Predicting El Niño Characteristics using Recursive Models, Data Trees, and an Evolutionary Algorithm

Nathaniel Kim

Palos Verdes Peninsula High School

February 17, 2016

**Table of Contents**

Introduction……………………………………………………………………………….. ...1

Hypothesis…………………………………………………………………………………. ..5

Materials and Methods…………………………………………………………………….…6

Results………………………………………………………………………………………..9

Discussion and Conclusions…………………………………………………………………13

Future Research……………………………………………………………………………...15

Acknowledgements………………………………………………………………………….17

References…………………………………………………………………………………...18

Appendix…………………………………………………………………………………….20

**Abstract**

The objective of this project was to predict future values of El Niño characteristics using recursive equation models, data trees, and an evolutionary algorithm. In addition, the project attempted to find how number of generations and size of the tree influenced accuracy of the predictions.

In order to generate a final model and test its accuracy, a Python program was written and run. First, the program generated a number of models using a data tree. The program then tested all the potential models against training data. Poor models were removed, new models were generated as replacements, and random changes were carried out. This process of testing and changing the set of models, called a generation, was repeated various times. Finally, the best model throughout the entire process was outputted, and its accuracy was measured by comparing its predictions to testing data.

3 different characteristic values of El Niño were tested. These values measured air pressure, sea surface temperature, and precipitation levels each in different areas. For each value, the program was used to generate and test models after 10, 50, and 100 generations. Models created with different tree sizes were also generated.

Ultimately, this program had success in predicting future values of 2 El Niño characteristics, but also did not succeed at predicting the temperature characteristic. In addition, number of generations had little effect on accuracy, while increased tree size led to increased accuracy. The successes of the model did satisfy the objective and also indicate potential for improvements in future testing. Furthermore, the program’s generality indicates possible success both in predicting other El Niño characteristics and in predicting any other time series as well.

**Introduction**

El Niño, one of the major meteorological phenomena of today’s world, is a period in which ocean temperatures in the Equatorial Pacific are unusually warm (Tropical Ocean Atmosphere Project) and is more formally defined as the occurrence of a “sea surface temperature departure from normal (for the 1971-2000 base period) in the Niño 3.4 region greater than or equal in magnitude to 0.5 degrees C (0.9 degrees Fahrenheit), averaged over three consecutive months” (NOAA News Online). It is characterized by “unusually high atmospheric sea level pressures… in the western tropical Pacific and Indian Ocean regions, and unusually low sea level pressures… in the southeastern tropical Pacific” (Tropical Ocean Atmosphere Project). El Niño causes increased rainfall across the southern tier of the US and in Peru, which has caused destructive flooding, and drought in the West Pacific, sometimes associated with devastating brush fires in Australia. Unfortunately, damages, such as that shown in Figure 1, linked to El Niño can cost billions of dollars (Pielke and Landsea 2030), making methods of predicting El Niño so important.

Figure 1: Damage caused in Malibu by the 1997-1998 El Niño. (Source: L.A. Times)

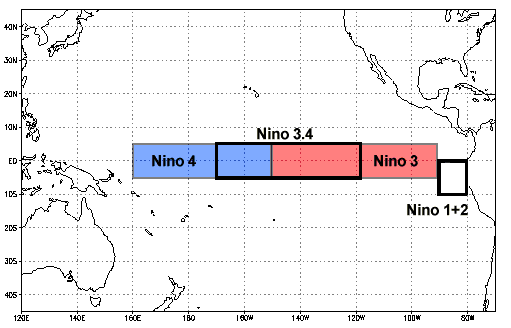
One of the most important facets of El Niño predictions is the use of indices. An index is “a number scale in which all the individual factors needed to describe a complicated phenomenon are boiled down to a single number” (Barnston) and is easy to track and analyze over long periods of time. Due to this, indices are common and widespread ways to summarize El Niño status. There are many different El Niño indices used in practice today, including the Southern Oscillation Index (SOI), sea surface temperatures (SSTs) in different regions, and the Multivariate ENSO Index (MEI). The SOI is a “a standardized index based on the observed sea level pressure differences between Tahiti and Darwin, Australia” (National Centers for Environment Information: Southern Oscillation Index). Negative values of the SOI indicate a higher air pressure in Darwin than in Tahiti and often indicate El Niño episodes (Australian Government Bureau of Meteorology). SSTs of regions in the Equatorial Pacific Ocean are also commonly used as El Niño indices. One such region which is extremely important is the Niño 3.4, given by the region ranging from 120° W to 170° W and 5° N to 5° S (NOAA News Online) and shown in Figure 2. The SST of this region is also known as the Niño 3.4 (National Centers for Environment Information: Equatorial Pacific Sea Surface Temperatures), which is the definition which shall be used for the rest of this paper. The Japan Meteorological Index, or JMI, is a similar index. It measures recent average temperature anomalies in the region given by 90° W to 150°W and 4° N to 4° S (Álvarez *et al*.). Unlike the previous three indices, the MEI uses many different variables in its calculations (as the name suggests). Specifically, it incorporates “sea-level pressure, zonal and meridional components of the surface wind, sea surface temperature, surface air temperature, and total cloudiness fraction of the sky” (Wolter).

