COS710 A3 report

David Walker - 19055252

June 2023

1 Introduction

Grammatical Evolution is a subgenre of Genetic Programming which evolves programs using a Backus-Naur Form grammar which maps to chromosomes. In this assignment, I've implemented a grammatical evolution model to try and solve the Seoul Bike Ride Duration problem. This was interesting, albeit somewhat challenging. Unfortunately, it did not lead to significantly improved results over my canonical GP implementation: however, it did highlight some significant shortcomings I faced in that process.

2 Data

2.1 Important Note

I used the same simplified, cleaned dataset from Assignment 1 and 2 for this assignment. I have included the explanation of that processing work below, but note that it has not changed since the last assignment.

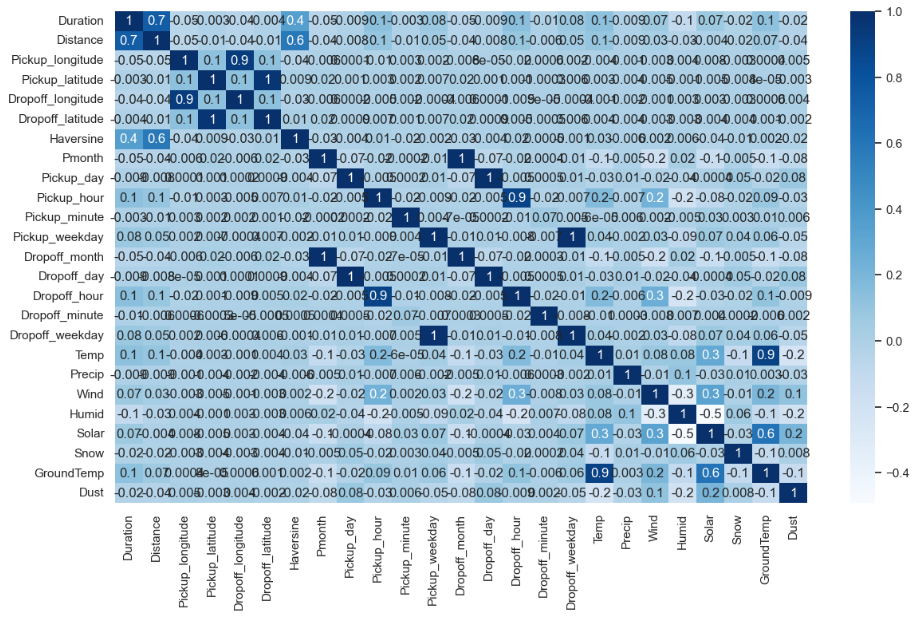
2.2 Dataset description

The dataset provided to use came in the form of a 9 601 139 line long CSV file. It contained 25 columns, each with a different feature relating to a given bike trip.

The first step I took on this dataset was to rename the columns, as the initial names given were somewhat confusing. With that done, I initially decided to just dive into the GP programming, splitting the data into training and testing data, on a 70/30 split respectively.

However, upon working through the GP, I realised that my accuracy was quite limited, and decided to turn back to my dataset to find out why. The first thing I did was some moderate statistical trimming on the dataset. I first removed all entries which had any null values. I also then removed any duplicates. To be fair, the dataset didn't seem to have major issues ion this sense, but by trusting an algorithm to ensure that those criteria are met, I could be more confident in my dataset not being a problem. I then also used a Z-score based metric to remove any outliers from the data: if the Z-score of a given row was greater than 3, it was removed. This concluded the ”cleanup” of the data.

I also wanted to simplify the dataset, as 25 features was quite processing intensive, and duplication of closely-related features might be harming my training accuracy. To do this, I examined the Pearson coefficient of the features, using a heatmap.



Using this heatmap, we can clearly see that come features are very closely correlated. The strongly correlated features include dropoff and pickup months, dropoff and pickup days, and pickup and dropoff days of the week, implying that those elements of a bike trip are almost always going to be an equal pairing. The same is true of pickup and dropoff latitude and longitudes, implying that the bikes will always be picked up and dropped off in the same place. Ground temperatures and air temperatures are also very closely related.

For each of these closely related features, I dropped one of the two features. This was to ease computational complexity and aid ”accurate” solution-finding, by making the search space less redundant. The dropped features are ’GroundTemp’, ’Dropoff latitude’, ’Dropoff longitude’, ’Dropoff month’, ’Dropoff day’, and ’Dropoff weekday’.

With this, I was able to begin working on the GP itself.

