UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Analiză a performantei Retelelor Neuronale Artificiale si a Random Forests pe telemetria autovehiculelor

Conducător ştiinţific

Prof. dr. Horia F. Pop

Absolvent

Antoniu Ficard

2024

BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION INFORMATICS

DIPLOMA THESIS

Performance analysis of Artificial Neural Networks and Random Forests on automotive telemetry

Supervisor

Prof. dr. Horia F. Pop

Author

Antoniu Ficard

2024

ABSTRACT

This study explores the application of Artificial Neural Networks (ANN) and Random Forest (RF) algorithms on automotive telemetry data to analyze driver behavior and predict fuel consumption. The research leverages data collected from the On-Board Diagnostics II (OBD-II) port, GPS, and accelerometers. The primary focus is on evaluating the performance of ANNs and RFs in classifying road corners and estimating fuel consumption for different driving scenarios.

The methodology involves collecting extensive telemetry data, calculation precise fuel consumption using engine data, preprocessing it to ensure accuracy, and segmenting the road into corners and straight sections. These segments are then classified using ANNs and RF models. Additionally, the study assesses the models' effectiveness in predicting fuel consumption by incorporating various features such as engine load, speed, and acceleration.

The results indicate that while Random Forests exhibit robust performance with limited data, Artificial Neural Networks outperform when extensive features are provided. This demonstrates the significance of feature engineering in enhancing model accuracy. The study concludes with recommendations for future work, including further classification of driver behavior and exploring additional predictive features to refine the models.

This research contributes to the automotive industry by providing insights into optimizing fuel consumption and improving vehicle performance through advanced machine learning techniques.

Table of Contents

1. INTRODUCTION

Today, transportation accounts to nearly a quarter of all carbon dioxide emissions and the climate change caused a push towards greater fuel efficiency. However, the fuel consumption of a vehicle is affected by many variables, like engine size, volumetric efficiency, thermal efficiency, vehicle wear, driver behaviour, road profile, traffic density, weather conditions, tire pressures, the amount of energy consumed by auxiliar functions of the vehicle (lights, air conditioning, power steering pump) etc. and correlations between these inputs can be made, using past values, to properly approximate future fuel consumption values.

Computer science and artificial intelligence have developed rapidly in the past decade and the scientific community has tried to apply such concepts in many fields, ranging from medicine, astronomy, economy, finance to many others, but in the automotive industry the data that was collected and used to train artificial intelligence models is relatively scarce because it is necessary to use dedicated equipment to collect data and it is costly on large scale. After cars increasingly added more electronic modules and emission control systems, at the beginning of the millennium, automotive manufacturers had to implement standardised diagnostics ports and protocols on their products and as such every vehicle manufactured in the past 20 years is equipped with an OBD-2(on-board diagnostics) port corresponding to SAE J1962 and transmitting data over protocols defined in SAE J1979 and SAE J1978. These protocols were implemented over standards ISO 9141-2 or SAE J1850 in older vehicles and now over CAN BUS.

The purpose of this study is to use data provided by the OBD-2 port in synergy with GPS and topographic data and evaluate the performance of random forests and deep learning methods to provide insights on driver behaviour and road profiles, as well as estimate fuel consumption for a given route.

2. THE EVOLUTION OF VEHICLE TELEMETRY AND PAST STUDIES

In the last two decades, the OBD-II port has come to equip most vehicles around the world, with slight variations depending on each country's regulation. Moreover, it has come to support newer standards like CAN to make it easier for devices to communicate. The purpose of the OBD-II connector was to standardise emission control system diagnosis, since automobiles have become increasingly dependent on electronic. Previously, Wickramanayake et al [1] used parameters like load, fuel level, acceleration, bearing, altitude and GPS coordinates to predict the fuel consumption of fleet vehicles, without considering necessarily the characteristics of the road itself. They have compared the accuracy of Random Forrests, Artificial Neural Networks and Gradient Boosting, but such models require training on a particular route. When a vehicle would take a new route, the model would need to be trained again, not being able to achieve wide applicability. The purpose of their study was to compare the performance of different techniques and concluded that random forests were the most effective.

Similarily Zhao et al [2] have attempted to predict the fuel consumption of light-duty petrol powered vehicles using OBD-II data as verification. They have used a broader data collection method, using 143 vehicles and a custom developed Android application to predict different values. Traffic density was Fuzzy classified depending on speed and vehicles were Fuzzy classified depending on their engine displacement and intake technology (turbocharged, naturally aspirated, etc.). After that, a correlation was made using a linear regression model between traffic speed and engine category, providing satisfactory results for traffic speeds above 20 km/h. Again, similarily to the previous study made by Wickramanayake et al [1] the input data set is not complete, disregarding other elements like driver behaviour.

Also taking advantage of OBD-II equipment, Yao et al [3] have used the fact that taxis in Beijing come with OBD-II monitors and together with an Android application that records data, they attempted to obtain a complete analysis of fuel consumption in relation to driver behaviour. The team responsible for this study has recorded instantaneous fuel consumption, average fuel consumption, average fuel consumption excluding idle time and gyroscope, accelerometer and GPS data from the mobile phones. The data between the two devices was combined using Pearson correlation and attempted to analyze the impact of driver behaviour on fuel consumption. They have used Artificial Neural Networks, Support Vector Machines and

Random Forrests to compare the performance of each model. The study has proven high accuracy results and the prediction using Random Forrests has shown the best performance overall. Their goal was mainly to predict the fuel consumption with the help of just the smartphone sensors, but their study was mostly concentrated among taxi drivers in Beijing, showing just urban driving. This could be improved by analysing different road profiles and also taking into consideration road characteristics.

Karaduman et al [4] had a different approach and initially used the sensors on a smartphone to determine the starting end ending point for road shapes, Fuzzy classified the corners in the road as left curved, right curved or straight and depending on acceleration values classified the driver behaviour as safe or aggressive. This shows a rudimentary approach of classification, but with methods that can be scaled. Using a similar approach, road sectors can be split, but further classified into more categories, also depending on the corner radius. This parameter can vary as there exist curves with variable radius (that either tighten or widen), but such corners can impact driver behaviour and a wide, long, high speed corner could be treated differently by a driver compared to a tight hairpin. An important factor for such behaviours can be the vehicle condition and wear, suspension type and performance or even the amount of weight loaded in the car. Considering these elements, Karaduman et al [4] had a different, but insightful approach.

3. DATA COLLECTION AND MEASUREMENT METHODOLOGY

The vehicle used for data is a Chevrolet Spark M250, built by General Motors Korea in 2007. At the time of this paper's writing, the odometer reads in between 110000km and 120000km. The engine that this vehicle is using has the internal code A08S3. The F8CV engine built by Daewoo is identical to the A08S3. The F8C engine has none to very minor modification, being a predecessor for the F8CV. The most important updates from the F8C to the F8CV include the use of an electronic distributor, rather than a mechanical distributor and the use of lower friction materials, slightly softer alloys for rollers, bearings and cams. Considering the similarity between the two engine codes and the scarcity of performance data, they were used interchanginbly for calculations.

OBD-II queries can be made to find live parameter values, using mode 1. The structure of an OBD-II query is as follows, mentioned in [5]:

Header bytes (Hex)			Data bytes								
Priority/Type	Target address (hex)	Source address (hex)	#1	#2	#3	#4	#5	#6	#7	ERR	RESP
Diagnostic request at 10.4 kbit/s: SAE J1850 and ISO 9141-2											
68	6A	F1	Maximum 7 data bytes							Yes	No
Diagnostic response at 10.4 kbit/s: SAE J1850 and ISO 9141-2											
48	6B	ECU addr	Maximum 7 data bytes							Yes	No
Diagnostic request at 10.4 kbit/s (ISO 14230-4)											
11LL LLLLb	33	F1	Maximum 7 data bytes							Yes	No
Diagnostic response at 10.4 kbit/s (ISO 14230-4)											
10LL LLLLb	F1	addr	Maximum 7 data bytes							Yes	No
Diagnostic request at 41.6 kbit/s (SAE J1850)											
61	6A	F1	Maximum 7 data bytes							Yes	Yes
Diagnostic response at 41.6 kbit/s (SAE J1850)											
41	6B	addr	Maximum 7 data bytes							Yes	Yes

Figure 1 – OBD-II frame structure [5]

Also mentioned in document [5], PID 0x00 provides available PIDs from 0x01 to 0x20. Interrogating mode 1 for PID 0x00 revealed that the following PIDs were available:

PID 0x01 - Monitor status since DTCs cleared.

PID 0x03 - Monitor Fuel system status.

PID 0x04 - Calculated engine load

PID 0x05 - Engine coolant temperature

PID 0x06 - Short term fuel trim Bank 1

PID 0x07 - Long term fuel trim Bank 1

PID 0x0B - Intake manifold absolute pressure

PID 0x0C - Engine speed

PID 0x0D - Vehicle speed

PID 0x0E - Timing advance

PID 0x0F - Intake air temperature

PID 0x11 - Throttle position

PID 0x13 - Oxygen sensors present

PID 0x14 - Oxygen sensor 1

PID 0x15 - Oxygen sensor 2

PID 0x1C - OBD-II standards this vehicle conforms to

To estimate the fuel consumption of the vehicle, the volumetric efficiency of this engine had to be calculated. In the service manual of the vehicle [6] the graph (Figure 2) was shown:

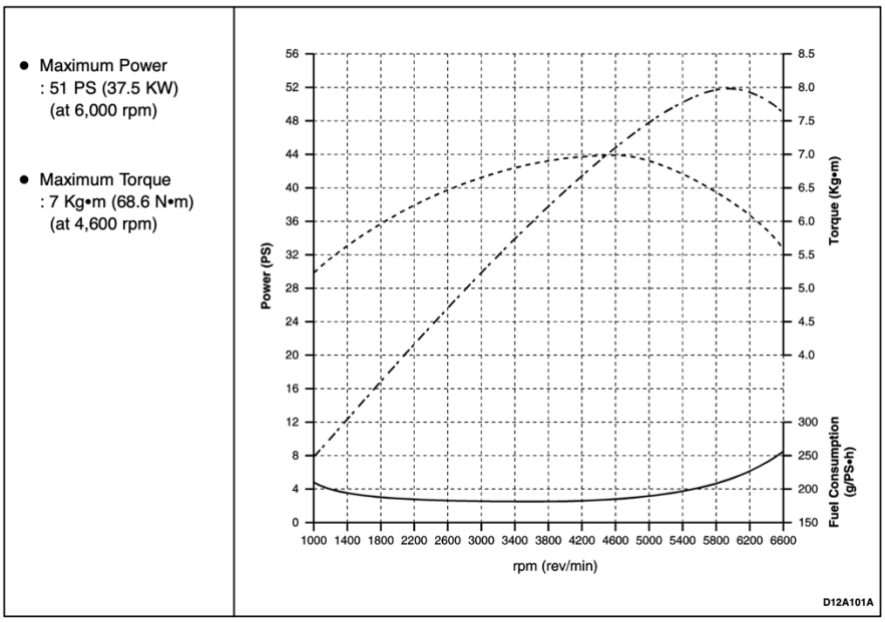

Figure 2 – Power, Torque and BSFC graph for F8C [6]

Using the graph given above, data was extracted to indicate engine torque, power and brake specific fuel consumption and was constructed in the table below(Table 1, the table is not specifically given in the document [6]):

Engine speed (RPM)	Torque(Nm)	Power(PS)	Fuel consumption(g/PSh)
900	51.39	6.59	210.34
1000	51.39	7.32	210.34
1100	52.54	8.23	204.48
1200	53.48	9.14	200.42
1300	54.37	10.06	197.15
1400	55.37	11.04	194.81
1500	56.20	12.00	192.68
1600	57.05	13.00	191.06
1700	57.79	13.99	189.66
1800	58.55	15.01	188.68
1900	59.25	16.03	187.23
2000	59.93	17.07	186.84
2100	60.60	18.12	185.79
2200	61.24	19.18	185.34
2300	61.82	20.24	185.18
2400	62.46	21.34	184.48
2500	62.93	22.40	184.48
2600	63.44	23.49	184.11
2700	63.94	24.58	183.62
2800	64.36	25.66	183.62
2900	64.80	26.76	183.35
3000	65.23	27.86	182.76

3100	65.64	28.97	182.76
3200	66.01	30.08	182.76
3300	66.35	31.17	182.76
3400	66.61	32.25	182.76
3500	66.94	33.36	182.76
3600	67.16	34.42	182.76
3700	67.40	35.51	182.76
3800	67.63	36.59	182.76
3900	67.83	37.66	182.76
4000	68.01	38.73	182.76
4100	68.12	39.77	182.76
4200	68.23	40.80	183.33
4300	68.27	41.80	183.77
4400	68.35	42.82	184.48
4500	68.49	43.88	185.32
4600	68.56	44.91	186.37
4700	68.464	45.81	187.5
4800	68.28	46.66	188.52
4900	67.92	47.38	189.53
5000	67.615	48.13	190.5
5100	67.16	48.76	191.51
5200	66.72	49.40	193.18
5300	66.25	49.99	195.21
5400	65.68	50.49	197.48
5500	65.078	50.96	200.1
5600	64.41	51.36	202.17
5700	63.70	51.70	205.69
5800	62.97	52.00	209.02
5900	62.19	52.25	213.15
6000	61.43	52.48	217.15
6100	60.58	52.62	222.08

6200	59.63	52.64	227.37
6300	58.70	52.65	233.26
6400	57.60	52.49	240.33
6500	56.27	52.07	247.54
6600	55.10	51.78	257.76

Table 1 – Power, torque and BSFC for A08S3, extracted from graph

The first observed pattern is operation in closed-loop mode. This is the default mode of operation that the fuel system is using, unless special conditions are met. In the closed-loop operation, the upstream oxygen sensor read value is oscillating around $0.45mV$, indicating that the fuel system is trying to achieve stoichiometric values relying on feedback.

The second observed pattern is in open-loop mode and it occurs during insufficient temperature. There is insufficient data about this mode in order to find the target air-fuel ratio, as this mode occurs for a very short amount of time while the oxygen sensor heats up. Also, this data is irelevant as the manufacturer recommends not driving the vehicle for the first 30 seconds of engine operation. This amount of time is plenty for the oxygen sensors to reach a sufficiently high temperature.

The third observed pattern is in open-loop mode and it occurs during engine braking (defined as $MAP < 32kPa$ and $TP < 10\%$). In this operation mode, the oxygen sensor reads and the short term fuel trim indicates , meaning that under this condition no fuel is injected.

The final observed pattern is in open-loop mode and it occurs under maximum throttle plate opening and high engine speeds (defined as $MAP > 90kPa$ and $TP \geq 90.98\%$). This operation mode is targeting rich fuel mixtures in order to obtain maximum power output while also protecting the engine from excessive wear or damage.

Defining input variables and predefined data

Variables:

ω - engine speed, read from OBD-II (given in RPM)

$LOAD_{PCT}$ - calculated engine load, read from OBD-II (given in $\%$)

U_{O_2} - voltage sent back from the pre-catalytic converter oxygen sensor, read from OBD-II (given in mV)

$MAP \in [20{,}105] \cap \mathbb{N}$ - manifold absolute pressure, read from OBD-II (given in kPa)

IAT - intake air temperature, read from OBD-II (given in $°C$)

TP - throttle position (given in $\%$)

AAT – ambient air temperature

AAP – ambient air pressure

Constants:

WOT - wide-open throttle

STP - standard temperature and pressure, $p = 1atm$ & $T = 25°C$

$Cyl = 3$ - the number of cylinders

$Bore = 68.5mm$ - cylinder bore

$Stroke = 72mm$ - cylinder stroke

$AFR_{petrol} = 14.7$ - stoichiometric air-fuel ratio for petrol

$AFR_{E10} = 14.05$ - stoichiometric air-fuel ratio for E10 petrol.