Figure 2: The Niño 3.4 Region (source: National Centers for Environment Information)

There are many different methods which can be used to develop El Niño predictions, one of which is the use of an evolutionary algorithm. Evolutionary algorithms have many uses, one of which is the development of a predictive model. In this use, evolutionary algorithms usually follow this basic structure (What is an Evolutionary Algorithm?):

1) Generate many random models

2) Test each model for effectiveness

3) Create new models using characteristics of the better models of step 2

4) Apply mutations by randomly changing characteristics of present models or by adding random new models

5) Delete models with low effectiveness

6) Repeat steps 2 through 5 (known as a generation) until a desired effectiveness is reached

As can be seen, the evolutionary algorithm is named as such due to its similarity with Charles Darwin’s theory of evolution (Jones 1).

Data trees are a fundamental part of the model used in this project. Trees consist of many objects called nodes. One of these nodes is designated as a root. All other nodes are connected to by a directed edge from one, and only one, other node. This directed edge connects from the parent to the child. A node which has no children is called a leaf. Finally, the depth of a node is defined by the number of directed edges between the root and this node. (Carnegie Mellon School of Computer Science)

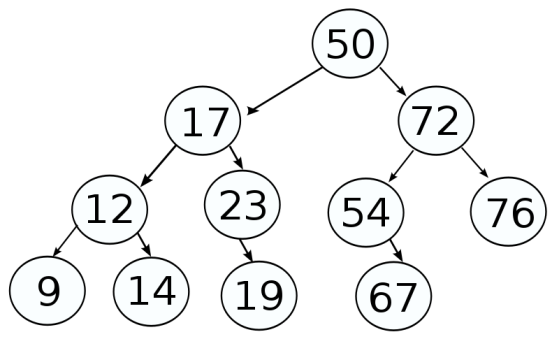
To illustrate these concepts about nodes, a diagram is very helpful. Figure 3 shows a simple example of a data tree. In this example, the nodes are represented by circles and the directed edges are represented by arrows. The node labeled “50” is the root, since it has no parent. The nodes labeled   
“17” and “72” are the children of the root. In addition, the leaves are the nodes labeled “9”, “14”, “19”, and “67”, and they all have a depth of 3.

Figure 3: A data tree. (From Wikimedia Commons)

Previous work has been done demonstrating the effectiveness of an El Niño prediction model which made use of an evolutionary algorithm (Álvarez *et al*.). However, this work is quite limited. First, the model only looked at the Japan Meteorological Agency (JMA) Index, the sea surface temperature of a region in the tropical Pacific. Other El Niño prediction models use many other factors and indices, as noted previously. Furthermore, the study only produced a model but did not test it for accuracy. These issues could be improved through the incorporation of additional and diverse factors into the model. In fact, by using different indices, insights regarding the relative importance of each factor in predicting El Niño may arise. This could be done by implementing different models incorporating different factors and observing which models work best. Also, the models could be tested for accuracy by looking at how well they would have predicted past El Niños. This could be achieved by using previous data which excludes the last several years to develop the models, and then by comparing the models’ predictions during these several years to the actual data.

**Hypothesis**

The objective of this study was to predict future values of El Niño indices by developing a data-tree-based model structure and an evolutionary algorithm to improve these models, and then using historical data to actually produce these models. The hypothesis is that these future values could be predicted to some reasonable degree of accuracy. A secondary objective was to determine whether the accuracy of the models increased with more generations simulated. The hypothesis is that as more generations were simulated, the accuracy would increase. Finally, a third objective was to determine the effects of number of total models per generation on accuracy. It was hypothesized that as this number increased, the accuracy of the models would also increase.

