tackle the issue of too simple data set, additional records will be reproduced artificially. Data should be supplemented with records that will emphases the relationship between those values, so different pendulums should be used.

Calculations will be performed with predefined lengths, therefor formula for period T should be derived.

$$T = \sqrt{\frac{4\pi^2 l}{g}}$$

Data prepared using this method is presented in table 13.2. These records describe the relation better but still there is possible issue — one may point out that in this scenario, the best formula describing g can be stated as follows.

$$g(l, T) = 9.80665$$

Thanks to the nature of GE this vulnerability of data set may not get exploited allowing valuable result to be obtained.

As a quick conclusion, it may be said that the dataset should contain a representative sample of the whole data. Providing only a subset may have a negative effect on the produced results, as not all characteristic features of the data may be included in the subset of data. As an easy theoretical example, symbolic regression may be used. Let us assume that one is willing to achieve a polynomial of 5th degree specifying only 2 points and grammar that describes any mathematical expression, then there are infinitely many possible functions that come through these two points and there is no defined way to determine the right one.

13.6. Second attempt

Second iteration had been executed, however evolution on enriched data set does not provide satisfactory results. After 365 seconds best genome was returned with surprisingly good

$T[s]$	$l[m]$	$g[ms^{-2}]$
1.447	0.522	9.80665
1.459	0.522	9.80665
1.443	0.522	9.80665
1.447	0.522	9.80665
1.441	0.522	9.80665
1.444	0.522	9.80665
1.434	0.522	9.80665
1.453	0.522	9.80665
0.25	1.003204646	9.80665
0.75	1.737601418	9.80665
1	2.006409293	9.80665
1.25	2.243233784	9.80665
1.5	2.457339491	9.80665
1.75	2.654230008	9.80665
2	2.837491233	9.80665
2.25	3.009613939	9.80665
2.5	3.172411642	9.80665

Table 13.2. Data set from table 13.1 with additional artificially computed results to improve generated results

fitness value being equal to $2.5400306357448104 * 10^{-22}$, however it lacks simplicity therefore regardless of high accuracy that was obtained it cannot be considered as a good solution. The phenotype of solution is presented below:

```
Phenotype: psqrt(plog(np.exp(96.17)+np.exp(88.27)+pdiv(85.02*np.exp(x[:,
    ↪ 0])-69.29,np.pi)+np.exp(x[:, 0])-pdiv(pdiv(pdiv(x[:, 1],14.87),60.10)
    ↪ ,plog(03.16)*psqrt(np.exp(42.30))+46.14)+np.exp(80.42)*93.79-plog(
    ↪ plog(72.73)+88.29)*pdiv(90.82*np.tanh(x[:, 1])-np.tanh(34.71),np.exp
    ↪ (50.43))*plog(03.68)*psqrt(np.exp(plog(10.39)*39.89))))
```

One conclusion from this experiment is that one may get a very good result that will describe the provided data set and encapsulate all relations present in the data set, but still have it in a very complicated, hardly explainable form. As literature suggests the reasonable solution is to adjust fitness function to incorporate the simplicity measure to value simple solutions more, therefor standard PonyGE2 regression fitness class is not able to optimize search taking simplicity into consideration. Having said that, there are two reasonable possible solutions:

- Reduce the search space constraints — allow more entities in the population or allow more generations. That would allow more evolutionary processes to be accomplished, and therefore the result may be simplified.
- Modify the fitness function to incorporate the simplicity measure. This however is considered as a challenging problem, as appropriate definition of simplicity in reference to the solution should be developed. Some sources [2] suggest that length could be such a measure of simplicity, however that may not always be the case. For example having variable `x[:, 0]` as in this experiment, according to length simplicity measure, using this variable is equally simple as introducing `psqrt(...)` as their lengths are equal, and it is obvious that introducing a variable decreases simplicity far less than using the square root. Taking into consideration all these facts, there is no single universal simplicity measure.

13.7. Conclusion

Presented solutions may further increase the accuracy of solution and achieving feature of solution explainability. The conclusion from this experiment is that first of all the data set have to describe all features of the population not only a subset, otherwise not explicitly emphasized features may not get included in the result. Secondly it appears that even if the result describes well the data set, it may not be simple enough to satisfy explainability condition, therefor fitness should take the simplicity into consideration, with respect to correct, according to problem specification, definition of simplicity.

14. Regression over data set with introduced noise using PonyGE2

This chapter presents an artificial use case of regression over a data set that is derived from a known function formula with introduced noise. The goal of this experiment is to show if it would be possible to retrieve the formula resembling the original one.

14.1. Introduction and data set

To perform the experiment, a quadratic function $f(x) = 6x^2 - 3x + 19$ will be used. The function is fairly simple and without noise should not pose an issue for the algorithm — that is going to be checked and proved during the test Trial. Each record from the data set is of shape $\{x; f(x)\}$. After test trial, formula will be slightly changed to introduce the disturbance modeling real life example: $f_1(s) = 6x^2 - 3x + 19 + \texttt{random\_noise}$.

14.2. Script used for data set generation and data set details

The program presented below is used to generate the data set and create the random disturbance by the creation of noise according to normal distribution. The mean of the distribution is set to 0 and the standard deviation to 400 to make the disturbance visible on the plot.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl

# Creating np array for x
x = np.arange(-100,101,1)
# Computing accurate function values
y = 6*x*x - 3*x + 19

```

```
11
12 # Noise specification
13 mean = 0
14 standard_deviation = 400
15
16 # Generation of the noise
17 noise = np.random.normal(mean,standard_deviation, len(x))
18
19 # Introducing the noise
20 realistic_y = y + noise
21
22 WIDTH_SIZE = 10
23 HEIGHT_SIZE = 5
24
25 fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(WIDTH_SIZE,HEIGHT_SIZE))  #
    ↪ Create figure for plots and plots.
26 ax1.plot(x, y, '.')            # Plot accurate data on the axes1.
27 ax2.plot(x, realistic_y, '.')  # Plot noisy data on the axes2.
28
29 ax1.set_xlabel('x')
30 ax1.set_ylabel('y')
31 ax1.set_title('Pure data')
32
33 ax2.set_xlabel('x')
34 ax2.set_ylabel('y')
35 ax2.set_title('Data with noise')
36
37 plt.show()
38
39 test_Trial = zip(x,y)
40 normal_Trial = zip(x,realistic_y)
41
42 print("### Pure data ###")
43
44 for x,y in test_Trial:
45     print(x,y,sep='\t',end='\n')
46
47 print("### Data with noise ###")
48
49 for x,y in normal_Trial:
50     print(x,y,sep='\t',end='\n')
```

The presented code was inspired by online sources [58], [13], [14].

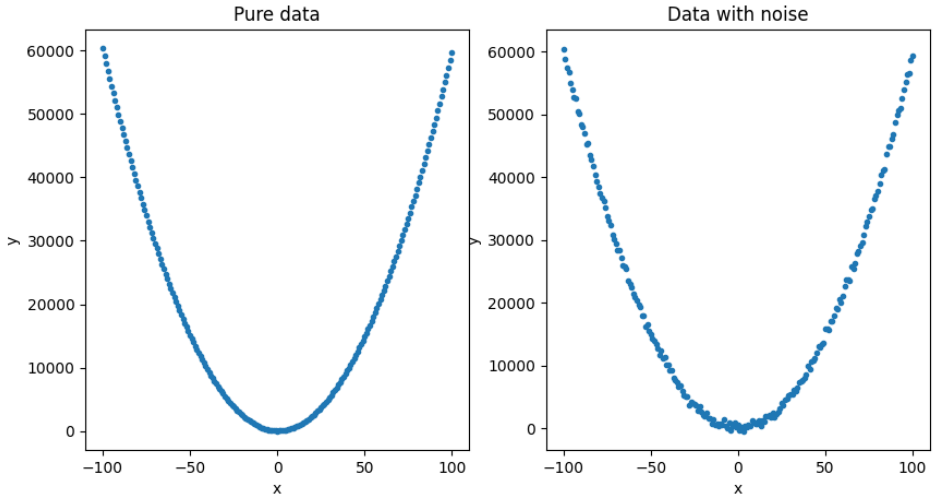

Figure 14.1. Plot presenting the data set generated using function $f(x) = 6x^2 - 3x + 19$ with introduced noise generated according to normal distribution.

14.3. Test trial

After performing the test trial of the experiment, it went out that the evolved formula was close enough to the desired formula. Evolution took 30 seconds. It was performed using parameters specified below.

```
CACHE:                  True
CODON_SIZE:             100000
CROSSOVER:              variable_onepoint
CROSSOVER_PROBABILITY:  0.75
DATASET_TRAIN:          regr_noise_pure/Train.txt
DATASET_TEST:           regr_noise_pure/Test.txt
DEBUG:                  False
ERROR_METRIC:           mse
GENERATIONS:            50
MAX_GENOME_LENGTH:      500
GRAMMAR_FILE:           supervised_learning/regression-noise.bnf
INITIALISATION:         PI_grow
INVALID_SELECTION:      False
MAX_INIT_TREE_DEPTH:    10
MAX_TREE_DEPTH:         17
MUTATION:               int_flip_per_codon
POPULATION_SIZE:        500
FITNESS_FUNCTION:       supervised_learning.regression
REPLACEMENT:            generational
SELECTION:              tournament
```

```
TOURNAMENT_SIZE:        2
VERBOSE:                False


```

The result of the Trial is presented below in a form of phenotype generated by PonyGE2.

```
    x[:, 0]*psqrt(36.03)*x[:, 0]
```

According to agreed convention, the result can be interpreted as $x * \sqrt{36.03} * x$ that could be simplified to $x^2 * \sqrt{36.03}$. Comparison with desired formula $f(x) = 6x^2 - 3x + 19$ had been prepared using WolframAlpha tool and presented on figure 14.2. Taking into consideration the limitations on search, the result is very good and satisfying. It proves the concept.

14.4. Test on data with noise

Having completed the test trial, proving that GE is capable of retrieving the formula in a data set that does not contain disturbances, the data set with artificial noise is going to be explored. To perform this test, the same parameters were used, the only difference was in values of the second column in the data set.

The raw result of the test is presented below. Total time consumed by evolution is 40 seconds.