$M_{air} = 28.9647 \frac{g}{mol}$ - molar mass of air

$R = 8.31446261815324 J \cdot K^{-1} \cdot mol^{-1}$ - universal gas constant

As the engine does not operate outside the interval $[900{,}6600]RPM$, we will define functions in relation to engine speed to use RPM with domain $[900{,}6600] \cap \mathbb{N}$ (better OBD-II compatibility).

We define engine power $@WOT\&STP$ through the following function of engine speed:

$$P: [900{,}6600] \cap \mathbb{N} \to \mathbb{R}, P(\omega)$$

We define engine torque $@WOT\&STP$ through the following function of engine speed:

$$\tau: [900,6600] \cap \mathbb{N} \to \mathbb{R}, \tau(\omega)$$

Expressing power in relation to torque and engine speed, using $N \cdot m$, RPM and obtaining a result in PS the following equation results:

$$P(\omega) = \frac{\pi \cdot \tau(\omega) \cdot \omega \cdot 1.35962}{30 \cdot 1000}$$

Expressing brake specific fuel consumption as a function of engine speed:

$$BSFC: [900,6600] \cap \mathbb{N} \to \mathbb{R}, BSFC(\omega)$$

Expressing target air-fuel ratio as a function of engine speed and throttle position (noticed behaviour during testing):

$$\lambda_{target}: ([900,6600] \cap \mathbb{N}) \times [20,105] \cap \mathbb{N} \to \mathbb{Q}, \lambda_{target}(\omega, MAP)$$

Volumetric efficiency is expressing how well the engine is filling up its displacement with air mass. Using this data alongside manifold absolute pressure, intake air temperature, engine load and engine speed, an estimation of airflow and air mass entering one cylinder at each intake stroke can be made (this method is called the speed-density method, usually applied to vehicles not equipped with a mass-airflow sensor).

Using the $BSFC$ function and the P function, the function FCR_MAX of ω that specifies the fuel consumption rate $@WOT\&STP$ is defined.

$$[BSFC] = \frac{g}{PS \cdot h}$$
$$[P] = PS$$
$$[\omega] = RPM$$

$$FCR\_MAX: [900,6600] \cap \mathbb{N} \to \mathbb{R}, FCR\_MAX(\omega) = BSFC(\omega) \cdot P(\omega), [FCR\_MAX] = \frac{g}{h}$$

$$60\frac{g}{h} = 1\frac{g}{min}$$

Furthermore, the mass of fuel used to complete one engine cycle can be calculated, knowing that one cycle occurs every 2 revolutions.

$$RPM = \frac{revolutions}{minute}$$

$$\frac{[FCR\_MAX(\omega)]}{[\omega]} = \frac{\frac{g}{min}}{\frac{revolutions}{minute}} = \frac{g}{min} \cdot \frac{min}{revolutions} = \frac{g}{revolutions}$$

$$2revolutions = 1cycle \Longrightarrow 1\frac{g}{revolutions} = 2 \cdot \frac{g}{cycle}$$

As an engine cycle is unit-less, the measurement unit can be simplified to g. To obtain the amount of fuel entering each cylinder, the value is divided by the number of cylinders and the following function of engine speed for intake stroke fuel mass gets defined:

$$ISFM\_MAX\colon [900,6600] \cap \mathbb{N} \to \mathbb{R}, ISFM\_MAX(\omega) = \frac{FCR\_MAX(\omega)}{\frac{\omega}{2} \cdot 60 \cdot Cyl} = \frac{FCR\_MAX(\omega)}{30 \cdot \omega \cdot Cyl}$$

$$[ISFM\_MAX] = g$$

$$AFR = \frac{m_{air}}{m_{fuel}} \Longrightarrow TAI(\omega) = ISFM\_MAX(\omega) \cdot AFR_{petrol}$$

$$[TAI] = g$$

To determine the mass of air that can be displaced in one cylinder $@STP$, the swept volume of one cylinder must be determined.

$$V = \pi \cdot r^2 \cdot h$$

$$\Longrightarrow V_{swept} = \pi \cdot (\frac{Bore}{2})^2 \cdot Stroke = \pi \cdot Bore^2 \cdot Stroke \cdot 2^{-2} = 265340.486319mm^3$$

$$d = 2r$$

Using the ideal gas law (stated in [7] atmospheric air behaves like an ideal gas for low pressures and temperatures), air mass can be calculated:

$$pV = mR_{specific}T$$

$$R_{air} = \frac{R}{M_{air}} = \frac{8.31446261815324J \cdot K^{-1} \cdot mol^{-1}}{28.9647g \cdot mol^{-1}} = 0.28705502J \cdot K^{-1} \cdot g^{-1}$$

$$1Pa = 1J/m^3$$
$$1atm = 101325Pa$$
$$1m^3 = 10^9 mm^3$$

$$m_{cyl\_displacement} = \frac{pV}{R_{air}T} = \frac{101325Pa \cdot 265340.486319 \cdot 10^{-9}m^3}{0.28705502J \cdot K^{-1} \cdot g^{-1} \cdot 293K} = 0.31965931661g$$

Using everything calculated above, volumetric efficiency can be estimated as follows as a function of engine speed:

$$\eta_v = \frac{m_{actual}}{m_{cyl\_displacement}}$$

$$VE: [900,6600] \cap \mathbb{N} \rightarrow (0, \infty], VE(\omega) = \frac{TAI(\omega)}{m_{cyl\_displacement}}$$

Volumetric efficiency can exceed a value of 1 especially using forced induction as high pressure are achieved, but it can also be exceeded in naturally aspirated engines. The best naturally aspirated engines can achieve a peak volumetric efficiency of 1.15.

A volumetric efficiency of 0 is not possible as that would mean no air would enter the engine and no fuel would be used. As volumetric efficiency is calculated $@WOT$, a situation where no air enters the cylinders is impossible on Earth.

For the A08S3 engine the following values were estimated $@WOT\&STP$ (Table 2):

Engine speed (RPM)	Volumetric efficiency (%)	Fuel consumption (g/h)	Fuel consumption for each cylinder per combustion cycle (g)	Theoretical air intake (TAI) for each cylinder per combustion cycle (g)	Brake specific fuel consumption (g/PSh)
900	48.48337525	853.9804	0.010542968	0.154981628	210.34
1000	50.51346732	988.598	0.010984422	0.161471007	210.34
1100	56.03993406	1206.432	0.012186182	0.179136873	204.48
1200	60.59065022	1422.982	0.013175759	0.193683661	200.42
1300	64.31599224	1636.345	0.013985855	0.205592064	197.15
1400	66.83416352	1831.214	0.014533444	0.213641633	194.81
1500	69.57267414	2042.408	0.015128948	0.222395538	192.68
1600	71.99780407	2254.508	0.015656306	0.230147692	191.06
1700	74.10672652	2465.58	0.016114902	0.236889059	189.66
1800	76.05524321	2679.256	0.016538617	0.243117674	188.68
1900	77.03724085	2864.619	0.016752158	0.246256721	187.23
2000	78.76100693	3082.86	0.017127	0.2517669	186.84
2100	80.01359046	3288.483	0.017399381	0.2557709	185.79
2200	81.35715152	3502.926	0.017691545	0.260065718	185.34
2300	82.27799349	3703.6	0.017891787	0.263009275	185.18
2400	83.26478439	3910.976	0.01810637	0.266163644	184.48
2500	84.45876998	4132.352	0.018366009	0.269980331	184.48
2600	85.38930635	4344.996	0.018568359	0.272954877	184.11
2700	86.17778996	4553.776	0.018739819	0.275475338	183.62
2800	86.78589936	4755.758	0.018872056	0.277419217	183.62

2900	87.54667572	4968.785	0.01903749	0.279851109	183.35
3000	88.09145491	5172.108	0.019155956	0.281592547	182.76
3100	88.86462738	5391.42	0.019324086	0.284064065	182.76
3200	89.58947656	5610.732	0.019481708	0.286381113	182.76
3300	89.9874162	5811.768	0.019568242	0.287653164	182.76
3400	90.63660399	6031.08	0.019709412	0.289728353	182.76
3500	91.24869535	6250.392	0.019842514	0.29168496	182.76
3600	91.82678162	6469.704	0.019968222	0.293532867	182.76
3700	92.37361999	6689.016	0.020087135	0.295280886	182.76
3800	92.64593222	6890.052	0.020146351	0.296151358	182.76
3900	93.14372373	7109.364	0.020254598	0.297742595	182.76
4000	93.61662567	7328.676	0.020357433	0.29925427	182.76
4100	94.06645922	7547.988	0.020455252	0.300692205	182.76
4200	94.78958693	7791.525	0.0206125	0.30300375	183.33
4300	95.20945993	8012.372	0.020703804	0.304345913	183.77
4400	95.97587497	8264.704	0.020870465	0.30679583	184.48
4500	96.3746266	8487.656	0.020957175	0.308070477	185.32
4600	96.6768534	8703.479	0.021022896	0.309036573	186.37
4700	97.02816532	8925	0.021099291	0.310159574	187.5
4800	97.12902025	9124.368	0.021121222	0.310481967	188.52
4900	97.04000882	9305.923	0.021101866	0.310197433	189.53
5000	96.94865114	9486.9	0.021082	0.3099054	190.5

5100	96.70284902	9652.104	0.021028549	0.309119671	191.51
5200	96.61933701	9832.862	0.021010389	0.308852717	193.18
5300	96.54527181	10014.273	0.020994283	0.30861596	195.21
5400	96.60672385	10209.716	0.021007646	0.308812398	197.48
5500	96.10863184	10345.17	0.020899333	0.3072202	200.1
5600	95.36887944	10452.189	0.02073847	0.304855513	202.17
5700	95.32708616	10634.173	0.020729382	0.304721916	205.69
5800	95.20019667	10806.334	0.020701789	0.304316302	209.02
5900	95.43580059	11019.855	0.020753023	0.305069432	213.15
6000	95.60631494	11226.655	0.020790102	0.305614497	217.15
6100	95.615915	11414.912	0.020792189	0.305645185	222.08
6200	95.37766967	11573.133	0.020740382	0.304883611	227.37
6300	94.21423308	11616.348	0.020487386	0.301164578	233.26
6400	92.86687395	11631.972	0.020194396	0.296857619	240.33
6500	90.48413426	11510.61	0.019676256	0.289240969	247.54
6600	89.45977036	11555.3808	0.019453503	0.285966495	257.76

Table 2 – Volumetric efficiency and theoretical air intake for A08S3

An estimation of the amount of air entering the cylinders at any time of engine operation can be made using the speed-density method and parameters hard-coded in the ECU. However, these constant values cannot be directly accessed, and they must be deduced using available parameters and data calculated above.

Using PID 0x04 defined in [5] with the formula for calculated engine load ($LOAD\_PCT$), the current airflow can be deduced and calculated.

$$LOAD_{PCT} = \frac{current\_airflow}{peak\_airflow@WOTSTP(\omega) \cdot \frac{BARO}{29.92} \cdot \sqrt{\frac{298}{AAT + 273}}}$$

$$[AAT] =^{\circ} C$$

$$[BARO] = inHg$$

The peak airflow of the engine can be calculated using the $TAI(\omega)$ function defined earlier. As $TAI(\omega)$ is calculated for one engine stroke, we can multiply by engine speed to obtain peak airflow ($AIRFLOW_{MAX}(\omega)$):

$$AIRFLOW_{MAX}: [900,6600] \cap \mathbb{N} \rightarrow [0, \infty)$$

$$[AIRFLOW_{MAX}] = \frac{g}{s}$$

$$\frac{g}{min} = \frac{g}{60s}$$

$$AIRFLOW_{MAX}(\omega) = TAI(\omega) \cdot \omega \cdot \frac{1}{60} \cdot \frac{1}{2}$$

Thus, mass air flow (MAF) can be defined as a function of intake air temperature (IAT), manifold absolute pressure (MAP) and engine speed(ω).

$$MAF: [900,6600] \cap \mathbb{N} \times [0, \infty) \times [0, \infty) \times [0,100] \rightarrow [0, \infty)$$

$$MAF(\omega, MAP, IAT, LOAD_{PCT}) = LOAD_{PCT} \cdot AIRFLOW_{MAX}(\omega) \cdot \frac{AAP}{101.325} \cdot \sqrt{\frac{298}{AAT + 273}}$$

$$1atm = 101.325kPa$$

Using the λ_{target} function and AFR_{E10} an estimation of current fuel consumption can be made.

$$fuel\_flow: [900,6600] \cap \mathbb{N} \times [0,\infty) \times [0,\infty) \times [0,100] \rightarrow [0,\infty)$$

$$fuel\_flow(\omega, IAT, MAP, LOAD_{PCT}) = \frac{MAF(\omega, AAT, AAP, LOAD_{PCT})}{\lambda_{target}(\omega, MAP) \cdot AFR_{E10}}$$

$$[fuel\_flow] = \frac{g}{s}$$

To give the reading in $\frac{L}{100km}$, the relations below can be used:

$$\frac{g}{s} \cdot (\frac{m}{s})^{-1} = \frac{g}{m}$$

$$\frac{g}{m} \cdot (\frac{g}{L})^{-1} = \frac{L}{m} = 1000 \cdot \frac{L}{km} = 10 \cdot \frac{L}{100km}$$

$$\frac{km}{h} = \frac{10}{36} \cdot \frac{m}{s}$$

also

$$\frac{g}{s} = 3600 \cdot \frac{g}{h}$$

$$\frac{g}{h} \cdot (\frac{km}{h})^{-1} = \frac{g}{km}$$

$$\frac{g}{km} \cdot (\frac{g}{L})^{-1} = \frac{L}{km} = 100 \cdot \frac{L}{100km}$$

$$\rho_{E10} = 743\frac{kg}{m^3} = 743\frac{g}{L}$$

To read OBD-II parameters, a Carly Universal Scanner was used along with a Python script communicating with the device over Bluetooth-Low-Energy handle 24. The specific Bluetooth handle allows the direct serial communication of the script with the vehicle's ECU, allowing the program to directly send OBD-II queries to the car. The adapter and the bluetooth package act just as a proxy between the computer and the engine management unit.

A Python script communicates with the ECU of the car and queries the parameters. After a query has been made, the computer calculates the mass air flow, mass fuel flow and updates the global variables for the parameters, the travelled distance since the program has begun and the amount of fuel used since the program has begun, in grams.

