COS710 Assignment 2 report

David Walker - 19055252

2023

1 Introduction

The purpose of this assignment was to build a structure-based genetic program (GP) which would be able to accurately predict the duration of a bike trip when rented from a company in Seoul, given a set of data about the rental itself and various atmospheric and circumstantial factors surrounding it.

My strategy for tacking this problem was to begin at the high level, by leveraging various existing GP libraries and technologies to build an understanding of the given problem. Should said libraries at any point prove insufficient, I would then dive down to a deeper level with less abstraction, and try to overcome the issues with manual programming. This led to a greater understanding of various approached which have already been taken, and helped me to better understand the intricacies and minor differences between various GP approaches.

1.1 Important note

Please note that large parts of the below document are the same as Assignment 1. To avoid any need for disambiguation, and ensure all details are able to be found, all information that was provided with A1 is below. Apart from the structure-based approach section which was added, and the new results and discussion thereof, the only other section with significant changes is the "fitness evaluation" section, which explains in detail how the structure of each program was used in this approach.

A second important note is that, for full disclosure, I did not have the opportunity to run as many runs or do as much parameter tuning as I would have liked on this model. This is due to a lack of available time: a poor excuse, but one that I felt was worth mentioning regardless. Specifically, the length of time it takes for this model to be trained was prohibitive when load shedding limited my ability to use my desktop PC for a maximum of only 6 hours uninterrupted at a time. Because of this, the number of test runs and tuning runs was limited, which limited my ability to improve model accuracy quite significantly.

2 Structure based approach

To turn my canonical GP from assignment 1 into a structure-based GP for assignment 2, I took some inspiration from the Iterative Structure-Based Algorithm approach. This approach uses a similarity index (i.e., a calculated value describing the similarity between two trees) to apply pressure to the GP, altering its traversal of the search space by either rewarding or punishing generated programs for their similarity to past programs. My implementation of this concept was centered around exploring more of the search space, as I felt that my previous canonical implementation had too easily fallen into local minima, and a push towards greater exploration would lead to better results.

The specific implementation I used necessitated two changes to my GP implementation. Firstly, in terms of fitness evaluation, I adjusted the custom fitness function. A further description of how this was done is below, in the appropriate section of this document, but the shortened explanation is that I allowed a run of 'n' generations to occur naively, and then saved the best tree from that run into a array (for the sake of simplicity, let's call that array 'arr-trees'). I then ran 'i' more runs, whereby each individual's fitness was also evaluated based on similarity to each tree in 'arr-trees'. Should it, on average, be too similar, the individual's fitness score was worsened, on a sliding scale. Given this metric, then, pressure was applied such that the tournament selection method would be much more likely to choose solutions which diverge from past runs' solutions. This new solution was then added to 'arr-trees'. To alleviate the unfair fitness score advantage faced by the first run, then, an additional comparison between all solutions in 'arr-trees', which evaluated raw fitness without taking into account similarity, was then employed at the end to determine the overall best solution from the series of runs.

While this method is definitely very runtime-intensive, it seems to be more effective at creating an exploration of the search space over multiple runs than merely adjusting the random seed each time .

From a technical perspective, an interesting challenge I was faced with was that GPLearn, the library I had completed assignment 1 with, didn't allow me to view individual's structures in the fitness function, as confirmed by a GitHub issue [1]. This necessitated rewriting all additional elements in a more low-level framework, specifically DEAP, which set back my progress significantly, especially given DEAP's relatively slow execution time compared to GPLearn. Ultimately, however, this rewrite did prove successful.