```
psqrt(x[:, 0])+x[:, 0]*x[:, 0]*psqrt(81.56*np.sin(psqrt(45.29))+np.
sin(np.sin(psqrt(45.29))+np.tanh(pdiv(psqrt(x[:, 0]),np.exp(x[:, 0])))))+
    ↪ plog(ps
qrt(65.78)+plog(x[:, 0])-x[:, 0])
```

To enhance readability this could be further transformed into the following form, that can be passed to wolframalpha.com to achieve a simpler equivalent formula that is easier to compare with original function

```
sqrt(x)+x*x*sqrt(81.56*sin(sqrt(45.29))+sin(sin(sqrt(45.29))+tanh((sqrt(x)/
    ↪ exp(x))))+log(sqrt(65.78)+log(x)-x)
```

The simplified result returned by WolframAlpha is presented below. It appears that for longer evolutionary process it would tend to transform into quadratic formula, but in this particular case the result is over-complicated, and it is not possible to observe obvious similarities to the original formula $f(x) = 6x^2 - 3x + 19$.

$$x^2\sqrt{\sin\tanh e^{-x}\sqrt{x} + 0.431901 + 35.2258} + \sqrt{x} + \log -x + \log x + 8.11049$$

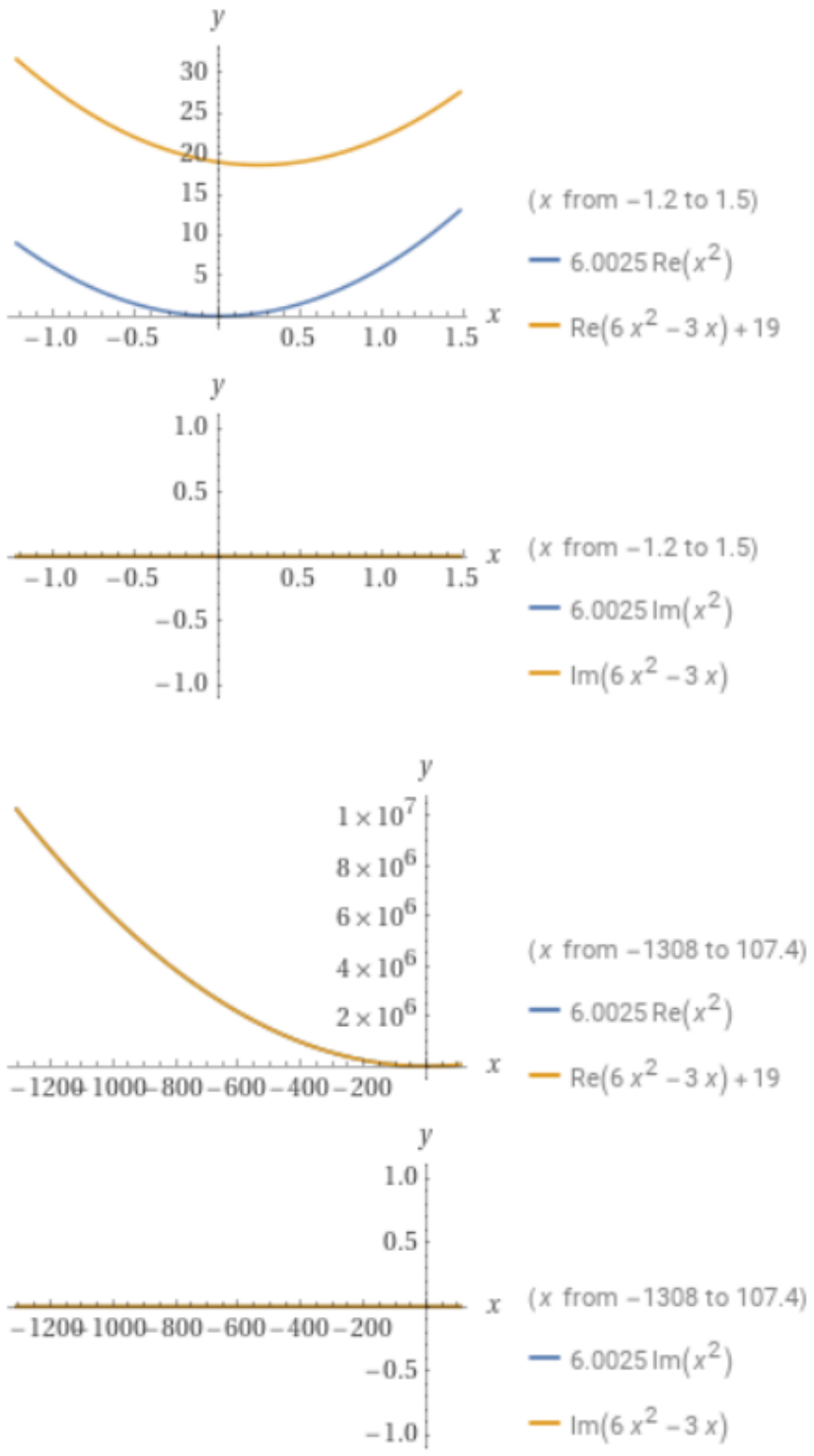

Figure 14.2. Comparison of the desired function $f(x) = 6x^2 - 3x + 19$ and the one evolved during the test trial, $f_t(x) = x^2 * \sqrt{36.03}$ prepared using https://www.wolframalpha.com/. Detailed chart can be obtained by use of link https://www.wolframalpha.com/input?i=x*sqrt%2836.03%29*x+vs+6x2%E2%88%923x%2B19. Blue line in the second pair of chart is almost fully covered by orange, desired, function.

Comparison of the desired formula and the achieved one is presented in figure 14.3. It is surprising how close evolved formulas real part is to the original ones for $x \geq 0$. Judging by this fact, the Trial could be described as a partially succeed, however the result does still lack explainability — the formula is over-complicated and hard to work with.

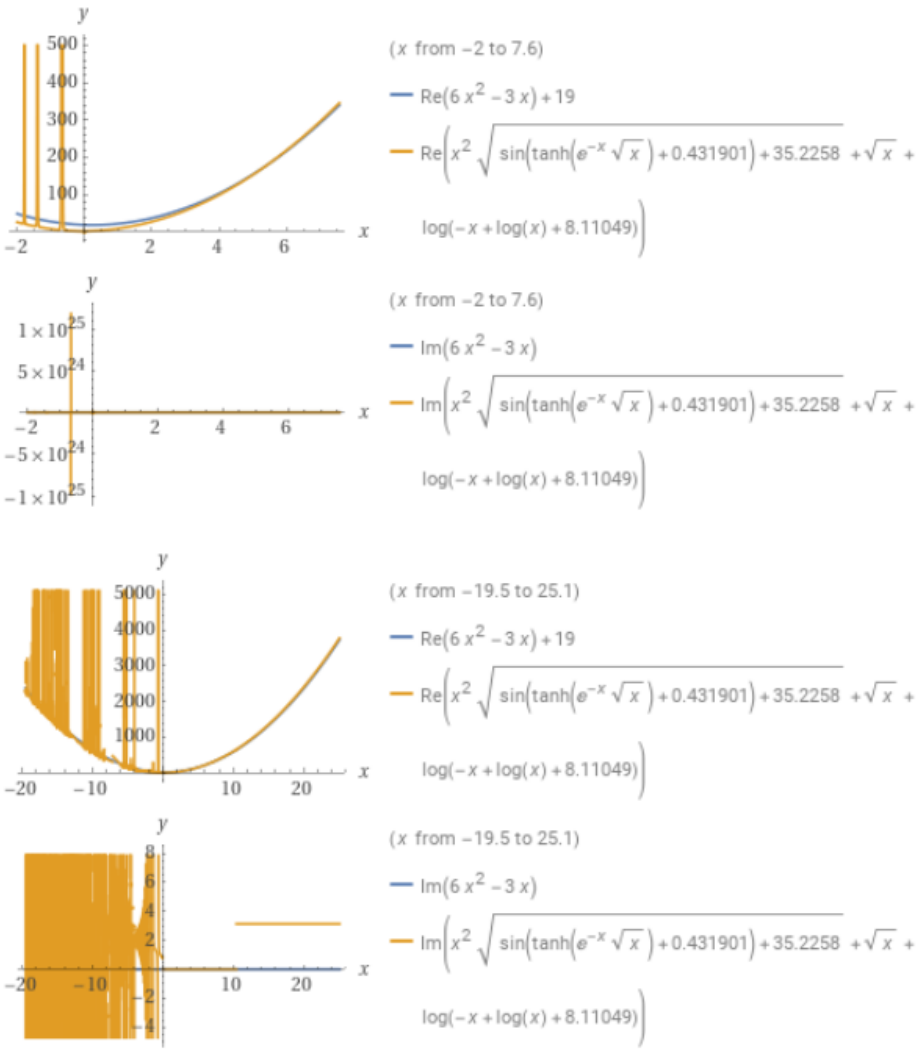

Figure 14.3. Comparison of the desired function $f(x) = 6x^2 - 3x + 19$ and the one evolved during the real Trial $f_t(x) = x^2\sqrt{\sin\tanh e^{-x}\sqrt{x} + 0.431901 + 35.2258} + \sqrt{x} + \log -x + \log x + 8.11049$ prepared using https://www.wolframalpha.com/

Closer examination of the figure 14.3 is possible via a following link `https://www.wolframalpha.com/input?i=sqrt%28x%29%2Bx*x*sqrt%2881.56*sin%28sqrt%2845.29%29%29%2Bsin%28sin%28sqrt%2845.29%29%29%2Btanh%28%28sqrt%28x%29%2Fexp%28x%29%29%29%29%2Blog%28sqrt%2865.78%29%2Blog%28x%29-x%29`.

14.5. Conclusions

The presented experiment has showed that it is hard to eliminate the impact of presence of noise in the input data therefor to achieve best results, data set should be prepared carefully with special attention to preciseness of data, however it is still possible to further enhance the result by introducing assumptions like definition of simplicity and connecting its value to fitness, or limiting number of allowed derivations. These approaches are possible but requires either additional knowledge or many experiments to determine if the assumption is feasible.

15. Evolving formula for function composed of at least one periodic function using PonyGE2

The goal of this task is to check how well does GE approach work with periodic functions incorporated in the data set. To perform the test, two data sets will be used, one with pure data and the second with noise. Then results are going to be compared and discussed.

15.1. Data set

The function that is going to generate the data set is going to be $f(x) = \sin(0.1 * x) * 20 + x - 10$. It does contain the periodic function that is going to introduce some periodic behavior, but it is also accompanied by a simple linear formula to further test abilities of GE. Comparison of pure and noisy data is presented in the figure 15.1.

15.2. Trial with pure data

To perform the experiment, parameters presented below were used. Due to the complexity of the problem, more generations were allowed in order to evolve simpler and better in terms of fitness formula. Despite efforts, the solution that came out was very complicated and hardly explainable. To ensure that the outcome is simple enough, the size of derivation tree was reduced significantly, disallowing huge formulas to be generated by marking them as illegal.