Because the polling rate is limited to 6Hz, variables are updated depending on their volatility and their importance. As a result, parameters were updated periodically depending on importance in the following cycle(each is updated once every $\frac{1}{6}s$):

```
UPDATE_CYCLE = [OBDLiveDataPIDs.MAP,
                OBDLiveDataPIDs.RPM,
                OBDLiveDataPIDs.CALCULATED_ENGINE_LOAD,
                OBDLiveDataPIDs.SPEED,
                OBDLiveDataPIDs.MAP,
                OBDLiveDataPIDs.RPM,
                OBDLiveDataPIDs.CALCULATED_ENGINE_LOAD,
                OBDLiveDataPIDs.MAP,
                OBDLiveDataPIDs.RPM,
                OBDLiveDataPIDs.CALCULATED_ENGINE_LOAD,
                OBDLiveDataPIDs.SPEED,
                OBDLiveDataPIDs.MAP,
                OBDLiveDataPIDs.RPM,
                OBDLiveDataPIDs.CALCULATED_ENGINE_LOAD,
                OBDLiveDataPIDs.MAP,
```

```
OBDLiveDataPIDs.RPM,
OBDLiveDataPIDs.CALCULATED_ENGINE_LOAD,
OBDLiveDataPIDs.SPEED,
OBDLiveDataPIDs.MAP,
OBDLiveDataPIDs.RPM,
OBDLiveDataPIDs.CALCULATED_ENGINE_LOAD,
OBDLiveDataPIDs.IAT]
```

Apart from engine data, accelerometer data is also logged. An ADXL313 sensor was coupled to an Arduino UNO compatible board and glued at the bottom of a plastic cup (Figure 3) that has the shape of a conical section. The plastic cup was wrapped with rubberized insulation tape to diminish vibration (Figure 4) and inserted into the cup holder of the vehicle to ensure that it is permanently in the same position (Figure 5). The program loaded on the microcontroller was written to support both requests from the computer, but also it can send interrupts with the accelerometer data to the Python script. A separate thread, no. 2 communicates via USB at a baud rate of 115200 with the microcontroller and updates the X, Y and Z global variables periodically.

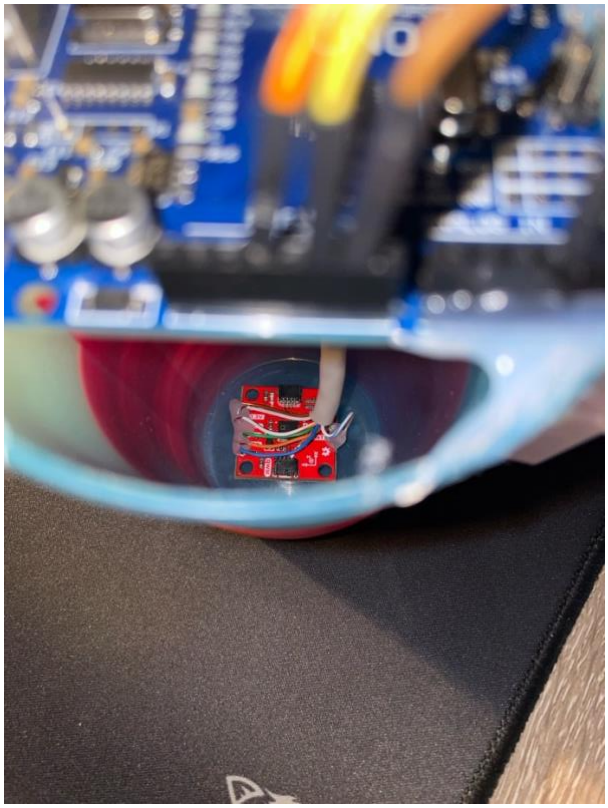

Figure 3 - ADXL313 accelerometer unit glued at the bottom of the plastic glass

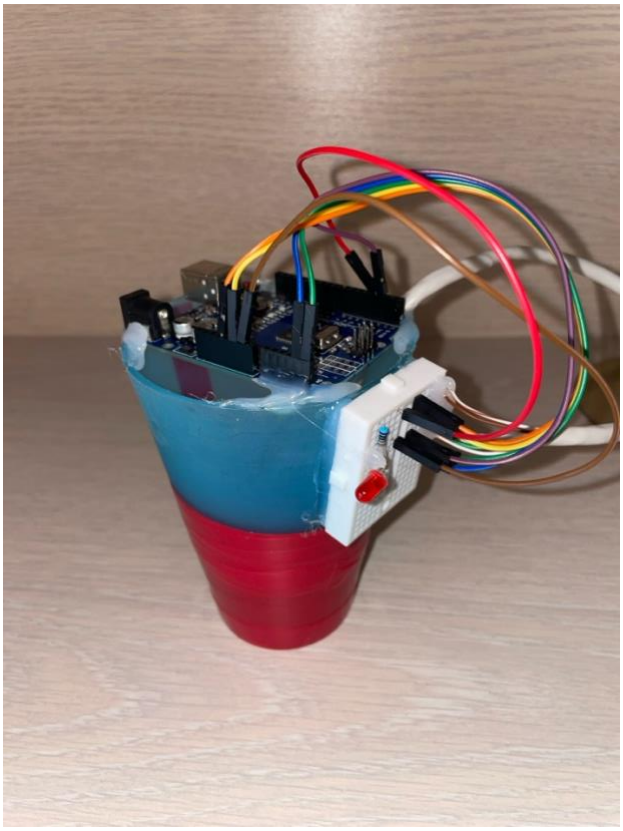

Figure 4 - Arduino UNO compatible board connected to ADXL313 in the plastic glass housing

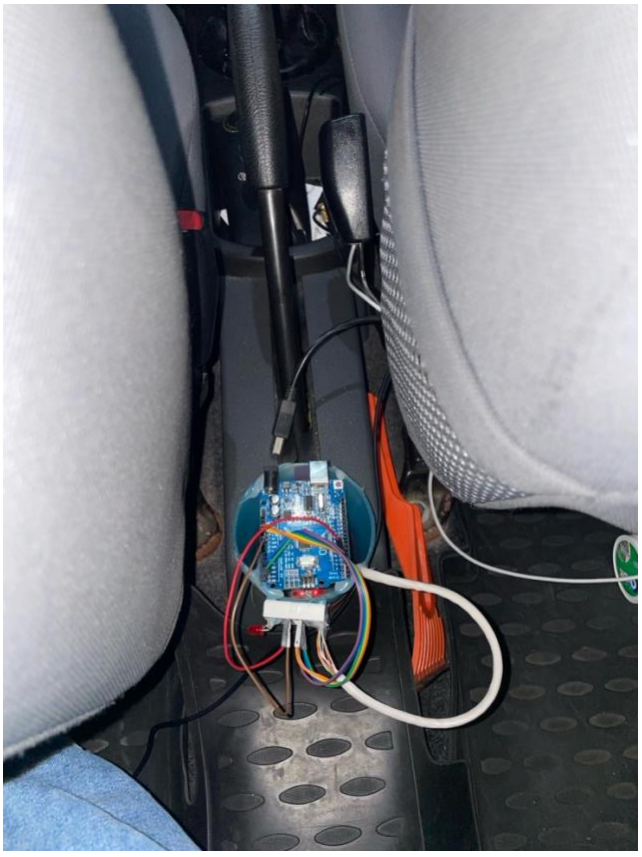

Figure 5 - Accelerometer unit in the rear cupholder of the vehicle (between front seats), aligned forward

To collect geo-position data, a U-blox NEO N8M GPS module (Figure 6) was used. An active SMA antenna (Figure 7) was coupled to the module and then it was connected via USB to the computer. The module continuously transmits NMEA sentences at a baud rate of 9600 and in the Python script a new thread, no. 3 was created to handle communication. Using the Python package pynmea2 [8], the data is parsed and then variables for the latitude, longitude, altitude, GPS speed and possibly heading are stored in variables.

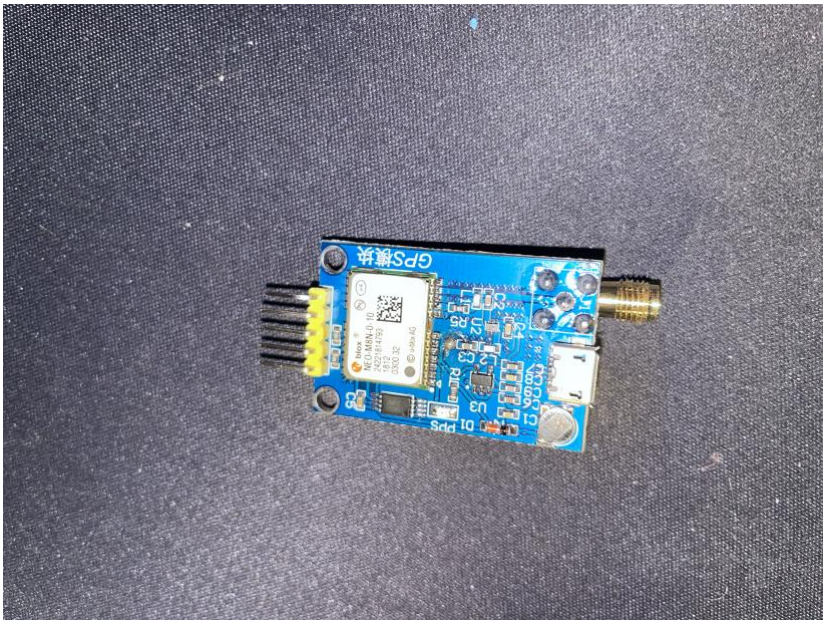

Figure 6 - GPS circuit board

Figure 7 - GPS SMA antenna mounted to the roof of the vehicle

Another thread, no. 4 reads from OpenWeather API [9] the ambient air pressure and ambient air temperature every 20 minutes and sleeps in between API calls.

Using the Python program along with the auxiliary devices, a test has been made over multiple road trips, testing both motorway driving, urban driving and curved road driving (Table 3). To test the validity of the program, the common “full-to-full” method was used. The petrol tank of the vehicle was filled and the odometer reading was written down. Immediately after starting the engine, the Python script was launched into execution and the road-trip took place. Even if the engine was temporarily turned off, the program was immediately launched again into execution upon engine restart. At the end, the program was stopped and the fuel tank was filled again, until the fuel pump clicked. The odometer reading was written down and the amount of petrol pumped in the tank was noted.

Odometer distance (km)	Fuel pump reading (L)	Python program distance (km)	Python program fuel amount (L)	Relative distance error	Relative fuel amount error
326	15.26	316.0248119	15.02	-0.015727392	-0.030598736
156	11.46	148.98546	11.22	-0.020942408	-0.044965
56	5.84	53.48	5.43	-0.070205479	-0.045
127	8.71	119.179	7.86	-0.061582499	-0.096904444

Table 3 – Road trip test results and error

4. DATA PREPARATION

For a better understanding of the road profile, a route must be first divided into sections, on for each corner. Typically, a trip that has taken place on a curvy road with many hairpins would require slowing down and speeding up repetitively, which would increase rolling resistance and the probability of using the brakes, thus converting kinetic energy into heat and increasing fuel consumption. As such, a road must be first divided into corners and the corners must be classified considering their severity. In rallying, a pacenote system has already been decided on, assigning corners numbers from 1 to 6 (1 is the smallest radius and 6 is the largest radius) and additionally add the categories of square, hairpin and acute (in order of decreasing radius). The categories were compressed to just 3, to increase classification accuracy. With all rallying categories, the models would end up underfit. After classifying each corner, the road can be then be defined by a sequence of corners. The consumption can be calculated for each corner and the results summed to obtain the final result.

Firstly the data has to be sanitized, to eliminate bad reads. Rows that have missing data, in regards to engine parameters, large spikes in fuel consumption (over 2 g/s) or no increase in distance (the car is standing still), missing GPS coordinates or missing accelerometer values are eliminated.

The second task is to get the latitude and longitude coordinates of each measurement taken from the car and then modify the coordinates to be precisely on the median axis of the road, to eliminate the error of the sensor. This can be achieved using the OpenStreetMap Overpass API [10] and sending a query with the current coordinates and a radius to get the current way. A way is a subpart of a road and it is composed of multiple segments. A segment connects two nodes that have coordinates like latitude, longitude and optionally, elevation. Upon request, the API sends the ways that are in the specified radius (in the program the radius was specified at 10m and if the API does not return a way, then it retries with a radius greater by 2m until it finds one) and the nodes that define it. Each set of coordinates read from the GPS sensor is then moved to the centreline of the road by either moving it to overlap with the position of a node, or the orthogonal projection to one of the segments defined by two nodes in the way. When going through all sets of coordinates (sequentially), the program tries to overlap the coordinates to the same way and only when the current latitude and longitude are outside the way (the projection

on the line is outside the segment) the program makes a new request to the API, because it assumes that the current set is not in that way anymore. To find the projection the program goes through all nodes that make up the way sequentially, as the API gives the nodes in the order they appear. The program takes two adjacent nodes, constructs a segment and attempts to find the orthogonal projection. If the projection is outside the segment or the distance is greater than the number of lanes multiplied by a lane width (369e-7) added to a leeway of 28e-6, then it proceeds and tries to find another segment. If no projection is found then the algorithm tries again with a leeway of 738e-6. For all segments tested, it constructs an array containing the projections and the distance to it.

$$AB: \frac{y - y_A}{y_B - y_A} = \frac{x - x_A}{x_B - x_A}$$

$$AB: x(y_B - y_A) - y(x_B - x_A) - x_A y_B + x_B y_A$$

$A(x_A, y_A)$ – node in the map

$B(x_B, y_B)$ – node in the map

$$d(AB, C) = \frac{| ax_C + by_C + c |}{\sqrt{a^2 + b^2}}$$

$$x_H = \frac{b(bx_C - ay_C) - ac}{a^2 + b^2}$$

$$y_H = \frac{a(ay_C - bx_C) - bc}{a^2 + b^2}$$

$C(x_C, y_C)$ – the point that was read from the GPS sensor

$H(x_H, y_H)$ – the projection of C on AB

A problem for this algorithm is near intersections, because multiple ways are returned by the API and sometimes a point (defined by latitude and longitude) can be projected on a wrong way. Because the GPS sensor has a small error, a different road that the path is intersecting with might be closer to the point read by the GPS. To address this, the array of projections is sorted by distance and then the algorithm also looks at a point ahead (3 readings generally) and a point

behind. For the three points it tries to compute a similarity grade by comparing the properties of the road. The road usually has properties like *name*, indicating the street name, *ref* giving the indicative assigned to the road by authorities, *surface* representing the material of the pavement and *highway* indicating if it is a motorway, a residential road, a primary highway or secondary highway. If the *name* or *ref* are equal, then the similarity grade is 1. If the *highway* type is the same, but the *name* or *ref* differ, then the similarity grade is 0.6. Lastly, if the surface is the same, but no other properties align the similarity grade is given as 0.3. For the three points it chooses, the algorithm multiplies the similarity grades and if the result is greater than the threshold, set experimentally at 0.3 then that *way* has precedence. The ways are sorted by the computed similarity grade, descending and also by distance as a second criteria. The projection that is on the first position is then picked.

$$similarity(w1, w2, w3) = similarity(w1, w2) \cdot similarity(w2, w3)$$

$$similarity(w1, w2) = \begin{cases} 1, & if\ w1 = w2 \\ 0.6, if\ w1.type = w2.type \\ 0.3, if\ w1.surface = w2.surface \\ 0, otherwise \end{cases}$$

After each entry of the CSV file has been moved to the median of the road, the program can then start separating the corners from the straight sections. To do that, the program goes through each set of coordinates, using three adjacent measurements and calculates the determinant of the matrix:

$$\begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix}$$

$A(x_A, y_A)$ – first point, in order of reading
$B(x_B, y_B)$ – second point, in order of reading
$C(x_C, y_C)$ – third point, in order of reading

If the determinant is negative, it is a left curve, if it is positive, it is a right curve and if it is 0, then the points are collinear. By keeping track of the sign, the program can detect when a curve ends and when a curve begins (when the sign changes). Because the readings can be slightly offset, the program will also calculate the radius of the curve and if it exceeds a threshold of 0.005 or the area determined by the determinant is less than 2e-10, the points are marked as being in

a straight section. To find the radius of the curve, it is first assumed that a circle can be overlapped to the three points, and the three points are on the circumference of the same circle, which means an equal distance from the centre of the circle:

$O(x_O, y_O)$ – the centre of the circle

$$d(A,O) = d(B,O) = d(C,O)$$

$$\sqrt{(x_A - x_O)^2 + (y_A - y_O)^2} = \sqrt{(x_B - x_O)^2 + (y_B - y_O)^2} = \sqrt{(x_C - x_O)^2 + (y_C - y_O)^2}$$

$$\begin{cases} 2x_O(x_A - x_B) + 2y_O(y_A - y_B) = x_A^2 + y_A^2 - x_B^2 - y_B^2 \\ 2x_O(x_A - x_C) + 2y_O(y_A - y_C) = x_A^2 + y_A^2 - x_C^2 - y_C^2 \end{cases}$$

$$y_O = \frac{(x_A - x_C)(x_A^2 + y_A^2 - x_B^2 - y_B^2) - (x_A - x_B)(x_A^2 + y_A^2 - x_C^2 - y_C^2)}{2\big((y_A - y_B)(x_A - x_C) - (y_A - y_C)(x_A - x_B)\big)}$$

$$x_O = \frac{x_A^2 + y_A^2 - x_B^2 - y_B^2 - 2y_O(y_A - y_B)}{2(x_A - x_B)}$$

After the coordinates of the centre have been determined, the radius is calculated using the euclidian distance formula.

