University of Pretoria

COS710 - Artificial Intelligence

Investigative Report

The use of Genetic Programming in developing a predictive Load-Shedding model.

Adrian Rae - 19004029

28-03-2022

Abstract

In the course of this report, we explore the use of iterative evolutionary processes to develop and refine a predictive model for "Load-Shedding": a semi-structured regime of controlled power outages implemented in South Africa instituted to reduce strain on the national electrical grid. This investigation serves as an academic 'foray' of sorts, into the mechanisms, methodologies and decision-making processes underlying the practical uses of evolutionary methods for creating useful, real-world models that address a concrete set of unknowns. This constitutes the primary goal of undertaking this process: "To gain knowledge of the underlying mechanism, over-and-above the concrete application". The subject matter is thus intended to reflect such a set of unknowns prompting the development of a model and subsequently, this report should not be misconstrued as a venture to qualify the finer points of the Load-Shedding rationale, but rather to use the subject matter as a vessel for realising the benefits of evolutionary processes in model development.

Contents

1 Problem Statement

This task revolves around the use of Genetic Programming practises to evolve a model that can be used to make predictions of the current load-shedding state in South Afica, given a number of inputs are considered. An overview of the data, programming structures and evolutionary processes will be given, along with a motivation where required. The implementation of these structures will be considered, with the results evaluated for their efficacy, ability to generalize and consistency across particular evaluations.

2 Data and Addressing Biases

The datasets used for feature analysis and generating fitness cases were gathered at this site.

Three datasets were gathered from this repository, which each make up a selection of metadata that can be used to infer the resulting fields of fitness cases. [1]

2.1 Stage Data

`south_africa_load_shedding_history.csv`

The first dataset contains records which pair a datetime to a load shedding stage.

$$(d_n, s_n) \in \mathbb{DT} \times \{0, ..., 6\} \tag{1}$$

with datatimes ranging from 1/9/2015 11:47:19 to 9/7/2020 20:03:18 and stages 0 to 6. It should be noted here that while there are 6 stages present in the data, there are in fact, at the time of writing, 8 potential load-shedding stages being implemented in South Africa: the latter two being so severe, they have yet to be implemented in practise. The years 2016 and 2017 are excluded as they had not experienced any load shedding.

2.2 Electrical Production Data

`Total_electricity_production.csv`

This dataset maps the date to the electricity produced in South Africa in gigawatt hours, from 1985 to 2018, measured quarterly.

$$(d_n, x_n) \in \mathbb{D} \times \mathbb{R}^+ \tag{2}$$

2.3 Electrical Indicators

`Electricity_indicators.csv`

[1] See Feature Identification.

This is a compound dataset that provides electrical indicators for African countries across the years 2000 - 2020. There is no consistent record format, but one is able to discern periodic formats every 19 records, as a 19-record group forms a logical "3rd" dimension across which time and Country vary. That is to say, there exist 3 axes in this data: *Country*, *Year* and the *Property* in question, across which that data varies.

As the task necessitates the development of a model for South Afica, only the first 19 records of the file are of significant interest.

3 Problem and Solution Domain

In this section, we discuss the sub-spaces from which input and output of the model are derived. The formal identification of features that will be extracted from the dataset and used in the construction of models of varying complexity, will be delayed until the following section, on Feature Identification.

3.1 Target Values

A pertinent quality of the input data is that of the target value, the property of the input data that the model will be used to emulate, given other fields of input data. Differing tasks require considerations for varying target types and domains: such as those used for classification or continuous functional regression.

Here, it is evident that we are dealing with a classification problem: in that our predictive model should (upon some kind of final evaluation or mapping) produce either a concrete symbolic classification, or an index which maps to the resultant classification.

$$c_1, c_2, c_3, ..., c_k \tag{3}$$

For a fixed k classification classes. It is worth noting here, that if indices are used for the mapping, *i.e.* $1 \mapsto c_1, 2 \mapsto c_2...$, then the range of values used for the indices are particularly susceptible to *clumping* when the fitness metric of the model is not perceptive to the semantics of the output range. For example, if the targets are the indices $1, 2, 3$, then a model with a poorly-defined fitness metric might cause outputs to tend to 2, the median of classification indices, as it minimizes the predefined loss correctly, but does not align with the semantics of the output range.[2]

3.2 Input Values

Comparatively simpler, is the discussion around the input values used for the model. Based on the results of Feature Identification, one will arrive at a set of fields that comprise a subset of the input dataset records. An input to the model will thus be an ordered set of m values:

$$(x_1, x_2, ..., x_m) \in \mathbb{F}^m \tag{4}$$

Ranging across a field of values $\mathbb{F}$ that encompasses several type fields:

$$\mathbb{F} = \mathbb{F}_1 \cup \mathbb{F}_2 \cup ... \cup \mathbb{F}_n \tag{5}$$

[2]Two solutions to this issue are possible. One involves using a reduction function R, that maps $\{1, ..., k\} \mapsto \mathbb{V}$ for some field $\mathbb{V}$ and an inverse R^{-1} that accomplishes the reverse. The model is evaluated based on the range $\mathbb{V}$, but the output elements are mapped to the classification space using R^{-1} for the final result, such that a poorly defined metric is less susceptible to the linearity of the output range. Another solution is to instead treat the output targets as members of $\mathbb{U}^k$, where $\mathbb{U}$ is the open unit interval $(0, 1)$ and each component represent a *heuristic confidence* of each classification. The exact classification can be chosen as the component index, corresponding to the value with maximum norm.

That might include boolean, integer and real values $\{0,1\} \subset \mathbb{Z} \subset \mathbb{R} \subseteq \mathbb{F}$ as well as strings $\mathbb{S} \subseteq \mathbb{F}$, so long as the model includes a method for operating on such values [3].

[3]Realistically, this approach is computationally impractical. It relies upon consistent indexing and function structure that make any adaptation or generalization to different structures impossible. Thus, a dictionary approach is favoured: *i.e.* an input would preferably be treated as a dictionary functor LU which maps a keyword to a particular value:

$$LU(k_i) = x_i \in (x_1, x_2, ..., x_m) \tag{6}$$

This way, the indexing set $I = \{1, ..., m\}$ can be replaced with a generic indexing set $I = \{k_1, ..., k_m\}$. A tuple in itself is just a dictionary functor with the former as its indexing set, so this just provides a smarter, more general way of addressing input components in a way that doesn't rely upon order or size.

4 Feature Identification

4.1 Input Parameterization

4.1.1 The Foundational Model M_0

These are parameters that are necessary for a predictive model to determine a load-shedding stage at some point in time. This is developed from data in the first dataset.

- $t \in \mathbb{R}$: A unitless time quantity. For this task, we make use of a timestamp for the sake of its ubiquity and programmatic practicality.
- $s \in \{0, ..., K\}$: The suspected load-shedding stage at time t. The choice of seemingly including the load-shedding stage as part of an input parameter may seem strange, but is prompted by the explanation in the following sections.

4.2 Model Extensions

These parameters are not strictly essential for determining a load-shedding stage at some point in time, but allow for greater specificity in the model's parameterization, allowing for the model to arrive at a classification with greater accuracy.

4.2.1 First Model Extension M_1

This is an extension of the model M_0 used to improve model accuracy based on the knowledge of more underlying fields which influence the data. This is prompted by the data fields in the second dataset. **This is the model that will be used for the concrete evaluation to follow**.

- t, s: as defined in M_0.
- $p \in \mathbb{R}^+$: The electrical production in Gigawatt hours, at time t. [4]

4.2.2 Second Model Extension M_2

This is an extension of the model M_1 used to improve model accuracy based on the knowledge of more underlying fields which influence the data. This is prompted by the data fields in the third dataset.

- t, s, p: as defined in M_1.
- $p_d \in \mathbb{R}^+$: population density.
- $p_t \in \mathbb{R}^+$: population total.
- $g \in \mathbb{R}^+$: gross domestic product.