**Materials and Methods**

The materials needed were a computer used to run Python (a Windows 7 laptop was used), Python 3.4.3 to run a modeling program, and a Python development environment to actually develop the model (IDLE, Python’s default development environment, was used.) In addition, to actually create and test models, various sources of index data were used. These were monthly SOI values in PSI from January 1991 to January 2016 (found at <https://www.longpaddock.qld.gov.au/seasonalclimateoutlook/southernoscillationindex/soidatafiles/MonthlySOI1933-1992Base.txt>) , monthly Niño 3.4 values in °C from January 1950 to January 2016 (found at <http://www.cpc.ncep.noaa.gov/products/analysis_monitoring/ensostuff/detrend.nino34.ascii.txt>), and annual inches of rainfall in Los Angeles from 1877 through 2016 (found at <http://www.laalmanac.com/weather/we13.htm>).

A Python program, written in Python 3.4.3 for Windows 7, to generate models was developed and run. What follows are details about the program.

First of all, the program used a data tree to represent the set of models. Each node in the tree consisted of an operator and a value. The operator was a randomly chosen fundamental operator of arithmetic (namely +, -, \*, and /.) Values were either random floating-point numbers from -2 to 2 inclusive, or previous values of the relevant index ranging from f(t-1) to f(t-8) where f(t) is the value of the index at time t. The only exception for this was the root, which had no operation and a value of 1.

Each path directly from the root to a leaf represented a model. For example, if the nodes in such a path were “1”, “+f(t-1)”, "\*2", and "+f(t-3)", in that order, then the model was given by f(t) = 1 + f(t-1) \* 2 + f (t-3).

In order to create a model, the program followed the following steps:

1. The program first took in input data of a given index. To develop the models, all values not from the past 5 years were used as training data, as the values from the past 5 years were instead used to later test the data.

2. The program then generated a tree containing between 90 and 110 random leaves, all of which had depth less than 10. Thus there were between 90 and 110 models.

3. For each mode, the program predicted values of the given index and tested these predicted values against the real values. It then calculated each model’s accuracy by taking the arithmetic mean of all percentage errors. The lower that this accuracy value was, the more accurate the model was. The most accurate model and its accuracy were recorded.

4. The program deleted the branches of the worst half of models (i.e., those which have the lowest accuracy).

5. To replace these models, the program duplicated about 35 models and created about 15 new models by attaching leaves to pre-existing nodes. The duplication was similar to reproduction, and the creation was similar to mutation.

6. Next, the program performed approximately 5 crossovers. A crossover consisted of swapping an entire subtree with another subtree, while still keeping these subtrees and the rest of the tree intact. This process preserved the number of leaves in the tree and thus the total number of models as well.

7. The program performed approximately 5 mutations. A mutation consisted of replacing random nodes in the tree having sufficiently high depth with new, randomly generated nodes. Again, this process also preserved the number of leaves in the tree and the total number of models. Note that these mutations differed from those described in step 4.

8. Steps 3 through 7 (a generation) were repeated a large number of times to carry out more generations.

9. Finally, the model with the lowest average percent error out of all models from all generations and this percent error were recorded. Furthermore, this model was compared against the actual data from the last 5 years, and its average percent error (used as accuracy) at predicting this data was recorded.

(Note: Source code is found in the Appendix.)

To now generate actual models and test accuracies and hypotheses, the aforementioned SOI, Niño 3.4, and Los Angeles precipitation data were used. For each value, 10 models were created using a tree of approximately 100 leaves and 10, 50, and 100 generations each. (These different number of generations were used to test the effect of number of generations on accuracy.) For each of these sets of 10 models, the average percent error in training and the accuracy at predicting “future” values were recorded. Furthermore, to test the effects of tree size on accuracy, 10 models was created using a tree of approximately 200 leaves and 50 generations. Again, the average percent error in training and the average percent error in predicting were recorded. (In all, 120 models were created.) In addition, for each of the 3 sets of data, the most accurate set of 10 models (out of the 4 sets) was used to predict 5 future values of the data.

