AutoTracker Technical Manual

Students:

Brian Quilty - 18373856

Conor Marsh - 19728351

Supervisor:

Paul Clarke

05/05/23

1. Abstract

This project aims to improve the users' driving. It uses machine learning and the phones built in sensors to predict and analyze the users driving style. The app detects the speed limit of the road that the user is on. It also detects the speed of the vehicle that the user is in. Using these two values it determines if the speed limit has been broken by the user. The app aims to get the distance and average speed as well as their driving style from the user's journey. It uses object detection and the phone's camera to detect oncoming traffic lights and notify the user through speech that there is either a red or green traffic light approaching. The app aims to detect the distance of the car in front of the user and determine if it has broken the stopping distance limit.

2. Motivation

Our motivation for our project stemmed from our desire to help improve drivers' driving ability with the intention of stopping accidents and improving people's overall driving skills. This means that our project is really well suited for learner drivers who are just beginning to drive and need extra help to improve their driving skills. As the app is able to detect oncoming traffic lights, the user can have extra guidance in case they may be colorblind for example or have general problems identifying colors, or they are drowsy and not fully paying attention, having an extra guide guiding them can be very useful in preventing crashes due to a red light, or them getting a fine from running a red light. Learner drivers also may need this extra guidance as they can be very nervous and keeping track of everything when starting off can be daunting. As the application is able to determine the stopping distance of cars, this can help people’s driving skills. This is because most people don’t know exactly how much distance they should maintain, which can lead to accidents due to sudden brakes. Having an application that conveys this knowledge can help teach the drivers how much of a distance that they should be maintaining. This is especially the case for learner

drivers who again have a lot to keep track of when starting off, so this knowledge can help teach them how much of a distance that they should be maintaining with the car in front of them. The application keeps track of the driving type behavior of the user. This is useful for preventing accidents, as the user can see an overall view of how aggressive, slow or normal they are driving. If the percentages are skewed towards aggressive or slow, there is more concern for an accident, they want the percentages to be as normal as possible. This is also helpful for learner drivers who are just beginning to drive. Having an indication of the type of driver they are, can help them and help change their driving style to a more normal one as they have a constant indication of their driving style. Showing this to their instructor could also be beneficial in helping shift their style.

3. Research

3.1 database

Throughout the duration of our four years in the course, we learned about databases but did not fully know how to implement a local database and integrate it into functioning data that could be used through a codebase.

There were many different options we could have chosen, such as an online database using firebase where an internet connection would be required, we elected against firebase for this reason as our app was going to be used in vehicles were there is no easy access to Wi-Fi, instead mobile data would be needed to connect to the database. As a result of this we had to choose an offline database, again there were various options to choose from such as PostgreSQL, SQLite, MySQL, MongoDB and Couchbase. We decided to go for SQLite. SQLite will store data into the user's local phone into a file. The main reason we used this was because it was an easy-to-use lightweight database and android studio comes in with built in SQLite database implementation.

It is easy to use as there is a lot of documentation about its use in android studio online so there was a lot of information, we could use to ensure that it ran smoothly in our application. It is also mainly based around the most popular language to use for SQL, MySQL. Throughout the course, we were used to using

the syntax that SQLite provided so we did not have to learn a completely different syntax than we were accustomed to which led us to choosing this language.

One of the main drawbacks from using a local database was that we could not link the database data to the user's account. If we were to use an online database like Firebase, then the user could login to their account from any device and still have the data that they were using throughout the app. Therefore, we had to decide the tradeoffs between online use vs offline use. If we used an online database the data could be easily corrupted due to faulty internet connection as it would rely on mobile networks that can become increasingly unstable if you were for example on a backroad in the middle of nowhere, also you could have no internet connection whatsoever if you had no mobile data so the app would lose a lot of its functionality. Using an online database would have made our data more flexible and accessible, but we elected to go for an offline database to ensure that we had a larger set of users that could use our app against a smaller portion of users that were given more flexible functionality.

SQLite gave us an easy way to store various amounts of data that would be used throughout the app, such as the total average speed, total distance travelled, total driver type, and total number of journeys. Also, it could store data for individual journeys such as driver type percentages for the journey, average speed and distance travelled. As well as storing the sensor values throughout the journey that can be used to predict the driver type. We wanted to ensure that the database was not taking too much space on the user's space, to implement this we set a limit on how much individual journeys the database could store, so if there was 25 journeys already in the database, the user could not start a journey and would have to delete a journey before starting another journey.

3.2 location in android studio

We needed to get the location of where the vehicle was for various reasons such as, calculating the speed of the current vehicle, the distance travelled for the journeys, the average speed for the journeys and the speed limit of the current road the vehicle was on. There were a few options that we could have gone for to implement this functionality and acquire the current latitude and longitude co-

ordinates. We could have used google play services and their fused location provider, instead we went for the location manager, which will provide location, latitude and has a built-in speed function which will get the current speed of the car that you are in. After testing the speed function of the location manager and using our own function which would use the latitude and longitude points to estimate the speed by using the current points and the previous points and the time between the points and then plugging the details into the speed formula: speed = distance / time. After multiple tests we elected to go for the location manager's built-in speed function as we felt it was more accurate and gave better results.

To use the location manager, we would have to use the phone's built-in GPS feature, to ensure this we would have to ask for permission from the phones user to use it, if they decided to not give permission then we would not get the functionality for average speed, distance travelled.

The latitude and longitude co-ordinates were vital in ensuring the functionality of the application. For acquiring the distance travelled we needed to plug the latitude and longitude co-ordinates that the location manager gave us into a formula that would calculate the distance between the two points. Our original implementation had us getting the longitude and latitude points at the start of the journey and the latitude and longitude co-ordinates at the end of the journey and then calculating the distance between those two co-ordinates. After doing research we realized that this would not work as if the user drove from point a to point b and then drove back to point a, the distance travelled would be 0 as the co-ordinates would be the same as the end and start points were the same. To calculate the average speed, we originally thought that we were going to need the latitude and longitude co-ordinates, where we would use a function like the distance, getting the distance and time between the start and end points and then using the speed formula to get the average speed. We, however, faced the same problem we faced when calculating the distance. If the user went from point A to point B and back to point A the average speed would be 0. After research we decided to add all the speed values and the total count of speed values and then divide them to get a more accurate average speed throughout the journey.

3.3 Sensors

After doing research on Kaggle and looking for ways to implement machine learning into our project that would take advantage of the phones technology to help give the user information about their vehicle and their driving style we came across this dataset ; https://www.kaggle.com/datasets/outofskills/driving-behavior. This is a dataset based on using the phone's gyroscope and accelerometer sensors to predict the driving type behavior of the driver which could either be slow, normal or an aggressive driving type. Kaggle has given it a usability of 10 which means they believe it to be a faultless dataset, it also expected to be updated annually and has a paper discussing its validity and the research that they went through to get the values and ensure that the values were correct, this can be found here; https://www.researchgate.net/publication/365629800_Building_a_Driving_Behaviour_Dataset. We also looked in other places for similar datasets to ensure that we picked the right one, for example we searched for "driving behavior" on roboflow a public website where there were accessible datasets, we couldn't find any viable datasets there, we also rigorously looked throughout google for driving behavior datasets , a lot of these datasets were also based on sensors from the phone. There was a similar study we found from; https://data.mendeley.com/datasets/9vr83n7z5j/2, were there was also a study paper based on the dataset which can be found here; https://www.sciencedirect.com/science/article/pii/S2352340922002037. However, there was no usability rating associated with it and there were less downloads than the one we picked. However, this dataset was a good backup incase that we ran into issues with the dataset that we chose.