```
CACHE:                  True
CODON_SIZE:             100000
CROSSOVER:              variable_onepoint
CROSSOVER_PROBABILITY:  0.75
DATASET_TRAIN:          pfunc/pure/Train.txt
DATASET_TEST:           pfunc/pure/Test.txt
DEBUG:                  False
ERROR_METRIC:           mse
GENERATIONS:            200
```

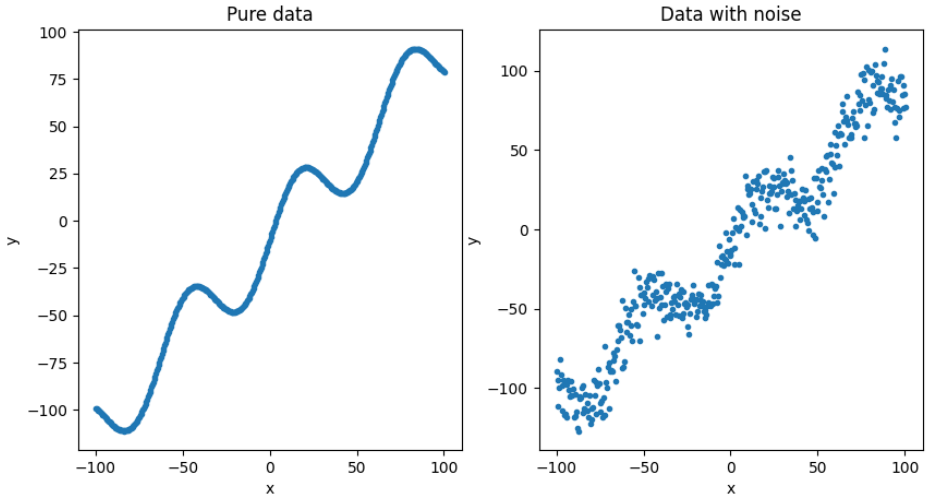

Figure 15.1. Visualization of pure data set generated by function $f(x) = \sin(0.1 * x) * 20 + x - 10$ and $f_n(x) = \sin(0.1 * x) * 20 + x - 10 + \texttt{noise}$ that contains noise generated according to normal distribution with following parameters: mean $\mu = 0$ and standard deviation $\sigma = 10$

```
MAX_GENOME_LENGTH:      500
GRAMMAR_FILE:           supervised_learning/regression-noise.bnf
INITIALISATION:         PI_grow
INVALID_SELECTION:      False
MAX_INIT_TREE_DEPTH:    10
MAX_TREE_DEPTH:         6
MUTATION:               int_flip_per_codon
POPULATION_SIZE:        500
FITNESS_FUNCTION:       supervised_learning.regression
REPLACEMENT:            generational
SELECTION:              tournament
TOURNAMENT_SIZE:        2
VERBOSE:                False
```

Without limiting the size of output, it was possible to achieve the result presented below. At first, it looks promising as it contains the desired trigonometry function and is fairly simple, however comparison to the original function shows a lot of differences. A simplified generated formula is presented below.

$$x - 0.159394(-x \sin 4.37551 - \sqrt{x} - \sin x + x(-\log(-0.0423908x)) + 12.05 + 65.15)$$

After limiting the size of the derivation tree as stated in the parameters above, another attempt was performed, producing the result presented below.

$$x - \sqrt{83.50 - x}$$

Plots in figure 15.2 show clearly that the trial did succeed to some extent, as the created plot of the real part is close enough to the desired formula to call it approximation. It becomes even clearer when comparing the result to the non-periodic part of function - $x - 10$. Visualization is presented in figure 15.3.

To tackle the issue of close to linear approximation, search parameters were adjusted as described below, allowing broader search.

- GENERATIONS: 300
- MAX_TREE_DEPTH: 8
- POPULATION_SIZE: 600

Despite those changes, search seems to be stuck around the quasi linear approximation. Interpretation of the trial's result is presented below.

$$x - \sqrt{\frac{x - 29.99}{\frac{44.52}{x+x}} - 86.83 - 84.19}$$

It is worth to mention that interpretation above is simplified as the output of the program contains predefined safe functions that for example have predefined value for zero division.

15.2.1. Conclusion on the test

It appears that shape introduced by sin function had been treated as impurity in data that should be removed. There are a couple of potential solutions:

- Increase number of generations, allowing more evolutionary processes to happen.
- Increase size of population to achieve higher diversity among population.
- Adjust the fitness function to value simple individuals more.

15.3. Test with noise

In this trial, noisy data is going to be passed to PonyGE2 configured according to parameters file specified below.

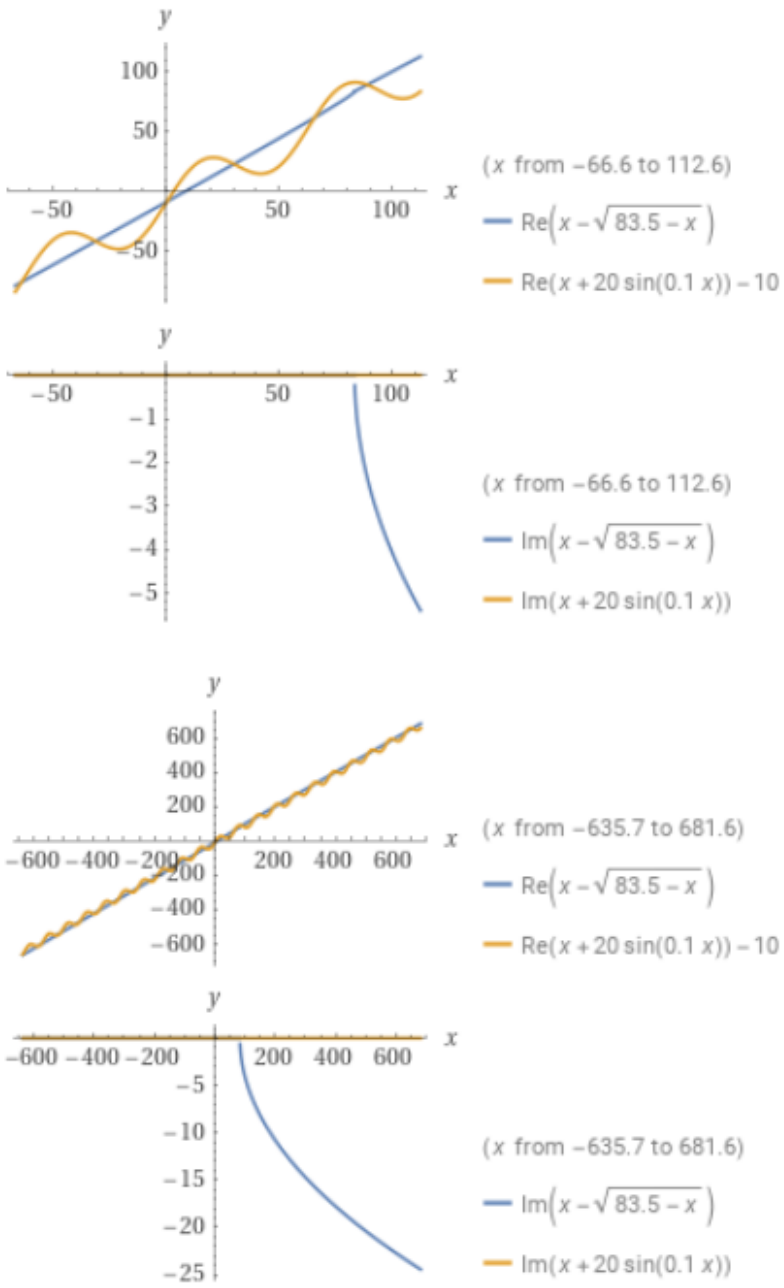

Figure 15.2. Comparison between result generated by GE $x - \sqrt{83.50 - x}$ and the desired function $\sin(0.1 * x) * 20 + x - 10$ prepared using wolframalpha.com. Closer inspection of figure is possible via a link https://www.wolframalpha.com/input?i=x-sqrt%2883.50-x%29+vs+sin%280.1*x%29+*+20+%2B+x+-+10. In the second pair of charts, blue line is almost covered by sine shape of the desired function.

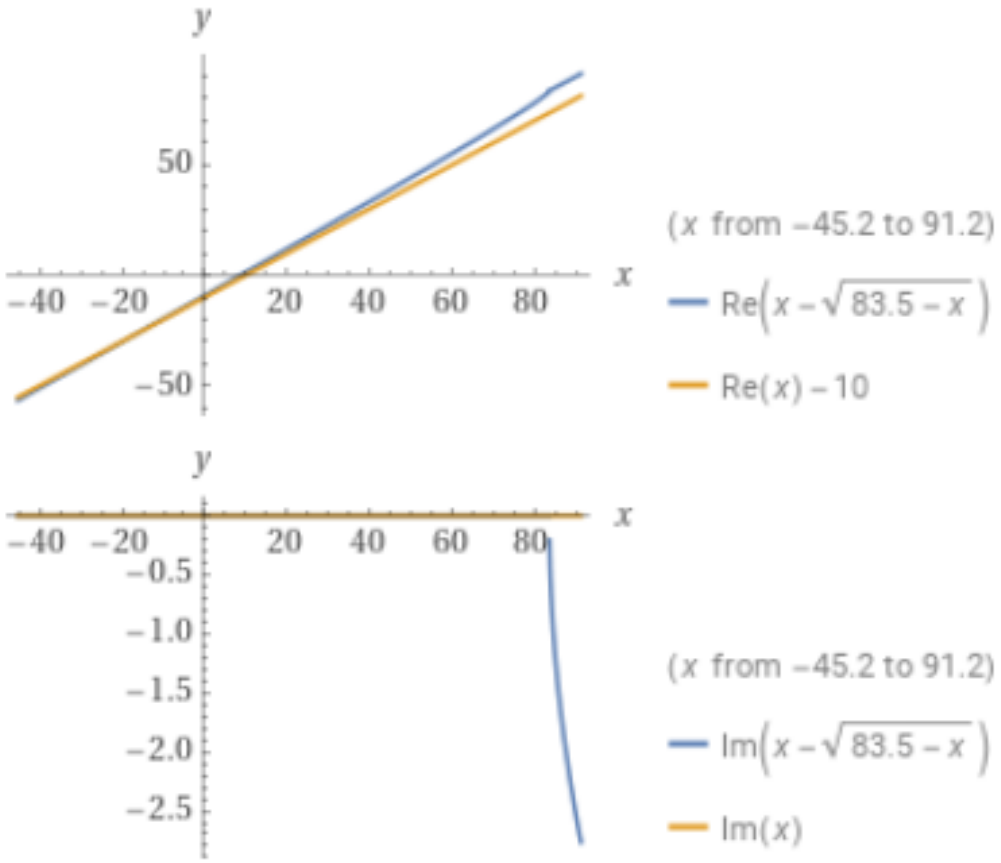

Figure 15.3. Comparison between result generated by GE $x - \sqrt{83.50 - x}$ and part of the desired function $x - 10$ prepared using wolframalpha.com. The plot can be viewed using a following URL https://www.wolframalpha.com/input?i=x-sqrt%2883.50-x%29+vs++x+-+10.

```
CACHE:                  True
CODON_SIZE:             100000
CROSSOVER:              variable_onepoint
CROSSOVER_PROBABILITY:  0.75
DATASET_TRAIN:          pfunc/noise/Train.txt
DATASET_TEST:           pfunc/noise/Test.txt
DEBUG:                  False
ERROR_METRIC:           mse
GENERATIONS:            300
MAX_GENOME_LENGTH:      500
GRAMMAR_FILE:           supervised_learning/regression-noise.bnf
INITIALISATION:         PI_grow
INVALID_SELECTION:      False
MAX_INIT_TREE_DEPTH:    10
MAX_TREE_DEPTH:         8
MUTATION:               int_flip_per_codon
POPULATION_SIZE:        600
FITNESS_FUNCTION:       supervised_learning.regression
REPLACEMENT:            generational
SELECTION:              tournament
TOURNAMENT_SIZE:        2
VERBOSE:                False
```

The returned result is also tending to be close to linear approximation of the function. The result is presented below accompanied by visual interpretation in figure 15.4