[4] Given that the datasets are not naturally structured in a way that permits "joining" them on time, a heuristic is needed to map actual values of p to t for developing fitness cases. In light of this, the closest time t_p to t, which corresponds to p will be used as the linking field.

- $g_+ \in \mathbb{R}^+$: gdp growth.
- $g_c \in \mathbb{R}^+$: gdp per capita.
- $e_t \in \mathbb{U}$: proportion population with electricity access (total).
- $e_r \in \mathbb{U}$: proportion population with electricity access (rural).
- $e_u \in \mathbb{U}$: proportion population with electricity access (urban).
- $c \in \mathbb{U}$: emissions from production.
- $u_c \in \mathbb{U}$: proportion production from coal.
- $u_h \in \mathbb{U}$: proportion production from hydroelectric.
- $u_g \in \mathbb{U}$: proportion production from natural gas.
- $u_n \in \mathbb{U}$: proportion production from nuclear.
- $u_o \in \mathbb{U}$: proportion production from oil.
- $r \in \mathbb{U}$: proportion renewable energy output.
- $t_c \in \mathbb{R}$: time to realise energy output.

While this model is particularly descriptive and seems the shining choice to model the input space on, we are left with some problems which need to be addressed:

- The dataset from which these fields are derived is largely incomplete. It would be near impossible to accurately and prescriptively create fitness cases which source such fields from the dataset as it would require either a NULL substitution strategy for each field, or the development of operators in the function set that act explicitly on NULL types.
- The lack of specificity in the joining field, being the time at which these results are derived, means it would be disingenuous to collate most of the results from this dataset with those in dataset 1, as they could be out of date by any amount within the period of a year.
- Finally, perhaps most relevant is the notion that our model should be accessible, making use of as few input parameters as possible to come to a concrete classification. It is unlikely that anyone who would make use of this model would have access to these data fields, or even an accurate approximation of such.

It is for such reasons that this models serves as an academic illustration: an ideal for input parameterization in a world where all the relevant data is accessible on demand [5], and will not be consulted further in this report in the context of implementation or evaluation.

[5] And cows are spheres.

4.3 Target Values

Following on from the last section, the target values are classes that represent the present stage of load shedding. Supposing there are K stages of load-shedding [6] including the *NULL* stage, we arrive at the following structure: an output of the model is a value

$$p_i \in \mathbb{U} = (0, 1) \tag{7}$$

with $i \in \{0, ..., K\}$ that represents the "certainty" of stage s at time t.

4.4 Motivating Factors

The choice to treat the model as a certainty index is not arbitrary (being mentioned earlier in the discussion on target values). This is done to ensure the semantic understanding of the task matches one that can be formulated in a deterministic, numerical process.

Suppose for the sake of illustration that a naive model takes in t, a time parameter, in order to output s_t, the concrete load-shedding stage at time t. If there are 9 load-shedding stages: s_0-s_8, one might define the error of a particular classification c as the numerical difference between the stages, which is aggregated as an average across fitness cases.

Here is where out first problem arises. Suppose the data used to develop this model is uniformly distributed across stages. It is entirely likely that the model will converge towards a function that outputs a constant stage for any time t. In this example, the model might output 4 as it is the median of the stages - minimizing the expected Euclidean distance with any other stage. By the notion of "fitness" given this model is well-adapted, but by the semantic understanding of the task at hand, this model is evidently a poor choice to use in practise.[7]

The choice of features used in this task are such, as they factor in multiple regressions made upon distinct stages to arrive at a classification. This changes the semantics of the question being asked from the incorrect *what stage s is expected to be closest to the real stage s_t at time t* to the correct *what is the certainty of a particular stage s being closest to the real stage s_t at time t*, which can be aggregated [8] to arrive at the more accurate stage classification.

4.5 Classification

Let us begin by establishing a model of M_0 that aligns with the above expectations, with K load-shedding stages, or $K + 1$ classification classes. M is able to take in an input $\{t \mapsto t_0, s \mapsto s_0\}$ and derive a numerical expression x_0, which is then reduced by a reduction function R to form the actual

[6] At the time of writing, there are 8 degrees of load shedding possible in South Africa, along with the *NULL* stage corresponding to no load shedding, for a total of 9 possible states. The generality is there to ensure the model is scalable.

[7] The cause of this discrepancy is subtle, but crucial to note: this form of model is useful for continuous regression, but not classification, as it particularly susceptible to changes brought upon by spatial errors.

[8] Via an *argmax* application

confidence $p_0 = R(x_0)$.

If s_t happens to be the actual stage corresponding to time t, the following steps are used to refine the model:

1. The *target* corresponding to $s = s_t, t = t_0$ will be $y_{target} = 1$.
2. A Boolean indicator of the classification is gathered as $y = \lfloor p_0 \rceil$.
3. A Boolean indicator of error is derived as $|y - y_{target}|$
4. The model is adjusted according to the refinement strategy based on this error and its aggregation strategy.

Given the same operation is taken on the other classification classes, one can derive the most probable stage of load shedding for some time t as follows:

$$c_t = argmax\{p_0, p_1, ..., p_K\} \tag{8}$$

4.6 Example

Now, let us form a concrete understanding by way of actual values, given a simple model of type M_0. Let $K = 8$, so that there are 9 classification classes. Let $t = 1648318499$. If there was load-shedding of stage-4 at this time, we would expect to see fitness cases

$$(\{t \mapsto 1648318499, s \mapsto 4\}, 1) \tag{9}$$

$$(\{t \mapsto 1648318499, s \mapsto s_0\}, 0), s_0 \in \{0, ..., 8\} - \{4\} \tag{10}$$

which would subsequently be used to refine the model M. Now suppose for $s_0 = 4$ that M generates $x_0 = 4.59$. We make use of a reduction function [9] to reduce this to $p_0 = 0.98$, indicating a substantially "strong" inclination toward this being a stage likely to occur at this time. Conversely, suppose for $s_0 = 2$ that M generates $x_0 = -4.59$. We reduce this to $p_0 = 0.01$, indicating a substantially "poor" inclination toward this being a stage likely to occur at this time.

Given all p_{c_i} are calculated for $c_i \in \{0, ..., 8\}$ as

$$P = \{0.16, 0.56, 0.01, 0.04, 0.98, 0, 78, 0.43, 0.89, 0.34\} \tag{11}$$

The most likely stage can be found by applying

$$c_t = argmax(S) = 4 \tag{12}$$

Indicating stage-4 to be the most likely stage given the time t. Further input parameters, such as those outlined in the sections above, do not affect the classification logic, but provide mechanisms for extending the specificity of the model in a way that allows for a classification of greater accuracy.