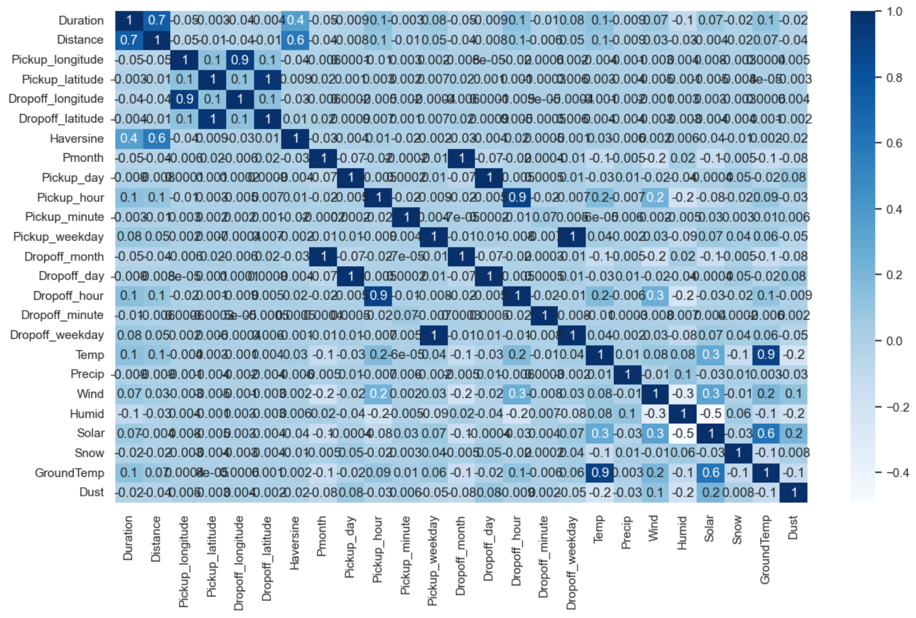
3 Dataset information

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.

4 Genetic program representation

4.1 GPLearn

To begin gaining familiarity with the idea of genetic programming, I decided to employ a library by the name of GPLearn. This library simplifies genetic programming quite elegantly, allowing me to see the outcomes of any changes and decisions I make without having to code them all myself - useful, albeit not perfect, as we'll see later. The representation of the genetic program in this library, as with most GPs, is as a tree, whereby nodes may be either functions or terminals. One can then use said tree to create an expression which is used to evaluate the desired output.

I decided to keep the function set small enough that it could maintain efficiency, while also aiming to allow the program to deal with the large dataset. The functions I settled on were addition, subtraction, multiplication, and division (protected, to avoid division by zero).

The terminal set came very naturally - I left it at its default setting, which meant that all 18 features from the given dataset were possible terminal inputs.

4.2 DEAP

While GPLearn was fascinating, I wanted to go slightly lower-level and play with a more hands-on version of genetic programming. Hence, I decided to also try the DEAP (Distributed Evolutionary Algorithms in Python) library, which offers greater flexibility and user control. Here, I was once again in charge of defining the function set.

Due to a lack of parallel evaluation when ran through Jupyter Notebooks on Windows, DEAP runs slower on my system than GPLearn, so I decided to simplify the function set slightly by removing the square root option. I felt this fair, as GPLearn indicated that it was barely being used when I trained the GP through it. I also played with the idea of using the power function in the function set, but due to the huge changes it made to the fitness scores, leading to wild fluctuations in accuracy, I decided to remove it too. Hence, I only used the addition, subtraction, multiplication, and protected division operators in my function set when using DEAP.

To represent the terminal set, I once again simply used all 18 features from the given dataset after it had been cleaned, naturally ensuring that each of those features would be selected from the same row in the training set when the tree is evaluated.

5 Initial population generation

In all cases, I elected to use the "grow" method for my population generation. I did this to avoid unnecessary bloat and clutter: should a more full tree be required, it would evolve, rather than being forced. In addition, allowing smaller trees to be parsed where possible is always good for evaluation speed. I used a population size of 100, which is small but seemed to prove adequate, for the sake of runtime efficiency.