```
x-np.exp(np.tanh(10.00))*plog(80.49-x)
```

15.3.1. Conclusion on the step of the experiment

According to presented data, the model tends to oversimplify the function, loosing precision. However, that could be explained by strict limitations on search space, leaving huge room for improvements. Despite the emerged issue, the trial could be said to be partially success, as the output provides approximation of the desired function.

15.4. Test with noisy data and reduced constraints

To check if results could be easily improved by reducing constraints on population size and generations, an additional trial was conducted, having the number of generations set to 500.

Execution took 110 seconds and returned output similar to the one from the previous trial. The result is presented below along with visualization in figure 15.5.

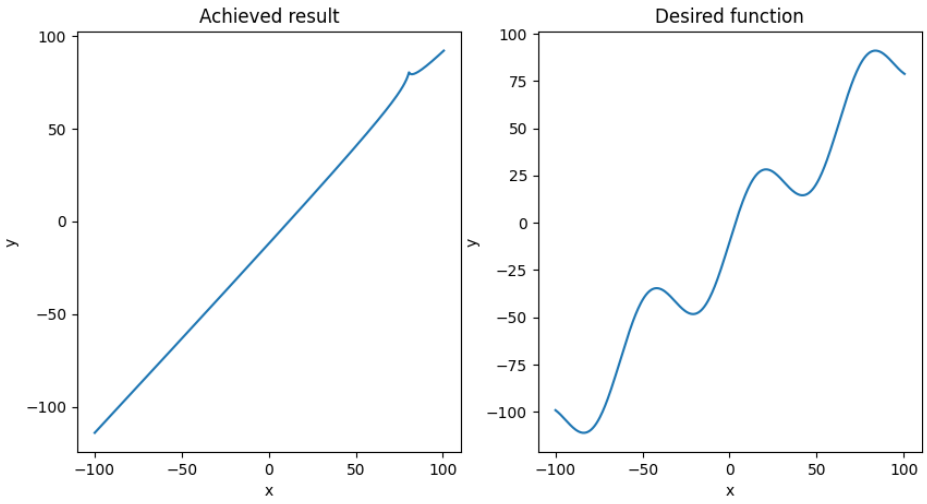

Figure 15.4. Comparison between result generated by GE as a result of trial with data containing noise, described in section 15.3 and the desired function prepared using script described in chapter 20

```
x-psqrt(87.49-x)*plog(pdiv(x-99.98,x-47.48))
```

It appears that this time shape is starting to adjust to the desired curve. Around $x = 4$ there is a visible spike that tends to the desired shape. Presence of this spike shows that reducing the constraints could possibly lead to better approximation, proving the hypothesis stated previously.

After changing parameters to allowing 1000 generations with population size 1000 the result started to contain a sin function making it even closer to the desired formula, taking 810 seconds to compute. The result is presented below and visualized in figure 15.6.

```
x-np.sin(pdiv(19.56+45.02,x))-10.54+np.sin(np.tanh(x))*psqrt(30.47)
```

Allowing 3000 generations made possible to achieve a shape even more resembling the desired one, taking 2435 seconds to compute. The result is presented below and in figure 15.7

```
x-np.sin(pdiv(66.00,x))-psqrt(79.94-pdiv(99.99,np.tanh(x))+41.63*np.sin(
    ↪ plog(x)))
```

15.4.1. Conclusion on the step of the experiment

Allowing more evolutionary processes to occur enhances the result, as showed in the trial above. Based on gained knowledge, it is safe to say that increasing mentioned parameters gives the desired outcome and is worth extra computational time.

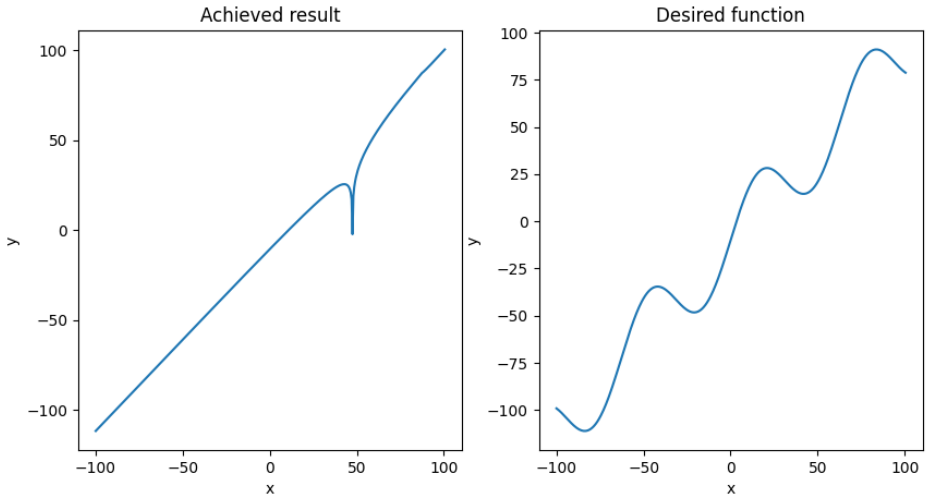

Figure 15.5. Comparison between result generated by GE as a result of trial with data containing noise and reduced evolutionary constraints, described in section 15.4 and the desired function prepared using script described in chapter 20

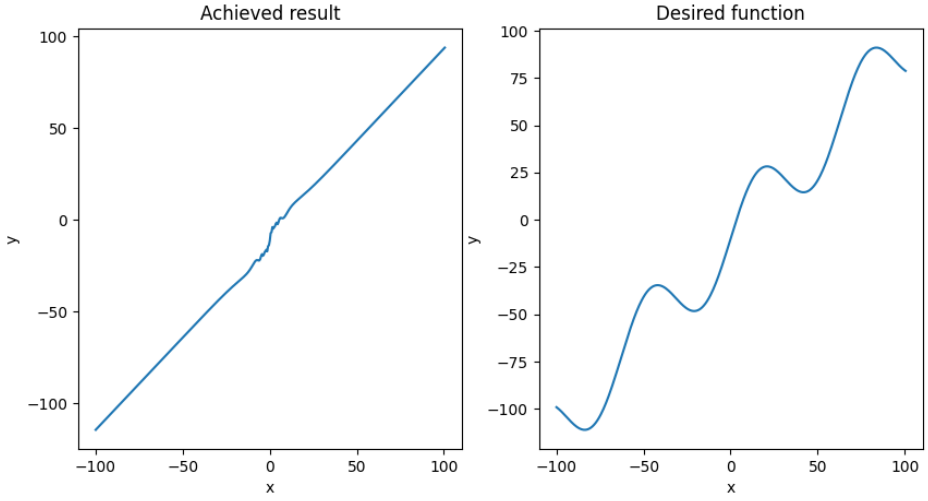

Figure 15.6. Comparison between result generated by GE as a result of trial with data containing noise and even more reduced evolutionary constraints to number of generations equal 1000 having populations of size 1000, described in section 15.4 and the desired function prepared using script described in chapter 20

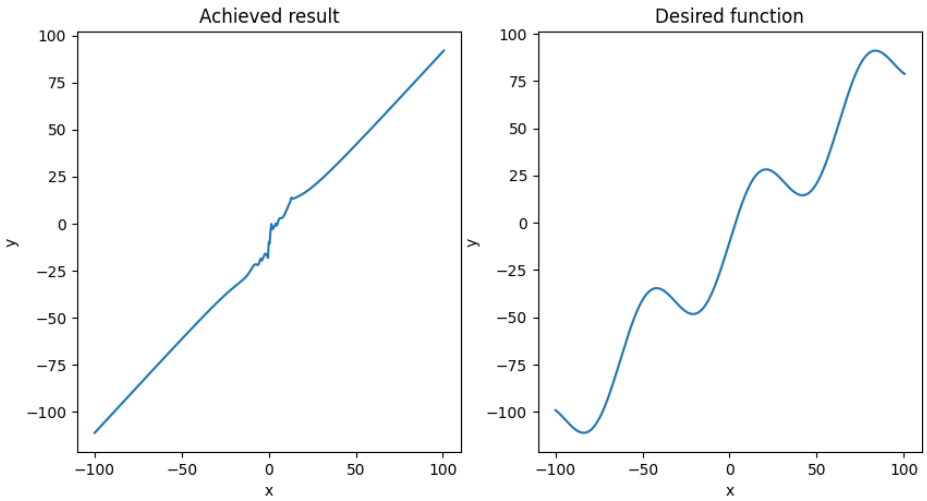

Figure 15.7. Comparison between result generated by GE as a result of trial with data containing noise and even more reduced evolutionary constraints to number of generations equal 3000 having populations of size 1000, described in section 15.4 and the desired function prepared using script described in chapter 20

15.5. Conclusion

The presented experiment showed that evolving periodic function can pose issue as in process of evolution it may get approximated by a function close to linear, taking values similarly to ones got in process of linear regression. Additionally, it was presented that introducing noise does not have significant impact on the shape of output. As a last remark, one may point out that increasing evolutionary search parameters allows getting values closer to desired function, proving that search is not getting stuck.

16. Evolving a valid Python program to compute the sum of elements in a delivered array

16.1. Introduction

The goal of this problem is to evolve a Python program that will be able to compute the sum of elements contained in an array of constant, predefined, lengths. It may be assumed that the elements of the array are integer numbers. To accomplish this task, PonyGE2 will be used.

The first attempt of this experiment was prepared with the use of the PyNeurGen library, however taking into consideration its limitations and constantly emerging issues with correct processing of grammar describing more complex solutions and issues caused by encapsulating fitness calculation function into the evolved individual it was decided to switch to PonyGE2.

16.2. Method of individual validation and calculating fitness value

16.2.1. Evaluation

To avoid overfitting, a single individual is going to be evaluated on three different randomly generated arrays of integer numbers. Evaluation based only on a single array may promote code that accidentally provided a constant that was close enough to the desired value.

16.2.2. Calculating fitness

As described in the section above, every individual is going to compute sums of three different arrays. Then fitness is the sum of absolute values of differences between returned and desired values. The formula below expresses that idea:

$$\texttt{fitness} = \sum_{i=0}^{n} (\texttt{desired\_value[i]} - \texttt{returned\_value[i]})$$

16.3. Used grammar

As a simplification, the following code snippet will be provided to guide the search in the right direction.

```
def sum_of_array(input):
    result = 0
    <code>
    return result
XXX_result = sum_of_array(input)
```

To calculate fitness according to the convention of PonyGE2, another fitness function should be developed in the form of a class derived from `base_ff`. The following piece of code presents the algorithm according to which fitness gets calculated, which is going to be used inside the fitness class.

```
import random

def right_answer(input):
    sum_of_elements = 0
    for element in input:
        sum_of_elements+=element
    return sum_of_elements

def calculate_fitness():
    fit = 0
    for _ in range(3):
        input = [random.randint(0,100) for _ in range(10)]
        fit+=abs(sum_of_array(input)-right_answer(input))
    return fit

fitness = calculate_fitness()
```

Having defined the starting point, it is time to create a BNF description of the language. Real Python grammar[27] seems to be too complex for such an example. Using the entire definition would overcomplicate the execution and consume a lot more resources than is necessary to test the performance of the tool and prove the concept, therefor grammar being a subset of original rules will be used.

The first attempt of this experiment was to be performed using PyNeurGen therefore it was planned to use a tool that creates correct indentation, available via URL https://svn.python.org/projects/python/trunk/Tools/scripts/pindent.py. However, during the execution there have emerged so many issues that it was decided to change the tool.

The used grammar is based on the one available in the GitHub repository entitled NC-GA served under a URL address https://github.com/Padam-0/NC-GA/tree/master that is supplementary material to the article https://towardsdatascience.com/introduction-to-ponyge2-for-grammatical-evolution-d51c29f2315a. Grammar does describe simple python programs but is fine-tuned for the case described in the article, therefore it requires modifications to be used in this experiment. A ready version of the grammar is presented below:

```
<fc> ::= <deff>{::}<callf>
<deff> ::= def sum_of_array(input):{:result = 0{::}<code>{::}return result :}

<callf> ::= XXX_result = sum_of_array(input)