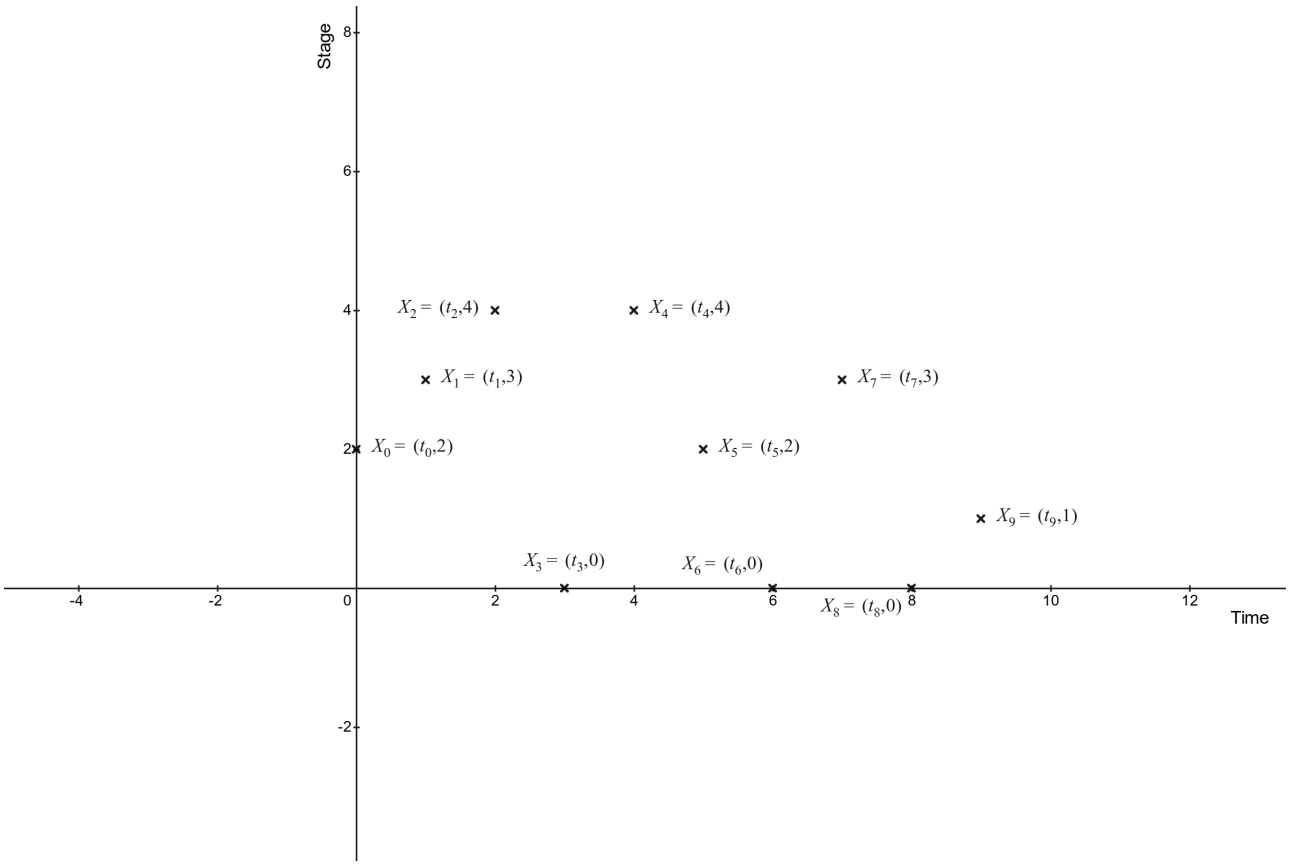
[9] In this case, the logistic function $\frac{1}{1+e^{-x}}$

4.7 A word on fitness cases

It should be noted here that an immediate concern is prevalent:

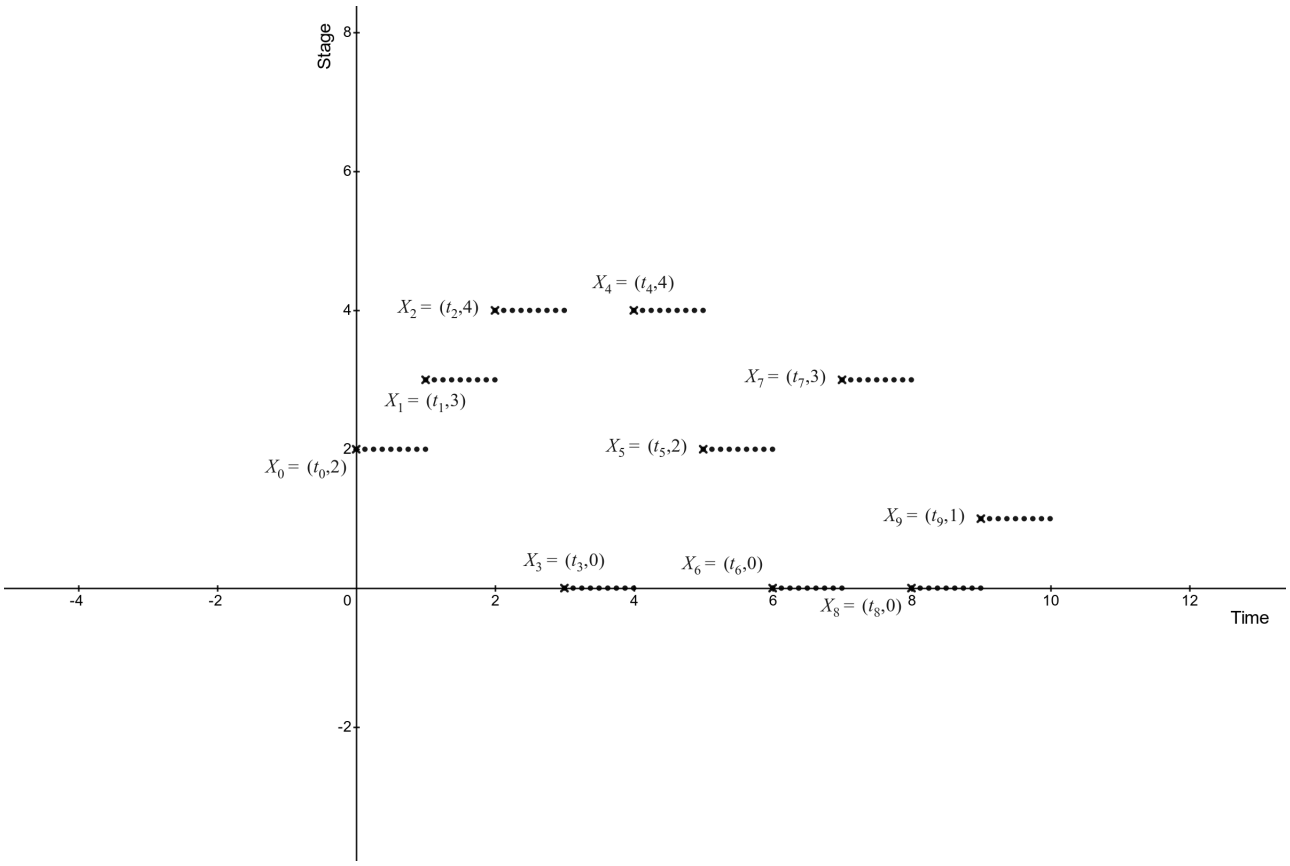
> The spareness of data will lead to the model being unable to form critical associations between variables and thus being unable to express the dicontinuity between stages which occur at time intervals. [10]

This *Logical Floor* deficit is illustrated as follows: consider the training cases for stages (y-axis) over time (x-axis) given below.