6 Fitness evaluation

6.1 Error function

To determine an individual's error, I decided to use the average of the raw error over the course of all evaluations of said individual against the sample of the training dataset. See the formula below, where "n" is the number of entries in the sample of the training dataset. I chose to use the average to avoid overly punishing very high but isolated incidents, where only very particular sets of values led to undesirable outcomes - using the raw error without taking the average across the sample led to overly inconsistent and fluctuating behaviour.

$$\frac{\sum_1^n |predicted - actual|}{n} \tag{1}$$

6.2 Fitness function

Given the individual error function defined above, it's clear that this is a minimisation problem: that is, the objective is to reduce the error value as much as possible. In GPLearn, I simply had to point it to the "duration" data as a label and it understood that. However, one of the advantages of using DEAP as well is that it allowed me the chance to write my own error function, and thus specify the evaluation of said function. Hence, I converted the above function into code, and initialised a fitness calculator which minimised single-objective error scores.

For assignment 2, I decided to implement a structural element into my fitness function. The idea that I implemented was to save the representation of the tree which was developed by an initial naive run of the GP, and then to use a similarity index to force exploration in subsequent runs by penalising overly-similar trees. This penalisation would then occur if the new run generated any trees similar to the best result of a past run. This decision was made due to the high likelihood of early convergence that I observed during assignment 1. After the initial run, all future runs have an additional calculation applied to their fitness value, which is made up of the formula below, where "distance" is the Levenshtein distance between the generated tree and the best tree of a given past run (smaller distance implies a more similar tree):

$$\frac{10}{0.1 + distance} \tag{2}$$

This formula was developed with the intention of penalising overly-high similarity indexes: by putting the similarity score in the denominator of the new term, we cause the additional amount added to decrease when the distance between saved past trees and the present tree being evaluated are very dissimilar, and that same figure grows quite large when the similarity figure is too similar: as a worst case scenario, an identical tree would apply a penalty of 100 to the raw fitness score, which would cause the selection method to deem it very unfit for the next generation. In that instance, trees which may have been deemed less fit on a raw fitness basis will be given an advantage if they offer an exploratory benefit to the GP. Put together, this leads to the below function being used to evaluate the total fitness of an individual, where n is the sample size used, s is the similarity index score, and i is the number of trees in the "past runs' best" array:

$$\frac{\sum_1^n |predicted - actual|}{n} + \frac{\sum_0^i s}{i} \tag{3}$$

This applies selection pressure on exploring a wider part of the search space. By penalising exploration of the existing search space, we can generate new solutions which break away from any local minima found in past run.

This poses the problem, however, of the first run being unfairly prioritised as it has no similarity index with which to compare, and thus has only a negligible penalty applied to it. To combat this, at the end of the evolution, the entire array of discovered best individuals from past runs is evaluated on only raw fitness, and the "best" program is selected to be the one with the lowest raw fitness score after this procedure. In doing this, we do take a fairly significant performance penalty, as we need to perform more runs, to generate additional solutions in new parts of the search space. By the end we should be able to determine whether it was worth the additional computational cost.

7 Selection method

I elected to use tournament selection, with a tournament size of 20 - this was decided after each GP with tournament sizes of 5, 10, and 20, after which 20 seemed to give the best fitness scores. Tournament selection works by selecting a random subset of individuals from the population (in this case, 20 individuals), and then comparing them against each other to see which is fittest. It then returns the fittest individual from the random selection, which is then used as a parent for the next generation.

The tournament size of 20 was chosen as it is an adequately small subset of the overall population, and will not cause early convergence by forcing the same trees to be used as parents in each selection process.

8 Genetic operators

In all cases, I used 40% crossover and 60% mutation. This is because I was frequently faced with the issue of premature convergence - while I couldn't quite explain which of my parameters was causing it, I was forced to resort to fairly extreme mutation probabilities in order to prevent it from re-occurring, otherwise I kept getting unusually small trees with relatively low accuracy.

The crossover method selects two trees using the given selection method, and chooses a random point in that tree to combine those trees, creating a new tree which is then added to the population. Mutation, on the other hand, selects random points in existing trees which are then changed into new random nodes.