Old name	**New name**
Duration	Duration
Distance	Distance
PLong	Pickup_longitude
PLatd	Pickup_latitude
DLong	Dropoff_longitude
DLatd	Dropoff_latitude
Haversine	Haversine
Pmonth	Pickup_month
Pday	Pickup_day
Phour	Pickup_hour
Pmin	Pickup_minute
PDweek	Pickup_weekday
Dmonth	Dropoff_month
Dday	Dropoff_day
Dhour	Dropoff_hour
Dmin	Dropoff_minute
DDweek	Dropoff_weekday
Temp	Temp
Precip	Precip
Wind	Wind
Humid	Humid
Solar	Solar
Snow	Snow
GroundTemp	GroundTemp
Dust	Dust

3 Representation

My representation of choice was a fairly simple one: given that I was aiming to produce a program which aimed to output a ”good”, correct value for the given problem, the grammar I chose to use wasn’t particularly complex. Please find it here:

```
<expr>  ::= <expr> <op> <expr>
| <var>
| <n>
<op>::= '+' | '-' | '*'
<var>::= Distance
| Pickup_longitude
| Pickup_latitude
| Haversine
| Pmonth
| Pickup_day
| Pickup_hour
| Pickup_minute
| Pickup_weekday
| Dropoff_hour
| Dropoff_minute
| Temp
| Precip
| Wind
| Humid
| Solar
| Snow
| Dust
<n>:== 1 | 2 | 3
```

4 Initial population generation

The initial population starts life as a randomly-generated list of integers. This list is of size 100. Each element of the population is then generated, first, by starting with the start symbol as an entry point of the generated program. From there, each number in the list is calculated mod 'n', where 'n' is the number of options available from which the start symbol of the grammar can be expanded. The 'n'th option is then selected. Should the selected symbol be a terminal, we then move to the next non-terminal symbol; should it be non-terminal, we continue that expansion process in a similar fashion until eventually all paths of the generated programs end in terminals. This process is repeated until the end program is generated. In this way, we generate the entire initial population.

5 Fitness evaluation

Fitness evaluation is done in a similar way to previous assignments, except that the output of a generated program is evaluated rather than the output of a generated expression. I use the mean average error of the program to work out the fitness score. A pseudocode representation of the algorithm is below, where y is the correct duration, *pred(y)* is the predicted duration, and n is the number of entries in the dataset.

$$\frac{\sum |y - pred(y)|}{n}$$

6 Selection method

I used fitness proportionate selection for this particular implementation. This was done, firstly, because I wanted to try avoid early convergence, which was a major issue in this project. What this meant is that a random selection of parent was made, but the likelihood of a parent being chosen was proportionally increased if said chromosome proved fitter than its counterparts. While it still isn't perfect, I usually find that roulette wheel selection produces more exploration and less exploitation, which is what I was looking for.

7 Genetic operators

This project uses both crossover and mutation. Crossover, in this context, swaps genes between two parent chromosomes, such that numbers in the list are swapped between parents to create a new offspring. Mutation, on the other hand, is the generation of random new genes which replace existing genes in parents, altering the outcomes when the program is generated. I also used a slight amount of elitism, where the very fittest individual from each generation is carried over to the next generation without changes.

I experienced a very serious issue of premature convergence, which I ultimately was unable to completely work around. For this reason, to mitigate that effect as much as possible, the mutation likelihood was quite high, at 65%.

8 Experimental setup

8.1 Parameters

- Population size: 48
- Iterations: 50
- Mutation likelihood: 0.65
- Crossover likelihood: 0.35

8.2 Development environment

This program was developed in R. I used R rather than my traditional preference, Python, on two primary basis: firstly, the existence of the GramEvol library for R [1], which takes a lot of "boilerplate" code work out of the scope of my work, and allows me to focus on optimizing the implementation instead. In addition, I chose R due to the efficient vectorisation capabilities it has, which allowed it to more easily interpret my given dataset as a vector list, and thus perform computation on said list much more quickly, and in a much more parallel fashion. In terms of parallelisation, another benefit of R is that the *lapply* backend for multithreading runs on Windows, while the Multiprocessing library I used in Python required me to switch operating systems and use Linux whenever I was working with my project: not a big deal, but an inconvenience nonetheless.

All programming was done in Visual Studio Code, using R 4.3.0, apart from the Python-centric data adjustment done in Assignment 2, the work of which carried over to this project.