The program then constructs objects in sequence for all corners. The road is represented as an array of corners. A corner can be left, right or straight and the class also encodes properties like the sets of coordinates (read from the CSV and projected on the median axis of the road), slope (rise / run, tangent of the angle at which the vehicle is ascending or descending), speeds of the vehicle, radiuses, fuel used through the corner, distance travelled, lateral accelerations, acceleration and deceleration. For all properties represented as arrays, properties like median, average, minimum and maximum are calculated to also obtain scalar representations for said array.

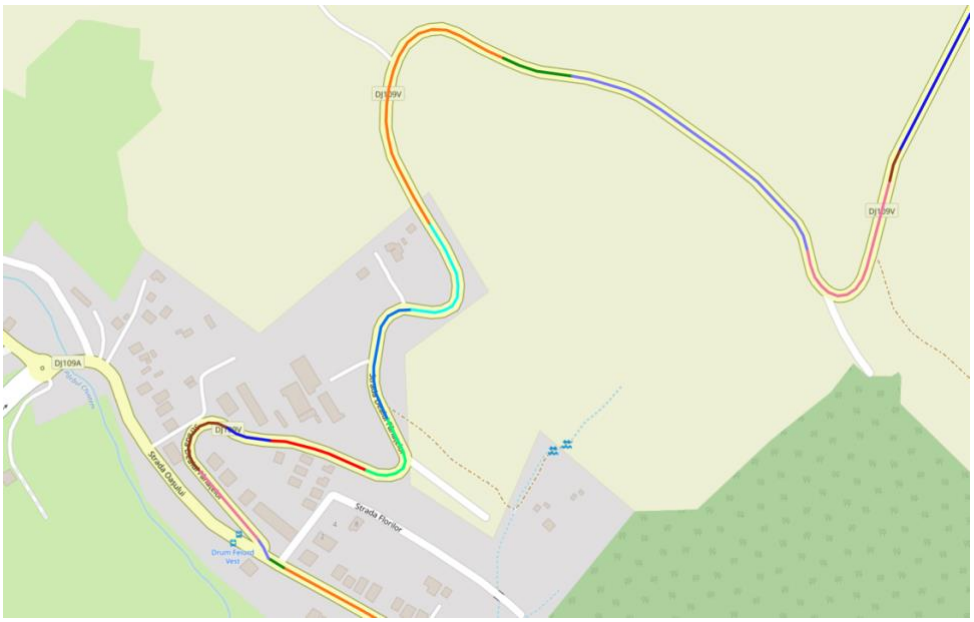

Figure 8 – Map section resulted from the corner split algorithm

Each resulting corner (Figure 8) was exported to a json file and the corners were manually labeled with pacenotes 0, 1 and 2, 0 meaning the tightest of corners and 2 including the corners with the largest radiuses and straight sections.

5. SOFTWARE IMPLEMENTATION

For the purposes of this paper multiple Python scripts and dependencies were implemented, each with its specific purpose. The first step in the project, was collecting engine data to analyse it and develop the data measurement methodology.

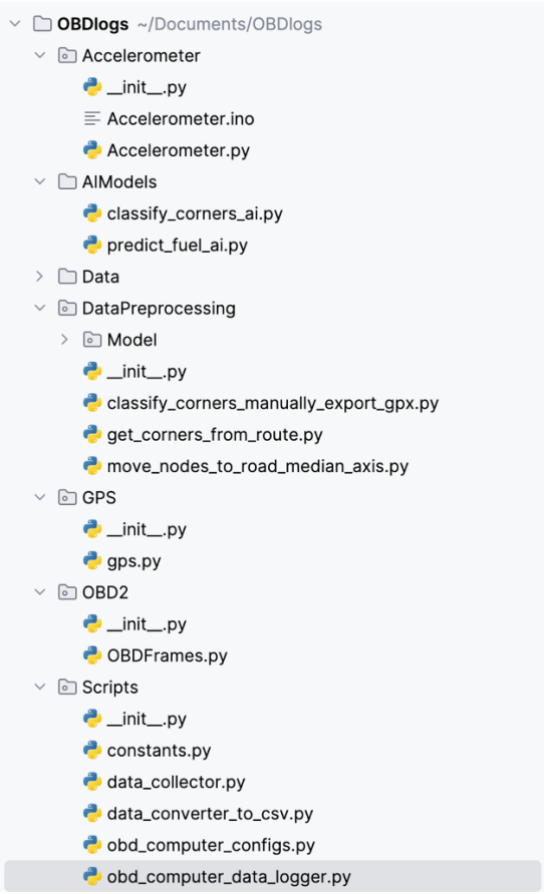

Figure 9 - Project file structure

All the engine relevant data was collected in the path Data/EngineCollected using the Scripts/data_collector.py script and was written in .log files. Little parsing was done to the data to maximise performance. To define the OBD2 queries, the class OBDLiveDataPIDs was defined in OBD2/OBDFrames.py and its properties hold the byte values for each query. The first byte holds the mode defined in SAE J1979 [5], 01 for interrogation and the second byte holds the PID hex value.

```
class OBDLiveDataPIDs:
    RPM = bytes('010C\n\r', 'ascii')
    MAP = bytes('010B\n\r', 'ascii')
```

```
SPEED = bytes('010D\n\r', 'ascii')
IAT = bytes('010F\n\r', 'ascii')
THROTTLE_POSITION = bytes('0111\n\r', 'ascii')
CALCULATED_ENGINE_LOAD = bytes('0104\n\r', 'ascii')
COOLANT_TEMPERATURE = bytes('0105\n\r', 'ascii')
SUPPORTED_PIDS_A = bytes('0100\n\r', 'ascii')
SUPPORTED_PIDS_B = bytes('0120\n\r', 'ascii')
SUPPORTED_PIDS_C = bytes('0140\n\r', 'ascii')
OXYGEN_SENSOR_1 = bytes('0114\n\r', 'ascii')
OXYGEN_SENSOR_2 = bytes('0115\n\r', 'ascii')
TIMING_ADVANCE = bytes('010E\n\r', 'ascii')
FUEL_STATUS = bytes('0103\n\r', 'ascii')
```

A Bluetooth console was used and the Carly Universal Scanner device was analyzed. It was noticed that under characteristic 24 of Bluetooth Low Energy, queries can be dirrectly sent to the ECU and a response is given. The Bluetooth Low Energy characteristic handle was taken, as well as the device address and using the Python bleak [11] library a connection was initialised. Parameters were read and decoded from the OBD2 queries and written to a .log file in the following format: {pid}{a}{b}{time_ns}, as 25 character strings, containing the PID as a hexadecimal value, A and B bytes defined in SAE J1979 [5] as a hexadecimal value and the time in nanoseconds.

A second script, Scripts/data_converter_to_csv.py reads the data from the log file and then creates object of the class OBDFrame defined in OBD2/OBDFrames.py. The class allows for easy parsing, converting the frames from the two given bytes into human readable data, with measurement units. Using the class methods the program writes the data to a CSV file, constructing columns for each read PID in the .log file. The files resulted from such measurements can also be found in the Data/EngineCollected directory. The resulting data was used to analyse and understand the engine management unit's algorithm for taking decisions regarding fuel injection. The pseudocode for the parse algorithm is as follows:

for line ∈ lines:

OBDFrame ← ParseLine(line)

for column ∈ OBDPIDsAndUnits:

if column.PID = OBDFrame.PID:

outputLine ← column.ParsedValue

columnDone ← True

minTime ← ∞

for column ∈ outputLine:

if column = NIL:

columnDone ← False

if column.time < minTime:

minTime ← column.time

if columnDone:

WriteRow(outputLine)

outputLine ← [NIL, …, NIL]

From the above sequence it can be calculated that the algorithm has a $\Theta(n \cdot m)$ complexity, where n is the number of lines read and m is the number of PIDs or columns the output CSV file should have.

To read GPS data, the GPS/gps.py script was defined. It uses the pynmea2 [8] Python library to decode data read from USB. The U-blox module was connected via USB to the computer and it continuously transitted NMEA phrases at a baudrate of 9600. To get the valuable information and store it in variables, the script defines the following variables:

```
LATITUDE = None
LONGITUDE = None
SPEED_KMH_GPS = None
HEADING = None
ALTITUDE = None
ALTITUDE_UNITS = None
```

```
TIME_GPS = None
DATE_GPS = None
```

Latitude and longitude are stored as floating point values, rather than degrees, minutes and seconds. The serial connection timeout is set to 1.05s and it constatly reads the phrases sent in an infinite loop. It uses the pynmea2 [8] library and decodes the information to update the variables.

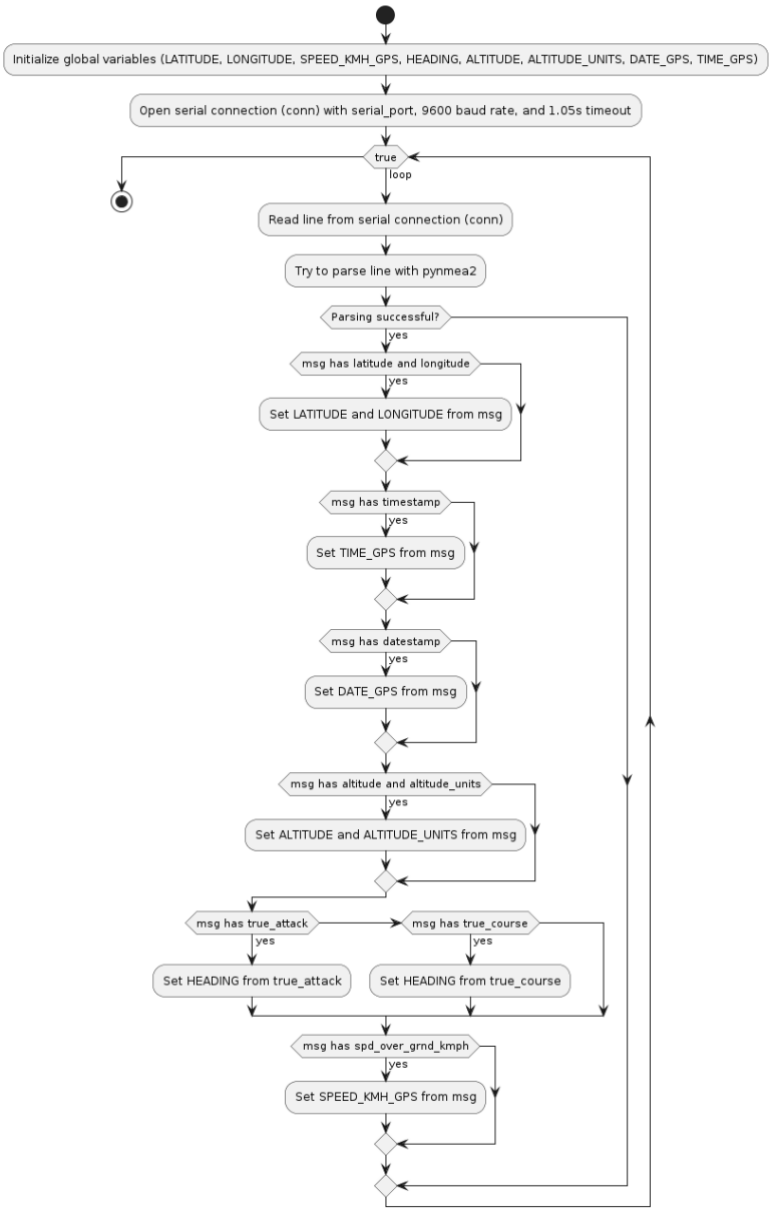

Figure 10 - Flow chart diagram for GPS reads

To read the acceleration values, the ADXL313 accelerometer was connected to an Arduino UNO compatible board and an LED was connected to indicate wether the accelerometer is in

interrupt mode or not. Code from Accelerometer/Accelerometer.ino was compiled and loaded to the board and the class AccelerometerADXL313 defined in Accelerometer/Accelerometer.py was used to handle communications. To maximise versatility and minimise the number of times the ode is compiled and loaded to the board, the microcontroller was programmed to respond to queries and commands sent via USB. The computer could sent one of the following messages encoded by ASCII via serial at a baudrate of 115200:

- **BEGIN**

  This message is received by the microcontroller to initialise the serial connection. On success, the board sends back the message **SER_SUCCESS**, or on failure **SER_FAIL**. Until the begin message is received by the board, the LED mapped to pin 13 on the Arduino UNO should blink. After the **SER_SUCCESS** message is sent back by the board, the accelerometer is initialised and the board sends the message **ACC_SUCCESS** or **ACC_FAIL** on failure.
- **END**

  This message is sent by the computer to close the connection with the Arduino and reset the board.
- **ENABLE_INTERRUPTS**

  The message sent by the computer tells the microcontroller to switch to interrupts mode. In this mode, the accelerometer sends an interrupt on pin 2 to indicate when activity or inactivity is detected. If the switch is successful, the board sends back the message **ENABLED_INTERRUPTS**.
- **DISABLE_INTERRUPTS**

  This message indicates to the board that it should disable the interrupts mode and resume normal operations. The response to this message should be **DISABLED_INTERRUPTS**.

- **SET_RANGE**

  The **SET_RANGE** message is followed by a numeric paramter, defined in the ADXL313 Arduino library [12] and it represents the measurement range that the accelerometer should use. Under normal operation the board sends back the parameter value. If the operation is not permitted, the board sends back **ILL_ACT** (illegal action). The operation is not permitted under interrupt mode.

- **SET_ACT_THR**

This message is also followed by a numeric paramter. This is the threshold that the accelerometer should use in interrupt mode to define the sensitvity to activity or inactivity. On success, the board sends back the paramter value, or **ILL_ACT** if the interrupt mode is enabled.

- **SET_INACT_TIME**

  This message requires a numeric paramter and it sets the timeout in ms that the accelerometer should wait before inactivity can be signalled through an interrupt on pin 2. Again, this type of command should send back to the computer the parameter value on success or **ILL_ACT** if the interrupt mode is enabled. The parameter cannot be changed during interrupt mode.

- **SET_INT_INTERVAL**

  This message is used to set the interval in ms that the board uses to send frames while in interrupt mode. When activity is detected, the board sends back accelerometer frames continuously until inactivity is detected. The interval set in this command can be always changed, even during interrupt mode. This command also requires the parameter.

- **REQUEST_FRAME**

  The request frame query can be used at any time and as a response, the board sends back an accelerometer frame containing the x, y and z axis accelerations.

