COS710 Assignment 1 report

David Walker

2023

1 Introduction

The purpose of this assignment was to build a 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.

Ultimately, the GP which I ended up with was still not as accurate as I would have liked - however, I am willing to accept this, as an unfortunate outcome of the learning process. Luckily, functionality-wise, I think I've adequately understood how GP works, and have proven so over the course of this project's work.

2 Dataset information

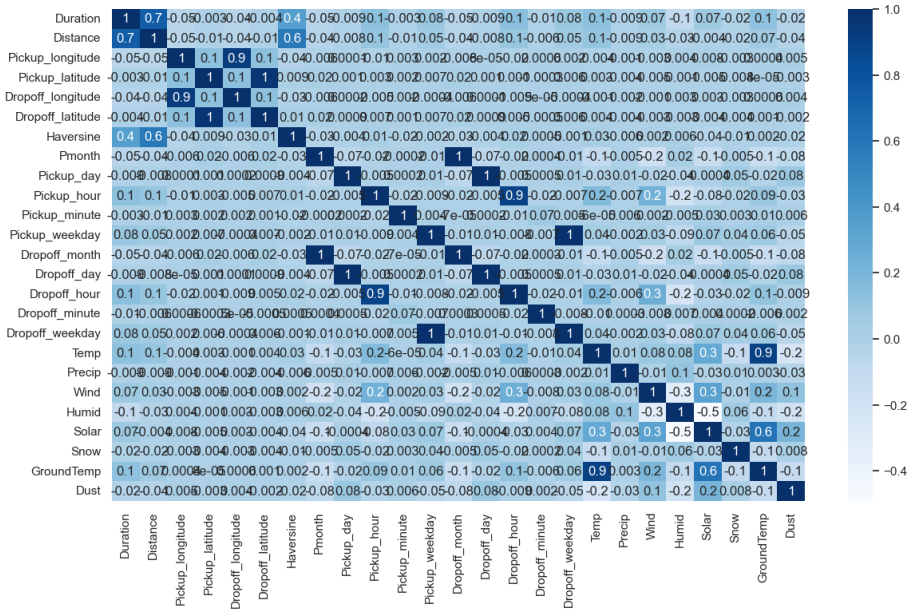
The dataset provided to use came in the form of a 9 601 139 line long CSV file. It contained 25 columns, each with a different feature relating to a given bike trip.

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.

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 Genetic program representation

3.1 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.

3.2 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, 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.

4 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 tournament size of 100, which is small but seemed to prove adequate, for the sake of runtime efficiency.

5 Fitness evaluation

5.1 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.

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

5.2 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.

6 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, amd 20, after which 20 seemed to give the best fitness scores.

7 Genetic operators

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

In the case of GPLearn, where options exist for point, hoist, and subtree mutation, I used a 50/50 split of point and subtree mutation, with the logic that hoist mutation might incidentally occur regardless, but I didn't want it to be a primary factor.

8 Experimental setup

8.1 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: 250
- Crossover likelihood: 40%
- Mutation likelihood: 60%
- Selection method: tournament
- Tournament size: 20
- Number of generations: 20

8.2 Development machine specifications

- CPU: AMD Ryzen 5 5600 (3.5GHz base, 4.4GHz boost)
- RAM: 16GB DDR4 (4000MT/s, CL18)
- Storage: Samsung 970 Evo Plus NVMe (2TB)
- GPU: AMD Radeon RX 6600XT (8GB)

9 Results

9.1 GPLearn

In GPLearn, I ran 10 runs, each with a different sequential random seed (from 1 through 10). Below are the results:

9.1.1 Results

Run number: 1(data: training)
MAE: 12.501677810330321
RMSE: 20.468111045784056
MedAE: 6.54177968626702
R2: -0.023738299382292327

Run number: 1(data: testing)
MAE: 12.530807433084027
RMSE: 20.523939292824636
MedAE: 6.546900433928089
R2: -0.02633974103580683

Run number: 2(data: training)
MAE: 7.390901538976721
RMSE: 13.60401068569152
MedAE: 2.9852319744325335
R2: 0.5477616885327259

Run number: 2(data: testing)
MAE: 7.404673117801045
RMSE: 13.631689785113263
MedAE: 2.9901336085130463
R2: 0.5472389155244958

Run number: 3(data: training)
MAE: 11.926645310467233
RMSE: 21.708249747009695
MedAE: 5.255941090108539
R2: -0.15155063957211068

Run number: 3(data: testing)
MAE: 11.933287673716174
RMSE: 21.300167546056564
MedAE: 5.250212440993687
R2: -0.10544144233474584

Run number: 4(data: training)
MAE: 11.041607496057393
RMSE: 19.05336752546425
MedAE: 4.60886996954383
R2: 0.11289115046572185

Run number: 4(data: testing)
MAE: 11.068932222792691

RMSE: 19.109742360245676
MedAE: 4.610640045483436
R2: 0.11022670588687555

Run number: 5(data: training)
MAE: 9.114207273636211
RMSE: 14.840314077708888
MedAE: 4.682813237945501
R2: 0.4618298484610621

Run number: 5(data: testing)
MAE: 9.11965732449673
RMSE: 14.857702983995036
MedAE: 4.676813939668005
R2: 0.46213529408075216

Run number: 6(data: training)
MAE: 13.531774396427853
RMSE: 19.42645431090037
MedAE: 9.836807435843635
R2: 0.0778097969136966

Run number: 6(data: testing)
MAE: 13.556806127573875
RMSE: 19.476238378267606
MedAE: 9.841144375447005
R2: 0.07577041649110294

Run number: 7(data: training)
MAE: 11.098365057937766
RMSE: 20.15239111617484
MedAE: 3.814407640585964
R2: 0.007600381165115633