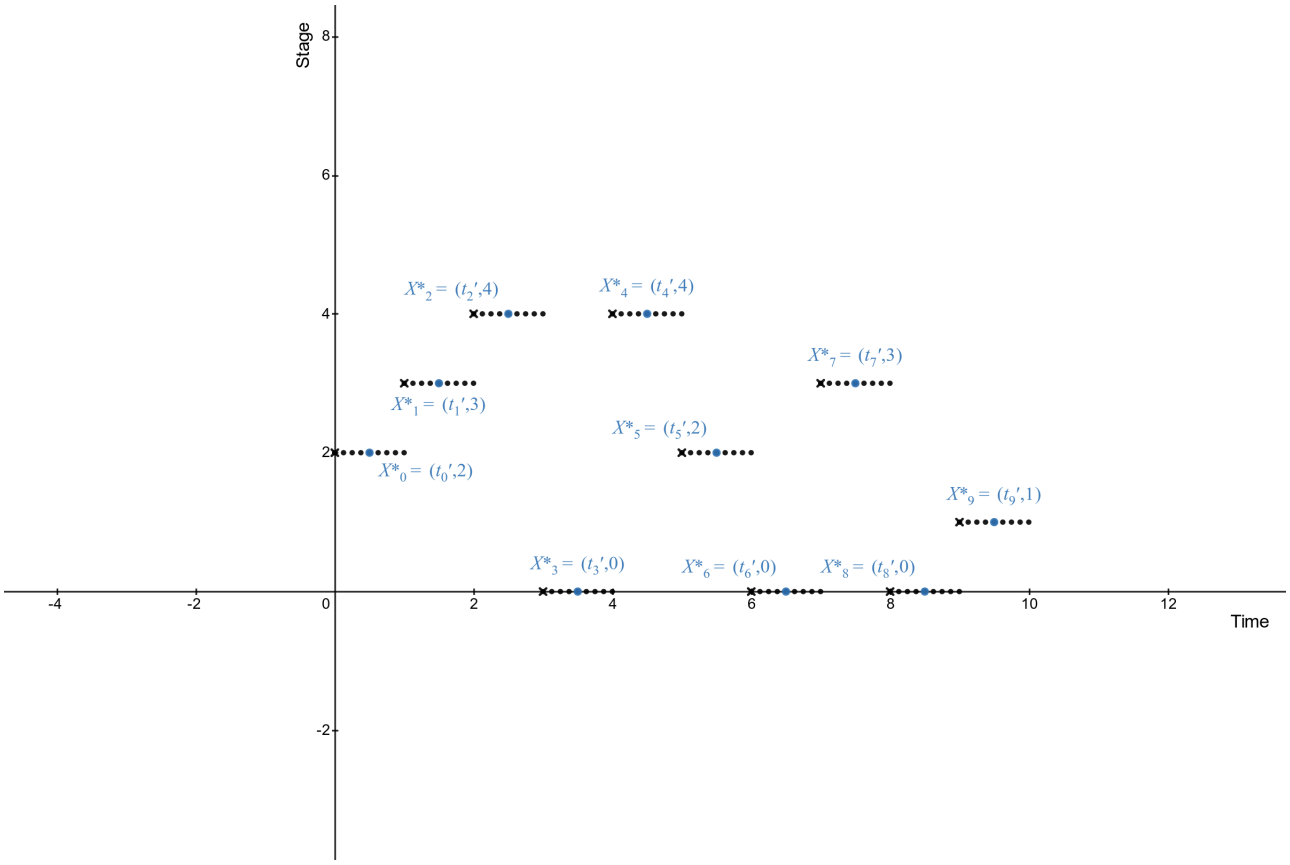


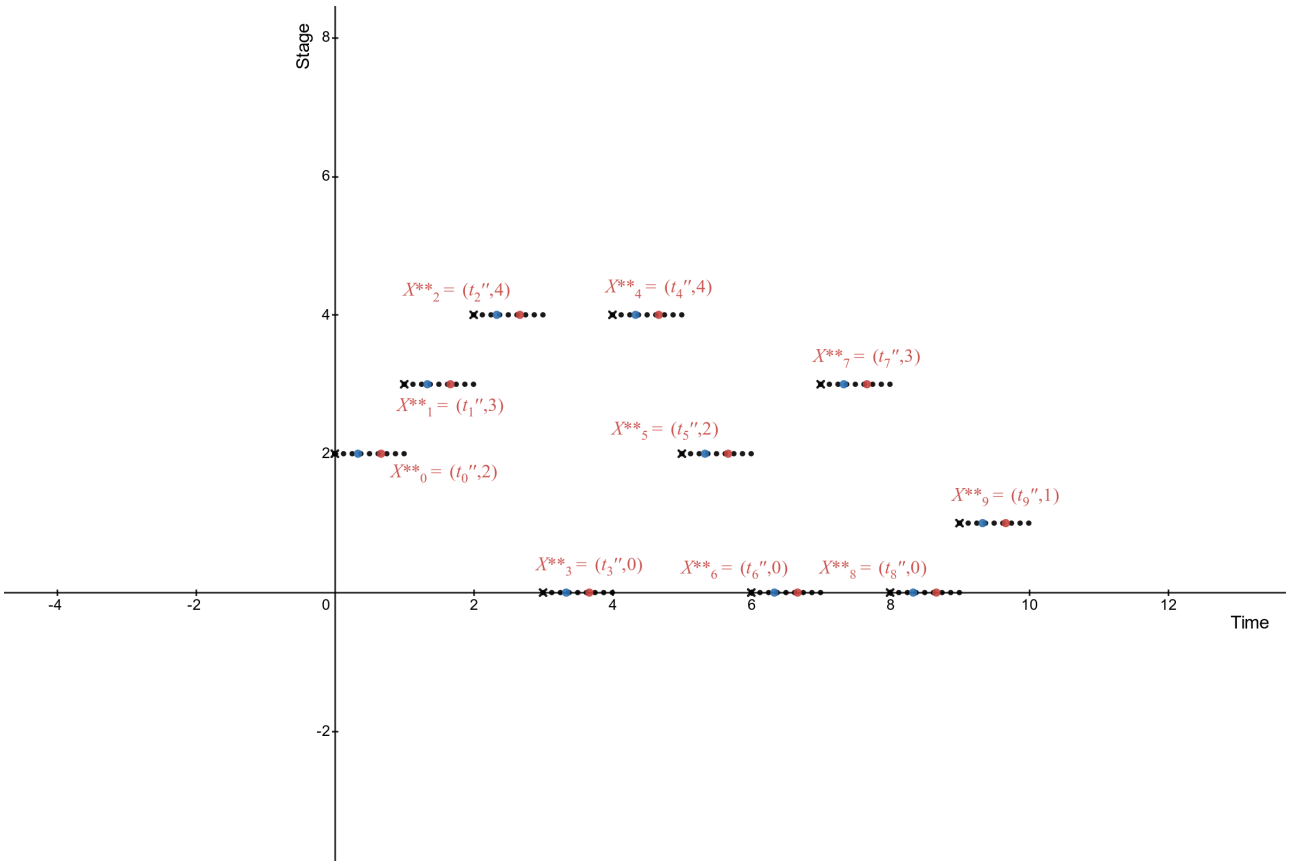
It is clear that, until a stage changes discontinuously, it remains constant. This is the *Logical Floor* that is present in the data. As humans, we can infer the underlying distribution to be:

[10] See the section mentioned in the hypothesis for greater elaboration.



But, this implicit data is not accessible to the model. As such, any continuous distribution that aligns with the given sample points could be inferred by the model. A solution to the problem is as follows: define a granularity k such that k new time points are inserted between every t_i and t_{i+1} with stage $s_0, s_1, ..., s_k$ corresponding to s_i. These new points are then treated as fitness cases in their own right. The following diagrams illustrate this for $k = 1$ and $k = 2$ respectively.





This is permissible as one can be assured these new fitness cases are totally accurate. This value will be known as the *insertion factor*.

5 Genetic Programming Structures

This section is intended to outline the implementation of evolutionary processes in a technology-agnostic environment. Some semblance of polymorphic structure and functional data types in the target environment are assumed, though are not strictly necessary, as either can be emulated with simpler constructs.

5.1 Atoms: Fundamental Elements

This section concerns Atomic elements: abstractions of elements that the final model makes use of in order to determine results. These are deemed "atomic", as there is no finer element utilized to represent the structure of an individual and each evaluation of an atom yields only a single output. In the case of the prototypical symbolic regression problem over $\mathbb{R}$, this would represent elements taken from the union of elements in $\mathbb{R}$, variables over $\mathbb{R}$, and functions which act over $\mathbb{R}^n$.