<code> ::= <stmt> | <stmt>{::}<code>

<stmt> ::= <var> = <expr> | <for> | <if> | <list-op>

<for> ::= for i in <list-var>:{:<fl-code>:}
<fl-code> ::= <fl-stmt> | <fl-stmt>{::}<fl-code>
<fl-stmt> ::= <var> = <expr> | <fl-if>
<fl-if> ::= if <cond>:<fl-if-opt>
<fl-if-opt> ::= {:<fl-code>:} | {:<fl-code>:}else:{:<fl-code>:} | {:<fl-code>:}elif <cond>:{:<fl-if-opt>:}

<var> ::= result | i
<list-var> ::= input

<number> ::= <num><num><num> | <num><num> | <num>
<num> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<op> ::= + | - | * | / | //

<if> ::= if <cond>:<if-opt>
<if-opt> ::= {:<code>:} | {:<code>:}else:{:<code>:} | {:<code>:}elif <cond>:<if-opt>
```

```
26
27 <cond> ::= <expr> <c-op> <expr> | <expr> <c-op> <expr> and <cond> | <expr>
       ↪ <c-op> <expr> or <cond>
28 <c-op> ::= "==" | "!=" | ">=" | "<=" | ">" | "<"
29
30 <expr> ::= <number> | <var> | <expr> <op> <expr> | len(<list-var>) | <list-
       ↪ var>[<neg><number>]
31
32 <list-op> ::= <list-var>.append(<expr>)
33 <neg> ::= "" | -
```

Having this grammar, it is worth reminding that PonyGE2 does provide functionality for handling the correct indentation. To take advantage of this function, the sequence `{:` and `:}` is used. Strings enclosed in these "brackets" undergo an indent by one level. Additionally, the sequence `{:::}` is a symbol of a new line.

16.4. Implementation of the fitness class

The implementation of the fitness class does contain the mentioned previously algorithm. The main function of the additional code present in the class is to retrieve the phenotype and then pass it to the defined algorithm. To call the evolved function program uses `exec()` a function surrounded by try block as the evolved program in some cases may cause runtime errors to be raised. If this happens, the program should eliminate this solution as it cannot be executed.

The complete code of the fitness class is presented below. Implementation is inspired by example of fitness class delivered in a package, located in file `/src/fitness/pymax.py` and the one described by mentioned article https://towardsdatascience.com/introduction-to-ponyge2-for-grammatical-evolution-d51c29f2315a.

```
1 from fitness.base_ff_classes.base_ff import base_ff
2 import random
3
4 class sum_of_array(base_ff):
5
6     maximise = False
7
8     def __init__(self):
9         super().__init__()
10
11    def evaluate(self, ind, **kwargs):
12
13        p = ind.phenotype
```

```
14
15          def right_answer(input):
16              sum_of_elements = 0
17              for element in input:
18                  sum_of_elements+=element
19              return sum_of_elements
20
21          def call_sum_of_array(input,p):
22              d = {'input': input}
23              try:
24                  exec(p, d)
25                  s = d['XXX_result']
26                  return s
27              except:
28                  return None
29
30          def calculate_fitness(p):
31              fit = 0
32              for _ in range(3):
33                  input = [random.randint(0,100) for _ in range(10)]
34                  result = call_sum_of_array(input,p)
35                  if result is None:
36                      return self.default_fitness
37
38                  fit+=abs(float(result)-right_answer(input))
39              return fit
40
41          return calculate_fitness(p)
```

16.5. Parameters

During the execution of the experiment, the following parameters were used in the form of a parameters file.

```
CACHE:                  True
CODON_SIZE:             100000
CROSSOVER:              variable_onepoint
CROSSOVER_PROBABILITY:  0.75
DEBUG:                  False
GENERATIONS:            400
MAX_GENOME_LENGTH:      500
GRAMMAR_FILE:           sum_of_array.pybnf
```

```
INITIALISATION:          PI_grow
INVALID_SELECTION:       False
MAX_INIT_TREE_DEPTH:     10
MAX_TREE_DEPTH:          17
MUTATION:                int_flip_per_codon
POPULATION_SIZE:         500
FITNESS_FUNCTION:        sum_of_array
REPLACEMENT:             generational
SELECTION:               tournament
TOURNAMENT_SIZE:         2
VERBOSE:                 False
```

16.6. Results

Despite the simplicity, the model was able to quickly accomplish the given goal. The final result is presented in the listing below.

```
def sum_of_array(input):
  result = 0
  for i in input:
    result = result + i
  return result
XXX_result = sum_of_array(input)
```

Evolution took only 95 seconds, reaching exactly the desired program, achieving a perfect fitness value equal to 0. Moreover, careful inspection of the plot of the best fitness values in the given generation, presented in figure 16.1 shows that the desired value of fitness required only about 30 generations to be reached.

16.7. Conclusions

The presented example emphasizes the advantages of PoyGE2 in terms of evolving syntactically correct Python code. It does have its own mechanism for handling correct white spaces, reducing the need to handle indentation in experiment code.

The tool also showed very good performance concerning the time of computations. Additionally, providing all parameters and algorithms was done in a very convenient way, shortening the preparation time.

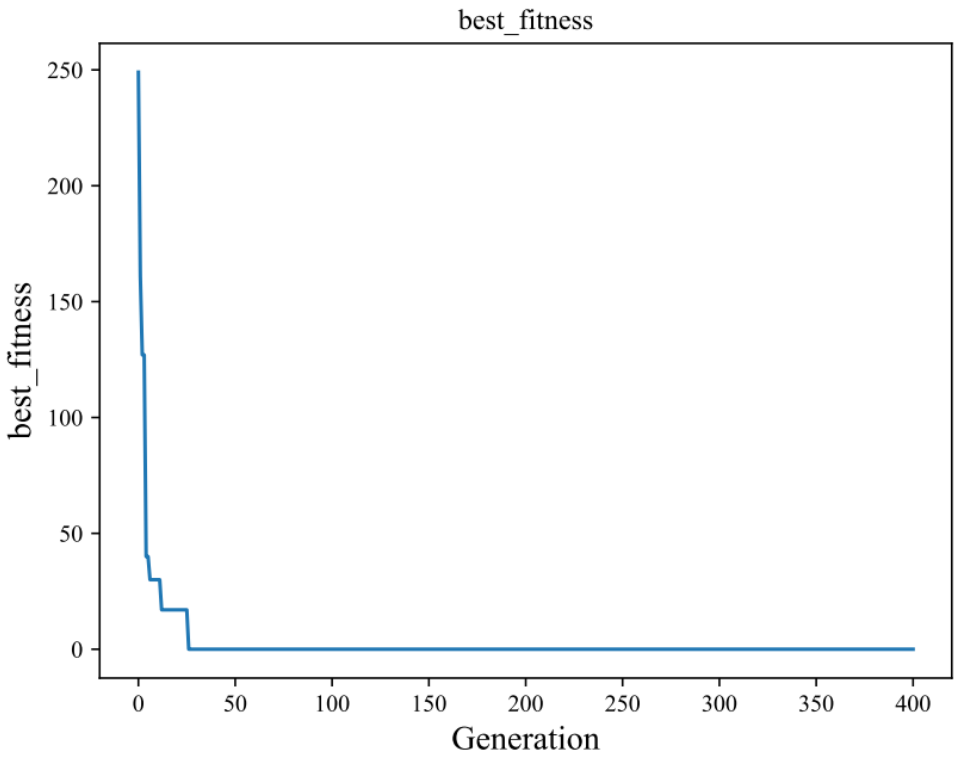

Figure 16.1. Plot presenting the best value of fitness achieved in a given generation. Plot was prepared automatically by PonyGE2

17. Simple regression on the dataset generated by a polynomial of 4th order using PyNeurGen

17.1. Introduction

This example is provided to show the possibilities offered by the PyNeurGen library. The task is to perform evolution having grammar describing mathematical expressions and a data set generated by a 4th order polynomial.

17.2. Data generation

To generate data, the polynomial presented below will be used. The selected formula makes it easier to visualize the shape of the function around interesting, non-trivial, points.

$$f(x) = 0.00001(x + 80)(x + 40)(x - 25)(x - 80)$$

This formula could be further transformed into the following one.

$$0.00001 * (x^4 + 15x^3 - 7400x^2 - 96000x + 6400000)$$

As PyNeurGen requires having fitness calculation encapsulated into the starting point code, the data generation script requires simple adjustments that will allow the script to partially generate the code. According to the method for calculating the value of fitness, described in section 17.3, the script should then prepare a ready-to-use array that contains values of parameter `x` and the desired function values. The code of the modified version is presented below.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 import matplotlib as mpl
5
6 # Creating np array for x
```

```
x = np.arange(-100,101,2)
# Computing value of the formula returned by the program
y = 0.00001*(x+80)*(x+40)*(x-25)*(x-80)


WIDTH_SIZE = 10
HEIGHT_SIZE = 5

fig, (ax1) = plt.subplots(1, 1,figsize=(WIDTH_SIZE,HEIGHT_SIZE))
ax1.plot(x, y, '.')           # Plot generated data on the axes1.

ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Dataset visualization')

temp = zip(x,y)
ds = []

for el in temp:
    ds.append([el[0],el[1]])

print(ds)

plt.show()
```

Code was inspired by online sources [13] [14].

17.2.1. Data set

Figure 17.1 shows a visualization of the data prepared for this trial. The dataset does contain a reduced number of points to test the capabilities of the tool to work with a limited amount of data.

17.3. Fitness calculation

To calculate fitness in PyNeurGen, the user is required to provide the exact code that is going to be executed. To do so, the following approach is going to be adopted. For each element of the dataset, let error `e` be the square of the difference between desired and achieved value.

$$\texttt{e} = (\texttt{desired\_value} - \texttt{achieved\_value})^2$$

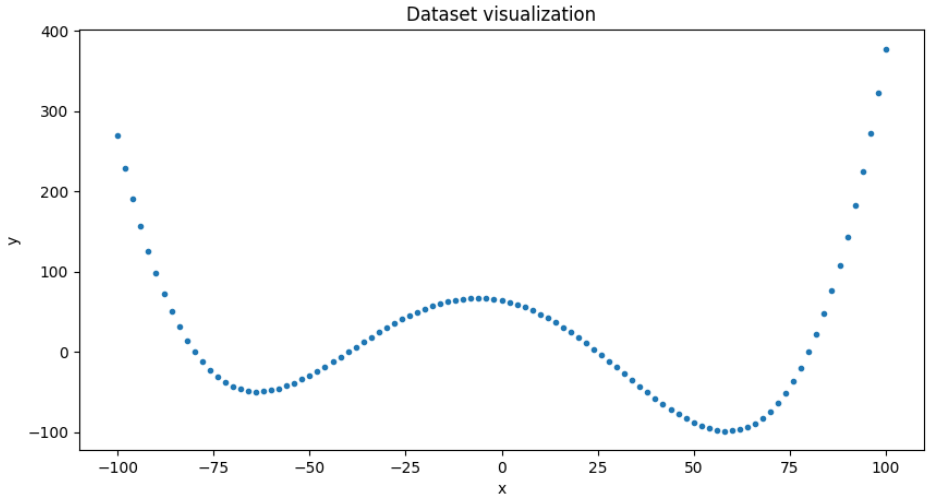

Figure 17.1. Visualization of the data set generated by polynomial $0.00001 * (x^4 + 15x^3 - 7400x^2 - 96000x + 6400000)$ prepared using tool described in section 17.2

Then, let the fitness be the sum of errors `e` of all records in the dataset. The formula is presented below.

$$\texttt{fitness} = \sum_{i=0}^{\texttt{len(dataset)}} e_i$$

$$\texttt{fitness} = \sum_{i=0}^{\texttt{len(dataset)}} (\texttt{desired\_value}_i - \texttt{achieved\_value}_i)^2$$

The presented approach can be then represented as a simple loop in code. Below, the implementation is presented. The data set variable is indexed by the number of record and by the column. The first column does contain `x` value and the second column provides the desired function output

```
1
2  def generated_fuction(x):
3      # ...
4      pass
5
6  fitness = 0
7
8  # dataset[record][col]
9
10 for record in dataset:
11     difference = generated_fuction(record[0])-record[1]
```

```
12      fitness+= difference*difference
```

Having that code, according to the conventions established by PyNeurGen, it is required to set the correct variable to transfer the fitness value into the library. The code presented below had been modified to perform all the mentioned necessary steps.

```
1
2  def generated_fuction(x):
3      # ...
4      pass
5
6  fitness = 0
7
8  # dataset[record][col]
9
10 for record in dataset:
11     difference = generated_fuction(record[0])-record[1]
12     fitness+= difference*difference
13 self.set_bnf_variable('<fitness>', fitness)
```

Having such a definition of fitness, it is required also to say that the optimization goal is to minimize its value. The ideal point is exactly at 0.