8.3 Development machine specifications

- CPU: AMD Ryzen 5 5600 (3.5GHz base, 4.4GHz boost)
- RAM: 48GB DDR4 (3200MT/s, CL18)
- Storage: Samsung 970 Evo Plus NVMe (2TB)
- GPU: AMD Radeon RX 6600XT (8GB)

9 Results

9.1 Notes

Please find data from all 10 training runs on the last page, should you wish to see what each run's final outcome was.

9.1.1 Training data evaluation

	Canonical	LR	GBM	KNN	RF	Structure	Gram. Evol.
MAE	7.38	10.11	7.35	5.53	1.21	11.73	14.56
RMSE	13.22	16.45	12.55	11.29	2.76	21.47	26.62
MedAE	3.53	6.9	3.74	2	0.4	3.78	4.69
R2	0.57	0.56	0.74	0.79	0.98	-0.125	0.12

9.1.2 Testing data evaluation

	Canonical	LR	GBM	KNN	RF	Structure	Gram. Evol.
MAE	7.39	10.12	7.37	6.83	2.92	11.74	14.54
RMSE	14.84	16.48	12.58	13.93	6.25	21.49	26.61
MedAE	3.54	6.9	3.75	2.59	1.2	3.78	4.68
R2	0.57	0.56	0.74	0.69	0.93	-0.126	0.13

9.2 Runtimes

Runtimes were exceptionally fast, owing to the multi-threaded, vectorised nature of the implementation I used. On average, each training run of 40 iterations took only around 70 seconds - for exact details, please refer to the per-run details on the final pages of this document. This is miles faster than the very slow DEAP structured implementation from assignment 2, which took 25 minutes per run, or 21 times the time each run of this grammatical evolution implementation took. Again, though, this is likely due to the implementation specifics rather than the overall method of grammatical evolution. It also compares favourably to the 2.06 minutes per run of my canonical implementation, albeit not by as much - nonetheless, running in only 56% of the time of the canonical implementation while doing twice as many generations is quite impressive.

9.3 Discussion

Unfortunately, while runtime performance was good, the actual accuracy of this model was quite poor. This is likely owing to a premature convergence issue I faced throughout, whereby the single elitist function generated early in the run started dominating the population, despite my attempts to mitigate that effect by pushing mutation likelihood to 65%. I think, with a more complex grammar, and with some additional parameter tweaking, we could do better: however, given my understanding and time constraints (which, to be fair, were self-imposed given my decision to re-learn R after not touching it for two years), this result does tell me that a good canonical implementation of GP is difficult to beat with more "sophisticated", but potentially more tricky, methods.

References

[1] F. Noorian, A. M. de Silva, and P. H. Leong, "gramevol: Grammatical evolution in r," *Journal of Statistical Software*, vol. 71, pp. 1–26, 2016.

10 Outputs and logs

10.1 Outputs of training runs

```
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  Haversine * 3 * 3
  Best Cost:        14.5555252132105
[1] "Time:"
   user  system elapsed
  65.11    6.58   71.98
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  Dropoff_hour + 3
  Best Cost:        17.7448156438864
[1] "Time:"
   user  system elapsed
  65.56    8.78   74.77
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  Haversine + (3 + (Dropoff_hour - 1))
  Best Cost:        17.0285068982117
[1] "Time:"
   user  system elapsed
  65.61    8.61   76.06
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  2 + (Dropoff_hour + (2 - 0.5))
  Best Cost:        17.7440369370037
[1] "Time:"
   user  system elapsed
  73.47   10.54   94.44
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  Dropoff_hour + Haversine
  Best Cost:        17.1067386216389
[1] "Time:"
   user  system elapsed
  67.53    8.14   79.45
[1] "GE:"
```

```
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  0.5 * (Haversine * Pickup_hour)
  Best Cost:        16.0728051061263
[1] "Time:"
   user  system elapsed
  63.52    6.52   70.26
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  Pickup_weekday + Dropoff_hour
  Best Cost:        17.6850566589246
[1] "Time:"
   user  system elapsed
  62.31    6.34   68.88
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  Humid * (0.5 * Haversine/3)
  Best Cost:        16.4712429727042
[1] "Time:"
   user  system elapsed
  63.67    6.46   70.36
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  0.5 * Pickup_longitude
  Best Cost:        17.6883428032855
[1] "Time:"
   user  system elapsed
  62.04    6.53   68.92
[1] "GE:"
Grammatical Evolution Search Results:
  No. Generations:  40
  Best Expression:  3 * (2 - (Precip - 3))
  Best Cost:        17.5802779299658
[1] "Time:"
   user  system elapsed
  62.68    6.34   69.27
```