5.1.1 Atoms: Encapsulating an Atomic Element

The most fundamental class construct is that of an Atom: an abstract class used to model an indivisible unit which can be evaluated, either in isolation, or with the residual evaluation of other Atoms. This class serves purely as a generalization of structure, allowing elements like Constants, Operators and Variables to be loosely grouped under a common label, and is not used in practice. With this in mind, the class defines several key interface methods:

- **arity:** A function which returns the number of Atoms which need to be resolved in order to evaluate the Atom. This generalizes the notion of non-parameterized Constants and Variables with arity 0 (being functionally equivalent to a function of no parameters), and parameterized Operators, with arity > 0.
- **instance:** A function which creates an instance of an Atom. This is to distinguish between the symbolic representation of an Atom, which might be contained in a terminal or functional set and its actual, bound counterpart in an Individual. To illustrate, one might define a Terminal set $\{..., X, ...\}$, to include a Terminal X; however, the object bound to each individual need not be this *same* object, but rather an instance $X_{i1}, X_{i2}, ...$ for each individual.
- **eval:** A method skeleton which is overridden by subsequent sub-classes. This is used to resolve the non-symbolic value of an Atom. Exactly $arity(A)$ Atoms are needed as parameters for the evaluation to be successful.

Further methods are outlined in this class which accomplish:

- Symbolic evaluation of the Atom: allowing loose resolution of the structures comprised of Atoms.
- Validation of input parameters for consistency with prescribed *arity*.
- Creation of a distinct copy of the Atom.
- Providing indicators for contextual use, which are redefined by sub-classes.

5.1.2 Terminals: Variables, Constants, ConstantRanges

A generalization of an elemental property is useful for defining the aggregating structures which represent Individuals, though lack the specificity of any concrete meaning or purpose. It is for this reason that specialized structures are introduced to serve a role in the task of classical symbolic regression.

Terminals are the most basic extensions of Atoms, representing trivial expressions. Despite being a functional extension of Atoms, they are still abstract and thus used similarly to Atoms for the purpose of encapsulating elements that serve similar roles as Terminating elements. The class does however specialize the functions:

- **arity:** Always returns 0 as no other Atoms are needed to evaluate a Terminating expression.

Various sub-classes of Terminal are implemented to realize the roles within the standard symbolic regression problem. They do so by implementing specific behaviour of the Atom class. These are each outlined below.

Variables

These are Terminators that represent an unknown/unbound value within an encompassing expression. They are useful for parameterizing models and largely drive the need for symbolic regression. They can be evaluated symbolically without any external resolution, but require a concrete binding before they can be evaluated. New state and behavioural context introduced by Variables include:

- **name and value:** State parameters that define a variable. Names are used as keys for the sake of identifying variables when their binding is required or evaluation is needed. Values are internal quantities used during concrete evaluation and can be set when the variable is created, or bound right before it's evaluation.
- **bind:** Method to bind a concrete value to the Variable's value
- **eval:** Method which evaluates to the Variable's state value.

Along with several methods for cloning, setting contextual flags and argument validation which follow, mutatis mutandis, from the Atom class.

Constants

These are Terminators that represent a concrete value within an encompassing expression. They can be evaluated, both concretely and symbolically, without any external resolution. New state and behavioural context introduced by Constants include:

- **value:** State parameter that define a constant. Values are internal quantities used during concrete evaluation and can be set when the constant is created.
- **eval:** Method which evaluates to the Constant's state value.

Along with several methods for cloning, setting contextual flags and argument validation which follow, mutatis mutandis, from the Atom class.

ConstantRanges

Often, it is the case that one requires one expression to account for a wealth of possible concrete values. As a case-in-point, the canonical $+C$ of indefinite integration is a syntactic shorthand for a constant that can take on an infinitude of values along a range. A similar structure is needed for symbolic regression, whereby it would be inefficient and imprecise to specify an infinitude of Constants, but rather to specify a single *ephemeral constant* from which instances of concrete Constants are obtained. As such, ConstantRanges are Terminators that represent a single concrete value within a predefined range, in an encompassing expression. New state and behavioural context introduced by ConstantRanges include:

- **upper and lower bound, choosing function:** State parameters which specify the range of values a Constant represented by the ConstantRange, can take. A choosing function can also be specified to specify how a value along the range is chosen (if for example, the domain is not simply that of the reals).
- **instance:** Method to generate a Constant with values defined by the state range. It should be noted here that, as the instance returned is a Constant with a concrete value, ConstantRanges will never occur inside aggregating structures, nor have a need to be evaluated. This ensures that they can be used as generating elements but not evaluating elements: that is to say *a ConstantRange C can be treated as an element in the Terminal set of generating values, whereas an Individual which makes use of it as a generator only receives a concrete Constant C_i as an instance.*

Note that, upon reading the next section, you might come to realize this kind of functionality could be achieved similarly with a 0-argument function, *rand()*, which is predefined to select a single value from a range of values. I however, would caution against this, for the sake of the semantics that differ between the two approaches. It might be beneficial to think of certain constants being introduced in an expression, rather than the output of a function that seems to suggest a stochastic element is involved in determining a model's output.

5.1.3 Operators

Aside from Terminating expressions, the symbolic regression problem necessitates a construct which acts upon Terminating expressions as a Non-Terminating expression. Operators (an alias for "Function", though used to avoid a conflict with the term used as a reserved word in most languages) are another basic extension of Atoms, representing non-trivial expressions. Despite being siblings of Terminals, operators are concrete and thus are implemented in practise. New state and behavioural context introduced by Operators include:

- **name:** The state attribute name of an operator, which can be specified during creation. Can be used for symbolic evaluation.

- **evaluation_procedure:** The state attribute which specifies a function undertaken to evaluate an expression, based on a set of input parameters. This is passed in during creation and is essentially an instruction for how future inputs should create an output. This means that an operator is a Functional acting upon a set of future input values. As a tangible example, if one wanted to create an analogue of the + operator for $\mathbb{R}$, called "add", the input for this state attribute would be $(x_1, x_2) \mapsto x_1 + x_2$.

- **arity:** In contrast to other Atoms, will return the number of parameters specified as input to the *evaluation_procedure*. This shows that several (or no) outputs are required in order to concretely evaluate an Operator's expression. These will later be passed in, as the value of child nodes passed to the operator.

- **eval:** Evaluation now requires countably many arguments, which are passed into the *evaluation_procedure* specified.

Several methods for cloning, setting contextual flags and argument validation which follow, mutatis mutandis, from the Atom class, are also defined.

5.2 ParseTrees: The "Individual" element

5.2.1 Nodes: Containers for Atoms

Nodes are structures which allow Atoms to be embedded within individual elements. They are merely structures that enclose Atoms [11] and provide a consistent structure for allocating children to expressions that need to be evaluated in a hierarchy. In addition, they provide several notable operations which can be undertaken:

- **eval:** evaluate the numerical or symbolic value of a Node [12] based on the evaluation of its child expressions.

- **arity:** get the number of subexpressions that are evaluated before the Node can be evaluated.

- **get depth:** get the depth of the subtree rooted at this Node.

- **random node:** get a random node from the subtree rooted at this Node. Includes options for excluding the root and only selecting non-terminal Nodes.

- **indicators:** indicator functions for whether the Node is terminating, contains a variable within the subtree and so forth.

- **helpers:** helper functions for genetic operators, such as swapping nodes with a specific lineage, deep-copying and operators overloaded to indicate descendance.

[11] That are thought of as the Node's value.
[12] Or rather, its enclosed Atom.

5.2.2 ParseTree: Interface for Node Interaction

This is the primary individual structure at which the control model's evolutionary processes are targeted during refinement. The containing structure for a subtree rooted at a Node. This structure provides the same function as a that of the subtree rooted at the root Node of a tree, but defines certain class methods for individual creation and evaluation.

- **random generation:** randomly produce a tree given a specified max height and terminal and operator sets. Set selection fairness and trivial tree restrictions are implemented here.

5.3 PopulationGenerators: Creating sets of Individuals

This structure is tasked with producing a population of ParseTree individuals based on selected criteria. [13] The primary goals of this structure are outlined in the following method:

- **generate:** generate a population of specified size given a specific generation method [14] and fairness/triviality conditions. This operation can be massively parallelized in order to increase the computational efficiency of the evolutionary process.