There were other datasets available on Kaggle that attempted to determine driver behavior using sensors that could determine if there were sudden breaks, right turns left turns acceleration etc. For these datasets it had a lower usability rating and did not have much research associated with it, so we would have had to just take their word for it which would have had more risk associated with it.

To use the dataset effectively and get the best results we were going to have to implement a machine learning algorithm which could Analyse the dataset. We were going to have to get the sensor values of the phone. After doing some

research we found out that phones have built in sensors like gyroscope and accelerometer which were the sensors that were used in the dataset to predict the driver type. From this we knew that this was a good idea to implement machine learning around this dataset. After doing some more research we found out that a phone can access the sensor values from the phone if the sensors were built into the phone. Now we can get the required sensor values every few seconds and use these values to predict the driving behavior of the person operating the vehicle. Sensors were built in to basically all modern-day phones, however if a user was using a very old phone there could be a possibility that the user will not be able to access this functionality.

3.4 machine learning

When implementing machine learning neither of us had much experience with it. We had a brief overview of it in Data warehousing and mining. In this module there was an assignment where you had to use machine learning briefly however the assignment wasn't entirely based around its implementation meaning for the most part, we were going in to using machine learning from a blind perspective.

To counter this lack of knowledge we spent a few days researching what machine learning was. We found out a lot of interesting information surrounding machine learning from this process. Machine learning is a big component in the advancement of a technology that is rising in popularity; Artificial intelligence. Machine learning in a simple expression does what it says in the tin, allows the machine to learn from itself and in our case learning from the dataset that we provide it. Machine learning aims to make a prediction given data, which suits us perfectly as we want to use it to predict the driver type. Machines learn from the data it is provided. From further research we determined that we were going to be using supervised machine learning, where we will be used as a guide to enable and teach the algorithm what decision to make, this will be done through the dataset using predefined classified labels such as aggressive, normal and slow. This is different than unsupervised machine learning, this is when a computer aims to identify complex patterns inside the data that it was provided, so the data will have no label, so the algorithm will have less guidance from the developer.

There were various options to choose from when deciding what sort of machine learning algorithm to implement. Originally, we implemented the machine

learning algorithm through python and connected it to android studio using chaquopy. We would use the sklearn library in python and specifically the decision tree classifier to implement it. in the end we decided that from a design perspective and maintenance perspective it would be easier to implement the machine learning in java instead. Also, we would have to pip install all the necessary libraries for chaquopy and therefore having a significant impact on the size of the application.

As an alternative we decided to implement the algorithm in java making it more in line with the language that we were already using throughout the project and helping improve the design element of the project.

To do this we use the weka library, which was a machine learning library specifically used for java. In weka there were various machine learning algorithms that we could have used like decision trees, but we decided to use a random forest algorithm to implement the machine learning for our project. This is a supervise machine learning algorithm which fits our dataset. It performs better when the dataset is classified which ours is. A real-world algorithm of how the algorithm works is imagine a person wants to know what the funniest tv show is, they decide to ask their friends, family members and co-workers they then decide that the funniest tv show is the one that was suggested by most people.

In this model a subset of data points and features is collected for constructing each decision tree. Then individual decision trees are constructed for each sample. Each decision tree will then generate an output. Then the final output is considered based on the majority voting for class-action. Majority voting is the majority classification instance that arises from constructing all the decision trees. Random forest uses parallelization meaning we can fully use the CPU as each tree is created independently. We also don't have to split the dataset into test and train data as there will always be 30% of the data that is not seen.

It is stable as it is based on majority voting. Comparing random forest and decision trees, random forests do not run into the problem of overfitting as it is based on majority ranking, whereas decision trees are allowed to grow without any control.

However, it is computationally slower. Random forests also don't use rules to make predictions whereas decision trees do. Random forests are known to be

more successful that decision trees as they are more diverse and acceptable and as our dataset has a usability rating of 10 and is a relatively large dataset with three thousand lines of data, random forest classifier is a good choice.

3.5 Speed Limit

When researching how to get to the speed limit of the road that the vehicle was currently on it became apparent quickly that we were going to need the latitude and longitude co-ordinates of the phone. As discussed already this was achieved through using location manager and the phone's GPS. After researching we realized that we had two options to choose from, googles road Api or Bings Road Api. In the end we elected to go for Bings Road Api but googles Api could have also been a good option, and acted as a backup option while we began using Bing, so if a problem arose, we could fall back on google. Here is a link to Bings Api; https://learn.microsoft.com/en-us/bingmaps/rest-services/routes/. Although from testing both in real-life and in unit testing the Api seemed to be accurate and gave no sign that the results were inaccurate, the accuracy of the Api cannot be guaranteed. The data is not in real-time, so if for whatever reason a road decides to change its speed limit tomorrow the information will be incorrect. For Ireland though through research we found that the speed limit of roads is mainly defined by what type of road it is example, motorways have a speed limit of 120 KM/h, regional roads are 80KM/h and National roads are 100km/h. This means that most roads are standardized and are unlikely to deviate from the set speed limit of the type of road it is. It also means it is easy to test the accuracy of the Api as we can give it regional, urban and motorway roads and know what the speed limit is for those roads.

Originally, we were going to use python and chaquopy to get the speed limit, but we decided from a design standpoint and maintenance standpoint it would be more convenient to use java. To get the speed limit we also needed internet connection, if there was no internet connection, we would let the user know that there was no connection available to get the speed limit.

3.6 TensorFlow

To detect the objects that would pass through the phone's camera we decided to use TensorFlow, through research we could have used something like YoloV7,

instead we used tensor flow and efficientDET. Object detection is the objective of identifying objects and their position on the screen. There are two main types, one-stage methods which prioritize speed and two-stage methods which prioritize detection accuracy.

Object detection in tensor flow works by dividing the network into two, the first division permits networks to isolate the task of locating and classifying them using R-CNN, the second division allows networks to predict classes and bounding boxes gin YOLO.

The model breaks down the input image that came from the camera in the phone, it breaks down that image into several components, then it draws boxes around them segmenting them. Each component follows feature extraction as the model intensifies. If there are any visual features in the bounding boxes, the model predicts the objects in that bounded box.

TensorFlow uses deep and machine learning for object recognition. Deep learning uses a programmable neural network that enables machines to make accurate decisions. It requires a lot of training to ensure that the learning process is correct and does not draw inaccurate predictions, hence the word 'deep'.

Machine learning has the help of a human to help make accurate predictions, deep learning however has to rely on its own architecture without the help of humans to make predictions. The different types of neural networks are CNN algorithms, which are specifically used for object detection. This filters through an image and assesses every element that is within it.