17.4. BNF grammar

In this case, a grammar developed by a library author for one of the examples, located in file `sample_grammatical_evolution.py`, is sufficient, therefore it is going to be used. with a modified starting point based on the previously mentioned fitness calculating code and slightly changed productions. The used version is presented below.

```
1  <expr>              ::= <expr> <biop> <expr> | <uop> <expr> | <real> |
2                          math.log(abs(<expr>)) | <pow> | math.sin(<expr> )
3  <biop>              ::= + | - | * | /
4  <uop>               ::= + | -
5  <pow>               ::= pow(<expr>, <real>)
6  <plus>              ::= +
7  <minus>             ::= -
8  <real>              ::= <int-const>.<int-const>
9  <int-const>         ::= <int-const> | 1 | 2 | 3 | 4 | 5 | 6 |
10                         7 | 8 | 9 | 0
11 <S>                 ::=
12 def generated_fuction(x):
13     <expr>
```

```
14    pass
15 fitness = 0
16 for record in dataset:
17     difference = generated_fuction(record[0])-record[1]
18     fitness+= difference*difference
19 self.set_bnf_variable('<fitness>', fitness)
```

17.5. Complete program

As it was described in chapter 7, PyNeurGen is just a library that needs a main program to run, therefor to perform this experiment one has to be written. The program does contain all the settings of the evolutionary process.

While adjusting the limitations, one has to be especially careful in setting the program length limit. It is important to take into consideration the length of the starting point and include it in the limit. If the specified limit is lower than the length of the starting point, the library will evaluate non fully terminated string resulting in fault and assigning fitness equal to fault penalty.

Additionally, during experiments, it was proven that PyNeurGen is not very stable, and allowing a higher number of generations may cause errors and unpredictable behavior.

The code used to perform this experiment is presented below. The implementation is inspired by example delivered with PyNeurGen in file `pyneurgen/demo/sample_grammatical_evolution.py`.

```
1  from pyneurgen.grammatical_evolution import GrammaticalEvolution
2  from pyneurgen.fitness import FitnessElites, FitnessTournament
3  from pyneurgen.fitness import ReplacementTournament
4
5  bnf="""
6  <expr>              ::= <expr> <biop> <expr> | <uop> <expr> | <real> |
7                          math.log(abs(<expr>)) | <pow> | math.sin(<expr> ) |
   ↪  x
8  <biop>              ::= + | - | * | /
9  <uop>               ::= + | -
10 <pow>               ::= pow(<expr>, <real>)
11 <plus>              ::= +
12 <minus>             ::= -
13 <real>              ::= <int-const>.<int-const>
14 <int-const>         ::= <int-const> | 1 | 2 | 3 | 4 | 5 | 6 |
15                         7 | 8 | 9 | 0
16 <S>                 ::=
17 import math
```

```
def generated_fuction(x):
   return <expr>
   pass
fitness = 0
dataset = [] # here goes gnerated dataset
for record in dataset:
    difference = generated_fuction(record[0])-record[1]
    fitness+= difference*difference
self.set_bnf_variable('<fitness>', fitness)

"""

ges = GrammaticalEvolution()

ges.set_bnf(bnf)
ges.set_genotype_length(start_gene_length=20,
                        max_gene_length=50)
ges.set_population_size(30)
ges.set_wrap(True)

ges.set_completion_criteria("g", 60)
ges.set_completion_criteria("f", "center", 0.01)

#ges.set_fitness_type('min', .01)

ges.set_max_program_length(3500)
ges.set_timeouts(10, 120)
ges.set_fitness_fail(10000000000.0)

ges.set_mutation_rate(.025)
ges.set_fitness_selections(
    FitnessElites(ges.fitness_list, .05),
    FitnessTournament(ges.fitness_list, tournament_size=2))
ges.set_max_fitness_rate(.5)

ges.set_crossover_rate(.2)
ges.set_children_per_crossover(2)
ges.set_mutation_type('m')
ges.set_max_fitness_rate(.25)

ges.set_replacement_selections(
        ReplacementTournament(ges.fitness_list, tournament_size=3))

```

```
61 ges.set_queue_size(0)
62 ges.set_garbage_collection(5)
63 ges.set_maintain_history(True)
64 ges.create_genotypes()
65 print ges.run()
66 print ges.fitness_list.sorted()
67 print
68 print
69 gene = ges.population[ges.fitness_list.best_member()]
70 print gene.get_program()
```

17.6. Results

Unfortunately, due to tool limitations in terms of efficiency and stability, it was possible only to achieve the following result.

```
1 def generated_fuction(x):
2     return 9.9
3     pass
```

It is not the right solution and cannot be considered a success. It emphasizes the issues of the tool, showing that it may not be the right choice for large projects.

17.7. Remarks on used tool

During testing the tool, it went out that it is the most convenient to use with so-called one-liners, as the tool does not perform very well in terms of generating whole functions due to the way the grammar parsing and fitness calculation are performed.

Despite having getting started tutorials, it is very difficult to quickly get the program working. The tutorial on GE presents just a simple case and does not provide extensive and exhaustive explanations of some functionalities, forcing users to reverse engineer the code.

18. Results, summary, and conclusion

18.1. Results of tools comparison

18.1.1. General conclusion on comparison

Having presented a few of the available tools and the usual criteria stated by software developers, it is not possible to point to one winner of the comparison. Presented tools do differ significantly in terms of delivered features and used technology. Additionally, they come with different advantages and disadvantages whose weights may differ among users. As an example of such features, the time efficiency and learning curve can be presented.

Available literature shows some mature tools that are still maintained and that are usable in current projects, as well as interesting but abandoned tools. There was also presented a single example of a currently rising tool with great potential to succeed, depending on the maintenance and further development.

18.1.2. Final recommendation on tools

As it was stated in the previous section it is impossible to select, a single, best tool, however, after careful analysis of the presented developers' needs, available data on developers' preferences, and features of tools it is possible to state some recommendations.

Two packages that are definitely worth of attention are PonyGE2 and gramEvol. Both tools are equipped with comprehensive documentation and supplementary resources covering potential use cases. Both are mature and still maintained. Moreover, both tools gathered a community of users who use it for real cases.

Special attention could be also paid to GRAPE, which could be a significant competitor of PonyGE2 in the near future, assuming it will be still maintained and improved.

18.2. Conclusions on performed experiments and prepared models

18.2.1. Final conclusions

Performed experiments show just a few of many potential use cases but emphasize the advantages of Grammatical Evolution and available tools. Experiments also show potential issues that may emerge during the development of the GE model.

Grammars and their correct definitions are very good tools to introduce additional knowledge into the system. Careful design may limit the search space in accordance with already possessed knowledge about the system. One of the grammars presented in the example from gramEvol documentation does introduce more knowledge than grammars used in other examples, showing this ability to encapsulate expert knowledge into the model.

Designing a correct model requires deep knowledge of the utility of the solution. It allows the user to create a correct fitness function that is being used to rate evolved solutions, therefore allowing the selection of the correct results.

Special attention should be paid also to the limitations of a derivation process — they differ among tools. Some do introduce the concept of limiting the depth of a derivation tree, whereas others use just the length of the final phenotype as a limitation.

As a final conclusion, one may say that building GE models does allow injecting knowledge into the system and has the efficiency of retrieving good solutions proportional to the amount of introduced knowledge. GE does not require a lot of theory knowledge from the final users, enabling them to adopt the technique relatively quickly.

18.3. Conclusion on GE potential applications

Previous chapters do present lots of potential applications of GE. Examining the sources clearly shows that it may be used with a very wide variety of problems.

Currently, it is mostly used in the field of evolving mathematical expressions, computer program generation, and classification.

Careful analysis of available publications on grammatical evolution applications shows that this approach, despite its clear advantages over other approaches, is still used by a minority of projects. For example, the number of available projects using neural networks and projects using GE is non-comparable.

In spite of much lower developers' and researchers' interest than the artificial neural networks receive, models developed by GE show significant advantages over their ANN competitors.

- Ability to work on limited dataset — In real-life scenarios, access to data sources may be limited, posing a serious issue in terms of preparing the training data.
- Efficiency of training/evolution — Neural networks may take a lot more time to get trained than GE to evolve a good solution. That is caused by additional, refined, knowledge introduced into the system by use of CFG.
- Explainability — One of GE's goals is to produce a solution that is simple and possible to understand by humans, which is hardly possible in terms of ANN.
- Safety — It is possible to perform extensive analysis of the expected behavior of the system in edge cases using formal methods.

To conclude, it may be said that GE is still not widely used, but it does provide solutions to many of current issues with other approaches.

19. Further work

19.1. Better comparison

Quality of tool comparison could be increased as current may suffer from some issues — mostly related to small developers' interest in particular tools caused by narrow tool specialization, like depending only on official sources that have no other sources maintained by 3rd parties to perform a cross-check of presented information.

19.1.1. Expert knowledge on each tool

Expert knowledge could be potentially used to increase the accuracy of tools' quality analysis. Most of the presented conclusions were based on information gathered from available sources and own experiments, leading to a danger of omitting some important features or information on the quality of the tool.

Using the knowledge of experts who use the tool would provide valuable notes on that tool. Unfortunately, access to that kind of knowledge is very limited due to the small size of communities around GE tools.

19.1.2. Exploring tools in depth

As an additional source of information about the tool, the code itself may be used. Such an approach would provide a clear insight into the mechanics behind the tool. Such an approach would result in gaining detailed knowledge, but it comes with a huge cost of an enormous amount of consumed time.

19.2. Further literature review

This text describes only a subset of available tools that were chosen based on the described features and the state of maintenance of the tool. It is possible that there exist still some new

tools that had not been found yet but do provide better quality. Searching for publications about new tools is complicated due to lack of popularity of new solutions. Often they are not listed with more popular tools, and it is required to extensively search to reach them. An example of such tool is GRAPE. It is not described widely yet and there is a single paper describing it, not to mention lack of its presence on lists of GE tools. That leaves a huge field for improvement, as it is even difficult to estimate the probable number of publications to be searched through.

Additionally, there are many more publications handling topics of fine-tuning the evolutionary process. Using better parameters may influence the comparison of tools significantly. Therefore, exploring the topic of fine-tuning GE models may result in changing execution times or improving accuracy of some tools.

