Seminar report in Automatische Programmierung (IN2107, IN4898)

Genetic Programming Applied to Time Series Forecasting

January 2018

Eivind Meyer
Supervised by Dr. Harald Rueß

Department of Informatics
Technische Universität München

Preface

This report was written as a part of the evaluation material for the seminar course Automatisches Programmieren *(Automatic Programming)* held by Dr. Harald Rueß at Die Technische Universität München, Department of Informatics. Out of a number of available topics related to the concept of automatic programming, I chose to conduct my study within the one labelled as *Genetic Programming for the Generation of Programs by Means of Natural Selection.*

Abstract

Time series forecasting is a topic of great theoretical and practical interest. In this paper, we examine how genetic programming can be applied to it. In short, the predictions are calculated by applying the same genetically trained function to each historic data point, and summing the results. A software implementation in Python has been created together with this report, yielding promising results on artificial and real-world test data.

Contents

List of Figures

1 Introduction

Time series forecasting, that is, the predicting of future values of a time-indexed vector of data points, is a research field of great academic and practical interest. The applications thereof range from weather forecasting to stock market predictions — applications with obvious importance to a lot of people.

Unlike classical approaches to time series forecasting, in which typically the parameters of a fixed-structure model are optimized, genetic programming (GP) allows the model in its entirety to be altered. In fact, that is the very basis of the technique; by resembling biological evolution as first formulated by Charles Darwin, genetic programming signifies the generation of increasingly capable computer programs through evolutionary mechanisms. Albeit the fundamental ones are reproduction and natural selection, genetic algorithms can be extended by a variety of other mechanisms.

In terms of the problem at hand, genetic programming lets us perform a so-called symbolic regression analysis of the time series, that is, fitting a function with mutable structure to the training data. Whereas the traditional regression technique linear regression alters the regression parameters $\hat{\boldsymbol{\beta}}$ so that the predictions yielded by $f(\hat{\boldsymbol{\beta}}, \boldsymbol{x}) = \hat{\beta}_1 x_1 + \hat{\beta}_1 x_2 + ... + \hat{\beta}_1 x_p$ most accurately fit the training data, symbolic regression is not held back by such a limitation and could end up with highly non-linear and complex solutions such as, let us say, $\hat{\beta} x_1^2 sin(x_1 + x_2)^{x_2}$.

The field of genetic programming is considered a sub-field of the more general concept of evolutionary algorithms, in which the solutions not necessarily have the form of computer programs. For instance, in another popular sub-field of evolutionary algorithms, namely that of genetic algorithms, solutions have the form of sequences of characters or numbers. Evolutionary programming, also a sub-field of evolutionary algorithms, resembles genetic programming, but only allows numerical parameters to be altered, while the structure of the program is kept fixed.

1.1 Motivation

The requirement of a predefined function whose numerical parameters are to be optimized can be considered a disadvantage of the classical approaches to time series forecasting and regression in general. Often, a tedious research process is required in order to discover an adequate function. Furthermore, the decision might be based upon domain-knowledge (or the delusion of such) and possibly be flawed.

In *A Field Guide to Genetic Programming*, (Poli, Langdon and McPhee, 2008), seven empirically derived success indicators for the application characteristics are outlined:

1. Unknown or poorly understood relationships between the variables.
2. The finding of the shape of the solution is an essential part of the problem.
3. Large amounts of training data are available.
4. Testing a solution is easy, finding one is hard.
5. Analytic solutions do not exist.
6. Approximate solutions are acceptable.
7. Small improvements in the solutions are very valuable.

Even though point 3 may or may not hold true, depending on the problem at hand, all of the other properties are with certainty present within the field of time series forecasting. Consequently, testing the performance of genetic programming as a forecasting procedure for time series forms an interesting research topic. And given the many useful applications of time series forecasting, the topic is not only of academic interest.

1.2 Report structure

The first chapter serves as an introduction to the report. In Section 2, an explanation of the relevant theoretical concepts is provided. Section 3 describes the composition of the genetic programming software that has been developed alongside this report, while Section 4 presents and discusses the results obtained from said software. Section 5 concludes the report, and discusses potential improvements to the approach.

2 Theory

Firstly, we formulate the problem of time series forecasting in mathematical notation. For a single training sample, we label $\boldsymbol{x} = [x_0, x_1, ..., x_n]$ as the vector representing the time series. The prediction of the upcoming unknown value, namely x_{n+1}, is denoted $\hat{y}$.

2.1 Programs

2.1.1 Structure

In genetic programming, the programs are usually represented as trees [1], in which every inner node has two children nodes. The inner nodes represent bivariate functions that take their respective children as input. These functions are usually simple operations like addition, multiplication or exponentiation, but may be arbitrarily complex. The set of available functions, labelled as the *function set*, is predefined. On the other hand, each of the leaf nodes embodies a variable or constant. The set of available variables and constants is also predefined and is labelled as the *terminal set*. Additionally, each node may be transformed by some univariate function, for instance $sin(x)$ or e^x.