RNN has feedback loops that allow the algorithm to remember past data. They can then use this past memory to inform their understanding and predict the future, so the more the model is trained the more accurate it becomes as it uses past understanding.

The objects that we were trying to predict with TensorFlow were traffic lights and cars. Originally to find the dataset we used Kaggle and roboflow to find the image classified datasets that would be used to train the TensorFlow model. The datasets had limits, it needed to be in jpeg, and had to be in pascal VOC format, we tried various datasets in Kaggle and roboflow, we trained them but found their accuracy to be wayward and inaccurate. We then decided to research image

datasets and specifically traffic light and car datasets using the google search engine, through thorough research we found two datasets that provided us with high accuracy when trained using TensorFlow. For detecting cars, we originally wanted to detect license plates as they would be more lightweight but found that it couldn't detect plates at a long distance. So, we focused on cars instead and found the pascal VOC dataset, here; http://host.robots.ox.ac.uk/pascal/VOC/. This dataset had various objects that it classified, including people, cars, animals etc. We filtered the dataset and removed anything that wasn't cars from the dataset, leaving it only classifying cars and nothing else. This is one of the biggest and most well-known datasets, and through testing we found it was also the most accurate dataset we used for detecting cars.

For the traffic lights dataset there were a lot of accurate datasets that could predict a traffic light. However, we wanted the dataset to be able to be more in-depth, being able to predict green and red traffic lights to notify the user if the traffic light was red, notifying them to slow down. Also, if it changed to a green light, notifying them that they can go and not hold up traffic. The dataset we used was; https://github.com/Thinklab-SJTU/S2TLD, this is a big dataset with over 5000 images, giving us a diverse dataset and a lot of data to train from.

3.7 Detect Objects from Camera

To detect the objects using TensorFlow we had to pass in the current image from the video, to get this we had to use the camera from the phone. Through research we learnt that a bind preview could give a preview of and access to the mobile phone's camera. From this we could analyze the current image from the camera and pass it to the detection module which would then use the methods described in the TensorFlow section to make predictions based on the image. We also need permission from the user to use the camera from the user's phone or else the functionality won't be there.

3.8 Object Detector Android Studio

We also had to do extensive research to ensure that the TensorFlow model will work in android studio, through research we found a way to implement the model into our application.

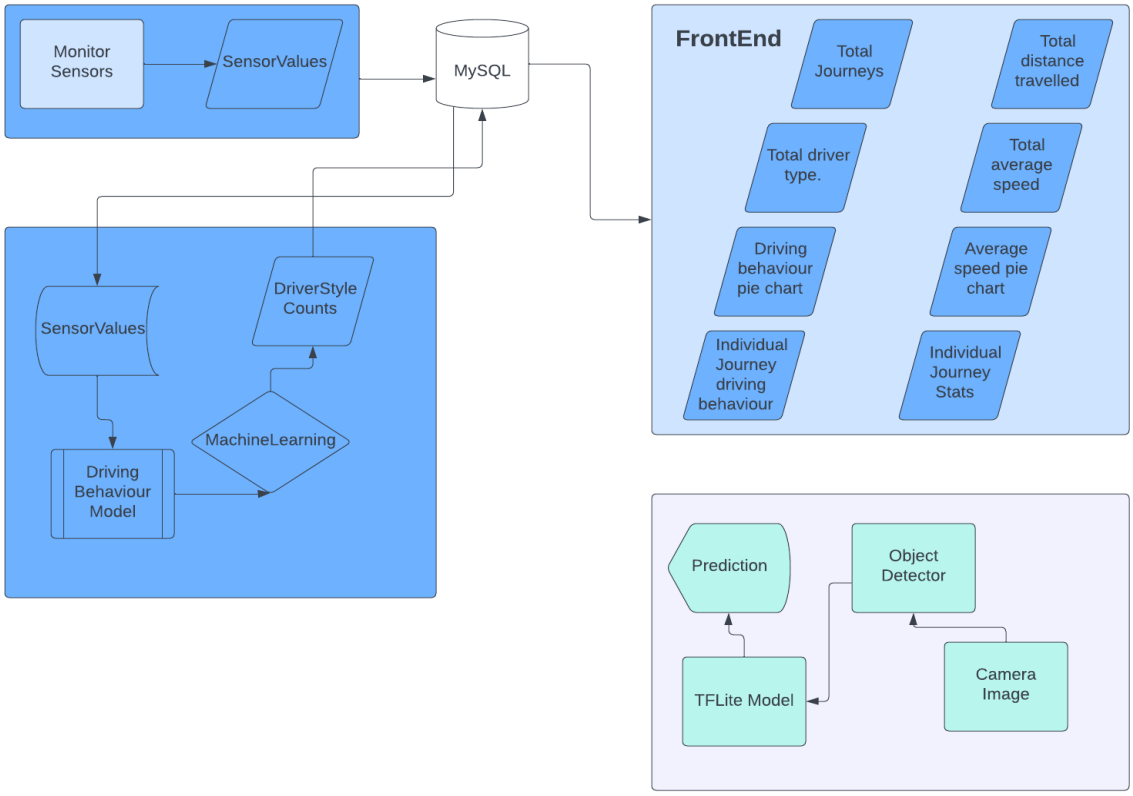
To do this we imported various TensorFlow libraries, like the object detector library, which can help predict the images that are being passed through from the detection activity. Also, we used the objectdetectorlistener to listen for updates on results that are got from the TensorFlow model.

To conduct this research, we used various methods such as thoroughly looking through google, looking at various you tube videos that implemented TensorFlow models in android studio and going through android studio documentation about TensorFlow which really helped make things possible.

Originally, we were ensuring if this was even possible due to the limitation of using a phone to make this possible however through the research, we were surprised by the phones ability to run such architecture.