19.3. Preparing own solution based on gained experience

19.3.1. Motivation

Based on gained information about available tools and their issues, it is possible to design new tool, combining approaches presented by different tools.

19.3.2. Usability

Having gathered all advantages and features, that have significant influence on each tools' usability, it is possible to maximize the usability of new tool.

Some of such features were for example handling of indentation, designing and passing own fitness function to the tool in convenient way, precision, automatic report generation, ease of installation.

Moreover, it would be crucial to remember about correct and clear documentation, numerous examples and guides explaining how to begin using the tool.

19.3.3. Potential benefits

A well-designed tool accompanied by a learning resources could increase popularity of GE approach, allowing its development and further propagation of XAI idea.

20. Program used to generate plots of functions using Matplotlib

The program presented in this chapter was used to generate visual interpretations of PonyGE2 results. It does contain definition of auxiliary functions taken directly from PonyGE2 sources (`src/utilities/fitness/math_functions.py`) to ensure right interpretation of result.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl



from math import ceil
np.seterr(all="raise")


def return_one_percent(num, pop_size):
    """
    Returns either one percent of the population size or a given number,
    whichever is larger.

    :param num: A given number of individuals (NOT a desired percentage of
    the population).
    :param pop_size: A given population size.
    :return: either one percent of the population size or a given number,
    whichever is larger.
    """

    # Calculate one percent of the given population size.
    percent = int(round(pop_size / 100))

```

```
    # Return the biggest number.
    if percent < num:
        return num
    else:
        return percent


def return_percent(num, pop_size):
    """
    Returns [num] percent of the population size.

    :param num: A desired percentage of the population.
    :param pop_size: A given population size.
    :return: [num] percent of the population size.
    """

    return int(round(num * pop_size / 100))


def aq(a, b):
    """aq is the analytic quotient, intended as a "better protected
    division", from: Ji Ni and Russ H. Drieberg and Peter I. Rockett,
    "The Use of an Analytic Quotient Operator in Genetic Programming",
    IEEE Transactions on Evolutionary Computation.

    :param a: np.array numerator
    :param b: np.array denominator
    :return: np.array analytic quotient, analogous to a / b.

    """
    return a / np.sqrt(1.0 + b ** 2.0)


def pdiv(x, y):
    """
    Koza's protected division is:

    if y == 0:
      return 1
    else:
      return x / y

    but we want an eval-able expression. The following is eval-able:
```

```
70
71      return 1 if y == 0 else x / y
72
73      but if x and y are Numpy arrays, this creates a new Boolean
74      array with value (y == 0). if doesn't work on a Boolean array.
75
76      The equivalent for Numpy is a where statement, as below. However
77      this always evaluates x / y before running np.where, so that
78      will raise a 'divide' error (in Numpy's terminology), which we
79      ignore using a context manager.
80
81      In some instances, Numpy can raise a FloatingPointError. These are
82      ignored with 'invalid = ignore'.
83
84      :param x: numerator np.array
85      :param y: denominator np.array
86      :return: np.array of x / y, or 1 where y is 0.
87      """
88      try:
89          with np.errstate(divide='ignore', invalid='ignore'):
90              return np.where(y == 0, np.ones_like(x), x / y)
91      except ZeroDivisionError:
92          # In this case we are trying to divide two constants, one of which
    ↪ is 0
93          # Return a constant.
94          return 1.0
95
96
97  def rlog(x):
98      """
99      Koza's protected log:
100     if x == 0:
101       return 1
102     else:
103       return log(abs(x))
104
105     See pdiv above for explanation of this type of code.
106
107     :param x: argument to log, np.array
108     :return: np.array of log(x), or 1 where x is 0.
109     """
110     with np.errstate(divide='ignore'):
111         return np.where(x == 0, np.ones_like(x), np.log(np.abs(x)))
```

```
112
113
114 def ppow(x, y):
115     """pow(x, y) is undefined in the case where x negative and y
116     non-integer. This takes abs(x) to avoid it.
117
118     :param x: np.array, base
119     :param y: np.array, exponent
120     :return: np.array x**y, but protected
121
122     """
123     return np.abs(x) ** y
124
125
126 def ppow2(x, y):
127     """pow(x, y) is undefined in the case where x negative and y
128     non-integer. This takes abs(x) to avoid it. But it preserves
129     sign using sign(x).
130
131     :param x: np.array, base
132     :param y: np.array, exponent
133     :return: np.array, x**y, but protected
134     """
135     return np.sign(x) * (np.abs(x) ** y)
136
137
138 def psqrt(x):
139     """
140     Protected square root operator
141
142     :param x: np.array, argument to sqrt
143     :return: np.array, sqrt(x) but protected.
144     """
145     return np.sqrt(np.abs(x))
146
147
148 def psqrt2(x):
149     """
150     Protected square root operator that preserves the sign of the original
151     argument.
152
153     :param x: np.array, argument to sqrt
154     :return: np.array, sqrt(x) but protected, preserving sign.
```

```
155     """
156     return np.sign(x) * (np.sqrt(np.abs(x)))
157
158
159 def plog(x):
160     """
161     Protected log operator. Protects against the log of 0.
162
163     :param x: np.array, argument to log
164     :return: np.array of log(x), but protected
165     """
166     return np.log(1.0 + np.abs(x))
167
168
169 def ave(x):
170     """
171     Returns the average value of a list.
172
173     :param x: a given list
174     :return: the average of param x
175     """
176
177     return np.mean(x)
178
179
180 def percentile(sorted_list, p):
181     """
182     Returns the element corresponding to the p-th percentile
183     in a sorted list
184
185     :param sorted_list: The sorted list
186     :param p: The percentile
187     :return: The element corresponding to the percentile
188     """
189
190     return sorted_list[ceil(len(sorted_list) * p / 100) - 1]
191
192
193 def binary_phen_to_float(phen, n_codon, min_value, max_value):
194     """
195     This method converts a phenotype, defined by a
196     string of bits in a list of float values
197
```

```
198     :param phen: Phenotype defined by a bit string
199     :param n_codon: Number of codons per gene, defined in the grammar
200     :param min_value: Minimum value for a gene
201     :param max_value: Maximum value for a gene
202     :return: A list os float values, representing the chromosome
203     """
204
205     i, count, chromosome = 0, 0, []
206
207     while i < len(phen):
208         # Get the current gene from the phenotype string.
209         gene = phen[i:(i + n_codon)]
210
211         # Convert the bit string in gene to an float/int
212         gene_i = int(gene, 2)
213         gene_f = float(gene_i) / (2 ** n_codon - 1)
214
215         # Define the variation for the gene
216         delta = max_value[count] - min_value[count]
217
218         # Append the float value to the chromosome list
219         chromosome.append(gene_f * delta + min_value[count])
220
221         # Increment the index and count.
222         i = i + n_codon
223         count += 1
224
225     return chromosome
226
227
228 def ilog(n, base):
229     """
230     Find the integer log of n with respect to the base.
231
232     >>> import math
233     >>> for base in range(2, 16 + 1):
234     ...     for n in range(1, 1000):
235     ...         assert ilog(n, base) == int(math.log(n, base) + 1e-10), '%s
    ↪  %s' % (n, base)
236     """
237     count = 0
238     while n >= base:
239         count += 1
```

```
240         n //= base
241     return count
242
243
244 def sci_notation(n, prec=3):
245     """
246     Represent n in scientific notation, with the specified precision.
247
248     >>> sci_notation(1234 * 10**1000)
249     '1.234e+1003'
250     >>> sci_notation(10**1000 // 2, prec=1)
251     '5.0e+999'
252     """
253     base = 10
254     exponent = ilog(n, base)
255     mantissa = n / base ** exponent
256     return '{0:.{1}f}e{2:+d}'.format(mantissa, prec, exponent)
257
258
259 # Code above is taken directly form source of PonyGE2 to
260 # ensure that visualisation is accurate.
261 ##########################
262 # Creating np array for x
263 x = np.arange(-100,101,.5)
264 # Computing value of formula returned by program
265 y = x-np.exp(np.tanh(10.00))*plog(80.49-x)
266 # Computing values of the deired folrmula
267 y_desired = np.sin(0.1*x) * 20 + x - 10
268
269
270 WIDTH_SIZE = 10
271 HEIGHT_SIZE = 5
272
273 fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(WIDTH_SIZE,HEIGHT_SIZE))
274 ax1.plot(x, y)           # Plot generated data on the axes1.
275 ax2.plot(x, y_desired)   # Plot desired data on the axes2.
276
277 ax1.set_xlabel('x')
278 ax1.set_ylabel('y')
279 ax1.set_title('Achieved result')
280
281 ax2.set_xlabel('x')
282 ax2.set_ylabel('y')
```

```
283 ax2.set_title('Desired function')
284
285
286 plt.show()
```

The code was inspired by examples from online sources [58], [13], [14] and does contain code taken from PonyGE2 repository to maintain a constant interpretation of special functions.

21. Program used to generate the dataset that contains a noise

The program presented below was used in experiments to prepare a data set, in a format used by PonyGE2, and introduce a noise that is generated according to the normal distribution with parameters specified in the code.

```
import matplotlib.pyplot as plt
import numpy as np

import matplotlib as mpl

# Creating np array for x
x = np.arange(-100,101,.5)
# Computing accurate function values
y = np.sin(0.1*x)*20+x-10


# Noise specification
mean = 0
standard_deviation = 10

# Generation of the noise
noise = np.random.normal(mean,standard_deviation, len(x))

# Introducing the noise
realistic_y = y + noise

WIDTH_SIZE = 10
HEIGHT_SIZE = 5

fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(WIDTH_SIZE,HEIGHT_SIZE))  # Create figure for plots and plots.
ax1.plot(x, y, '.')            # Plot accurate data on the axes1.
```

```
ax2.plot(x, realistic_y, '.')  # Plot noisy data on the axes2.

ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Pure data')

ax2.set_xlabel('x')
ax2.set_ylabel('y')
ax2.set_title('Data with noise')

plt.show()

test_trail = zip(x,y)
normal_trail = zip(x,realistic_y)

print("### Pure data ###")

for x,y in test_trail:
    print(x,y,sep='\t',end='\n')

print("### Data with noise ###")

for x,y in normal_trail:
    print(x,y,sep='\t',end='\n')
```

The code was inspired by examples from online sources [58], [13] and [14].

Bibliography

[1] Pączkowanie. //pl.wikipedia.org/wiki/P%C4%85czkowanie?oldid=71226831, wrzesień 2023. [Accessed: 2024-01-09 03:42Z].

[2] Peter Adam. Introduction to ponyge2 for grammatical evolution. https://towardsdatascience.com/introduction-to-ponyge2-for-grammatical-evolution-d51c29f2315a, July 2017.

[3] Hasanen Alyasiri, John A Clark, Ali Malik, and Ruairí de Fréin. Grammatical evolution for detecting cyberattacks in internet of things environments. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE, 2021.

[4] Nikolaos Anastasopoulos, Ioannis G. Tsoulos, and Alexandros Tzallas. Genclass: A parallel tool for data classification based on grammatical evolution. *SoftwareX*, 16:100830, 2021.

[5] Dimitrios Angelis, Filippos Sofos, and Theodoros E. Karakasidis. Artificial intelligence in physical sciences: Symbolic regression trends and perspectives. *Archives of Computational Methods in Engineering*, 30(6):3845–3865, April 2023.

[6] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Peter Naur, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. Revised report on the algorithmic language algol 60. *The Computer Journal*, 5(4):349–367, 1963.

[7] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, 1959.

[8] Tatenda Herbert. Chareka and Nelishia Pillay. A study of fitness functions for data classification using grammatical evolution. *2016 Pattern Recognition Association of South Africa and Robotics and Mechatronics International Conference (PRASA-RobMech)*, pages 1–4, 2016.

[9] B.J.. Copeland. artificial intelligence. https://www.britannica.com/technology/artificial-intelligence. [Accessed 19 October 2023.].

[10] Cleber A.C.F. da Silva, Daniel Carneiro Rosa, Péricles B.C. Miranda, Filipe R. Cordeiro, Tapas Si, André C.A. Nascimento, Rafael F. L. Mello, and Paulo S. G. de Mattos Neto. A multi-objective grammatical evolution framework to generate convolutional neural network architectures. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2187–2194, 2021.

[11] Allan de Lima, Samuel Carvalho, Douglas Mota Dias, Enrique Naredo, Joseph P. Sullivan, and Conor Ryan. Grape: Grammatical algorithms in python for evolution. *Signals*, 3(3):642–663, 2022.

[12] Anthony Mihirana De Silva and Philip HW Leong. *Grammar-based feature generation for time-series prediction*. Springer, 2015.

[13] The Matplotlib development team. matplotlib.axes.axes.plot - matplotlib 3.8.2 documentation. https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.plot.html#matplotlib.axes.Axes.plot.

[14] The Matplotlib development team. Quick start guide - matplotlib 3.8.2 documentation. https://matplotlib.org/stable/users/explain/quick_start.html#quick-start.

[15] Grant Dick and Peter A. Whigham. Initialisation and grammar design in grammar-guided evolutionary computation. https://arxiv.org/abs/2204.07410, 2022.

[16] Jessica Barbosa Diniz, Filipe R. Cordeiro, Pericles B. C. Miranda, and Laura A. Tomaz Da Silva. A grammar-based genetic programming approach to optimize convolutional neural network architectures. In *Anais do XV Encontro Nacional de Inteligência Artificial e Computacional (ENIAC 2018)*, ENIAC 2018. Sociedade Brasileira de Computação - SBC, October 2018.

[17] Jessica Barbosa Diniz, Filipe R Cordeiro, Pericles BC Miranda, and Laura A Tomaz da Silva. A grammar-based genetic programming approach to optimize convolutional neural network architectures. In *Anais do XV Encontro Nacional de Inteligência Artificial e Computacional*, pages 82–93. SBC, 2018.

[18] Dimiter Dobrev. A definition of artificial intelligence. https://arxiv.org/abs/1210.1568, 2012.

[19] Don Smiley <ds@sidorof.com>. Pyneurgenpython neural genetic algorithm hybrids api - grammatical_evolution module. https://pyneurgen.sourceforge.net/api/grammatical_evolution_api.html#GrammaticalEvolution.

[20] Don Smiley <ds@sidorof.com>. Pyneurgenpython neural genetic algorithm hybrids api - suggestions for using grammatical evolution. https://pyneurgen.sourceforge.net/ge_suggestions.html.

[21] Anthony Mihirana de Silva Farzad Noorian. Package grammatical evolution for r. https://cran.r-project.org/web/packages/gramEvol/gramEvol.pdf, October 2022.

[22] Philip H.W. Leong Farzad Noorian, Anthony M. de Silva. Grammatical evolution: A tutorial using gramevol. https://fnoorian.github.io/gramEvol/inst/doc/ge-intro.html.

[23] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2 wiki - about ponyge2. https://github.com/PonyGE/PonyGE2/wiki/Introduction#about-ponyge2. [Accessed: 27-11-2023].

[24] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2 wiki documentation. https://github.com/PonyGE/PonyGE2/wiki. [Accessed 13-11-2023].

[25] Michael Fenton, James McDermott, David Fagan, Stefan Forstenlechner, Erik Hemberg, and Michael O'Neill. Ponyge2: grammatical evolution in python. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17. ACM, July 2017.

[26] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[27] Python Software Foundation. Full grammar specification - python 3.12.1. https://docs.python.org/3/reference/grammar.html.

[28] Meghana Kshirsagar Gauri Vaidya. Pycom 2022 - workshop: Shield: Do it the safe way. https://python.ie/previous-pycons/pycon-2022/talks/, 2022. [Accessed 13-11-2023].

[29] David Gunning, Mark Stefik, Jaesik Choi, Timothy Miller, Simone Stumpf, and Guang-Zhong Yang. Xai—explainable artificial intelligence. *Science robotics*, 4(37):eaay7120, 2019.

[30] Krishn Kumar Gupt, Muhammad Adil Raja, Aidan Murphy, Ayman Youssef, and Conor Ryan. Gelab – the cutting edge of grammatical evolution. *IEEE Access*, 10:38694–38708, 2022.

[31] Branimir K. Hackenberger. R software: unfriendly but probably the best. *Croatian Medical Journal*, 61(1):66–68, February 2020.

[32] Thomas Helmuth and Lee Spector. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, page 1039–1046, New York, NY, USA, 2015. Association for Computing Machinery.

[33] R. Hurbans. *Grokking Artificial Intelligence Algorithms*. Manning, 2020.

[34] ISO. Iso/iec 14977:1996 information technology - syntactic metalanguage - extended bnf. Standard, ISO, 1996. [Accessed 15-11-2023].

[35] K.D. Jagreet. *Artificial Intelligence and Deep Learning for Decision Makers*. BPB Publications, 2019.

[36] Pedro José Pereira, Paulo Cortez, and Rui Mendes. evoltree: Evolutionary decision trees. https://pypi.org/project/evoltree/.

[37] D.E. Kazaryan and A.V. Savinkov. Grammatical evolution for neural network optimization in the control system synthesis problem. *Procedia Computer Science*, 103:14–19, 2017. XII International Symposium Intelligent Systems 2016, INTELS 2016, 5-7 October 2016, Moscow, Russia.

[38] John R. Koza and Riccardo Poli. *Genetic Programming*, pages 127–164. Springer US, Boston, MA, 2005.

[39] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford book. Bradford, 1992.

[40] S.J. Naik. *Think AI: Explore the flavours of Machine Learning, Neural Networks, Computer Vision and NLP with powerful python libraries (English Edition)*. ITpro collection. BPB Publications, 2022.

[41] Miguel Nicolau and Alexandros Agapitos. Understanding grammatical evolution: Grammar design. *Handbook of grammatical evolution*, pages 23–53, 2018.

[42] Adam Nohejl. Grammatical evolution. bathesis, Charles University in Prague, Faculty of Mathematics and Physics, 2009.

[43] Adam Nohejl. Grammar-based genetic programming. mathesis, Charles University in Prague, Faculty of Mathematics and Physics, 2011.

[44] Farzad Noorian, Anthony M de Silva, and Philip HW Leong. gramevol: Grammatical evolution in r. *Journal of Statistical Software*, 71:1–26, 2016.

[45] Farzad Noorian, Anthony M de Silva, and Philip HW Leong. Grammatical evolution: A tutorial using gramevol. *Massachusetts Institute of Technology*, 2016.

[46] User of geeksforgeeks.org. Bnf notation in compiler design. https://www.geeksforgeeks.org/bnf-notation-in-compiler-design/, Jul 2021. [Accessed 15-11-2023].

[47] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[48] Stack Overflow. Stack overflow developer survey 2023 - most popular technologies. https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup May 2023.

[49] A.P. Parkes. *Introduction to Languages, Machines and Logic: Computable Languages, Abstract Machines and Formal Logic*. Springer London, 2012.

[50] PROJECTPRO. Why r programming language still rules data science? https://www.projectpro.io/article/why-r-programming-language-still-rules-data-science/161, 2023.

[51] David Robinson. The impressive growth of r. https://stackoverflow.blog/2017/10/10/impressive-growth-r/, 2017.

[52] Conor Ryan, JJ Collins, and Michael O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Genetic Programming*, pages 83–96, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[53] Sergey Salihov, Dmitriy Maltsov, Maria Samsonova, and Konstantin Kozlov. Solution of mixed-integer optimization problems in bioinformatics with differential evolution method. *Mathematics*, 9(24):3329, 2021.

[54] Dominik Sepioło and Antoni Ligęza. Towards model-driven explainable artificialintelligence. an experiment with shallowmethods versus grammatical evolution.

[55] Daniel Shanks. *Solved and unsolved problems in number theory*, volume 297. American Mathematical Soc., 2001.

[56] Anupriya Shukla, Hari Mohan Pandey, and Deepti Mehrotra. Comparative review of selection techniques in genetic algorithm. In *2015 international conference on futuristic trends on computational analysis and knowledge management (ABLAZE)*, pages 515–519. IEEE, 2015.

[57] D Smiley. Pyneurgen: Python neural genetic algorithm hybrids. *Release 0.3, URL http://pyneurgen.sourceforge.net*, 2012.

[58] MS Somanna. Guide to adding noise to synthetic data using python and numpy. https://medium.com/@ms_somanna/guide-to-adding-noise-to-your-data-using-python-and-numpy-c8be815df524.

[59] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23(5):828–841, oct 2019.

[60] Tuong Manh Vu. Software review: Pony ge2. *Genetic Programming and Evolvable Machines*, 22(3):383–385, Sep 2021.

[61] Hao Wang, Yitan Lou, and Thomas Bäck. Hyper-parameter optimization for improving the performance of grammatical evolution. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 2649–2656, 2019.

[62] Eric W. Weisstein. Prime counting function. https://mathworld.wolfram.com/PrimeCountingFunction.html.

[63] Wikipedia contributors. Genetic programming — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Genetic_programming&oldid=1178930607, 2023. [Online; accessed 24-December-2023].

[64] Wikipedia contributors. Grammatical evolution — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Grammatical_evolution&oldid=1183182329, 2023. [Accessed: 16-11-2023].

[65] Wikipedia contributors. Kleene star — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Kleene_star&oldid=1165979969, 2023. [Online; accessed 13-January-2024].

[66] Wikipedia contributors. Ohm's law — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ohm%27s_law&oldid=1191985219, 2023. [Online; accessed 9-January-2024].

[67] Wikipedia contributors. Selection (genetic algorithm) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Selection_(genetic_algorithm)&oldid=1178932579, 2023. [Online; accessed 9-January-2024].

[68] Wikipedia contributors. Symbolic regression — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Symbolic_regression&oldid=1183921862, 2023. [Online; accessed 28-November-2023].

[69] Wikipedia contributors. Notacja EBNF [online]. wikipedia : wolna encyklopedia, 2023-11-12 00:06Z. [Accessed: 5-11-2023].https://pl.wikipedia.org/wiki/Notacja_EBNF?oldid=71821429.

[70] Wikipedia contributors. Genetic algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=1193124228, 2024. [Online; accessed 9-January-2024].

[71] Wikipedia contributors. Software quality — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Software_quality&oldid=1193814316, 2024.

[72] Saneh Lata Yadav and Asha Sohal. Comparative study of different selection techniques in genetic algorithm. *International Journal of Engineering, Science and Mathematics*, 6(3):174–180, 2017.

[73] Zhen Zhong, Ribhu Sengupta, Kamran Paynabar, and Lance A. Waller. Multi-objective allocation of covid-19 testing centers: Improving coverage and equity in access. https://arxiv.org/abs/2110.09272, 2021.