Each request and response is ended by a '\n' ASCII character. The accelerometer frames have the **ACC_FRAME** header at all times, both in interrupt mode and while responding to the **REQUEST_FRAME** query. When the board does not recognize a command or query, the parameter has an error on decode or there is an error in receives, the **UNKNOWN_ERR** message is sent back to the computer. The AccelerometerADXL313 class in Python was implemented to provide methods to its users and it creates an abstractisation layer when using the module. Its users should not have to use the defined queries and commands and by simple calls to the class methods all functionalities of the accelerometer can be used. To handle the interrupt mode, the AccelerometerADXL313 class creates a new thread that loops and continuously reads data from the serial connection. When an accelerometer frame is found, the function given as a parameter to the interrupt mode enable method is called. The programmer can then use the methods

get_x(), get_y() and get_z() to retreive the coordinates that were sent by the board. The coordinate retreival methods raise an error when the interrupt mode is disabled and in that case, if the programmer wants to get the acceleration values, he/she should use the get_frame() method, which returns the three coordinates. The source code compiled for the microcontroller has no repetitive loops and only decision statements. The only repetitive element is that by design, the board loads repetitively the loop() function (Figure 11 and Figure 12). This means that the time complexity for one cycle of the board is $\Theta(1)$.

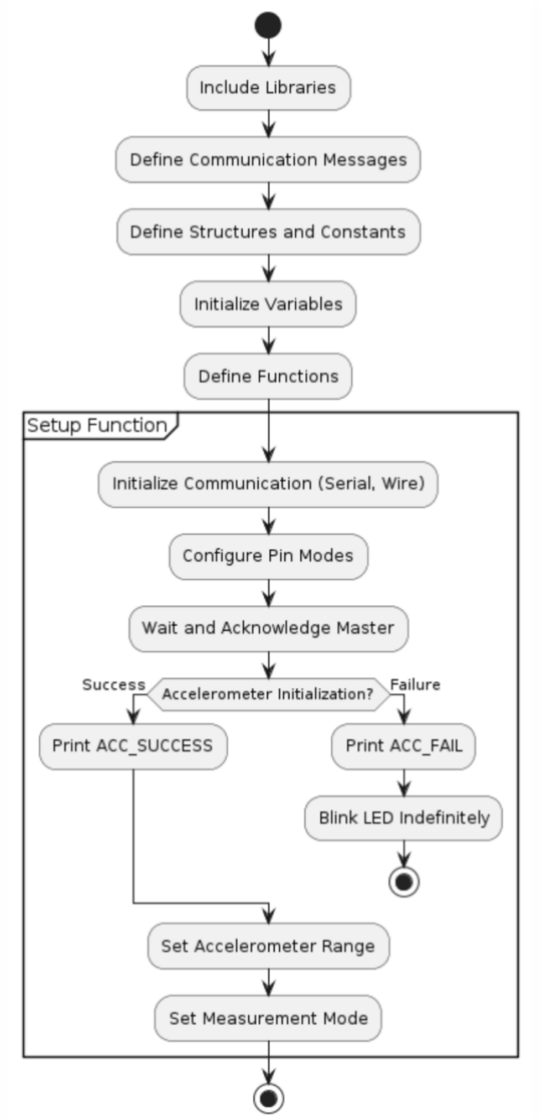

Figure 11 - Microcontroller initialization sequence flow diagram

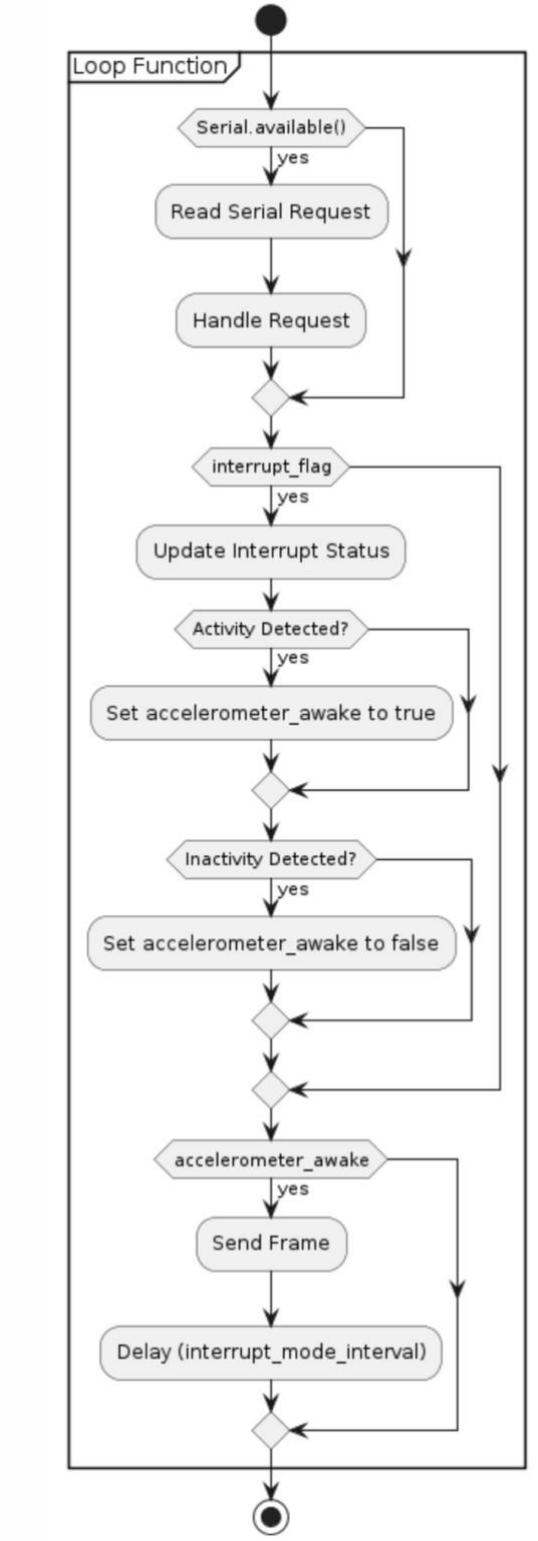

Figure 12 - One loop cycle flow on the microcontroller

To collect all the data that should be used in the artificial intelligence models, the script Scripts/obd_computer_data_logger.py should be used. In the Scripts/obd_computer_configs.py, values can be set for the output files, device addresses, serial connection ports, logging interval, a correction coefficient, OpenWeather API [9] key and the option to also print information to

console. In this file, the update cycle which defines in which order parameters should be cycled through is also defined. This is a necessity since only one parameter can be read through OBD2 at a time. After this script is run, the output CSV file should contain on each row GPS, acceleration, engine and computed infromation. The Scripts/obd_computer_data_logger.py implements the formulas defined in the data measurement methodology chapter and the constants defined in that chapter are added to the Scripts/constants.py file. Each of the threads loops continuously to read data from the input sources and update the variables or write the variables' values in the output CSV file.

The print_cycle thread loops continuously, computes the fuel consumption using the formula and prints all values in the CSV file. In one cycle there is no other repetitive structure, giving a time complexity $\Theta(1)$ for each cycle in the infinite loop.

$$FUEL\_CONSUMPTION = \frac{FUEL\_USED}{DISTANCE} \cdot 100$$

The engine_data_read_cycle thread loops continuously, reads and updates the parameters read from the OBD scan tool, computes the fuel used, instant fuel consumption and distance travelled. On each cycle of the infinite loop, the program goes through each parameter of the engine update cycle. After each read, the computed parameters (i.e. the fuel used, instant fuel consumption and distance travelled) are updated. Let the number of parameters defined in the update cycle (they can repeat) be noted with k. In this case, the time complexity of one update cycle is $\Theta(k)$ (Figure 13).

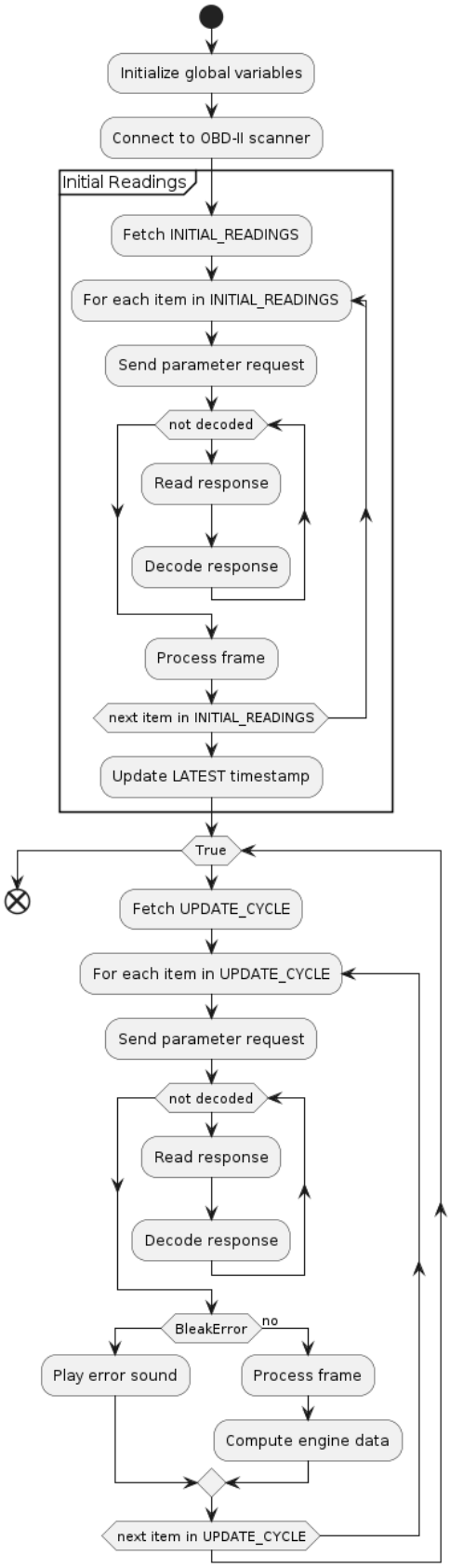

Figure 13 - Engine data thread flow chart

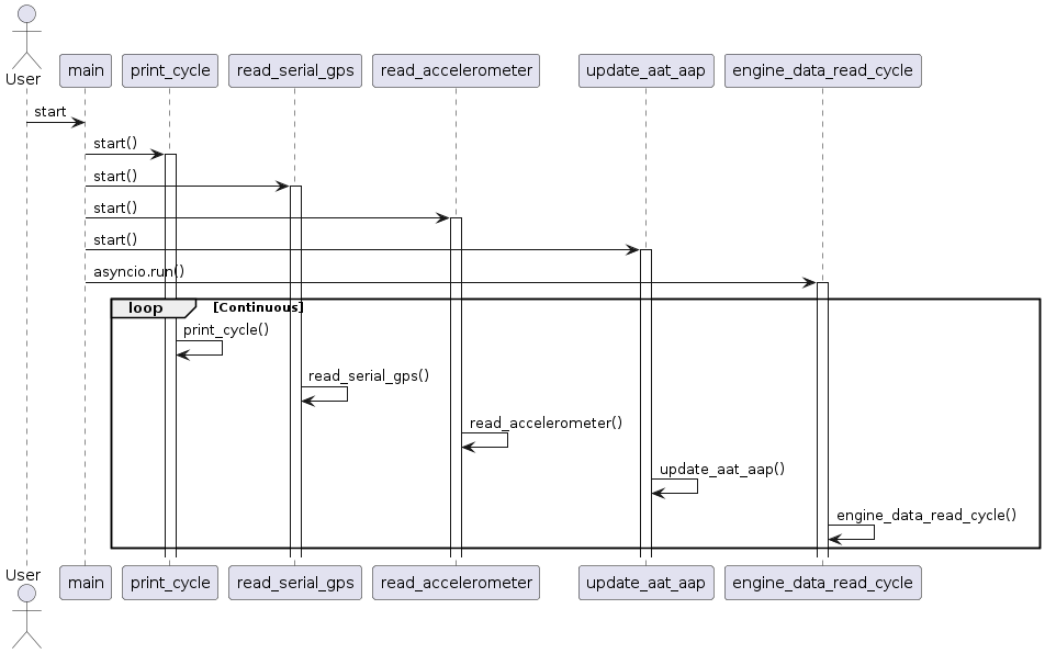

Figure 14 - Sequence diagram for obd_computer_data_logger.py

The DataPreprocessing/move_nodes_to_road_median_axis.py script plays a vital role in the performance of the artificial intelligence model and it should be run on the resulting CSV file from the Scripts/obd_computer_data_logger.py file. In this script, the formulas and procedures that are defined in the data preparation chapter are applied using functions to move all coordinates on the median axis of the road. For this script, the input is a CSV file, and the output file is nearly identical to the input file. The only difference is that rows with missing values are eliminated and the latitude and longitude coordinates are moved to be on the median axis of the road. The process_data() function is called and it parses all data from the CSV file to the AutomotiveDataRow class, defined in DataPreprocessing/Model/AutomotiveDataRow.py. To get the way data, the script uses the Requests package [13] for Python and using http requests to the Overpass API [10] it gets information about the ways in XML format. To decode the information, the built-in packages for XML parsing and using the class ElementTree the information is retreived. Data sanitation and way retrieval takes a complexity $O(n \cdot r)$, where r is the number of retries to get the necessary ways. Finding the projections takes $\Theta(n \cdot m)$, where m is the number of segments in the way, and finding the best projection takes $\Theta(n \cdot k)$, where k is the number of projections found. The overall complexity of this script is $O\big(n \cdot (m + r)\big)$, as $\forall k \ \leq m$.