9 Experimental setup

9.1 Parameters

Most parameters can be inferred by the above information, but for simplicity, here are the major ones worth discussing in one place:

- Population size: 100
- Crossover likelihood: 40%
- Mutation likelihood: 60%
- Selection method: tournament
- Tournament size: 10
- Number of generations: 20

9.2 Development machine software configuration

In order to overcome the Windows-based limitation of only being able to run Jupyter notebooks in single-threaded mode, I used an Ubuntu 22.04 virtual machine running inside of Windows to perform the training of this model. This enabled multi-threading support, which sped runtimes up significantly: an important factor, since load shedding meant that getting a complete run without having to interrupt it for power outages could be challenging without it. The virtual machine was assigned 5 CPU cores (10 threads), and 16GB of RAM. Of course, there is a performance penalty applied here, compared to running the program on a bare-metal installation of an operating system, but the convenience tradeoff made it worthwhile in this case.

9.3 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)

10 Runtimes

10.1 DEAP

Performing 4 runs of training with this model, where each subsequent run was forced to explore a new area of the search space through the penalisation of similarity in the fitness function, took a total of 101 minutes and 55 seconds. This works out to 25 minutes per run of 20 generations. Unfortunately, this is exceptionally slow: an unexpected outcome, given the relative lack of complexity of this implementation, and the fact that it was able to run in a multi-threaded environment once I used the Linux environment described above. It's particularly inefficient when I compare it to the 2.06 minutes per run of my canonical implementation in GPLearn. Upon examining the program as it ran, I couldn't see any specific reason why it was running so poorly: a possible reason is the virtualised nature of said environment, but even then, the performance hit should be negligible for a processor-based task like this one. The most likely reason, then, is a possible lack of optimisation in DEAP itself, exacerbated by my lack of familiarity with the framework. In the future, I would like to investigate this further and see whether I can find ways to optimise this implementation and improve those strangely slow runtimes.

11 Results

11.1 Best run comparison to provided research paper

Structure implementation is the structure-based GP from this assignment, canonical implementation is my canonical GP implementation from Assignment 1, and the rest are values taken to compare from the reasearch paper ”Seoul bike trip duration prediction using data mining techniques”, by Sathishkumar V E, Jangwoo Park, and Yongyun Cho. [2]

11.1.1 Training data evaluation

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

11.1.2 Testing data evaluation

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

11.1.3 Per-run metrics

These are the fitness scores obtained by the final population of each run after 20 generations.

Run number	Minimum	Average	Maximum
1	23.236	1.7e+08	1.7e+10
2	22.086	8476.11	672961
3	23.4322	7863,89	456311
4	21.796	353.68	8025.4

11.1.4 Discussion

While it seems that the strategy I employed was semi-successful in aiding exploration, the lack of time available to do fine-tuning and adaptation of this new paradigm given the limited available development time (largely due to the unfortunate influence of Eskom's current woes) led to this implementation not having the finesse of the canonical implementation, and thus not achieving the results which I suspect it was capable of given a little more time and training.

A very promising takeaway from my results was the gradual improvement in fitness scores over the course of the runs that were performed: this shows that each new run, which aimed to explore a new area of the search space, seemed to hone in on a new, potentially more effective minima than the last run, as a result for the last minima deliberately being penalised. This implies that, given a longer set of runs, we would be able to create a gradually more and more accurate system. Furthermore, with some improvement on the penalisation formula, we could use this technique to better balance an accurate result that might be a global minimum.

11.2 Additional notes

Regarding my submission: included in the ZIP file, you'll notice there are no CSV files, for size reasons. Should you wish to recreate my setup, please use the given "For_modelling.csv" file, and run through the "A1_dataset_cleanup.ipynb" notebook to create all other referenced CSV files. Furthermore, should you want to access the code outside of this ZIP, please email me at u19055252@tuks.co.za, and I'll give you access to the GitHub repository containing it all.

References

[1] Trevorstephens. Is there any way to get the formula expresssion of each individual? thanks. · issue 281 · trevorstephens/gplearn, Nov 2022.

[2] Sathishkumar V E, Jangwoo Park, and Yongyun Cho. Seoul bike trip duration prediction using data mining techniques. *IET Intelligent Transport Systems*, 14(11):1465–1474, 2020.