**Results**

|  |  |  |  |
| --- | --- | --- | --- |
| Generations | Leaves | Average Percent Error | Accuracy |
| 10 | 100 | 33.73 | 58.45 |
| 50 | 100 | 33.19 | 59.20 |
| 100 | 100 | 32.40 | 55.60 |
| 50 | 200 | 32.60 | 66.31 |

Chart 1: Accuracy of the SOI Models

|  |  |  |  |
| --- | --- | --- | --- |
| Generations | Leaves | Average Percent Error | Accuracy |
| 10 | 100 | 10.51 | 3.78 |
| 50 | 100 | 10.51 | 3.74 |
| 100 | 100 | 10.51 | 3.74 |
| 50 | 200 | 14.34 | 3.18 |

Chart 2: Accuracy of the Niño 3.4 Models

|  |  |  |  |
| --- | --- | --- | --- |
| Generations | Leaves | Average Percent Error | Accuracy |
| 10 | 100 | 42.77 | 23.22 |
| 50 | 100 | 38.53 | 37.56 |
| 100 | 100 | 35.66 | 26.65 |
| 50 | 200 | 35.70 | 33.98 |

Chart 3: Accuracy of the L.A. Precipitation Models

The three charts above show the average percent error and the accuracy of each of the 10 sets of models. Also, for each value, the most accurate set of models (*i.e.*, the set with the lowest accuracy value) is shaded with yellow. As stated before, lower values of the right-hand column indicate more accuracy.

Chart 4 shows the accuracy of the models with approximately 100 leaves and illustrates the effect of generations on accuracy. Also included are average accuracies of the 3 models with 10 generations, of the 3 models with 50 generations, and of the 3 models with 100 generations.

Chart 5 shows the accuracies of each set of the 50-generation models. Again, averages are also included.

Charts 6, 7, and 8 show the real values and model-predicted values of each of the characteristics for the last 5 time intervals (months for SOI and Niño 3.4; years for L.A. precipitation.) In addition, for each characteristic, the values for each of the next 5 time intervals are shown as predictions.

**Discussion and Conclusions**

From Chart 1, the SOI was predicted with a 55.60% error, which is very high. Thus the model did not predict the SOI well. However, from Chart 2, Niño 3.4 was predicted with an astoundingly low 3.18% error. Thus the model did predict Niño 3.4 very well. One explanation for this accuracy is that values for Niño 3.4 do not change by much from month to month, and this was especially true for the 5 values used as testing data. These smaller changes indicate more stability of the Niño 3.4 values, thus making them easier to predict. Finally, from Chart 3, Los Angeles annual precipitation levels were predicted with 23.22% accuracy, a fairly low percentage error. This shows that the model had some success predicting the precipitation data. In all, the model had mixed results in predicting various indices and characteristics of El Niño, but its successes do demonstrate potential and support the first hypothesis.

Chart 4 illustrates the relationship between number of generations and accuracy. The average results indicate that the 10-generation models were most accurate, followed closely by the 100-generation models. The 50-generation models were least accurate. However, each of the three models illustrates different trends in this regard, suggesting that number of generations actually has little effect on accuracy. This result was quite counter-intuitive and went against the relevant hypothesis, but can be accounted for by the fact that the most accurate model with regards to the training data was not necessarily the most accurate model with regards to the testing data.

Chart 5 shows the effect of number of leaves on accuracy. In Niño 3.4 and L.A. precipitation predictions, the set of models with 100 leaves was much more accurate than the set with 200 leaves. However, this is not the case with the SOI models. Based on the majority and the fact that Niño 3.4 and L.A. precipitation models were much more accurate, however, it can be concluded that the models with 200 leaves are more accurate than those with 100, the same result that the average values indicate and that the hypothesis predicted. One possible explanation for this trend is that the increased number of leaves led to a larger pool of potential models at each generation and overall, allowing for more accurate models to be created.

Finally, Charts 6, 7, and 8 predict future values of each of the three characteristics using the most accurate set of models. Predictions made by Chart 6 are probably unreliable, as the models failed to predict the test data of SOI well. However, Charts 7 and 8 are reliable based on the models’ accuracy at predicting the test data. Chart 7 indicates a steady dropping of the Niño 3.4 in coming months, while Chart 8 indicates relatively low annual rainfall in Los Angeles in future years.

**Future Research**

Unfortunately, this study had a number of limitations. For example, the real-life data was influenced by random variation, which caused unpredictability and thus less accuracy. In order to improve accuracy of the models, the random variation of the input data could have been minimized through curve smoothing techniques. Also, the model was single input and single output. It only took in one independent variable, time, and only had one dependent variable, which was the characteristic being predicted. Changing the model to allow for more independent or dependent variables could possibly increase accuracy. In addition, the functions developed by the model were limited by the order of operations. This could be avoided by implementing priority of operation for each of the nodes, allowing for a wider range of possible models and thus the potential for better models. Another improvement that could be carried out is the weighing of more recent values. In the current model structure, all values (even those from more than a century ago) were given equal importance. However, in reality, perhaps the more recent values carry more importance than older values. Weighing these recent values more heavily would solve this problem. Finally, only 10 models were generated for each set of parameters. An improvement in the predictions could be made with a larger set size, such as 100. Since the current model is actually quite slow and inefficient, this larger set size may cause the program to take too long. Thus improvements in efficiency would be needed.