5.4 FitnessFunctions: The Foundation of a Loss Landscape and Convergence to Solution

This structure handles evaluation of an individual given a predefined metric and aggregation for how well an individual evaluation matches that outlined in a set of fitness cases. The essential behaviour of this class is encapsulated in the methods:

- **bind case:** bind an individual fitness case to the fitness function buffer so that it can be used for batch training.
- **fitness:** evaluate an individual based on a prescribed fitness metric. [15]
- **predefine aggregate:** pre-calculate total adjusted fitness across a population such that normalized fitness can be calculated without evaluating the sum every time.

5.5 Selectors: The Core of Selecting Parents from a Population

This is a structure that allows a subset of a population to be chosen as parents given preferred criteria. Fundamentally, this structure is relied upon for the behaviour:

- **select:** select a group of individuals from the population by way of a pre-established fitness function, a passed-in population and selection strategy. [16]

[13] See section on Parameterization.
[14] Such as GROW, FULL and RAMPED50-50
[15] Such as RAW, ADJUSTED and NORMALIZED
[16] Such as the TOURNAMENT and FITNESS_PROPORTIONATE schemes.

5.6 GeneticOperatorSets: Implementing Operators on Individuals

This structure is used to transform a collection of individuals into a new collection under some prescribed genetic operation. This behaviour is encapsulated in the operation:

- **operate:** use a predefined genetic selector to select a subset of the population as parents, from which a specified genetic operation [17] is undertaken to produce a collection of children.

5.7 ControlModels: Putting Evolutionary Processes into Action

This structure encapsulates all the behavior of the mechanisms listed above and provides a consistent interface for evolving a population over a number of iterations. The fundamental behaviour of the structure is laid out in the methods:

- **bind fitness cases:** provide external fitness cases that the model can use to refine individuals.
- **evolve:** run through an iterative cycle of selection, operation and evaluation under the conditions specified, until specific convergence criteria are met. This returns the optimal individual upon termination.

5.8 Classes in Context: An implementation of the above structure

As mentioned earlier in this section, the language chosen to implement the constructs defined above need not be expressive in its ability to encapsulate polymorphism and functional data types; though for the sake of readability, writeability and orthogonality, it is wise to choose one which fully encompasses such concepts. Better yet, as the performance of this evolutionary modelling is heavily dependent on the scale of the population in question, it would be wise to select a language whose execution can, either at compile or runtime, be significantly optimized by way of inline reductions or massive parallelism.

For such reasons, I have chosen to make use of Python as the target language for exacting these processes. In addition, the Python syntax allows for easy comprehension of program semantics, generator comprehensions, as well as a convenient keyword and array packing argument-passing scheme that allows for the rapid portability of methods and classes between operand sets of varying size and type.

The source code can be found here.

5.8.1 Terminal Implementation

Defining Terminal types based on the schema described above is simple. Here is an example each of a Variable, ConstantRange, and pure Constant definition.

```
x: Variable = Variable("x")
alpha: ConstantRange = ConstantRange(-1, 1)
one: Constant = Constant(1)
```

[17] Such as the CROSSOVER and MUTATION operators.

5.8.2 Operator Implementation

For example, this allows one to define Operators in the follow manner:

```
mult: Operator = Operator("*", lambda a, b: a * b, rep="({} * {})")
add: Operator = Operator("+", lambda a, b: a + b, rep="({} + {})")
sub: Operator = Operator("-", lambda a, b: a - b, rep="({} - {})")
sine: Operator = Operator("sin", lambda a: math.sin(a))
```

Notice how, for the sine function, no optional representation is specified because the default functional notation *sin(x)* will be used for its symbolic evaluation, instead of some other, natural, infix operation which would have to be specified explicitly.

5.8.3 Generating Sets

These Atomic elements can easily be grouped to form generating Terminal and Function sets:

```
t_set: List[Terminal] = [x, one, alpha]
o_set: List[Operator] = [mult, minus, add, sine]
```

5.8.4 Fitness Cases

Fitness cases can be simply specified as tuples, with the first component being a dictionary that maps variable names to input values. The second component is the observed (or target) value corresponding to the conditions laid out by the first component.

```
cases = [
    ({"x": 0}, 0),
    ({"x": math.pi / 2}, 2),
    # ...
]
```

5.8.5 Population Generation [Background Process]

Creating an Initial Population Generator is achieved by passing in the relevant Terminal and Function sets.

```
population_generator: PopulationGenerator = PopulationGenerator(t_set, o_set)
```

Which can be used to create a list of Individuals by way of passing in the necessary parameters.

```
population: List[ParseTree] =
population_generator.generate(
    population_size,
    max_tree_depth,
    generation_method # e.g. PopulationGenerator.Method.GROW
)
```

5.8.6 Fitness Function [Background Process]

A fitness function can be generated as follows:

```
fitness_function: FitnessFunction = FitnessFunction(
    fitness_objective, # e.g. FitnessObjective.MINIMISE
    error_aggregator, # e.g. lambda S: sum(S) - sum of case error
    error_metric, # e.g. lambda y, t: abs(y - t) - error r.t. target
    maximising_max_fitness,
    equality_threshold
)
```

With fitness of a specific individual *ind* calculated as:

```
fitness_function.fitness(
    individual=ind,
    measure=FitnessMeasure.NORMALIZED # or any other target measure
)
```

5.8.7 Genetic Selector [Background Process]

A genetic selection mechanism can be established as follows:

```
genetic_selection: Selector = Selector(
    fitness_function, # a pre-establised fitness function instance
    selection_method  # e.g. SelectionMethod.TOURNAMENT
)
```

Whereby a parent can be chosen from a population as follows:

```
parent1 = genetic_selection.select(population)
```

5.8.8 Genetic Operator Set [Background Process]

A Genetic Operator Set can be established as follows:

```
genetic_operator_set: GeneticOperatorSet = GeneticOperatorSet(
    genetic_selection, # a pre-establised genetic selector instance
    population_generator # a pre-establised population generator instance
)
```

A specific set of Genetic Operators can be established, following which a random operator can be applied to a collective of parents from a population as follows:

```
# choose a random operator
genetic_operator = random.choices(
    genetic_operators,
    weights=genetic_operator_weights,
    k=1
)[0]
```

```
# apply it to a collective of parents to form a new collective of offspring
new_subset = genetic_operator_set.operate(
    population,
    genetic_operator # e.g. GeneticOperatorType.CROSSOVER
)
```

5.8.9 Control Model

As the Population Generation, Selection and Genetic Operator classes are employed by a control model, all interfacing can be done by establishing a control model and feeding in parameters, instead of creating all the evolutionary components manually.

```
    # Create a control model
    model: GenerationalControlModel = GenerationalControlModel(
        # fill in several keyword arguments
        population_size=100,
        max_tree_depth=5,
        terminal_set=t_set,
        # ...
    )

    # specify fitness cases for binding
    for args, target in cases:
        model.bind_fitness_case(target, **args)

    model.evolve(
            # evolve population with actions for each iteration and on converge
    )
```

5.9 [Extra] Data Set Loading

I make use of a separate module, DatasetManager, to handle the loading of dataset records as well as the formation of fitness cases based on the parameterization defined above. This is also used to generate the model variable set based upon the parameters used as data fields. As this essentially automates the process of loading data into the control model, it will not be further discussed here; save some key functionality:

- The dataset sources are hard-coded, but can be overridden by a system argument array, to specify the data sources.
- There exists an enumerated list of model types from which one can select a target model, in order to generate the relevant fitness cases.