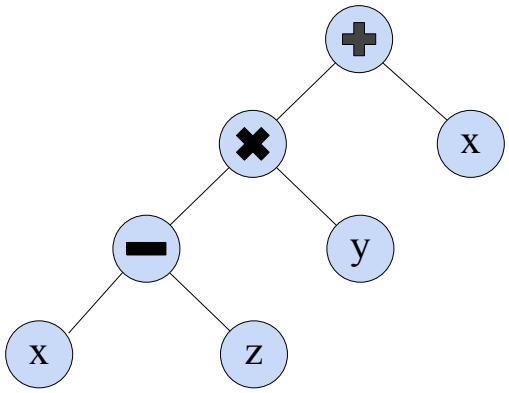

Figure 1: Graphical representation of $f(x, y, z) = (x - z)y + x$

In this study specifically, we make further assumptions of the program structure in order to adapt to the time series forecasting problem. Namely, we assume that the program can be written in the following form:

```
def predict(X, n):
    y_hat = 0
    consts = ...
    for i in range(n):
        x_i = X[i]
        y_hat += f(x_i, i, n, consts)
    return y_hat
```

Here, the function f is the symbolic representation of the tree that is being altered through genetic programming. Since the prediction now is defined as the sum of independent f outputs, we lose the ability of having interaction between different time series data points. However, in order to reduce the search space complexity, this decision has still been made.

2.1.1.1 Example: Predicting by averaging

As an example, imagine a program that simply calculates $\hat{y}$ by averaging the values in $\boldsymbol{x}$.

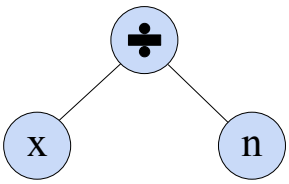

Figure 2: Graphical representation of f in the case of predicting by averaging

which would correspond to

```
def predict(X, n):
    y_hat = 0
    for i in range(n):
        x_i = X[i]
        y_hat += x_i / n
    return y_hat
```

2.1.2 Fitness

Analogous to natural selection in the domain of biological evolution, we want to ensure that the variants of the predictor function f that yield the most capable programs are passed on to later generations. Equivalently, we want to remove the bad ones. For this purpose we need some fitness measure through which we can rank the programs. Unlike in classical approaches to numerical optimization, there is no requirement imposed on the fitness measure having to be differentiable or continuous. Is goes without saying that the fitness measure has to somehow incorporate the deviation between the predicted value $\hat{y}$ and the actual value x_{n+1} summed over all the training samples.

Since there exists no a priori knowledge beyond that from which we can deduct an optimal fitness measure, we simply decide on a logistic fitness measure

$$\phi = \frac{1}{1 + exp(-R^2)} \tag{1}$$

where R^2 is the coefficient of determination, as known from the field of statistics. This statistic is the squared correlation between the the predicted y values and the actual y values.

2.2 Evolution

To facilitate the description of the evolution process, we introduce the term *population*, which is simply the collection of programs that exist at any given time in the process. We also use the term *generation* for a specific version of the population. The evolution process happens iteratively; a each iteration, the next generation is spawned from the current one through evolutionary mechanism. Analogous to the phrase *survival of the fittest*, the $|P|$ best programs, where $|P|$ is the size of the population, are transferred to the next generation. The *goodness* of a program is, of course, equivalent to its score according to the fitness function (4). We allow both the existing programs, as well as the ones produced through the evolutionary operators *crossover* and *mutation* to participate in this survival contest.

2.2.1 Randomized creation

Initially, the population must be filled with randomly generated programs from which the later generations originate from. The random creation of a program goes as follows:

1. Randomly decide the number of constants to be included in the terminal set, and assign to each of them a sampled value from some probability distribution.
2. Recursively decide the predictor function f by sampling operators from the function set while at inner nodes, and sampling the terminal set while at leaf nodes. Decide randomly whether a child node should be an inner node or a leaf node. It is helpful to have limit on the tree depth that the tree cannot surpass, in order to avoid ending up with over-complicated trees.

2.2.2 Crossover

The crossover is the main operator utilized in the evolution process [1]. It is analogous to reproduction in the biological sense. The most widely used form of crossover is *sub-tree crossover* [3], which happens in the following fashion:

1. Select two parents from the current population.
2. For both parents, randomly select a *crossover point* from the parent's respective predictor function tree. This could be either an inner node or a leaf node.
3. Insert the sub-tree rooted at the first parent's crossover point at the crossover point of the second parent, by replacing the existing sub-tree.

Given the structural assumptions made in section 2.1.1 with regards to the programs, it is also required to copy the parent's constants to the newly spawned child program. It can also be argued that adding a small drift to these constants might be helpful in order to facilitate the long-term creation of better programs.

2.2.3 Mutation

A mutation is a random alternation of a program. The most widely used form of mutation is *sub-tree mutation* [3], which consists of the following steps:

1. Select a program from the current population.
2. In the same manner as with the crossover operator, select a crossover point from the program's predictor function tree.
3. Replace the tree rooted at the crossover point by a randomly generated tree produced in the same way as described in section 2.2.1.

2.2.4 Selection

The genetic operators used in the evolution process, that is, crossover (2.2.2), mutation (2.2.3) and migration (2.3), require some sort of algorithm for selecting the programs that are having offspring, being mutated or migrating, respectively. In *A Field Guide to Genetic Programming* (Poli, Langdon and McPhee, 2008), *tournament selection* is extensively described, that is, to sample two programs at random, and select the best one. In the software implementation, however, *fitness proportional selection* is used. This method is as simple as sampling programs from a probability distribution in which the programs' sampling probability is proportional to their respective fitness [4].