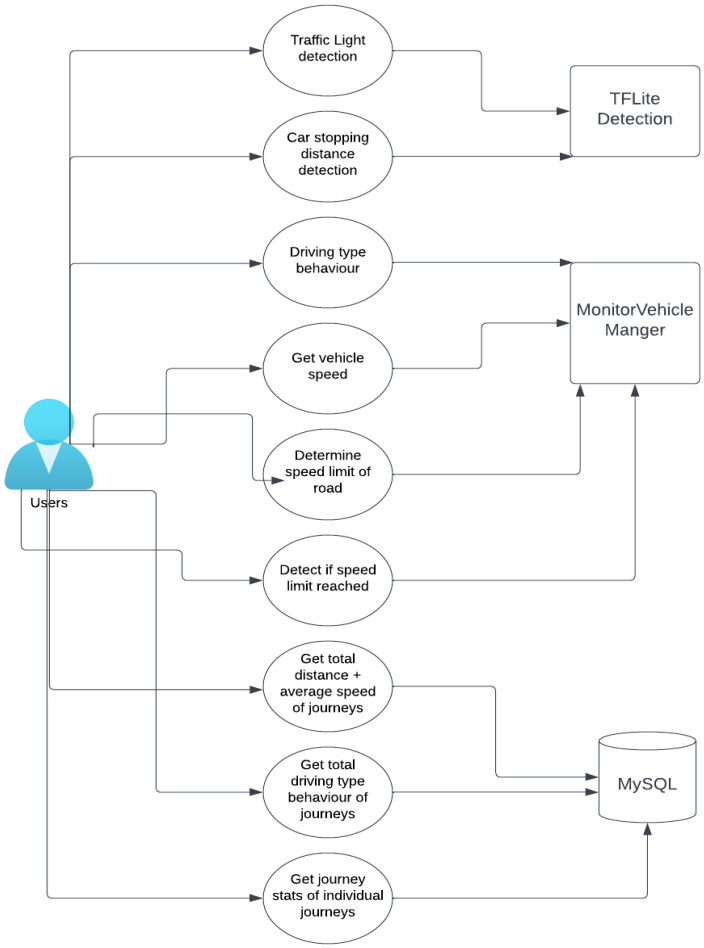
4. Design

4.1 System Architecture



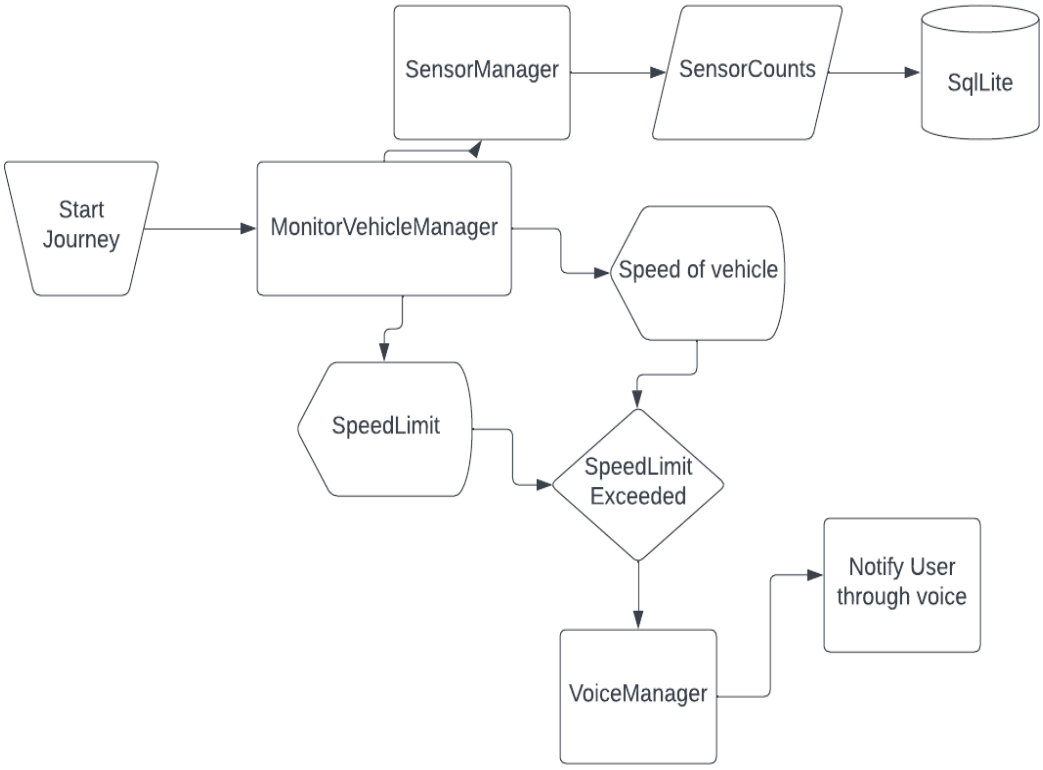
The system architecture is predominantly centered around the MySQL database. The monitorSensors class gets the sensor values of the current journey and passes it to the database. The database uses these sensor values, passes it to the driving behavior model which uses machine learning to determine the driver style counts, these counts are then stored inside the database. The frontend uses the database to display various values to the user. For example, the total journeys, total distance travelled, driving behavior pie chart etc. The detect objects architecture uses the image from the camera, passes it to the objectdetector class which uses the tflite model to make a prediction on the object that is returned, e.g., traffic light, car. From here it can give the user information such as whether to drive or stop depending on traffic light, or if they are too close to a car based on their speed.

4.2 Use case diagram



The users when using the app can detect traffic lights and detect stopping distance for cars using tflite detection. The application can determine driving type behavior, get the current vehicle speed, determine the speed limit of the road and determine if the speed limit is reached. This is done by using the MonitorVehicleManager. They can also get the total distance and average speed of all total journeys, as well as the total driving type behavior and journey stats for individual journeys. This is done by using the MySQL database which stores all the data from the journeys.

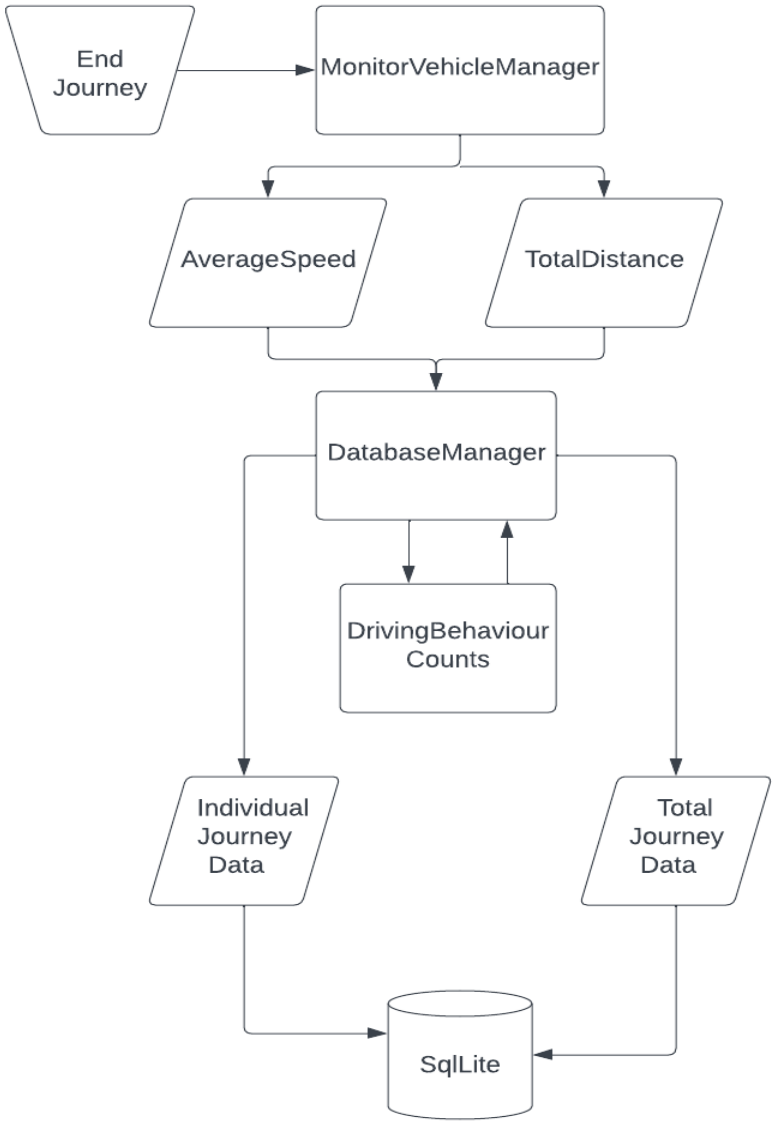
4.3 Start journey data flow diagram



For this data flow diagram, when the user starts the journey, they will start the operation of the monitorVehicleManager class, this class uses the sensorManager which gets the sensor counts from the sensors within the phone and stores the values inside the MySQL database. The monitorVehicleManger gets the speed of the vehicle and the speed limit of the road that the user is on. This class uses both

these values to determine if the speed limit has been exceeded. If it has been exceeded, it uses the voiceManager class to notify the user that the speed limit has been exceeded and that they should slow down.