6 Hypotheses

6.1 Training Data

I believe the training data made available for this task to be unsuitable for the development of a truly generalized, predictive model. While my concerns over the third dataset have aleardy been voiced [18] and the validity of model M_2 challenged, there are other, more general concerns that I have yet to voice:

- **The Load-Shedding schema is wildly inconsistent:** while the training data is well-structured and presents a wide-reaching chronological perspective of stages and electrical production in general, it does not take into account the many factors that have influenced the legislation behind the load shedding strata, along with what stages have been in consideration at what point in time. To clarify, it is disingenuous to say that *stage-4* at this point in time [19] (where there are currently 8 stages) would be equally as severe as the *stage-4* of 2015 (where there were only 4 stages). [20]

- **There is a proclivity for under-fitting due to the type of ranges involved:** The dataset only accounts for changes between stages and does not account for the implied *floor* between points in time. Case-in-point: suppose there are time-stage pairs $(t, s) = (t_0, s_0), (t_1, s_1)$. What stage should be relevant if $t_c \in (t_0, t_1)$ is chosen? It is clear to us that as stages change at t_0 and t_1, the relevant stage s_c at t_c should be s_0, as we have yet to encounter the discontinuous jump to s_1 at t_1. This is the implied *floor*. However, how would the model be perceptive of this? If $s_0, s_1 = 0, 6$, what course would the model have to predict $s = 0$ and not any other value in the range, along some distribution that begins at s_0 and terminates at s_1? If one follows the *GIGO* principle, it should be assumed that any lack of accuracy in the final model for such t values between those listed in the training set, are due to this lack of inference. [21]

6.2 Fitness Function

I expect the fitness function outlined in the methodology to follow will be suitable for the application to this task. There is a clear incentive to minimize the error in to the model's evaluation relative to a batch of training cases, but this does not take into account the structure of the individuals themselves: that is to say, should the expression itself contribute towards some notion of fitness, as opposed to the result it evaluates to? [22]

[18] See my discussion here.

[19] Presently, March 2022

[20] One might argue that the model is intended to be perceptive of this: I disagree. I find this akin to trying to develop a calculator where the value of any numeral changes over time - what would the output $3 + 4 => 7$ infer?

[21] A possible solution to this is mentioned here.

[22] I say, it should. As will later be mentioned, I allow fitness to be swayed by the presence or lack of parameterization in a tree: deeming the fitness of constant trees to be incredibly poor. That being said, without providing semantic rules for expression simplification, there is nothing to stop the model from deeming $\frac{2.58 \times x}{x} = 2.58$, a constant tree (that is still parameterized) from being the convergent solution.

7 Evolutionary Methodology

7.1 Terminal Set

These form terminating expressions in the ParseTree individuals generated. The set of Terminals is comprised as the union of Variables, Ephemeral Constants and True Constants:

```
t_set: List[Terminal] = V + E + C
```

The varieties of Terminal are as follows:

7.1.1 Variables

The variable set will be entirely determined by the DatasetManager and the choice of model selected[23], being automatically loaded into the control model. Supposing a simple model is used, the variable set developed will be:

```
V = [ Variable("t"), Variable("s")]
```

Which represent the time quantity t and suspected load shedding stage s, respectively.

7.1.2 Ephemeral Constants

This represents a symbolic constant whose specific value is bound upon instance in an individual ParseTree individual. This is specified explicitly and is independent of any set model.

For this task, it is important to weight the value of nested computations against the lack of specificity that comes from a large range over which the constant can range. *i.e.* Suppose the model would benefit from the true constant value of 2 appearing in a subtree expression, but the constant α only covers values in the unit interval $\mathbb{U}$. Depending on the choice of functions in the function set, it may take 2, 3, or countably many levels of subtree evaluation for bindings of α to evaluate 2 [24], whereas allowing α to extend to a larger range could allow it to assume the value 2 directly, but with a lower probability of assuming any other constant in some specified range.

It is for such a reason, that I create an ephemeral constant α that covers the range $(-2, 2)$: allowing signed fractional expressions to be derived quickly, but allowing real values with large modulus to be derived with functors taken on those values of α larger than 1.

```
E = [ ConstantRange(-2,2) ]
```

7.1.3 True Constants

Despite the presence of an Ephemeral Constant, it might be useful to include a True Constant, which is bound to a singular value [25], that could even be included in the range of the Ephemeral Constant.

[23] See here for more information on what variables are used for particular models.

[24] *e.g.* (0.3 + 0.7) / 0.5

[25] *e.g.* If a model converges to the function $\frac{1}{x}$, then it is unlikely that an ephemeral constant will, over the course of only a few generations, be bound to a value that is near to the value 1. It makes sense to have a Constant with value 1 that could immediately be bound to a subtree expression.

For this task, it is reasonable that 1 would form an appropriate Constant, for the direct evaluation of unit offsets and inverses. The additive identity 0 does not seem appropriate in this regard as most arithmetic expressions involving 0 evaluate to other expressions listed in this section and so it would be computationally impractical to facilitate its use. Furthermore, as trigonometric functions will be heavily employed, it is wise to include (an approximate) value of π [26].

```
C = [ Constant(1), Constant(Math.pi, "pi") ]
```

7.2 Function Set

Now that we have a set of consistent Terminating expressions, we develop mechanisms for combining them into non-terminating expressions.

7.2.1 Standard Operators

These are operators that are necessary for creating polynomial expressions, the most basic algebraic structures.

- Addition: standard additive group action on $\mathbb{R}^n$.
- Multiplication: standard multiplicative group action on $\mathbb{R}^n$.

```
mult: Operator = Operator("*", lambda a, b: a * b, rep="({} * {})")
add: Operator = Operator("+", lambda a, b: a + b, rep="({} + {})")
o_set: List[Operator] = [mult, add]
```

7.2.2 Extended Operators

These are operators included to allows group inverses:

- Subtraction: standard additive group inverse on $\mathbb{R}^n$.
- Division (Safe): standard multiplicative group inverse on $\mathbb{R}^n$. If the later input is 0, the function reduces to the constant 1. This ensures that there is no functional expression that evaluates to an infinite quantity.

```
divs: Operator = Operator("divs", lambda a, b: a / b if b != 0 else 1)
sub: Operator = Operator("-", lambda a, b: a - b, rep="({} - {})")
o_set += [divs, sub]
```

These are operators included to provide specific behaviour on terminating expressions:

- Sine: included to provide periodic behaviour on the target field. [27]

[26] Though, e is extraneous due to the the more practical use of the *exp* function.
[27] All trigonometric functions reduce to functions of sinusoids.

- Logarithm (Safe): included to allow massive value reduction based on the information contained in an expression, in *nats*.[28] If the argument is non-positive, the function reduces to the constant 0. This ensures that there is no functional expression that evaluates to an infinite, undefined quantity.

- floor: returns the greatest integer expression k less than a target expression x. [29] This is useful for approximations of expressions and allow faster convergence to integer expressions.

```
sine: Operator = Operator("sin", lambda a: math.sin(a))
logs: Operator = Operator("logs", lambda a: math.log(a) if a > 0 else 0)
floor: Operator = Operator("floor", lambda a: math.floor(a))
o_set += [sine, logs, floor]
```

7.3 Reduction

Reduction is the process used to transform a real-valued output into a probabilistic value within the unit interval $\mathbb{U}$. This is otherwise known as a sigmoid function. Various choices of such a function are possible:

$$R_{logistic}(x) = \frac{1}{1 + exp(-x)} \tag{13}$$

$$R_{tanh}(x) = \frac{1 + tanh(x)}{2} \tag{14}$$

$$R_{arctan}(x) = \frac{2 * arctan(x)}{\pi} \tag{15}$$

For this task, I make use of $R_{tanh}(x)$ as it is directly operable on large values of x, without an intermediate operation that would overflow, that a reduction like $R_{logistic}(x)$ would be susceptible to.

7.4 Control Model Parameters

This section is intended to outline parameter choices that the Control Model utilizes in order to perform tasks and refine the model being generated.

7.4.1 Population Generation

- **Terminal and Operator Sets:** As prescribed above.

- **Population Size [50]:** The amount of initial members generated. For the sake of computational efficiency as well as allowing the evolutionary process to take place without initial bias, I have chosen 50 members as an appropriate figure.

- **Max Tree Depth [15]:** The maximum nested depth of a generated individual. For the sake of computational efficiency as well as allowing the evolutionary process to take place without initial bias, I have chosen 15 levels of functional nesting.