2.3 Island models

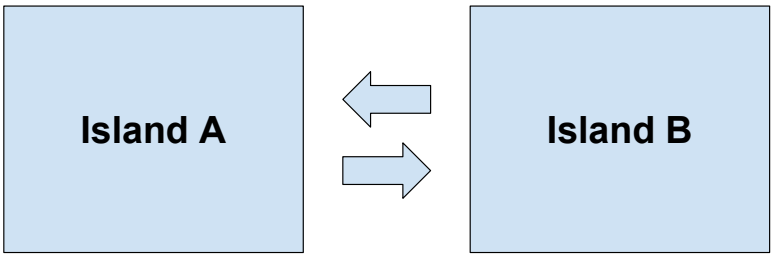

Figure 3: Island model consisting of two islands with intermigration

An extension of genetic programming is to let multiple populations evolve simultaneously on different *islands*, often by utilizing parallel or distributed computing implementations which would speed up the process. It is also possible to allow for intermigration, that is, mutual migration between islands. Again, this is an analogy for the happenings of biological evolution, in which evolution takes place in parallel at geographically separated locations. Empirically, the island model often performs better than a single population of the size equivalent to all the islands [7]. An informal rationale for this is that the island model ensures genetic diversity, as the islands follow different search trajectories. This, of course, relies upon the assumption that the mentioned intermigration is kept relatively small.

3 Experimental Setup

As mentioned, a Python implementation has been created in together with this report. This module can easily be used in other projects by importing it as is the case with other libraries. The module has also been tested on both artificial and real datasets.

3.1 Operator sampling

When sampling the *function set*, that is, the operators used upon the children of inner nodes, we sample from a probability distribution equivalent to the relative frequency table below.

Operator	**Relative frequency**
np.add	50
np.subtract	25
np.multiply	15
np.divide	10
np.power	2
np.mod	0.2
np.fmin	0.2
np.fmax	0.2
np.greater	0.1
np.greater_equal	0.1
np.less	0.1
np.less_equal	0.1
np.equal	0.01
np.not_equal	0.01

Table 1: Sampling distribution for binary operators

In the implementation, both leaf nodes and inner nodes can be transformed using some unary operator. The probability of this happening to an arbitrary node is 0.25. Whenever a transformation is introduced, it is sampled from the probability distribution representing the relative frequency table below.

Operator	**Relative frequency**
np.exp	4
np.sin	1
np.cos	1
np.tan	1
np.sign	1
np.fabs	1
np.floor	0.02
np.ceil	0.02
np.remainder	0.01

Table 2: Sampling distribution for unary operators

3.2 Mini-batch learning

In the case of huge datasets, computing the fitness of a program will take long — and increasingly so when the trees get bigger and bigger. And considering the fact that a GP population size often is in the range of thousands, the entire evolution process may take unacceptably long. However, evaluating the fitness over the entire training set may not be necessary in order to successfully evolve from one generation to the next. On the contrary, it might be unhelpful, as we risk ending up with populations where all or most of the programs are descendants of a program that performed well in the early generations [3]. Limiting the fitness evaluation to the program's predictive performance on a random fraction of the training samples, will introduce a randomness that serves to maintain diversity in the population.

3.3 Datasets

We simulate the algorithm on the following datasets:

3.3.1 Random Walk with Momentum

This artificial dataset is generated through the following stochastic procedure:

Initialization phase:

$$p = \mathcal{N}(0, 1) \tag{2a}$$

$$v = \mathcal{N}(0, 0.1) \tag{2b}$$

Iterative phase: $(0 \leq i < 20)$

$$p = p + v \tag{3a}$$

$$x_i = \mathcal{N}\left(p, \frac{|p|}{10}\right) \tag{3b}$$

$$v = v + \mathcal{N}(0, 0.02) \tag{3c}$$

3.3.2 Bitcoin USD

The *Bitcoin USD* data set [9] includes the closing price for Bitcoin on a daily basis since 2010. On a given day D, we declare the closing price for each of the 100 preceding days to be the time-series used for predicting the closing price on day D.

3.3.3 North Carolina Weather

The *North Carola Weather* data set [8] consists of daily temperature measurements from Raleigh-Durham International Airport since 2009. On a given day D, we declare the average temperature for each of the 40 preceding days to be the time-series used for predicting the average temperature on day D.

3.4 Regularization

From empirical observation, it became obvious that the algorithm would heavily overfit the training data if preventive measures were not taken. From a dataset that was generated artificially as described in section 3.3.1, the algorithm ended up with the awfully complicated solution shown in appendix A.1.1. Based on that experience, we extend the fitness function by additional terms in order to penalize program complexity.

$$\phi_2 = \frac{1}{1 + exp(-R^2 + 0.0005t_s + 0.005t_h)} \tag{4}$$

Here, t_s is the number of nodes in the tree, while t_h is the longest distance between the root and a leaf node. When evolving another program for the same dataset using this regularization technique, we then ended up with the much cleaner solution provided in appendix A.1.2. Even though the overcomplex solution surprisingly did not perform horrible on the validation set, the program yielded by the regularized algorithm produced better predictions.