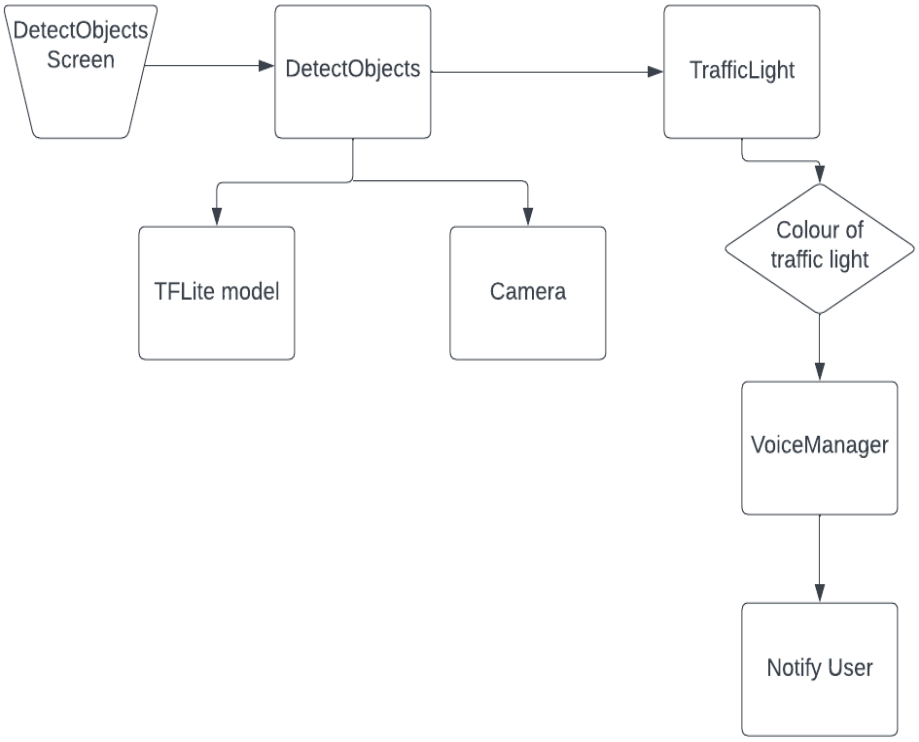
4.4 End journey data flow diagram



When the journey is ended by the user, it calls the monitorVehicleManager to determine the average speed of the journey and the total distance of the journey. These values are passed through to the databaseManager, which analyses the sensor count values that were stored in the database for the journey. It then uses

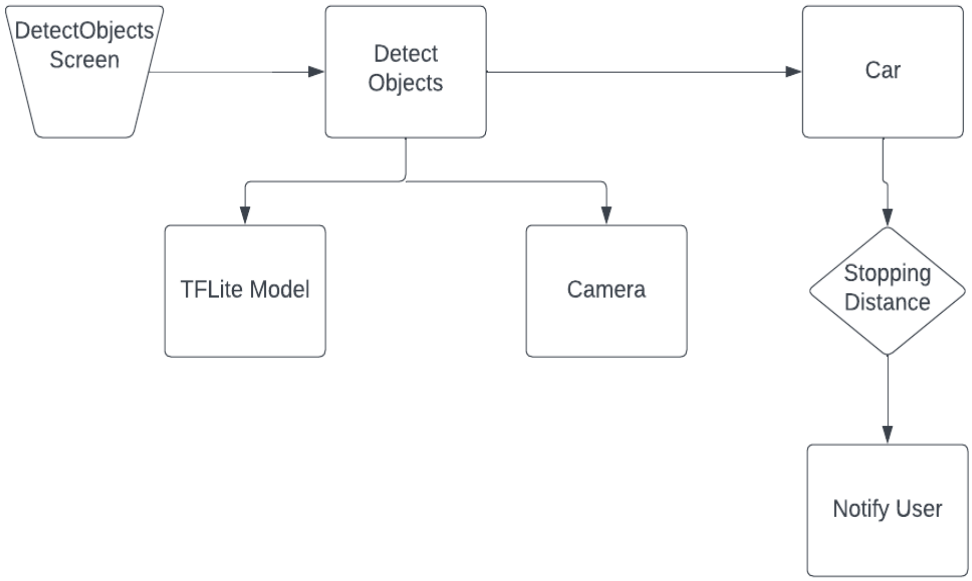
all this data that has been collected and uses it to store individual journey data as well as total journey data inside the MySQL database.

4.5 TrafficLight Detection data flow diagram.



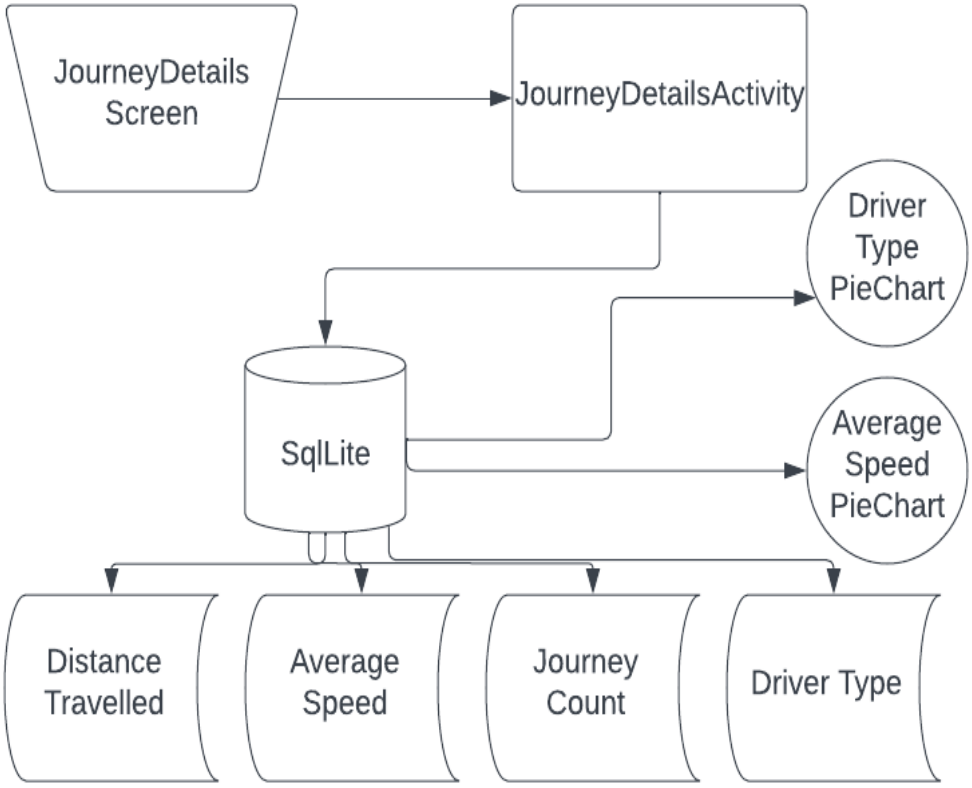
The detectobjects screen uses the detectobjects class which in turn uses the tflite model we created and the camera to analyze the image using the tflitemodel. When the object detector detects a traffic light in the frame, it will return either a green, yellow or red traffic light. When the light turns green a voice will be emitted notifying the user that the light has turned green, and they can drive. If the objectdetector detects that the light has turned red, a voice will be emitted notifying the user that the light is red and to slow down and stop.