[28] All logarithms are scalar multiples of the natural logarithm *log*.
[29] By $k = sup\{i \in \mathbb{Z} : i \leq x\}$. *round* and *ceil* are derived by way of *floor*.

- **Generation Method [GROW]:** This method is chosen over the FULL and RAMPED50-50 schemes as the structure of a potential solution is largely unknown and thus convergence would benefit from members of greater depth diversity. It is known from the outset that the eventual model will likely exhibit more complexity then a model comprising only a few levels of nested expression can account for. Thus, we opt for a scheme that is computationally most efficient for generating members.

- **Set Selection Fairness [False]:** Force the likelihood of selecting a terminal to be equally as likely as a non-terminal expression. [30] This is useful if there are few operators in comparison to terminals. In this case however, there are fewer terminals than operators and thus no need for explicit fairness.

7.4.2 Fitness Function

- **Fitness Objective [MINIMIZE]:** The objective of the function. We are minimizing as we wish to reduce the error between our classifier and the target classification across a batch of fitness cases.

- **Error Aggregation [$S \mapsto \frac{sum(S)}{len(S)}$]:** An aggregation of a set of error values across fitness cases. In this case, the mean of errors is taken as it accounts for the possibility of differing amounts of fitness cases being used to refine the model for differing inputs.

- **Error Metric [$y, y_{target} \mapsto |R(y) - y_{target}|^2$]:** The function used to measure error between the evaluated expression and the target expression: in this case, the square distance. The model returns an unbounded value in $\mathbb{R}$, but the target y_{target} is a boolean indicator, so the output is first reduced by R to return a probability in $\mathbb{U}$ which is then compared to the target.[31]

- **Maximizing Max Fitness [10^4]:** The maximum fitness value assumed by the function when maximizing. The signed quantity is used as a penalty for the following property.

- **Allow Trivial Expressions [False]:** A heuristic used to apply penalties in fitness to those individual that are not parameterized by a variable. This is a heuristic: that is, the Fitness Function error metric has no notion of the Tree structure, only the final evaluation and so this factor is needed to ensure a semantic governing the type of individual that is suited to the type of problem. I set this to true to ensure that the **individual used as a solution must be parameterized by a variable.**

7.4.3 Genetic Selection Mechanisms

- **Selection Method [TOURNAMENT]:** The strategy used to select parents from an existing population. I make use of tournament selection[32] as it ensures genetic diversity across generations while also preserving a strong component of fitness.

[30] Accomplished by using weighted selection over normal stochastic selection.
[31] Notice, the metric and aggregator together imply that a MS - mean-squared strategy is being used.
[32] In favour of Fitness Proportionate, which is also implemented.

- **Selection Proportion [0.3]:** The proportion of the population to use as subset for a tournament. 0.3 is utilized as it ensures that the fittest individual is included in every 3 of 10 selections on average, while offering enough scope for genetic diversity to be maintained, even when the control model begins to converge towards a solution. This also ensures that some genetic operators can reduce to reproduce with the same frequency, on average.

7.4.4 Genetic Operators

These are the mechanisms that allow the traversal of the solution space. The following operators act upon a tuple of parents taken from the population, in order to produce a tuple of children. In a naive strategy, every genetic operation takes place with equal likelihood: however, it is evident that one gains a degree of specificity by allowing operators to be chosen with some priority, as such, each operator is given a *weight* which corresponds to its frequency of occurrence. The weight is listed alongside the operator.[33]

- **Crossover [4]:** The most common of operators. Chosen as it allows exploitation of a local optimum in the solution space. It is given the highest weighting as it is one of the only exploitation operators and, given that it is chosen most frequently, is likely to reduce to Reproduction if identical crossover points are chosen. [34]

- **Mutation [3]:** Randomly alters an expression in an individual subtree. Is a crucial explorative operator which allows both negligable and notiable affects to the fitness of expressions and is chosen to allow consistent exploration of the solution space. This is the only application of the generation of trivial subtrees.

- **Permutation [2]:** Swap the order of countably many operands evaluations in an expression. Useful in the context of deriving inverses and reciprocals from expressions without mutating the underlying expressions.

- **Editing [2]:** Reduce a non-parameterized subtree to its simplest form. Used to ensure trivial models are eliminated from the genetic pool and to ensure computational efficiency across generations. Reduces to Reproduction when no editing is possible.

- **Inversion [2]:** Swap two non-dependent subexpressions in a tree. Reduces to Reproduction when there are no non-dependent node pairs.

- **Hoist [1]:** Promote a tree's non-trivial subexpression to a new tree in the population. This is useful for exploratory purposes but also hinders convergence as the child produced is often ill-suited to a complex relationship it's parent was likely suited to. This does, however, allow many extraneous non-terminal expressions [35] involving constant to be whittled away prior to evaluation.

[33] The weighting is assigned according to the operation's likelihood to reduce to Reproduction, the computational effort required to undertake the operation as well as the operator's relative usefulness is aiding exploration or exploitation.

[34] See here, for the reason why.

[35] Like those reduced by the Editing operator.

7.4.5 Control Model

- **Seed [<Input>]:** Used to assign a seed to the python RANDOM module such that control model processes are repeatable. Leave blank for a time-based seed.
- **Iteration Threshold [20]:** Number of iterations to undertake before convergence is deemed to be forced.
- **Explicit Convergence Condition [$f \mapsto f > 0.95$]:** The condition under which convergence is deemed to have occurred, based upon maximum fitness in the population. In this case, I claim convergence is attained if the maximum adjusted fitness is in excess of 0.95. Note, convergence is also assumed if the member with maximum fitness remains the same after $\lfloor \frac{N_{Iterations}}{10} \rfloor$ iterations occur.
- **Parallelization [5]:** Number of concurrent threads assigned to process the initial population generation and genetic operator application phases of the evolutionary processes.

7.5 Training Methodology

I will make use of a 50-50 Training/Testing subset scheme of fitness cases, with an insertion factor of 5 to counter the *Logical Floor* issue. [36]

[36] The problem statement is mentioned in this section, with the solution being described here.

8 Results

8.1 Training And Testing Accuracy Thresholds

The following table gives an overview of the performance of the evolutionary process, as measured by the accuracy $p \in \mathbb{U}$ of the the model when applied to a training and testing set, when 10 different seed values are used to seed stochastic processes used in the control model's operation.

Seed	**Training Set Accuracy**	**Testing Set Accuracy**
0	0.870586480	0.869711496
1	0.850347962	0.848303583
2	0.834431905	0.837687553
3	0.871302283	0.879582035
4	0.872905180	0.867090492
5	0.832749702	0.825047094
6	0.870349430	0.878051545
7	0.835863132	0.837582149
8	0.873096696	0.879039815
9	0.858274136	0.856732935
Max	0.8730	0.8795
Mean	0.8570	0.8578
St. Dev	0.0163	0.0188

9 Conclusions and Points of Consideration

I believe this model offers an appealing insight into the nature of Genetic Processes applied to the creation of a real-world, predictive model. While the accuracy of the models this ecosystem outputs happen to be consistently decent at correctly identifying the stage corresponding to a specific point in time, a few observations and thoughts should be voiced:

- This should not be taken as a fit-all solution: this process is heavily dependent on the data that was used to train the resultant models. Even though separate training and testing sets were used for this evaluation, all the fitness cases used are derived from the same chronological period. It is unclear, thus, whether the models generated would generalize well to parameters outside of the domain of training (like the current Timestamp) as boundary conditions on the model have not been imposed.

- As far as the practicality of this mechanism as a problem-solving technique is concerned, the sheer computational power required to leverage results from the models produced is ludicrous, unless the task that necessitates it is truly worth investing the time and resources: traditional regression practices, or even other heuristic-oriented approaches would be preferable, in the context of this task and the inconsistency of the training data utilized.