4 Evaluating Results

4.1 Random Walk with Momentum

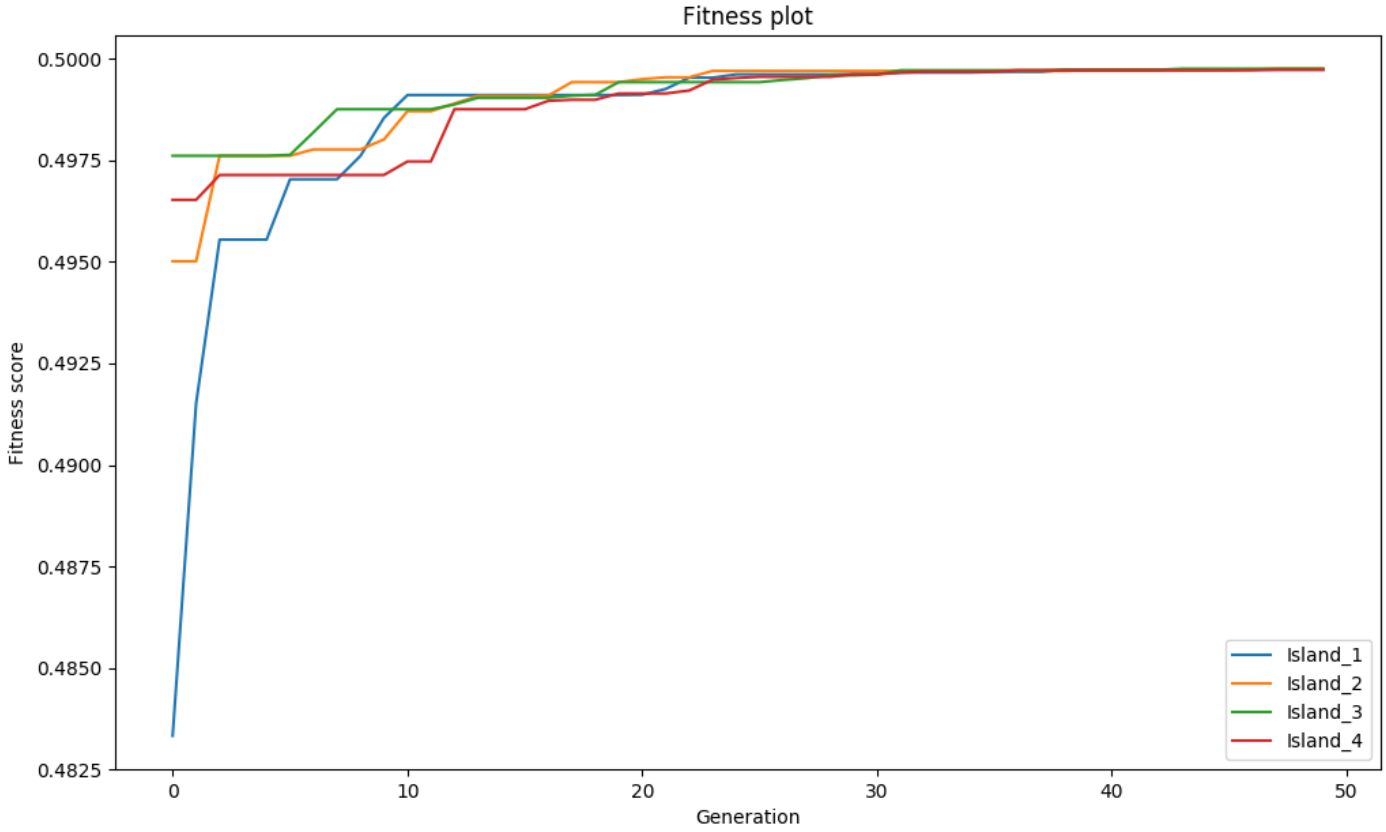

Figure 4: Fitness curve for 4 island-setup on the *Random Walk with Momentum* dataset

The above plot show the maximum training score for each island at each iteration. We observe that all of the islands eventually converge, partly due to the migration effects.