4.6 Car object detection data flow diagram



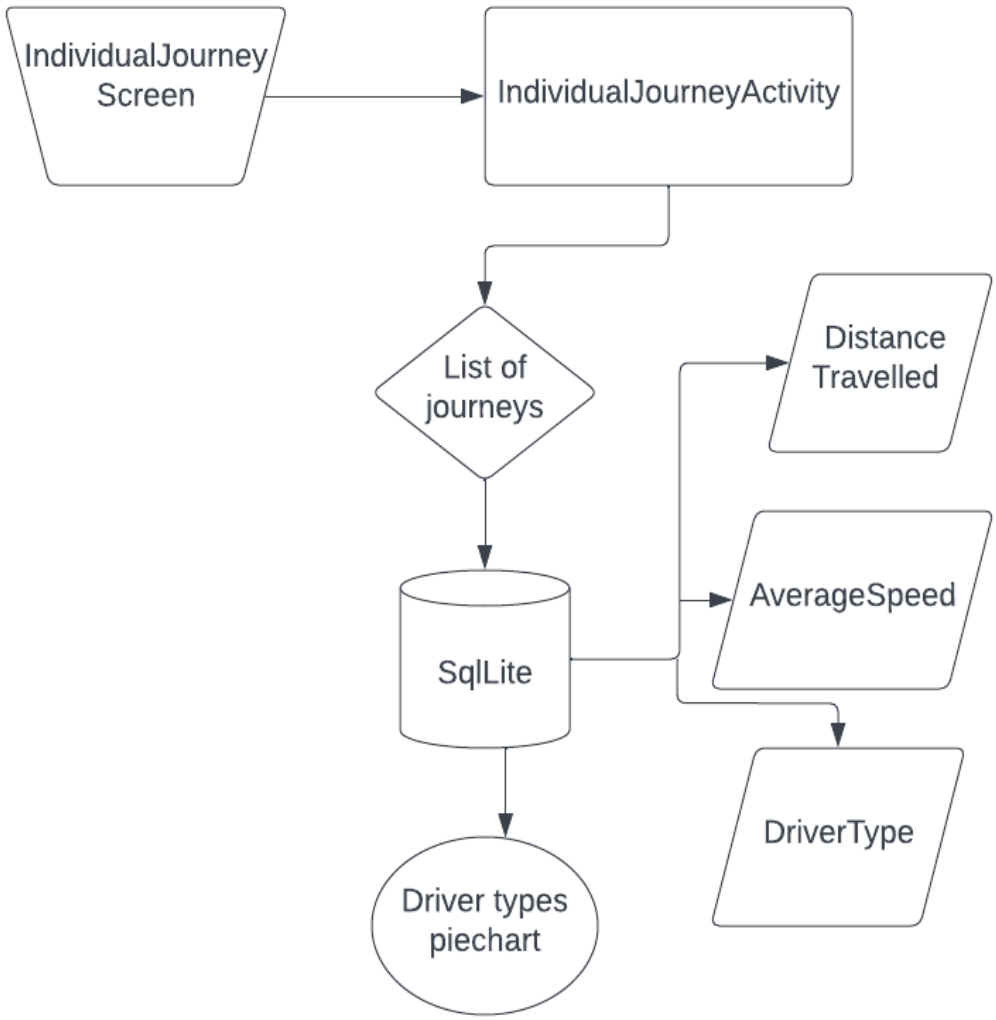
The user will go into the detectobjects screen which will use the detectobjects script. This script will use the tflite model and the camera to detect objects in the image. If it detects a car it will take into account, the speed of the vehicle and an estimate of how far away the vehicle is. If the stopping distance is met using these variables, then the user will be notified to maintain a distance with the car in front of them.

4.7 OverallStats data flow diagram



When the user wants to view the overall stats of all their journeys, they can go to the journeydetails screen. From here this screen will use the journeydetailsactivity to connect to the sql database and display the total stats of all the journeys. It will display the total distance travelled, average speed, total journey count and the driver type. It will also display the percentage of each driver type inside a piechart as well as the percentage of the percentages of the average speed in the form of a Piechart also. These values will be acquired from the values stored within the database.

4.8 Individual journey stats data flow diagram



When the user wants to view stats for individual journeys, they have to go to the individual journey screen. This screen uses the individual journeys activity, which will list all the journeys that the user has taken. It will display data from the MySQL database such as the distancetravelled, average speed and the driver type for each journey. It will also show the percentage of each driver type in the form of a pie chart for every journey as well.

Implementation;

5.1 TensorFlow model;

To build the TensorFlow model we used Google Collab to do so.

```
import numpy as np
import os

from tflite_model_maker.config import ExportFormat, QuantizationConfig
from tflite_model_maker import model_spec
from tflite_model_maker import object_detector

from tflite_support import metadata

import tensorflow as tf
assert tf.__version__.startswith('2')

tf.get_logger().setLevel('ERROR')
from absl import logging
logging.set_verbosity(logging.ERROR)
```

We imported the tflitemodel libraries first that would be used to create the model. Then to host the dataset online for the notebook to use it we hosted the dataset with both the traffic light dataset and the car dataset in google cloud storage.

```
[ ]  !gsutil cp gs://traffliclights/TF_Dataset.zip .
     !unzip -q TF_Dataset.zip
```

Then we downloaded the dataset from google cloud and unzipped it into the current directory's folder.

```
train_data = object_detector.DataLoader.from_pascal_voc(
    'TF_Dataset',
    'TF_Dataset',
    ['car','red','green','off','yellow']
)
```

We used the tflite module object detector and loaded the data in pascal voc format, which means that for every image in the dataset there was an associated xml file with its classification instance, e.g. red light, car, green light etc. We then built the train data and classified the images into car, red, green, off, yellow which would be the objects that the model was to predict.

```
[ ]  spec = model_spec.get('efficientdet_lite3')
```

We chose the model architecture Efficientdet_lite3 to use for our model, these architectures range from 0 to 4, and increase in size, latency and precision as the architecture increase, we decided to go for one of the higher models as we

wanted our results to be as precise as possible and didn't mind as much about speed.

```
[ ] model = object_detector.create(train_data, model_spec=spec, batch_size=4, train_whole_model=True, epochs=20)
```

Next, we will create the object detector using the train data and the architecture model we just defined. The batch size is the number of training examples used in each iteration. Train whole model means that it will not just tune the head layer but be more thorough to get better results. Epochs mean how many times the model is trained, so 1 epoch refers to one entire passing of training the data algorithm. So therefore, we will train the model 20 times in total.