Future research could incorporate these potential improvements. Also, more indices and characteristics could be tested. The accuracy for each of these could lead to additional insights regarding what makes an index or characteristic more predictable.

It is very important to note that although the models generated in this study predicted El Niño characteristics, this is not the study’s only use. Instead, the program presented in the study can actually be used to generate models which predict any time series, such as (but of course not limited to) ocean tides, sunspots, and stock market prices. It would be interesting to have the program generate and test models for a wide range of time series. Furthermore, comparing accuracies of models for many different time series could lead to further insights on predictability.

**Acknowledgements**

I would like to thank my parents, Mr. Sang Kim and Mrs. Eunsook Kim, for their unwavering support and motivation through the countless hours of carrying out this project. Furthermore, I would like to thank Mr. Peter Starodub for his guidance and numerous helpful resources. I would also like to thank all my friends who provided pointers, especially on how to correctly write a research paper. Finally, I want to acknowledge the authors and providers of all sources of information and data used in the project. Without any of these people, this project could not have been done.

**References**

Barnston, Anthony. "Why Are There so Many ENSO Indexes, Instead of Just One?" *NOAA Climate.gov*. National Oceanic and Atmosphere Association, 9 Jan. 2015. Web. 7 Dec. 2015.

"Definitions of El Nino, La Nina, and ENSO." *Tropical Ocean Atmosphere Project*. Pacific Marine Environmental Laboratory, n.d. Web. 27 Oct. 2015.

"EL NIÑO/SOUTHERN OSCILLATION (ENSO) DIAGNOSTIC DISCUSSION." *National Weather Service Climate Prediction Center*. National Oceanic and Atmosphere Association, n.d. Web. 7 Dec. 2015.

"Equatorial Pacific Sea Surface Temperatures." *National Centers for Environmental Information*. National Oceanic and Atmosphere Association, n.d. Web. 7 Dec. 2015.

Jones, Gareth. "Genetic and Evolutionary Algorithms." *Encyclopedia of Computational Chemistry* (n.d.): n. pag. *Wiley Online Library*. John Wiley and Sons, Ltd., 15 Apr. 2002. Web. 7 Dec. 2015.

Álvarez, Alberto, Pedro Vélez, Alejandro Orfila, Guillermo Vizoso, and Joaquín Tintoré. "Evolutionary Computation for Climate and Ocean Forecasting: “El Niño Forecasting”." *Proceedings of the Second International Conference on Euro* (1999): n. pag. *DIGITAL.CSIC*. Web. 26 Oct. 2015.

"NORTH AMERICAN COUNTRIES REACH CONSENSUS ON EL NIÑO DEFINITION." *El Nino Definition*. NOAA News Online, 23 Feb. 2005. Web. 26 Oct. 2015.

Pielke, Roger A., Jr., and Christopher N. Landsea. "La Niña, El Niño, and Atlantic Hurricane Damages in the United States." *Bulletin of the American Meteorological Society* (1999): 2027-033. *Hurricane Research Division*. Atlantic Oceanographic & Meteorological Laboratory of NOAA, 7 Apr. 1999. Web. 26 Oct. 2015.

"Southern Oscillation Index." *Australian Government Bureau of Meteorology*. Australian Government, n.d. Web. 7 Dec. 2015.

"Southern Oscillation Index (SOI)." *National Centers for Environmental Information*. National Oceanic and Atmosphere Association, n.d. Web. 7 Dec. 2015.

*Tree Data Structure*. Digital image. *Commons.wikimedia.org*. Wikimedia Commons, 14 Apr. 2006. Web. 17 Feb. 2016.

"Tree Data Structure." *Tree Data Structure*. Carnegie Mellon University School of Computer Science, n.d. Web. 17 Feb. 2016.

"What Is an Evolutionary Algorithm." *What Is an Evolutionary Algorithm?* Department of Computer Sciences at VU University Amsterdam, n.d. Web. 26 Oct. 2015.

Wolter, Klaus. "Multivariate ENSO Index (MEI)." *Earth System Research Laboratory Physical Sciences Division*. National Oceanic and Atmosphere Association, n.d. Web. 7 Dec. 2015.