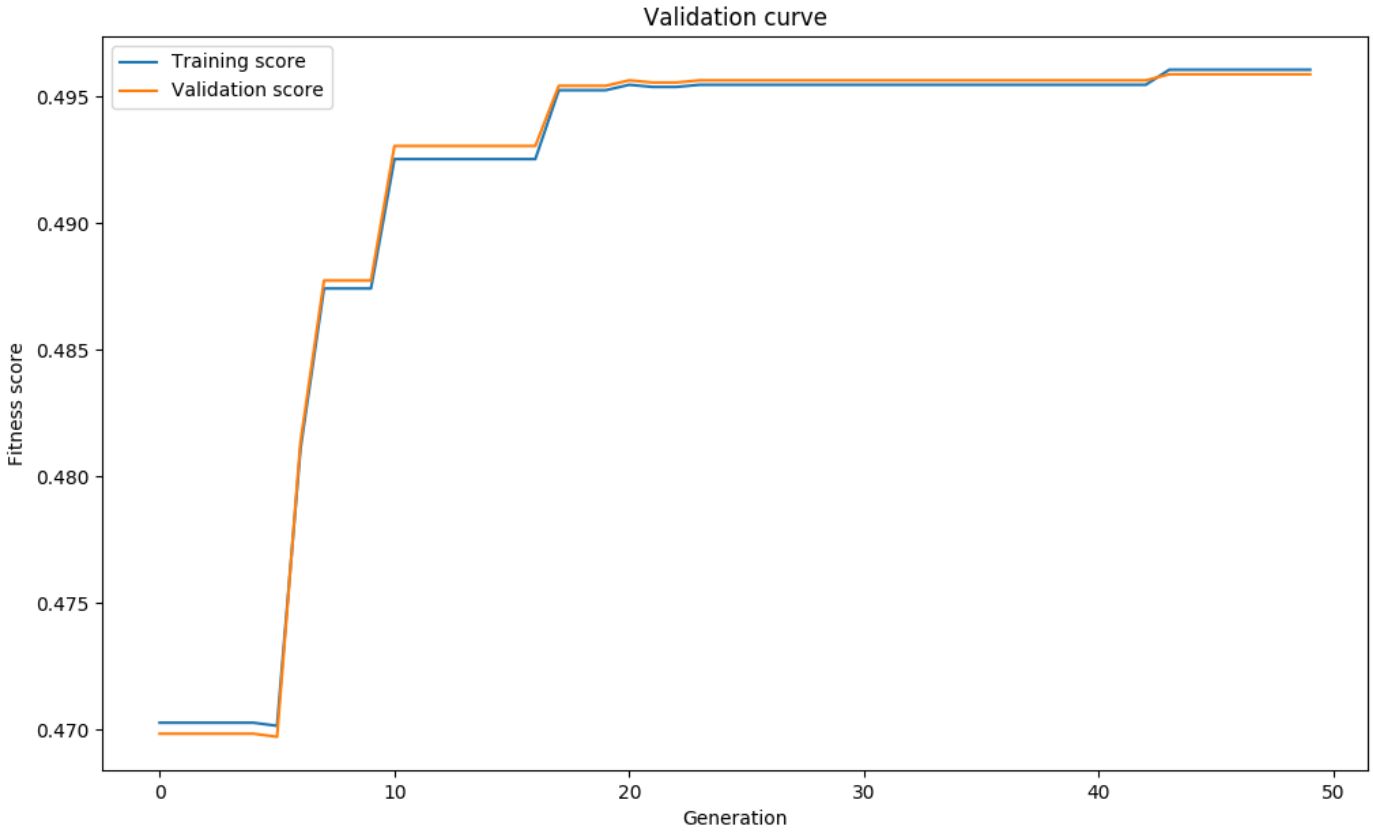

Figure 5: Validation curve for 4 island-setup on the *Random Walk with Momentum* dataset

The above plot shows the training and validation score of the program that has the best training score at each iteration of the evolution process. The fact that the two curves lie quite close to each other indicates the absence of any overfitting.

```
def predict(X, n):
        y_hat = sum(np.divide(np.add(X[i][0], X[i][0]), np.exp(np.subtract(n, i))) ...
        for i in range(n))
        return y_hat
```

The above program, which was the one with the highest prediction score on the validation set after the evolution process, weights time series samples with exponentially decreasing weight over time. This is intuitively a very reasonable strategy given the nature of the problem at hand.

Table 3: Example time series (left to right, top to bottom)

1.39	1.31	1.31	1.15	1.09	0.72	0.66	0.50	0.47	0.42
0.27	0.20	0.14	0.09	0.01	-0.07	-0.13	-0.21	-0.25	-0.39
-0.41									

While the actual y value is -0.41, the predicted $\hat{y}$ is -0.38. Of course, this single sample should not be used to draw any conclusion, but through visual inspection of multiple such samples, we can be reasonably confident that the algorithm is not very far off.

4.1.1 Bitcoin USD

Unfortunately, due to the the amount of data at hand, the evolution process was simply too time-consuming to be completed. Instead, the training was aborted after ten generations.