Now our model is created, and we can export it

```
model.export(export_dir='.', tflite_filename='VehicleDetection.tflite')
```

Once exported we will download it from google collab and pass it into android studio.

5.2 Location Service

To get the latitude and longitude and the speed of the vehicle we would need to use the location manager as already mentioned. As we had to get the speed of the vehicle in two separate instances, one where you are starting the journey and the other instance when determining the stopping distance for the car, we elected to put the location manager inside its own class and start it using a service so it would run in the background like so;

```
serviceIntent = new Intent( packageContext: this, LocationService.class);
startService(serviceIntent);
```

Then whenever the location service is started the function getlocation will be called inside the service which will use the location manager to listen for updates and acquire the latitude and longitude co-ordinates

```
@Override
public void onCreate() {
    getLocation();
}
```

Inside the getlocation function the app will determine if it has permission to use the phones gps, if it does then it will initialize the locationmanager and begin listening for updates.

```
if (ContextCompat.checkSelfPermission( context: this, Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED) {
    locationManager =
            (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER,  minTimeMs: 5000,  minDistanceM: 1,  listener: this);
}
```

Upon opening the app inside monitorvehicleservice.java the application will ask for permission to use the phone's GPS location.

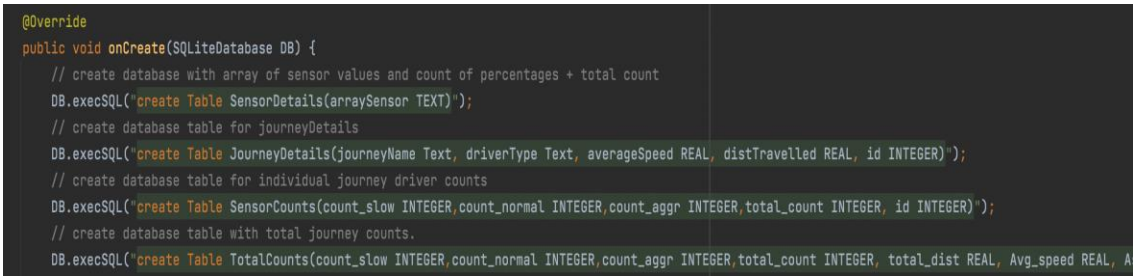
```
static void locationPermission() {
    // thread used to listen for location updates
    runnableLocationPermissions = () -> {
        // if locations permissions on phone not enabled sent notification
        if(ContextCompat.checkSelfPermission(mainActivity, Manifest.permission.ACCESS_FINE_LOCATION) ==
                PackageManager.PERMISSION_GRANTED &&
                ContextCompat.checkSelfPermission(mainActivity, Manifest.permission.ACCESS_COARSE_LOCATION) ==
                        PackageManager.PERMISSION_GRANTED) {
            locationPermission = true;
        } else {
            ActivityCompat.requestPermissions(mainActivity, new String[] {
                            Manifest.permission.ACCESS_FINE_LOCATION,
                            Manifest.permission.ACCESS_COARSE_LOCATION },
                    TAG_CODE_PERMISSION_LOCATION);
        }
    };
    handlerLocationUpdates.post(runnableLocationPermissions);
}
```

When the location has changed the service will pass the location into the monitor vehicle which will use the location object to get the speed of the vehicle, and the distance travelled.

```
@Override
public void onLocationChanged(Location Location) {

    // pass location to monitorvehicle
    MonitorVehicleManager.LocationChanged(Location);

    // look for more location updates
    getLocation();

}
```

5.3 Database

There are 4 tables inside the database that will have been used to store data around different functionality inside the system. The Sensor details table will be used to keep track of all the sensor values that are passed through when the user is on their journey. The journey details will keep track of individual journey details such as the journey name, the driver type of the journey, the distance travelled and average speed of the journey. The total counts will keep track of all the total counts of the driver types, the average speed and the distance.



The database SQL script was used to either insert, delete, update or select details into the various tables.

For example, this function was used to get or select a journey detail based on the detail that was passed into the function as a parameter and the unique journeyID that was also passed into the function as a parameter. The function would then return the details in the form of a cursor object.

```
public void renameJourney(String JourneyRename, int JourneyID){
    SQLiteDatabase DB = this.getWritableDatabase();
    String[] renameArgument = new String[]{JourneyRename};
    String renameSQLString = "UPDATE JourneyDetails SET journeyName= ? WHERE id=" + JourneyID;
    DB.execSQL(renameSQLString,renameArgument);
}
```

This function was used to update the name of the journey inside the journeydetails. It used the Update command in SQLite and used the journey name from the function parameter based on the unique journey id parameter. It would then execute the SQL based on the variables.

```
public void deleteJourney(int DeleteID){
    // delete journey details
    SQLiteDatabase DB = this.getWritableDatabase();
    String whereClause = "id=?";
    String[] deleteIdArg = new String[] { String.valueOf(DeleteID) };
    DB.delete( table: "JourneyDetails", whereClause, deleteIdArg);
    // delete sensor counts for journey
    DB.delete( table: "SensorCounts", whereClause, deleteIdArg);
}
```

This function was used to delete a journey from the database. It used the delete command in SQLite to delete the row from the table based on the journeyID, it also had to remove the row in sensor counts that corresponded to the journey details.

```
public Boolean insertSensorValues(String SensorValuesArray)
{
    SQLiteDatabase DB = this.getWritableDatabase();
    // insert sensor values into database
    ContentValues contentValues = new ContentValues();
    // originally string will be converted to array later
    contentValues.put("arraySensor", SensorValuesArray);
    long result = DB.insert( table: "SensorDetails",  nullColumnHack: null, contentValues);
    if(result==-1){
        return false;
    }else{
        return true;
    }
}
```

This function was used to insert the sensor values that were passed through into the sensordetails database. It would place the array sensor in the form of a string into content values then it would use SQLite's insert command and insert the content values into the table.

5.4 MonitorVehicleManager

The monitorvehiclemanager was used to get the speed limit of the road that you are currently on, the current speed of the car, the average speed of the car and the total distance travelled.