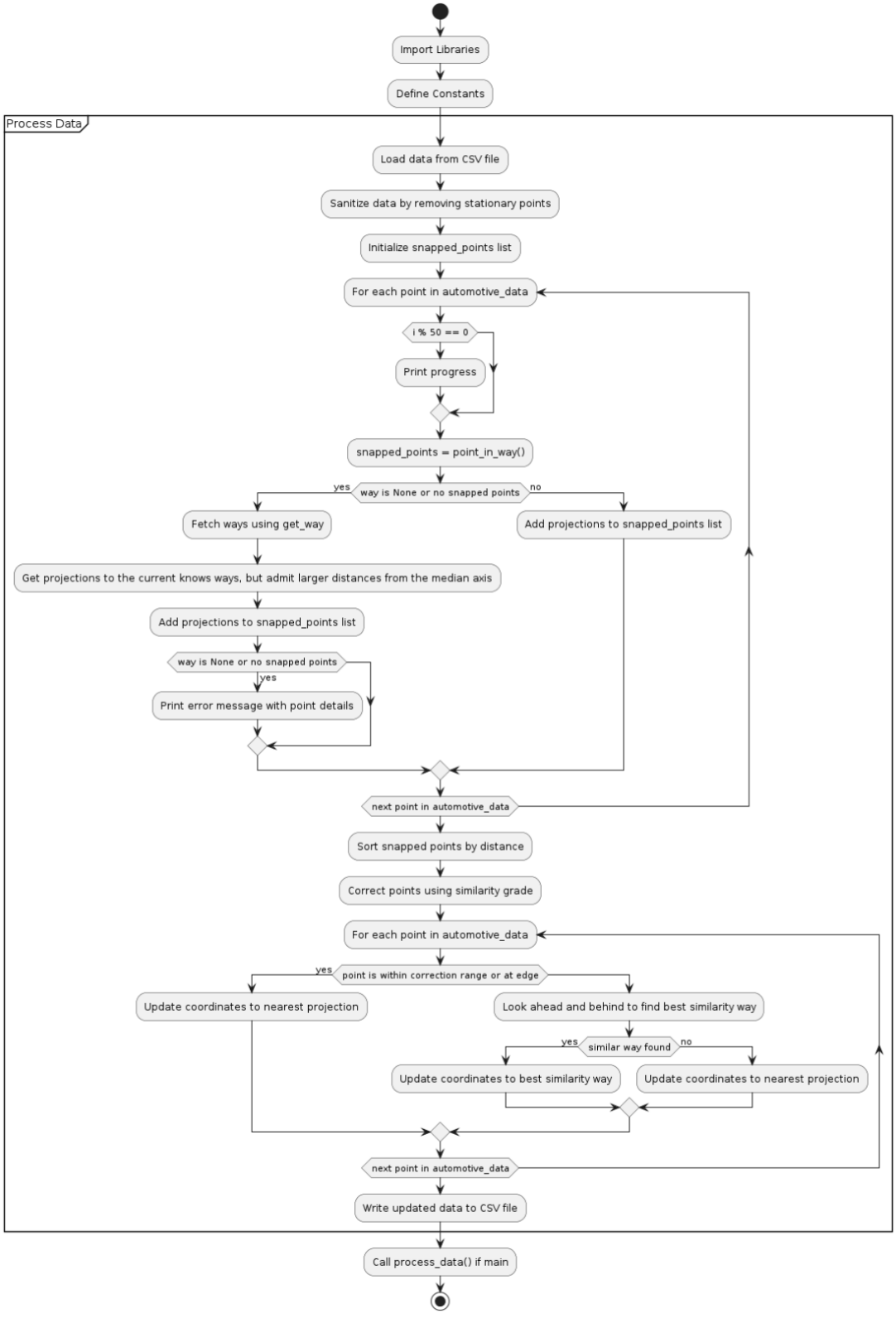

Figure 15 - Move points to median axis script flow chart

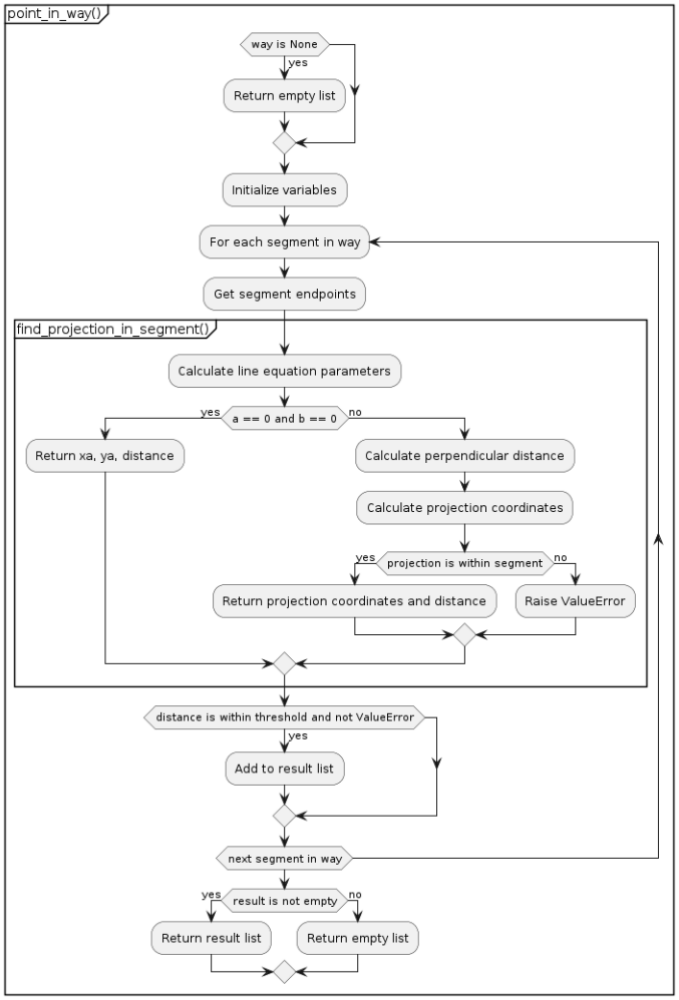

Figure 16 - Auxiliary functions flow chart for Figure 15

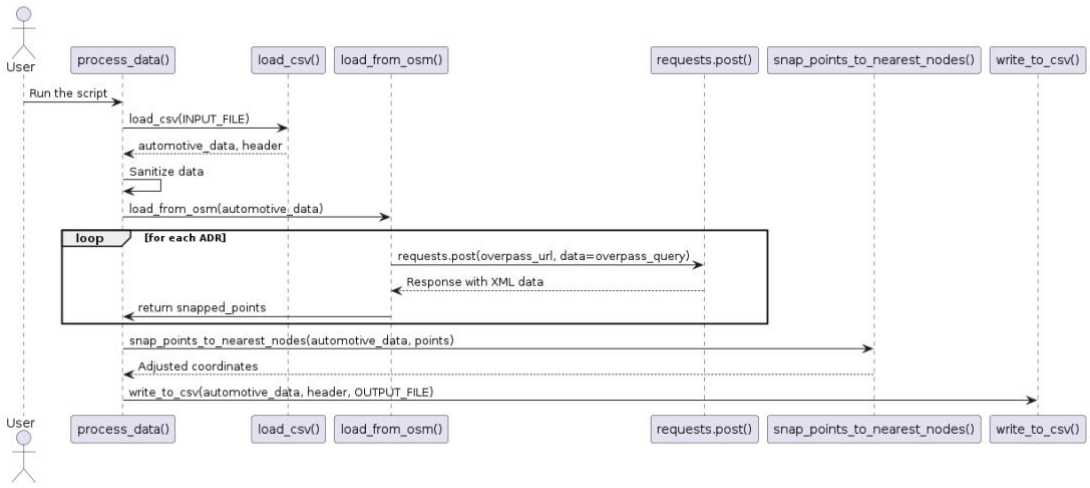

Figure 17 - Sequence diagram for move_nodes_to_road_median_axis.py

The next script that needs to be run is DataPreprocessing/get_corners_from_route.py. This script applies the steps shown at the data preparation chapter and splits the rows in sets to

build instances of the Corner class defined at DataPreprocessing/Model/Corner.py. The Corner class has a composition relationship with the Node class at DataPreprocessing/Model/Node.py and hold the array of nodes. The objects are later encoded to and decoded from arrays, using their respective methods. This is later used because the arrays are json encoded and written to files in Data/CornersSplitAndLabels. In this script's worst case, the program goes through all nodes and constructs a single corner. The Corner class constructor goes through all nodes to compute the properties of the corner. In this script, the algorithm's time complexity is $O(n^2)$.

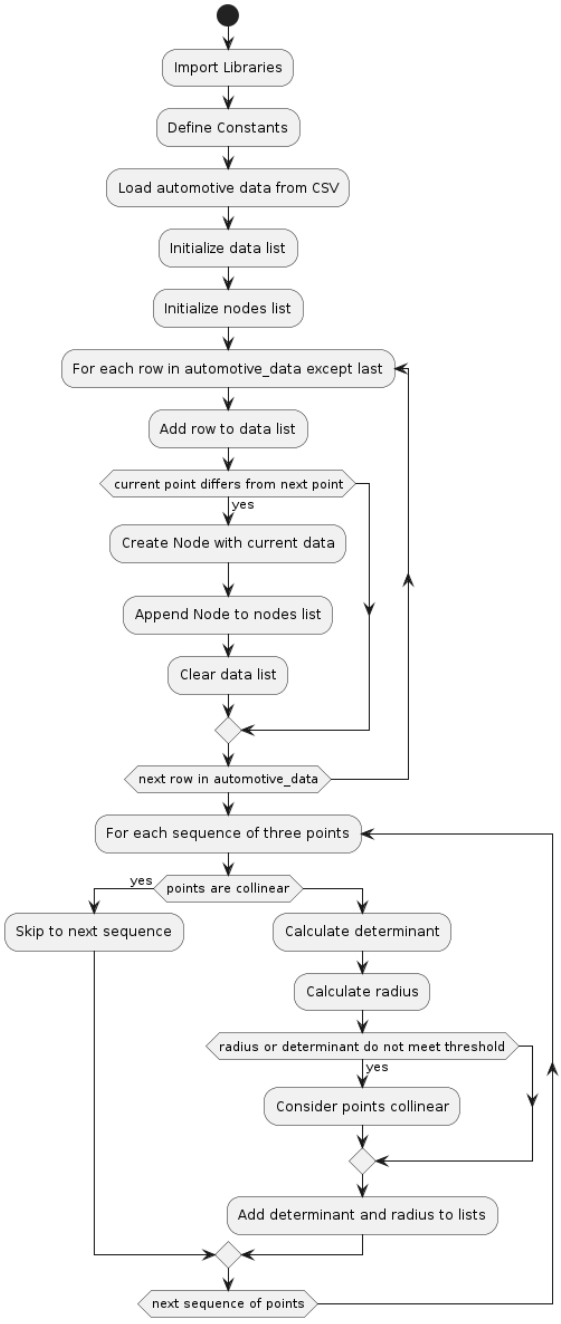

Figure 18 - Part 1 of the corner determination script flow chart

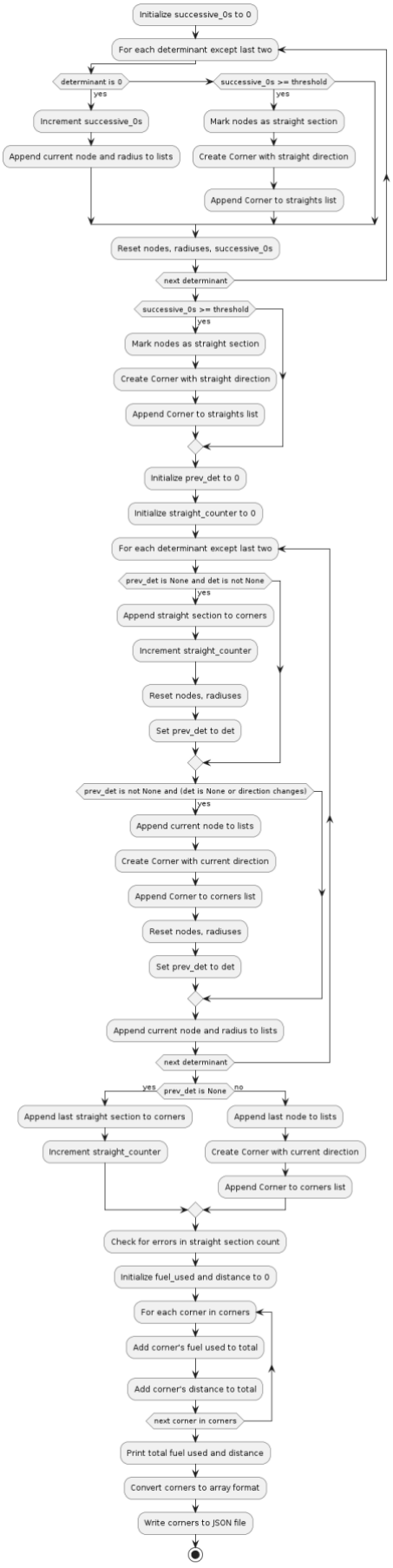

Figure 19 - Part 2 of the corner determination script flow chart

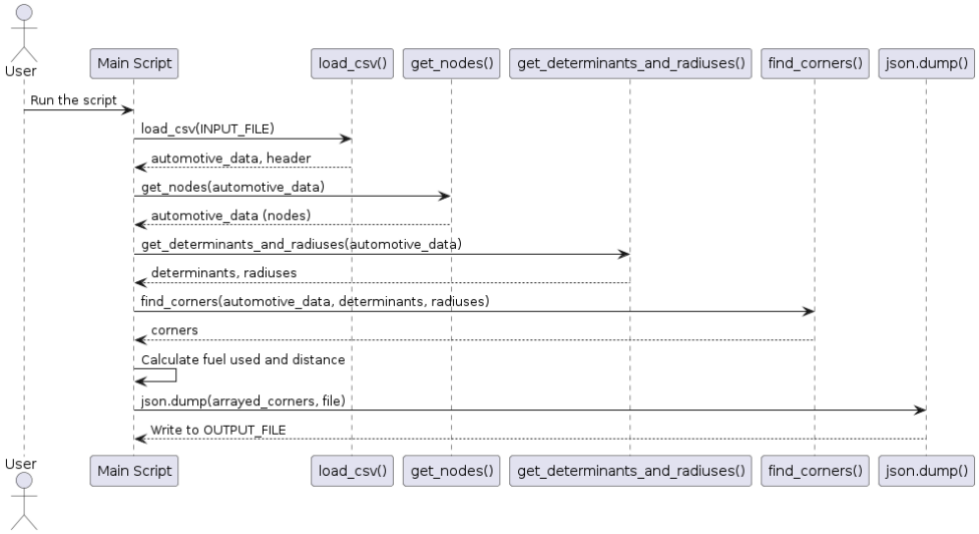

Figure 20 - Sequence diagram for get_corners_from_route.py

The DataPreprocessing/classify_corners_manually_export_gpx.py script reads the Corner objects from the given JSON file, constructs the objects and builds a GPX file with colour coded sections that can be visualised. The program also asks for input from the user to assign categories to corners and writes the labels in a separate JSON file. All of these files can be found under Data/CornersSplitAndLabels. To create the GPX files, the gpxpy library [14] is used.

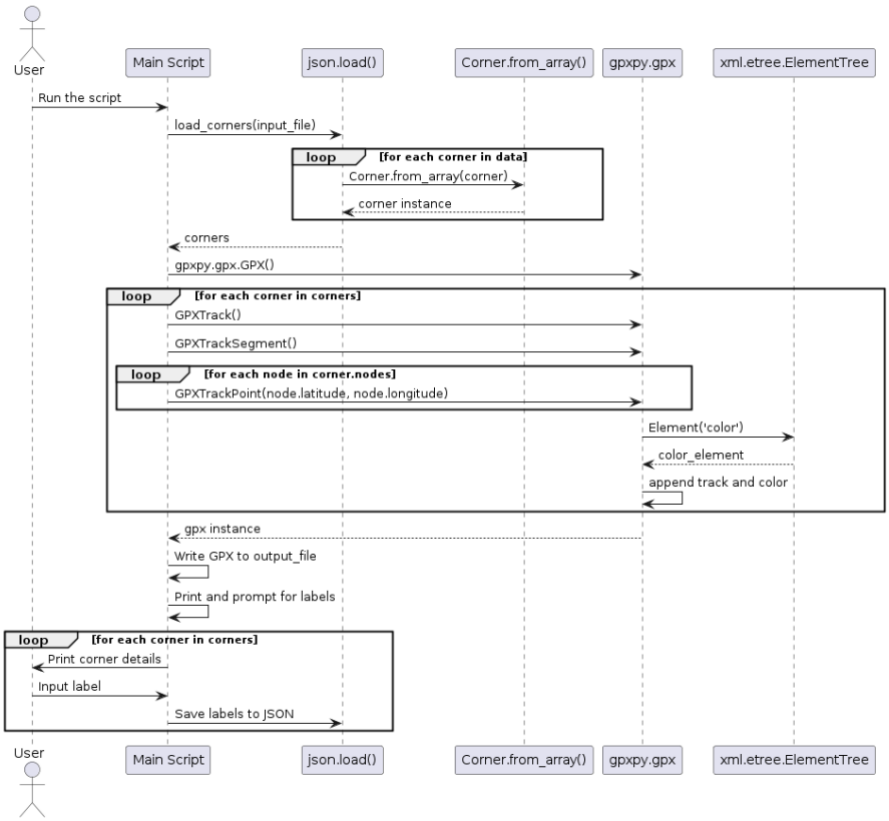

Figure 21 - Sequence diagram for classify_corners_manually_export_gpx.py

Lastly, the AIModels/classify_corners_ai.py and AIModels/predict_fuel_ai.py scripts are used to apply the artificial neural network and random forrest models to both described problems. For the predict_fuel_ai.py script, the performance metrics are stored in the Data/ModelPerformance directory as CSV files. These two scripts were implemented with the help of scikit-learn [15] , Keras [16] Python packages. To make the measurement process automatic, features were encoded in arrays and each configuration was defined as a set of boolean values:

```
names = ['Min Radius', 'Average Radius', 'Median Radius', 'Max
Radius', 'Distance traveled', 'Median Speed',
         'Min Speed', 'Max Speed', 'Average Speed', 'Min
Deceleration', 'Max Acceleration', 'Max Centrifugal',
         'Median Centrifugal', 'Average Centrifugal', 'Max
Rise/Run', 'Min Rise/Run', 'Average Rise/Run',
         'Median Rise/Run', 'Category', 'Entry Speed', 'Exit
Speed', 'Decelerated Speed', 'Accelerated Speed',
         'Min MAP', 'Max MAP', 'Average MAP', 'Median MAP', 'Min
Engine Load', 'Max Engine Load',
         'Average Engine Load', 'Median Engine Load',
         'Entry MAP', 'Exit MAP', 'Entry Engine Load', 'Exit
Engine Load']

features = [False] * 35
features[0] = True
features[4] = True
features[9] = True
features[17] = True
features[27] = True
features[28] = True
features[30] = True
test_features.append(features)
```

Multiple configurations were defined in a similar manner and tests were run 5 times, after which the results were centralised as means and written in a CSV. Using matplotlib [17] the results were plotted for each run to visualise performance.