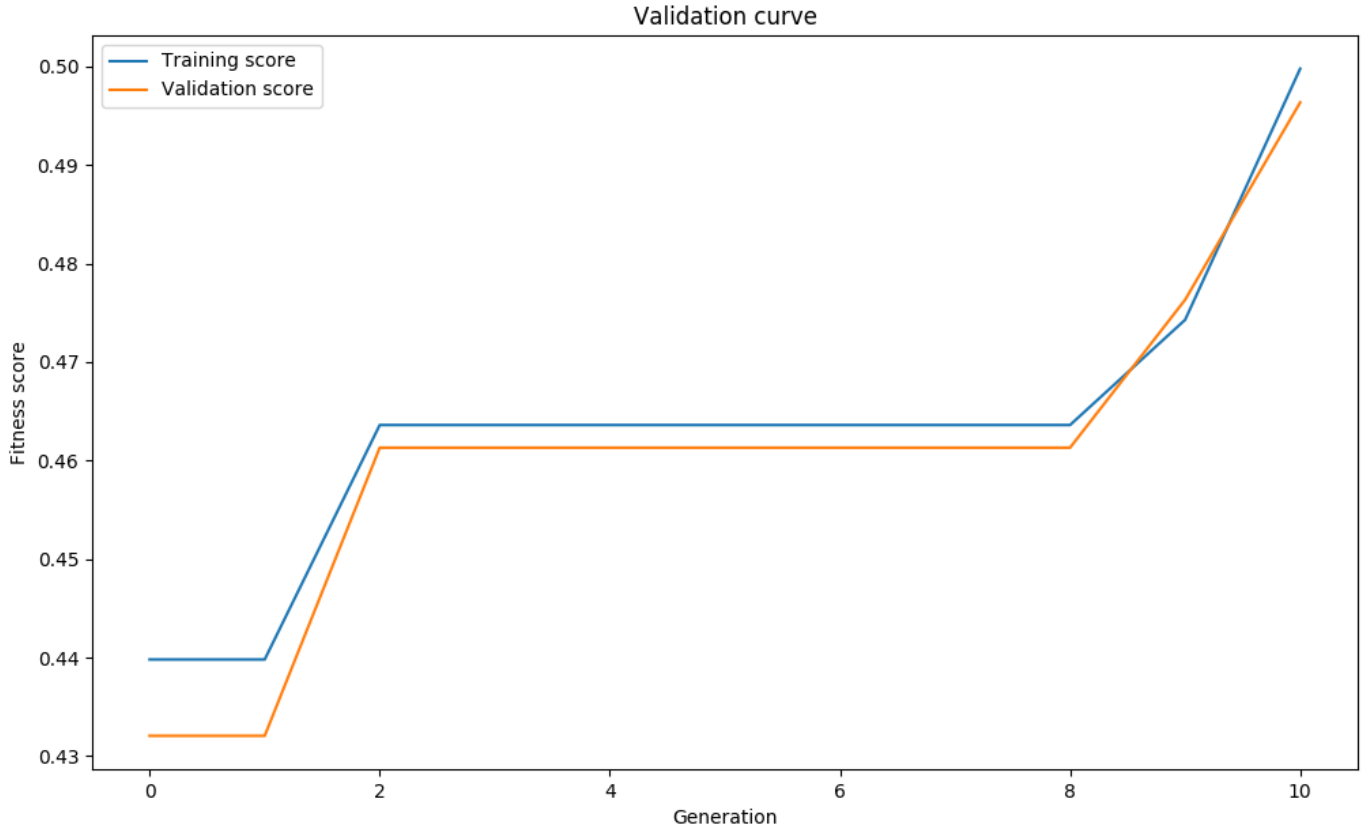

Figure 6: Validation curve for 4 island-setup on the *Bitcoin USD* dataset

```
def predict(X, n):
        const_98806283862962383509241869411 2 = 5.645514933677274
        const_73918215529208742249063870959 = 5.743234569110239
        y_hat = sum(np.divide(np.add(X[i][0], np.add(X[i][0], X[i][0])),
        np.divide(np.multiply(np.add(np.divide(np.add(np.exp(np.mod(np.add(
        np.add(X[i][0], X[i][0]), const_73918215529208742249063870959), X[i][0])),
        X[i][0]),
        np.divide(X[i][0], X[i][0])), X[i][0]), n),
        np.add(const_98806283862962383509241869411 2, X[i][0]))) for i in range(n))
        return y_hat
```

Listing 1: Highest scoring predictor function on the *Bitcoin USD* dataset

Considering the notable deviation between the training and validation scores, and the program complexity of the solution above, it is reasonable to assume that the penalty terms in the fitness function should have been weighted higher in order to avoid overfitting.