Run number: 7(data: testing)
MAE: 11.123939060613134
RMSE: 20.20263037384583
MedAE: 3.814886096794335
R2: 0.005544076529336528

Run number: 8(data: training)
MAE: 7.380212089360032
RMSE: 13.224084895888948
MedAE: 3.530777819506824
R2: 0.5726687241646284

Run number: 8(data: testing)
MAE: 7.394807288892565
RMSE: 13.254204256822643
MedAE: 3.5366000677990383
R2: 0.5719672286968922

Run number: 9(data: training)
MAE: 9.111385509029548
RMSE: 14.839341042399008
MedAE: 4.678639912728819
R2: 0.46190041858377795

Run number: 9(data: testing)
MAE: 9.116843922891372
RMSE: 14.856739382219748
MedAE: 4.672867118132496
R2: 0.4622050586438461

Run number: 10(data: training)
MAE: 12.501241300470536
RMSE: 20.59880358713073
MedAE: 6.541638829849578
R2: -0.03685354114221551

Run number: 10(data: testing)
MAE: 12.52768896171623
RMSE: 20.649699931006598
MedAE: 6.5440543997721825
R2: -0.03895608991908839

9.1.2 Discussion

These results prove that the GP is not particularly accurate. While we definitely have some stronger contenders than others over the course of the 10 runs we performed, the R-squared score of these solutions is never higher than .573, indicating only a moderate level of positive correlation between the predicted durations and the actual durations. However, with this being said, a median absolute error of only 3.53 in that same same run isn't horrific, indicating that with further training and refinement, this model could go somewhere. To take that particular run further, I would ideally have liked to let it run for an additional 30 generations, to take it to 50: however, time constraints prevented me from exploring this possibility.

One major battle I fought over the course of this training was premature convergence: while usually, bloat is an issue, the opposite kept happening in my training runs, whereby small solutions would dominate the mating pool and eventually lead to more complex but accurate solutions being ignored. To combat this, I initially tried using half-and-half tree generation: however, this went the opposite way and introduced major bloat and increased runtimes significantly, while providing similar accuracy scores, so I ended up scrapping that idea. Eventually, the best solution was just to increase the population size by 50 to 150, to increase the likelihood of mutation to 50% from 30%, and to decrease tournament size from 15 to 10. I also increased the initial depth of the trees, to prevent these overly-short solutions from popping up at all, since they clearly weren't accurate.

9.2 DEAP

Using DEAP, I on average had similar fitness values (especially MAE) to GPLearn, but evaluation of statistics became something of a challenge. Nonetheless, just for reference, the best run of my DEAP algorithm led to a MAE of 12.78, producing the program "protected_div(mul(Haversine, Precip), protected_div(Precip, Pickup_longitude))".

10 Runtimes

10.1 GPLearn

In total, running 20 runs (10 runs, each with one run for testing and one for training data) took 41 minutes and 17 seconds - a speedy average of 2.06 minutes per run of 20 generations. This is in no small part due to three significant hardware properties I took advantage of. Firstly, I increased the number of parallel processes for evaluation to 6, since I have a 6-core CPU. This allowed evaluations to happen in parallel, which massively reduced overall time taken. Secondly, I stored the dataset on an NVMe-protocol PCIe-based SSD, which sped up reading in the large dataset initially to RAM. And lastly, I used very fast RAM: at 4000MT/s, it has a significantly higher throughput than most DDR4, which was no doubt benefitical to operation speed, since it allowed the evaluations to occur much more quickly when recalling data from the dataset. Unfortunately, my understanding of vectorisation hadn't quite reached the level where GPU-based parellelisation was an option for me just yet.

10.2 DEAP

DEAP, sadly, does not have built-in multiprocessing, only allowing multiprocessing when executed using a library to map the processes to various Python instances. I was unable to use these libraries as a restriction on how Python spawns processes in Windows specifically (that is, the lack of a "fork" operation)

prevents interactive notebooks (such as the .ipynb notebook I used for this assignment) to run in multiple threads. I attempted to run it via a Python instance in a Windows Subsystem for Linux backend, but my IDE seemed decidedly unhappy with that solution. Hence, it took much longer to run: frustratingly so, to the point that I shifted my attention almost entirely back to GPLearn, despite DEAP's customisation and feature enhancements.

11 Comparatives

11.1 Best run comparison to provided research paper

GPLearn implementation is my own, 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. [1]

11.1.1 Training data evaluation

	GPLearn	LR	GBM	KNN	RF
MAE	7.38	10.11	7.35	5.53	1.21
RMSE	13.22	16.45	12.55	11.29	2.76
MedAE	3.53	6.9	3.74	2	0.4
R2	0.57	0.56	0.74	0.79	0.98

11.1.2 Testing data evaluation

	GPLearn	LR	GBM	KNN	RF
MAE	7.39	10.12	7.37	6.83	2.92
RMSE	14.84	16.48	12.58	13.93	6.25
MedAE	3.54	6.9	3.75	2.59	1.2
R2	0.57	0.56	0.74	0.69	0.93

11.1.3 Discussion

I was surprised to find that, although I thought my results were fairly inaccurate, they fall roughly in line with the gradient boosting machine model in most measures, only falling behind in the R-squared value. Of course, however, it comes nowhere close to the K-nearest-neighbours and random forest models.

The clear conclusion is that symbolic-regression GP is unlikely to be the ideal method to solve the bike trip duration problem, especially given that the random forest method is already so effective. However, a correlation does exist, and perhaps with more training and fine-turning, and a larger population in

which individuals can be evaluated more quickly and in an even more parallel manner, a "usable", albeit imperfect, solution here may be found.

11.2 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. Alternatively, I will host them on a Google Drive folder () for the remainder of the month, should you want to download them from there.

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] 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.