The set of scripts uses a procedural approach for algorithms, but also uses objects for more complex tasks or data handling. With the exception of classes Corner and Node, the classes defined in the project are independent. Class Corner and class AutomotiveDataRow have instances of class Node through composition.

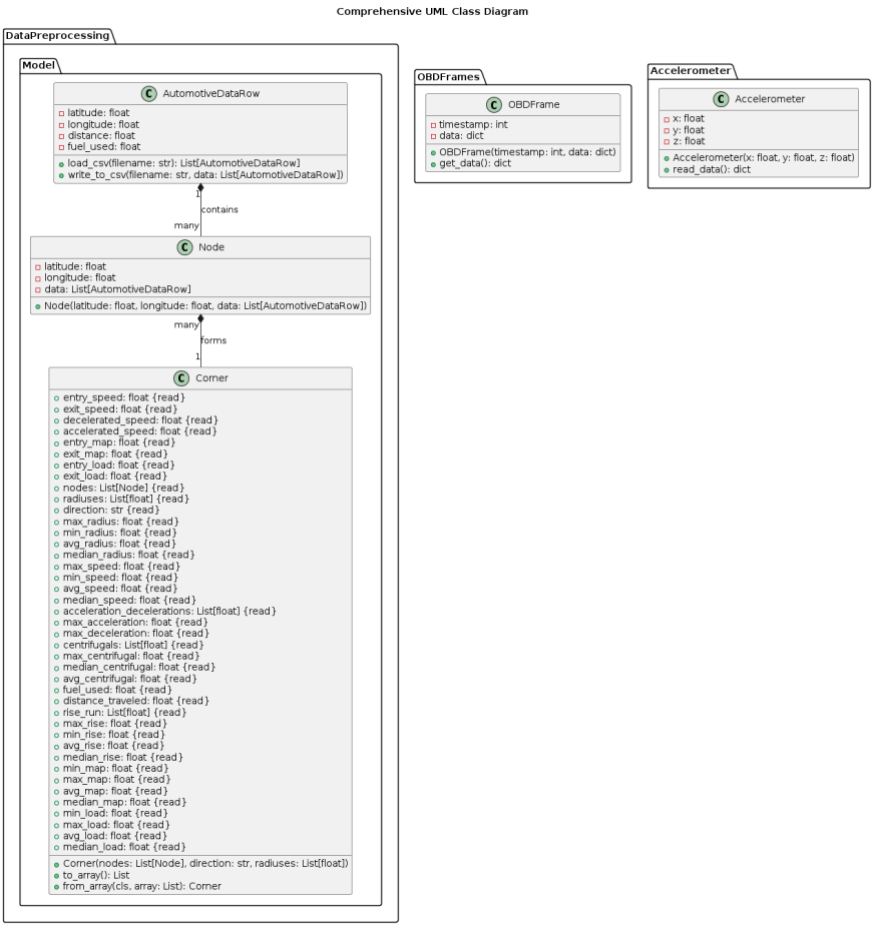

Figure 22 - Class diagram for the whole program

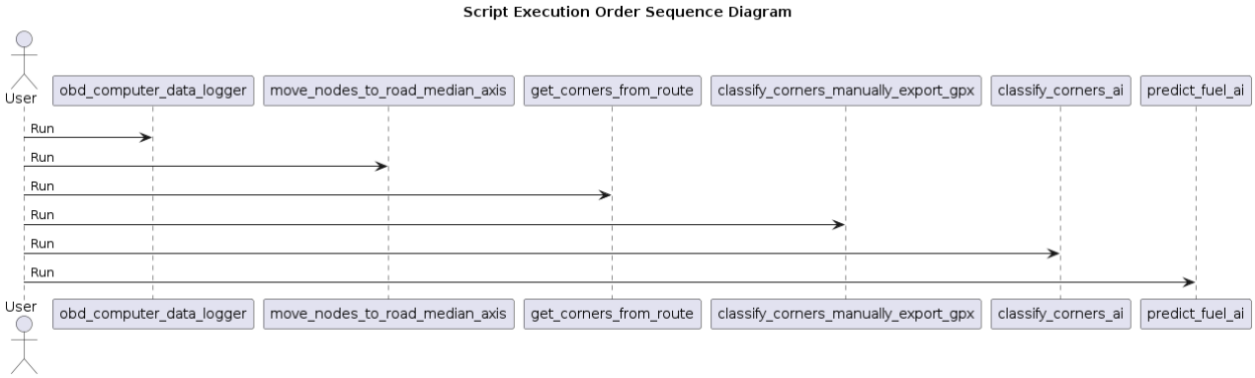

Figure 23 - Script execution order, pipeline like execution

6. CORNER CLASSIFICATION USING ARTIFICIAL NEURAL NETWORKS AND RANDOM FORESTS

The corners detected at the previous step, can be classified into multiple categories and the data can be later used in combination with engine and accelerometer data to provide insights on how a driver may be approaching different types of corners. Such information is valuable not only for predicting fuel consumption, but also road safety. As Rune Elvik has shown in his paper [18], smaller radius corners have a higher accident risk.

For this specific application, two models have been applied and their performance was compared. The corner objects previously constructed were turned into feature arrays and fed to two different models, one being an Artificial Neural Network and the other being a Random Forrest model. Recursive Neural Networks could also be used if the corners are reduced from an array of radiuses to a preset length, however that can be difficult to achieve considering that road curves have varying lengths.

The artificial neural network model was constructed using two dense hidden layers, each having a size of 16 neurons. Both hidden layers were configured to use the ReLu activation function because it is widely used and it usually shows the best performance in recognizing complex patters and it introduces non-linearity. The output layer was constructed with a neuron for each output class and it uses the softmax activation function as it is best suited for outputing raw probabilities in classification problems having multiple classes. Multiple optimizers were tested, but their performance did not affect the output of the ANN model in this specific problem and the metrics always seemed to converge to the same values for the specific data set. The number of epochs was tuned to around 700, point at which the metrics stabilised around a maximum value.

The random forrests were chosen as it has better performance than decision trees. The forest size was chosen at 200 and a random state of 42, commonly chosen values.

For initial testing, two different sets were used. One measurement set was used for training (approximately 45km of curved roads) and a different batch (approximately 7km of curved roads) from a different day was used for testing its performance. The corners were converted to feature arrays to contain only necessary information. To increase the applicability of this model, only data obtainable from GPS was used, like radius metrics(min, max, average,

median) and the corner length (m, circumference). Multiple combinations of parameters ware tested, to minimise similar parameters that would induce overfitting. In all cases, the radiuses that were set as infinity were reduced to 300 to bring the data to closer values. Corner radiuses are expressed as degrees (latitude and longitude euclidian distance) and a maximum distance of 202 degrees can be achieved using opposite points on the globe.

$$d_{max} = \sqrt{90^2 + 180^2} \approx 201.24$$

Was the feature used (TRUE / FALSE)					ANN			RF		
Min Radius	Median Radius	Average Radius	Max Radius	Length	Accuracy	Precision	Recall	Accuracy	Precision	Recall
TRUE	TRUE	FALSE	TRUE	TRUE	0.72	0.72	0.72	0.79	0.82	0.57
TRUE	TRUE	FALSE	TRUE	FALSE	0.72	0.72	0.72	0.79	0.82	0.57
TRUE	TRUE	FALSE	FALSE	FALSE	0.72	0.72	0.72	0.77	0.76	0.56
TRUE	FALSE	TRUE	TRUE	FALSE	0.72	0.72	0.72	0.77	0.43	0.41
TRUE	TRUE	TRUE	TRUE	TRUE	0.72	0.72	0.72	0.77	0.43	0.41

Table 4 - Performance of models depending on features used

Analysis of model performance (Table 4) reveals that the artificial neural network was more stable in regards to the features input in the model, but given the proper set of features the random forrests outperform the artificial neural networks.

7. FUEL CONSUMPTION PREDICTION USING ARTIFICIAL NEURAL NETWORKS AND RANDOM FOREST REGRESSORS

The same data that was previously used for classification as well as its results can be further used to predict future values for fuel consumption. Using the same principles as before, the corner objects must first be converted to feature arrays. The following properties were used to build feature arrays:

- Minimum corner radius
- Median corner radius
- Distance travelled
- Median speed
- Minimum speed
- Maximum speed
- Maximum deceleration
- Median lateral acceleration
- Median rise/run
- Corner label (one hot encoded, corner category from the previous chapter)
- Accelerated speed
- Minimum calculated engine load
- Maximum calculated engine load
- Median calculated engine load

The rise/run property is used to define the incline or decline of the road. Usually, road signs express inclines and declines in percentages, as a fraction of rise/run, where the run is the horizontal distance that the vehicle moves through and the rise is the vertical distance the vehicle moves through. This can also be expressed as the tangent of the angle that the road has relative to the horizontal plane.

The accelerated speed property represents the sum of the negative differences between two consecutive speed readings throughout the corner.

$$\sum_{v_i \in Corner} |max(v_i - v_{i+1}, 0)|$$

To obtain the best results, multiple parameter combinations (Table 5) were tried and their performance was evaluated. For each combination there were 5 tests made and for each metric the average was taken.

ID	Min Radius	Median Radius	Median Speed	Min Speed	Max Speed	Min Deceleration	Median Centrifugal	Median Rise/Run	Category	Accelerated Speed	Min Engine Load	Max Engine Load	Median Engine Load
1	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
2	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
3	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
4	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
5	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
6	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
7	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
8	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE
9	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 5 – Feature combinations used in the models

The artificial intelligence models were implemented in Python 3.9, using the scikit-learn package [15] and Keras [16].

To obtain the estimated fuel consumption data two models were constructed. The first used model is an artificial neural network comprised of 5 densely connected hidden layers, the first three having 32 neurons and the last two 16 neurons. When attempting to add more layers, the model seems to start overfitting. All hidden layers used the ReLu activation function. The output layer was configured to use one neuron with a linear activation function, like in all regression problems. A batch normalization layer was also tried between the first three and last three hidden layers, but it showed poorer performance on average (Table 7). The model was trained with 300 epochs and a batch size of 48.

Secondly, a random forest regressor was used to predict the fuel consumption data. To achieve peak performance from the random forest regressor, the parameters were hypertuned.

The following parameter grid was used for hypertuning:

```
param_grid = {
    'n_estimators': [25, 50, 100, 150, 200, 250, 300],
    'max_depth': [10, 15, 20, None],
    'min_samples_split': [2, 3, 4, 5],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2']
}
```

To hypertune the data for the random forest regressor, the GridSearchCV() class was used from the scikit-learn Python package.

The data was normalized using the StandardScaler() class available in Keras and fed through the model. To assess model performance, the root mean squared error and the mean absolute errors were used. Also, at the end of each prediction, the outputs from the models were summed to obtain the fuel used throughout the whole trip. To further evaluate performance, the absolute and relative errors were computed for the obtained sums.

Combination	Average of RMSE ANN	Average of MAE ANN	Average of RMSE RF	Average of MAE RF	Average of Corner sum absolute error ANN	Average of Corner sum relative error ANN	Average of Corner sum absolute error RF	Average of Corner sum relative error RF
1	5.1413	2.9525	5.7732	3.8668	40.5040	0.103294	23.5100	0.059955
2	5.7814	3.3294	5.0652	3.3917	30.5018	0.077786	18.0059	0.045919
3	4.2810	2.6748	4.3859	3.1849	6.5551	0.016717	6.3270	0.016135
4	4.1161	2.7437	4.6065	3.4236	12.0088	0.030625	4.8345	0.012329
5	2.2528	1.6008	3.7201	2.8434	17.5801	0.044833	15.9036	0.040557
6	3.3754	2.4294	4.3075	3.1798	51.2439	0.130683	1.5412	0.003930
7	2.5621	1.9517	4.3142	3.0078	8.1625	0.020816	14.6044	0.037244
8	2.7874	2.1480	3.5875	2.6124	16.4684	0.041998	9.9305	0.025325
9	7.0481	4.7520	5.8945	4.3049	24.1510	0.061590	46.5305	0.118663

Table 6 – Mean model performance for each feature combination, 5 tests

Combination	Average of RMSE ANN	Average of MAE ANN	Average of Corner sum relative error ANN	Average of Corner sum absolute error ANN
1	5.6652	3.5326	0.094792	37.170453
2	5.5739	3.3482	0.069706	27.333357
3	4.0898	2.8665	0.067117	26.318406
4	4.8605	3.4058	0.107354	42.096026
5	2.9320	2.2108	0.066387	26.032051
6	3.6353	2.5275	0.097748	38.329561
7	2.8915	2.2006	0.064768	25.397257
8	3.3581	2.5629	0.056907	22.314727