4.1.2 North Carolina Weather

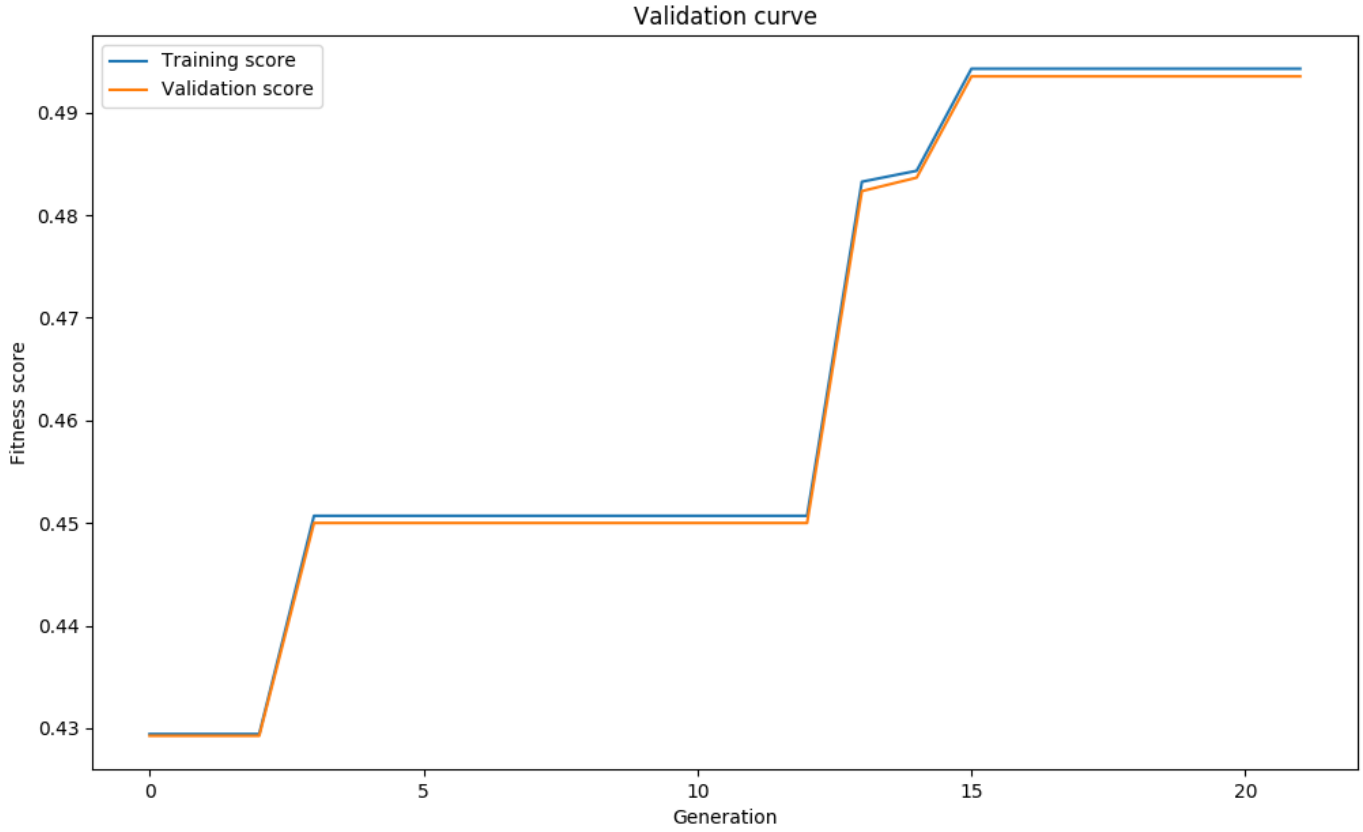

Figure 7: Validation curve for 4 island-setup on the *North Carolina Weather* dataset

```
def predict(X, n):
        y_hat = sum(np.divide(np.add(np.cos(np.add(np.add(X[i][0], np.divide(X[i][0],
        np.cos(np.divide(X[i][0], np.add(X[i][0], np.exp(np.add(X[i][0],
        np.cos(np.add(X[i][0], n)))))))))), np.exp(n))), X[i][0]), n) for i in range(n))
        return y_hat
```

Listing 2: Highest scoring predictor function on the *North Carolina Weather* dataset

Having observed the apparent overfitting at the Bitcoin USD dataset, the penalty terms for program complexity as described in section 3.4 were increased. It is possible that the cleaner results obtained in this section, relatively speaking, were due to that change.

5 Conclusion

This report has described the fundamental concept of genetic programming as well as outlining typical evolutionary mechanisms and implementation ideas related to the topic. Furthermore, development of a Python module for symbolic regression through genetic programming has been conducted. Using this implementation, we have tested the performance of genetic programming on both artificial and real-world time series, with promising results. Even though the training process is slow, applying genetic programming to time series forecasting might be worthwhile, considering the usefulness of even slightly better performing prediction techniques in the field.

5.1 Potential improvements

Firstly, it should be stressed that training a speed increase would be very beneficial. Currently, the evolution process requires days, if not weeks, of training if the training dataset is sizable. It can, however, be made faster by changing the configuration of the program. First and foremost, the training will be faster if the population sizes as well as the training batch sizes are reduced. A potential disadvantage would be reduced algorithm efficiency with regards to finding high-scoring predictor functions. Anyway, it is clear that the implementation code should either be optimized, or potentially partly or fully migrated into a faster programming language.

Secondly, the project would benefit from comparing the results obtained from the Python implementation with state-of-the-art prediction techniques for time series such as *autoregressive–moving-average (ARMA)*. Without it, it is impossible to decide if the genetic programming implementation in reality is a competitive method or not.

It would also be interesting to check if traditional gradient-based optimization methods can be successfully used to specify the values of the numerical constants.

References

[1] Volker Nissen (1997) *Einführung in Evolutionäre Algorithmen: Optimierung nach dem Vorbild der Evolution*. Vieweg+Teubner Verlag

[2] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu and Guido Lanza (2003) *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer US

[3] Poli R, Langdon WB, McPhee NF (2008) *A Field Guide to Genetic Programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza)

[4] Gareth Jones (2004) *Genetic and Evolutionary Algorithms*. University of Sheffield, UK

[5] *A Simple Genetic Programming in Python [Blog post]*. Retrieved from `https://zhanggw.wordpress.com/2009/11/08/a-simple-genetic-programming-in-python-4/`.

[6] I. De Falco1, A. Della Cioppa2, and E. Tarantino (2006) *A Genetic Programming System for Time Series Prediction and Its Application to El Niño Forecast*. Institute of High Performance Computing and Networking – CNR.

[7] Darrell Whitley, Soraya Rana and Robert B. Heckendorn (1998) *The Island Model Genetic Algorithm: On Separability, Population Size and Convergence*. Department of Computer Science, Colorado State University

[8] National Oceanic and Atmospheric Administration - National Centers for Environmental Information *Local Weather Archive*. Dataset freely available at `https://catalog.data.gov/dataset/local-weather-archive`.

[9] Yahoo! Finance *Bitcoin USD (BTC-USD)*. Dataset freely available at `https://finance.yahoo.com/quote/BTC-USD/history?p=BTC-USD`.