**Appendix: Source Code of the Program**

import random

# bunch of global vars

nodes = 0

leaves = 0

toPrint = []

toPrintList = []

nodeList = []

leafList = []

index2 = 0

fn = []

fnList = []

evalList = []

accList = []

accGen = []

fnGen = []

# constants

minLeaves = 180 # 90

maxLeaves = 220 # 110

opList = ["+", "-", "\*", "/"]

maxDepth = 10 # 10

inList = [8.52, 6.08, 5.85, 8.69, 20.2, 16.36, 9.08, 13.53]

realList = [8.52, 6.08, 5.85, 8.69, 20.2, 16.36, 9.08, 13.53, 3.21, 13.19, 37.96, 9.25, 16.42, 4.42, 17.94, 11.57, 9.09, 31.01, 12.4, 12.44, 24.35, 8.11, 27.36, 21.0, 11.99, 7.35, 8.08, 12.48, 7.66, 17.86, 12.82, 10.43, 31.28, 10.71, 8.96, 26.98, 19.67, 33.44, 12.3, 7.21, 14.35, 14.92, 21.26, 7.17, 12.32, 7.74, 27.47, 16.58, 22.0, 20.44, 13.68, 7.93, 8.38, 18.79, 4.85, 8.18, 5.58, 21.13, 9.54, 16.0, 11.94, 11.99, 9.46, 26.21, 8.21, 10.59, 7.99, 7.22, 12.36, 11.65, 11.59, 19.22, 18.17, 11.18, 32.76, 19.21, 13.07, 23.43, 22.41, 13.47, 21.66, 14.94, 11.88, 16.93, 12.52, 11.52, 12.66, 9.77, 18.03, 17.56, 7.94, 6.67, 9.59, 19.66, 13.65, 12.52, 8.58, 13.86, 15.26, 19.92, 17.05, 23.65, 13.42, 11.6, 16.18, 12.63, 19.18, 11.72, 19.3, 18.65, 19.52, 8.72, 19.32, 10.6, 16.29, 7.91, 5.59, 7.06, 18.83, 8.51, 16.11, 6.73, 26.28, 11.85, 13.36, 34.84, 19.28, 13.87, 14.05, 22.31, 9.21, 38.18, 12.16]

testList = [10.4, 13.13, 20.34, 11.35, 21.26]

newTerms = len(realList) - len(inList)

swapsPer = 5

mutationsPer = 5

gensDef = 1000

# class definitions

class Node:

def \_\_init\_\_ (self, operation, value, depth, parent, children, index):

self.operation = operation

self.value = value

self.children = children

self.depth = depth

self.parent = parent

self.index = index

def isleaf (self):

if len(self.children) == 0:

return True

else:

return False

def makeRoot (self):

global nodes

global leaves

global toPrint

global nodeList

global leafList

global index2

global fn

global fnList

global maxDepth

# initialize stuff

nodes = 0

leaves = 0

toPrint = []

nodeList = []

leafList = []

index2 = 0

fn = []

fnList = []

nodeList.append(self)

nodes += 1

index2 += 1

rootChildren = random.randint(1, 2)

for i in range (0, rootChildren):

if self.depth < maxDepth:

op = opList[random.randint(0, len(opList))-1]

if random.randint(0, 9) > 7:

val = str(4\*random.random() - 2)

else:

j = str(random.randint(1, 8))

val = "inListCopy[t-" + j + "]"

x = Node (op, val, self.depth+1, self, [], index2)

x.makeChildren ()

self.children.append(x)

def makeChildren(self):

global nodes

global leaves

global nodeList

global index2

global maxDepth

nodeList.append(self)

nodes += 1

index2 += 1

n = random.randint(0, 3)

if n == 0:

# leaf

leaves += 1

leafList.append(self)

else:

if self.depth < maxDepth:

for i in range (0, n):

op = opList[random.randint(0, len(opList))-1]

if random.randint(0, 9) > 7:

val = str(4\*random.random() - 2)

else:

j = str(random.randint(1, 8))

val = "inListCopy[t-" + j + "]"

x = Node (op, val, self.depth+1, self, [], index2)

x.makeChildren()

self.children.append(x)

else:

# no children generated --> leaf

leaves += 1

leafList.append(self)

def printTree (self):

global toPrint

# set to [] before call

global toPrintList

# finds every path down a tree and prints it

toPrint.append(self.index)

if self.isleaf():

toPrintList.append(toPrint)

toPrint = toPrint[:-1]

else:

for i in range (0, len(self.children)):

self.children[i].printTree()

toPrint = toPrint[:-1]

def getFns (self):