Table 7 - Average ANN performance with batch normalisation layer, 5 tests

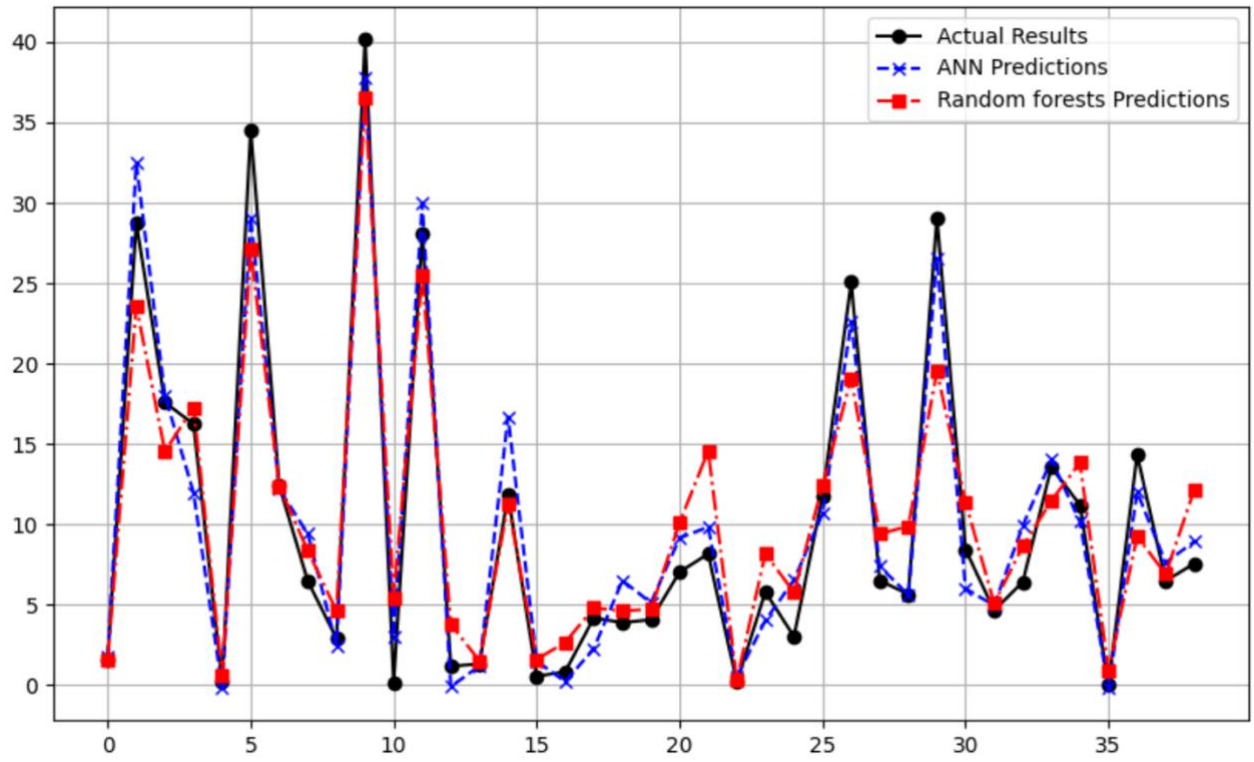

Figure 24 - Plot of model predictions, combination 7, test 2

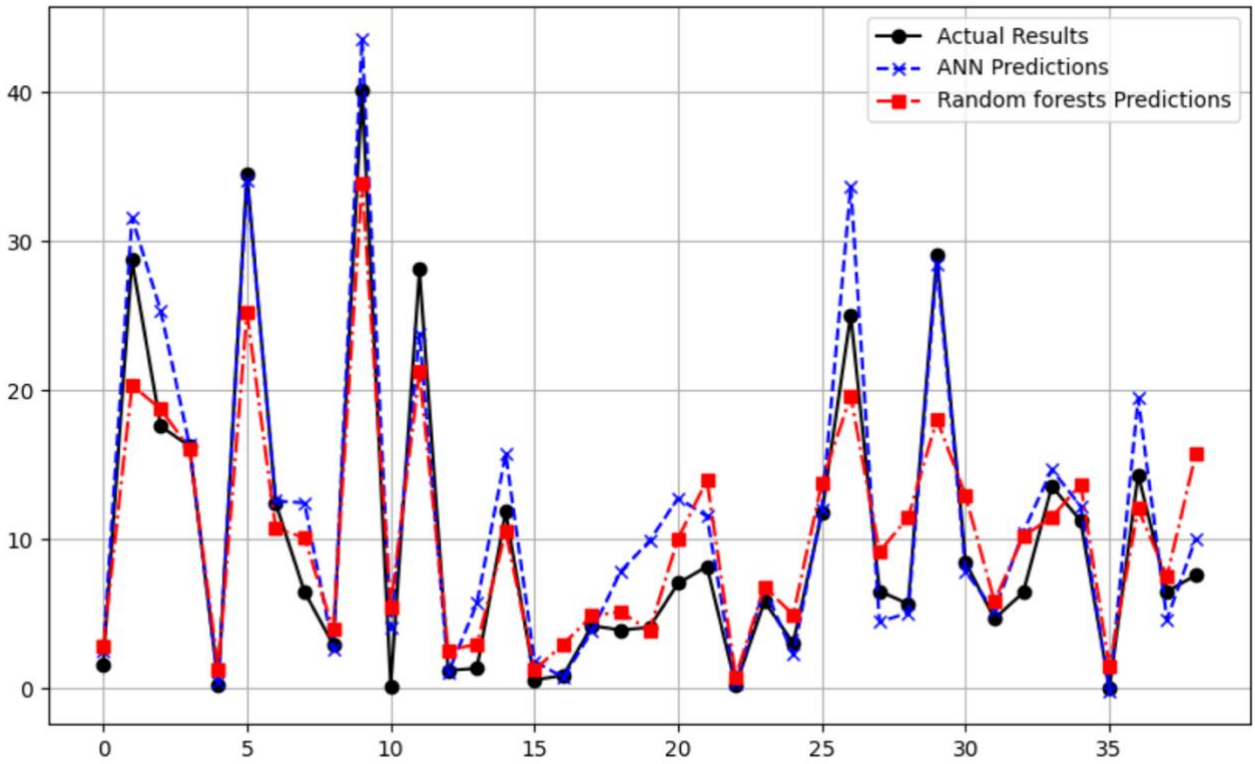

Figure 25 - Plot of model predictions, combination 6, test 3

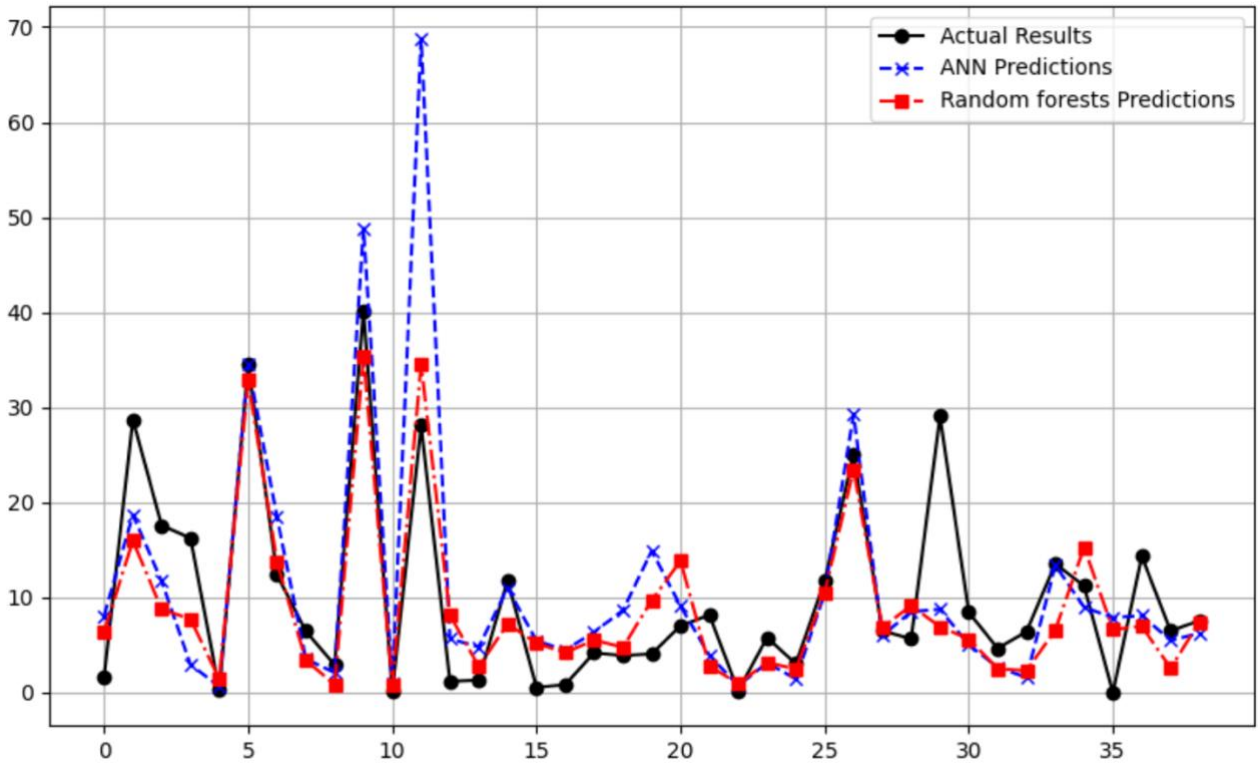

Figure 26 - Plot of model predictions, combination 9, test 1

Using Table 6, and Figures 9, 10 and 11 the results of the tests seem that as expected, the more properties the models are fed, the better are the predictions. Combination 9 is modeled to use just data that can be extracted from map and location services, to see the performance of a prediction without taking actual measurements in the field. The other combinations also reach satisfactory results and predict the used fuel for the trip with high accuracy.

Judging the data from Table 6, the results show that random forest regressors behave better than artificial neural networks when less data is available and the fuel consumption can be better estimated individually per corner. However, when more data can be used, like speed and acceleration, a well tuned artificial neural network can be better at prediting the fuel consumption throughout a corner, but even then, the random forest regressors provide marginally better results for the whole trip.

8. CONCLUSIONS AND FUTURE WORK

In previous studies, Yao et al [3] and Wickramanayake et al [1] proved that random forest regressors are the most effective model at predicting the fuel consumption of a vehicle. After taking a different approach and isolating road characteristics this paper explored the prediction capabilites of artificial neural networks and random forest regressors, using data dervied from vehicle telemetry. The features aimed to isolate road characteristics, like corner radius and incline, but also driver behaviour by analysing lateral accelerations and engine load among others.

Through rigurous testing, it was shown that artificial neural networks and random forest regressors showed different levels of accuracy, depending on the comprehensiveness of the feature sets used. For both models the richer features showed better performance, specifically feature combinations 3 and 5.

Random forests proved to be more robust and they handled better the scenarios where less features were given, also being less prone to overfitting. On the other hand, the artificial neural networks proved that they are better suited for complex problems with more features and the ANNs are better at learning patterns when sufficient data is available. Both models achieved satisfactory results during testing and proved that ANNs can outperform random forests with appropriate feature engineering.

It is recommended that random forest regressors are used when data is limited for their robustness and consistent performance, but in scenarios where extensive data is available properly tuned artificial neural networks should be used.

Future work could focus more on classifying corners and predicting driver aggressivity either as fuzzy classes or floating point coefficients for a certain road type, depending on the number of corners and their severity. Also, speed prediction throughout a corner might end up being more useful to generate new data that can be used with other models to predict values like fuel consumption. Moreover, the tests in this paper were mostly done on interurban roads with light traffic conditions. Having data on wether the road is urban or interurban, coupled to traffic conditions can be used to pick an appropriate model. Developing separate models for certain conditions might prove better performance in practice, rather than developing a highly complex model, prone to overfitting. Furthermore, for better and more precise results, more modern

vehicles that support OBD2 PIDs 0xA2 (Cyclinder fuel rate – the amount of fuel entering the cylinder on one intake stroke) and 0x66 (Mass air flow – g/s) could be used for fuel consumption estimations. Using the PIDs mentioned before could also be used to obtain data faster and with less effort for more types of vehicles.

Overall, this study demonstrates the importance of feature selection and model tuning in predictive models, providing valuable insights for both academic research applications, as well as practical applications in automotive telemetry and energy efficiency optimisation.

Bibliography

[1] S. Wickramanayake and H. D. Bandara, “Fuel consumption prediction of fleet vehicles using machine learning: A comparative study,” *Moratuwa Engineering Research Conference (MERCon),* pp. 90-95, April 2016.

[2] Q. Zhao, Q. Chen and L. Wang, “Real-Time Prediction of Fuel Consumption Based onDigital Map API,” *Applied Sciences,* vol. 9, no. 7, p. 1369, 2019.

[3] Y. Yao, X. Zhao, C. Liu, J. Rong, Y. Zhang, Z. Dong and Y. Su, “Vehicle Fuel Consumption Prediction Method Based on Driving Behavior Data Collected from Smartphones,” *Journal of Advanced Transportation,* vol. 2020, no. 11, pp. 1-11, March 2020.

[4] M. Karaduman and H. Eren, “Classification of road curves and corresponding driving profile via smartphone trip data,” *International Artificial Intelligence and Data Processing Symposium (IDAP),* pp. 1-7, Septmeber 2017.

[5] Society of Automotive Engineers (SAE), *SAE J1979: E/E Diagnostic Test Modes,* Warrendale, Pennsylvania: SAE, 2017.

[6] DAEWOO MOTOR CO., LTD., *DAEWOO M-150 BL2 - Daewoo Matiz MY2003 Service Manual,* Inchon, Koreea, 2003.

[7] G. W. Petty, A first course in atmospheric thermodynamics, Madison, Wisconsin: Sundog Publishing LLC, 2008.

[8] “pynmea2 GitHub repository,” [Online]. Available: https://github.com/Knio/pynmea2.

[9] OpenWeather, “Current weather data API documentation,” [Online]. Available: https://openweathermap.org/current.

[10] OpenStreetMap, “OpenStreetMap Overpass API documentation,” [Online]. Available: https://wiki.openstreetmap.org/wiki/Overpass_API.

[11] “bleak GitHub repository,” [Online]. Available: https://github.com/hbldh/bleak.

[12] SparkFun, “ADXL313 Arduino library GitHub repository,” [Online]. Available: https://github.com/sparkfun/SparkFun_ADXL313_Arduino_Library.

[13] K. Reitz, “Requests Python package documentation,” [Online]. Available: https://requests.readthedocs.io/en/latest/.

[14] “gpxpy Python library page,” [Online]. Available: https://pypi.org/project/gpxpy/.

[15] “scikit-learn,” [Online]. Available: https://scikit-learn.org/stable/.

[16] Keras, “Keras Official Page,” [Online]. Available: https://keras.io/.

[17] Matplotlib, “Matplotlib official website,” [Online]. Available: https://matplotlib.org/.

[18] R. Elvik, “Which is the more important for road safety—road design or driver behavioural adaptation?,” *Traffic Safety Research,* vol. 2, p. 9, 2022.

[19] S. A. F. Arkawazi, “The gasoline fuel quality impact on fuel consumption, air-fuel ratio (AFR), lambda (λ) and exhaust emissions of gasoline-fueled vehicles,” *Cogent Engineering,* vol. 6, no. 1, p. 1616866, 2019.

[20] Y. A. Cengel and M. A. Boles, Thermodynamics: An Engineering Approach (Mcgraw-hill Series in Mechanical Engineering), Boston: Tata McGraw-Hill, 2006.

[21] International Civil Aviation Organisation (ICAO), MANUAL OF THE ICAO STANDARD ATMOSPHERE extended to 80 kilometres (262 500 feet), Montreal: ICAO, 1993.

[22] Society of Automotive Engineers (SAE), *SAE J1962: Diagnostic Connector,* Warrendale, Pennsylvania: SAE, 2016.

[23] Society of Automotive Engineers (SAE), *SAE J1972: OBD-II Scan Tool,* Warrendale, Pennsylvania: SAE, 2022.

[24] EPI Inc., “EPI Inc.,” EPI Inc., 25 2 2019. [Online]. Available: http://www.epi-eng.com/piston_engine_technology/volumetric_efficiency.htm. [Accessed 26 4 2024].

ACKNOWLEDGEMENT

This work is the result of my own activity, and I confirm I have neither given, nor received unauthorized assistance for this work.

I declare that I used generative AI or automated tools in the creation of content or drafting of this document.

During the preparation of this work the author used OpenAI’s ChatGPT in order to draft the abstract of the paper and generate PlantUML code for figures 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23. After using this tool/service, the author reviewed and edited the content as needed and takes full responsibility for the content of the thesis.