A Appendix

A.1 Programs

A.1.1 *Random Walk with Momentum* algorithm output when not penalizing for program complexity

```
np.seterr(all = "ignore")
def predict(X, n):
        const_1116891152692538704168736279282 = 1.5937772166369832
        const_5826686172124708119854066322 32 = 8.708808574592355
        const_1177018530145800718407032802513 = 9.969132261867415
        const_1872501126106297068098073340 51 = 11.39602373865399
        const_5229970079996172895621322363 69 = 3.312402130175974
        y_hat = sum(
            np.divide(X[i][0], ...
            np.add(np.exp(np.add(np.subtract(np.add(np.exp(np.add(i, ...
            np.subtract(np.add(np.subtract(n, ...
            np.add(np.add(np.add(np.divide(np.multiply(np.cos(np.add(X[i][0], ...
            np.exp(X[i][0]))), ...
            np.subtract(np.multiply(np.add(np.exp(const_1177018530145800718407032802513),
            X[i][0]), X[i][0]), X[i][0])), np.add(np.exp( ...
            np.add(np.subtract(X[i][0], X[i][0]), ...
            np.exp(n))), np.fabs(np.divide(np.add(X[i][0], X[i][0]), np.add(X[i][0], ...
            X[i][0])))))), const_1177018530145800718407032802513), ...
            np.sign(np.divide(X[i][0], ...
            np.subtract(const_1177018530145800718407032802513, X[i][0])))), ...
            np.divide(np.exp(np.add(X[i][0], ...
            np.tan(np.add(np.divide(np.sin(np.subtract(X[i][0], X[i][0])), ...
            np.add(np.subtract(np.add(np.multiply(np.add(np.exp(i), np.sin(i)), ...
            np.cos(X[i][0])), np.exp(np.add(np.divide(X[i][0], ...
            const_1116891152692538704168736279282), X[i][0]))), ...
            np.exp(np.subtract(np.add(X[i][0], X[i][0]), np.subtract(n, ...
            np.multiply(np.sin(np.add(np.sin(np.add(X[i][0], np.add(X[i][0], ...
            np.multiply(np.subtract(n, np.exp(np.add(np.divide( ...
            np.multiply(np.add(X[i][0], ...
```

```
        np.exp(np.add(n, i))), np.fmax(i, X[i][0])), ...
        np.subtract(np.sign(np.add(np.exp(X[i][0]), X[i][0])), ...
        np.add(const_582668617212470811985406632232, np.sign(X[i][0])))), ...
        np.add(np.add(const_52299700799961728956213223 6369, ...
        np.sin(np.subtract(np.add(np.exp(np.divide(X[i][0], X[i][0])), ...
        np.divide(X[i][0], np.add(np.exp(np.add(np.subtract(np.add(X[i][0], ...
        np.multiply(np.divide(np.subtract(X[i][0], X[i][0]), np.exp(i)), ...
        np.mod(np.exp(X[i][0]), ...
        np.subtract(np.subtract(np.cos(np.subtract(np.divide(X[i][0], X[i][0]), ...
        np.tan(np.subtract(X[i][0], X[i][0])))), np.add(np.add(X[i][0], n), ...
        np.subtract(const_187250112610629706809807334051, n))), ...
        np.subtract(np.multiply(np.multiply(X[i][0], X[i][0]), ...
        np.exp(np.subtract(X[i][0], const_187250112610629706809807334051))), ...
        np.add(X[i][0], np.sin(np.add(X[i][0], X[i][0])))))))), X[i][0]), ...
        np.tan(np.add(n, np.subtract(X[i][0], X[i][0])))))), ...
        const_11770185301458007184070328025 13))), np.less_equal(np.add(X[i][0], ...
        X[i][0]), np.sin(np.subtract(np.add(np.add(np.add(X[i][0], X[i][0]), ...
        const_52299700799961728956213223 6369), np.subtract(X[i][0], X[i][0])), ...
        np.less_equal(np.add(X[i][0], X[i][0]), X[i][0])))))), n)))), ...
        np.add(np.exp(X[i][0]), X[i][0])))))), X[i][0])), np.divide(X[i][0], n)))))), ..
        const_11168911526925387041687362792 82)), np.cos(np.add(np.multiply(i, n), ...
        X[i][0])))))), n))), np.add(X[i][0], np.add(np.subtract(np.add(X[i][0], ...
        np.multiply(np.cos(X[i][0]), X[i][0])), np.subtract(np.subtract(X[i][0], ...
        X[i][0]), np.add(X[i][0], np.sin(X[i][0])))), np.subtract(i, np.add(X[i][0], .
        np.tan(n)))))), np.add(X[i][0], np.subtract(X[i][0], ...
        np.multiply(np.add(np.exp(np.add(X[i][0], np.add(X[i][0], np.exp(X[i][0])))),
        np.add(np.add(np.exp(X[i][0]), X[i][0]), np.add(X[i][0], np.exp(i)))), ...
        np.subtract(np.cos(const_11770185301458007184070328025 13), ...
        np.exp(np.divide(np.add(X[i][0], np.cos(X[i][0])), X[i][0])))))))))), n), ...
        np.add(i, np.floor(n))), np.add(np.add(np.divide(X[i][0], ...
        np.add(np.exp(np.add(np.tan(np.fmin(np.add(X[i][0], ...
        np.subtract(np.add(X[i][0], X[i][0]), np.divide(X[i][0], np.exp(n)))), ...
        X[i][0])), np.tan(np.add(n, np.add(const_582668617212470811985406632232, ...
        np.sign(X[i][0])))))), const_11770185301458007184070328025 13)), ...
        np.cos(const_11770185301458007184070328025 13)), ...
        const_11770185301458007184070328025 13))), ...
        const_11770185301458007184070328025 13)) for i in range(n))
    return y_hat
```

A.1.2 *Random Walk with Momentum* algorithm output when including regularization terms

```
np.seterr(all = "ignore")
def predict(X, n):
        y_hat = sum(np.divide(np.add(X[i][0], X[i][0]), np.exp(np.subtract(n, i))) ...
        for i in range(n))
        return y_hat
```