# gets a list of all functions

# similar to printTree

global fn

# set to [] before call

global fnList

# finds every path down a tree and prints it

fn.append(self.operation)

fn.append(self.value)

if self.isleaf():

fnList.append("".join(fn))

fn = fn[:-2]

else:

for i in range (0, len(self.children)):

self.children[i].getFns()

fn = fn[:-2]

def findNode (self, index):

# returns a node with a given index

if index == self.index:

return self

else:

for i in range (0, len(self.children)):

if self.children[i].findNode(index) != None:

return self.children[i].findNode(index)

def update (self, parent, depth):

# reupdates parent, depth, and index of each leaf, starting at root

# also recounts nodes and leaves

# set to 0 before call

global index2

global nodes

global leaves

# set to [] before call

global nodeList

global leafList

# update parent

self.parent = parent

# update depth

self.depth = depth

# update index

self.index = index2

index2 += 1

# update nodes, nodeList

nodes += 1

nodeList.append(self)

# update leaves

if self.isleaf():

leaves += 1

leafList.append(self)

# recursion

for i in range (0, len(self.children)):

self.children[i].update(self, self.depth + 1)

def killBranch (self, child = None):

# given the index of a node, kills the branch

# first, travel up to the start of the branch

if len(self.children) < 2:

self.parent.killBranch(self)

# remove from children

else:

self.children.remove(child)

# update tree after!

def duplicate (self, attachTo = None):

# duplicates branch, starting at top

newNode = Node (self.operation, self.value, self.depth, self.parent, self.children, -1)

newNode.children = []

if attachTo == None:

self.parent.children.append(newNode)

else:

attachTo.children.append(newNode)

if not self.isleaf():

self.children[0].duplicate(newNode)

# update after!

def topBranch (self):

# finds top of branch, given leaf

if len(self.parent.children) > 1:

return self

else:

return self.parent.topBranch()

def swap (self, node1, node2):

# swaps the position of two nodes

# more specifically, changes the children list of each parent

# be sure to check that both nodes are deep enough

# make sure that either node isn't in the other's subtree,

# or that they are not siblings (i.e. share same parent)

node3 = node1.parent

node4 = node2.parent

if node1.depth < 7 or node2.depth < 7:

return 2 # failed

if node1.findNode(node2.index) == None and node2.findNode(node1.index) == None:

node3.children[node3.children.index(node1)] = node2

node4.children[node4.children.index(node2)] = node1

return 0 # success!

else:

return 1 # failed

# update after!

def addNode(self):

# adds a random new node to self

global maxDepth

if self.depth == maxDepth:

return 1 # failed

else:

op = opList[random.randint(0, len(opList))-1]

if random.randint(0, 9) > 7:

val = str(4\*random.random() - 2)

else:

j = str(random.randint(1, 8))

val = "inListCopy[t-" + j + "]"

newNode = Node (op, val, self.depth+1, self, [], -1)

self.children.append(newNode)

# update after!

def changeNode(self):

# randomly changes self

if self.depth < 5:

return 1 # fail

else:

op = opList[random.randint(0, len(opList))-1]

if random.randint(0, 9) > 7:

val = str(4\*random.random() - 2)

else:

j = str(random.randint(1, 8))

val = "inListCopy[t-" + j + "]"

self.operation = op

self.value = val

return 0 # success!

## op = opList[random.randint(0, len(opList))-1]

## if random.randint(0, 1) == 0:

## val = str(4\*random.random() - 2)

## else:

## j = random.randint(1, 4)

## if j == 1:

## val = "inListCopy[t-1]"

## elif j == 2:

## val = "inListCopy[t-2]"

## elif j == 3:

## val = "inListCopy[t-3]"

## else:

## val = "inListCopy[t-4]"

## self.operation = op

## self.value = val

## return 0 # success!

# convenience functions

def updateRoot():

global nodes

nodes = 0

global leaves

leaves = 0

global index2

index2 = 0

global nodeList

nodeList = []

global leafList

leafList = []

root.update(None, 0)

def printRoot():

global toPrintList

toPrintList = []

root.printTree()

return toPrintList

def fnRoot():

global fnList

fnList = []

root.getFns()

return fnList

def evalFns():

global evalList

fnRoot()

evalList = []

for i in range (0, len(fnList)):

inListCopy = []

inListCopy2 = []

for k in range (0, len(inList)):

inListCopy.append(inList[k])

inListCopy2.append(inList[k])

for j in range (0, newTerms):

t = len(inList) + j

try:

inListCopy2.append(eval(fnList[i]))

except:

#print("ZeroDivisionError detected.")

inListCopy2.append(1000000000)

inListCopy.append(realList[t])

evalList.append(inListCopy2)

#print(evalList)

def accFns():

# compute RMSE of each function

global evalList

global accList

evalFns()

accList = []

for i in range (0, len(evalList)):

rmse = 0

fn = evalList[i]

for j in range (len(inList), len(inList) + newTerms):

try:

percentAcc = abs(100\* (fn[j] - realList[j]) / realList[j])

#rmse += (percentAcc)\*\*2

rmse += percentAcc

except OverflowError:

#print("OverflowError detected.")

rmse += 1000000000

#print(rmse)

#rmse = (rmse / newTerms) \*\* 0.5

rmse = rmse / newTerms

accList.append(rmse)

def killModels():

# kill a specified number of models

accFns()

global accList

global leafList

global leaves

#for i in range (0, kills):

for i in range (0, int(leaves/2)):

j = accList.index(max(accList))

accList.pop(j)

leafList[j].killBranch()

leafList.pop(j)

updateRoot()

def duplicateModels():

# duplicates a specified number of models

accFns()

global accList

global leafList

global leaves

accList2 = []

for i in range (0, len(accList)):

accList2.append(accList[i])

for i in range(0, int(leaves \* 0.9)):

j = accList.index(min(accList2))

leafList[j].duplicate()

accList2.remove(min(accList2))

if i + leaves > maxLeaves:

break

updateRoot()

def newModels():

# creates a specified number of models

global leaves

global nodeList

global maxLeaves

for i in range (0, int(leaves \* 0.5)):

j = random.randint(1, len(nodeList)-1)

nodeList[j].addNode()

if i + leaves > maxLeaves:

break

updateRoot()

def swapModels():

# carries out a specified number of SUCCESSFUL swaps

global nodeList

for i in range (0, swapsPer):

# for i in range (0, 3):

j = random.randint(1, len(nodeList)-1)

k = random.randint(1, len(nodeList)-1)

result = root.swap(nodeList[j], nodeList[k])

failCount = 0

while result != 0 and failCount < 20:

failCount += 1

j = random.randint(1, len(nodeList)-1)

k = random.randint(1, len(nodeList)-1)

result = root.swap(nodeList[j], nodeList[k])

updateRoot()

def mutateModels():

# carries out a specified number of SUCCESSFUL mutations

global nodeList

for i in range (0, mutationsPer):

j = random.randint(1, len(nodeList)-1)

result = nodeList[j].changeNode()

failCount = 0

while result != 0 and failCount < 20:

failCount += 1

j = random.randint(1, len(nodeList)-1)

result = nodeList[j].changeNode()

updateRoot()

def simGen():

# simulates a generation

killModels()

duplicateModels()

newModels()

swapModels()

mutateModels()

updateRoot() # for good measure

def simGens(gens = gensDef):

# simulates many generations

for i in range (0, gens):

simGen()

accFns()

accGen.append(min(accList))

j = accList.index(min(accList))

fnGen.append(fnList[j])

if i % max(int(gens/10), 1) == 0:

print(i)

# POP NAN

def testModel():

gen = accGen.index(min(accGen))

print(gen)

fn = fnGen[gen]

inListCopy = []

#inListCopy2 = []

testList2 = []

for i in range (0, len(realList)):

inListCopy.append(realList[i])

#inListCopy2.append(realList[i])

for j in range (0, len(testList)):

t = len(realList) + j

try:

#inListCopy2.append(eval(fn))

testList2.append(eval(fn))

except:

#inListCopy2.append(1000000000)

testList2.append(eval(1000000000))

inListCopy.append(testList[j])

print(testList2)

# compare testList and testList2

rmse = 0

for k in range (0, len(testList)):

try:

percentAcc = abs(100\* (testList2[k] - testList[k]) / testList[k])

rmse += percentAcc

except OverflowError:

rmse += 1000000000

rmse = rmse / len(testList)

return rmse

# building the tree

root = Node ("", "1", 0, None, [], 0)

root.makeRoot()

while maxLeaves < leaves or minLeaves > leaves:

# try again

root = Node ("", "1", 0, None, [], 0)

root.makeRoot()

# INITIAL MODEL DONE

#root.printTree()

#print(toPrintList). someuture values could be predicted with a reasonable degree of accuracty