The locationChanged function was an important function inside this script. It will be called as previously mentioned by the locationservice which will pass in the location object to it.

```
public static void LocationChanged(Location Location) {
    // get speed of vehicle
    getVehicleSpeed(Location);
    currentSpeed = 50;
    // set speed text in mainactivity
    mainActivity.setSpeedText(currentSpeed, String.valueOf(speedLimit));

    // analyse gps co-ordinates
    setCoOrdinates(Location);
    DetectionActivity.speed = currentSpeed;

    // start listening for road speed updates
    if(locationChanged == false){
        locationChanged = true;
    }

    // has speed limit been exceeded?
    try {
        if(speedLimit != null) {
            isSpeedExceeded(currentSpeed, speedLimit);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

This function will get the speed of the vehicle you are currently in by calling getvehiclespeed and passing the location object into it. It will also analyze the gps-coordinates which will be used to calculate the distance of the journey. It then passed the speed of the vehicle into the detectionactivity which would be used to determine if the car is going fast enough for it to notify the user about stopping distances. It will also determine if the speed limit was exceeded by passing in the current speed and the speed limit and emitting a voice if it was over the limit.

To get the speed limit of the road a thread was created and called inside the startlocationservice function that would be called whenever the user started their journey. The speedlimitthread was run using a runnable and a handler, inside the runnable the getspeedlimit function was called.

```
public int getSpeedLimit(double Latitude, double Longitude) throws InterruptedException, ExecutionException {
    System.out.println("spdlm" + speedLimit);
    SpeedLimit speedLimitInstance = new SpeedLimit();
    speedLimit = speedLimitInstance.getSpeedLimit(Double.toString(Latitude), Double.toString(Longitude));
    System.out.println("spdlm" + speedLimit);
    // null means no internet connection
    if(speedLimit == null){
        mainActivity.setSpeedText(currentSpeed, SpeedLimit: "No internet");
        // reset speed limit value.
        speedLimit = 0;
    } else{
        mainActivity.setSpeedText(currentSpeed, String.valueOf(speedLimit));
    }
    return speedLimit;
}
```

This function called the speedlimit class and specifically the getspeedlimit function inside this class, it used the current latitude and longitude values that were passed into the function that was got as previously mentioned by the locationchanged function inside this class. The speed limit class will determine if there is internet connection and return null if there is not, it will then set the text inside the mainactivity stating that there is no internet. If there is internet it will set the speed of the vehicle and the current speed limit inside the mainactivity.

```
public static void setCoOrdinates(Location Location){
    // get gps-co-ordinates used to determine speed of road
    currentLatitude = Location.getLatitude();
    currentLongitude = Location.getLongitude();

    // initialize latitude variables
    if(previousLatitude == 0 && previousLongitude == 0){
        previousLatitude = Location.getLatitude();
        previousLongitude = Location.getLongitude();
    }
    double distance = calculateDistance(currentLatitude, currentLongitude, previousLatitude, previousLongitude);
    // count every 10 meters
    if(distance >= 10){
        distanceTotal += distance;
        previousLatitude = Location.getLatitude();
        previousLongitude = Location.getLongitude();
    }
}
```

This function determines the total distance of the journey that you just travelled. It will store the current latitude and longitude inside variables then if the previous latitude and longitude haven't been initialized it will set them to the current location to ensure that a massive spike in distance is not occurring. Then it will calculate the distance between the current co-ordinates and the previous ones. Then if the distance has gone above 10 meters it will update the previous latitude and longitude values.

```
public static int getVehicleSpeed(Location Location){
    // get speed of location object
    // in m/s, convert to km/h by multiplying by 3.6 and round
    float vehicleSpeed = Location.getSpeed() * 3.6f;
    currentSpeed = Math.round(vehicleSpeed);
    averageSpeedTotal += currentSpeed;
    averageSpeedCount += 1;
    return currentSpeed;
}
```

This function gets the current vehicle speed, and it is used to get the average speed of the vehicle. It gets the current speed by using getSpeed() on the current location object and multiplying it by 3.6 to convert it from miles into kilometers per hour. It gets the average speed by getting a count of the total average speed throughout the journey and also keeps track of the number of times the function was called and divides the two to get the average speed for the journey.

5.5 SensorManager

First, we initialize the sensor class inside mainactivity, then when the journey has been started from the main activity, we call the onresume method like so;

```
public void startJourney(){
    serviceIntent = new Intent( packageContext: this, LocationService.class);
    startService(serviceIntent);
    // reset the list to be entered into the database
    databaseService.resetDriverList();
    // set journey state to true
    monitorVehicle.setJourneyState(true);
    // start the journey service
    monitorVehicle.startLocationService();
    // resume looking for sensor values
    sensors.onResume();
```

The onresume method inside the sensorservice registers the listeners for gyroscope and accelerometer, when the journey is ended the onPause function is called inside the sensorservice which unregisters the listeners for both the gyroscope and accelerometer.

The sensorservice class implements sensoreventlistener which determines if there has been a change in sensor values inside the phone when the sensors have been registered.

```
public void onSensorChanged(SensorEvent Event) {
    long startTime = System.nanoTime();
    sensorSecondsCurrent = (double) startTime / 1_000_000_000;

    // if accelerometer type and seconds passed > 1
    if(Event.sensor.getType() == 1 && sensorSecondsCurrent - sensorSecondsStarted >= 1) {
        setAccelerometer(Event);
    }

    // if gyroscope type and seconds passed > 1
    if(Event.sensor.getType() == 4 && sensorSecondsCurrent - previousSecondsGyroscope >= 1){
        setGyroscope(Event);
    }
}
```

When a sensor's value has been changed the onsensorchanged function is then called. This function determines which type of sensor has been changed and also if 1 second has passed between the previous change in that sensor. When those conditions are met it sets either the gyroscope or accelerometer values inside the duly named functions.

```
public static boolean setAccelerometer(SensorEvent Event){
    // reset seconds
    sensorSecondsStarted = sensorSecondsCurrent;

    // assign sensor values
    accelerometerXCord = Event.values[0];
    accelerometerYCord = Event.values[1];
    accelerometerZCord = Event.values[2];

    // add sensorarray value to database as a string separated by delimiter ","
    return databaseService.insertSensorValues(getDriverValues());
}
```

For example, when accelerometer values have changed, it gets the accelerometers x, y and z cords from the sensor event. It then inserts these values into the database. First however it converts both the accelerometer an

5.6 Driving Type Model

Whenever the journey is over and the databaseManager is an analyzing the sensor values in the database it calls the drivingstylepercentage function which will in turn use the driving type model to predict the driver's percentage

```
int[] drivingStylePercentage(int[] PreviousValues) throws Exception {
    // initalize python object
    InputStream targetStream = null;
    try {
        targetStream = mainActivity.getAssets().open( fileName: "driving_behaviour_dataset.csv");
    } catch (IOException e) {
        e.printStackTrace();
    }
    DrivingTypeModel model = new DrivingTypeModel();
    driversPercentage = model.PredictDriverType(targetStream,driverList,PreviousValues[0], PreviousValues[1], PreviousValues[2], PreviousValues[3]);
    return driversPercentage;
}
```

This is the function that implements this functionality. First it creates an input stream for the file that contains the drivingbehaviour dataset.

It then creates an instance for the drivingtypemodel, the function predictdrivertype takes the dataset file as an input stream as a parameter, a list of all the sensor values from the current journey inside the database, previousvalues are the previous slow, normal, aggressive and total counts from all the journeys within the database.

```
public Instances filterDataset(InputStream DrivingTypeDatasetCsv) throws Exception {
    // load data from CSV
    CSVLoader loaderCsv = new CSVLoader();
    loaderCsv.setSource(DrivingTypeDatasetCsv);
    // get driving type dataset from csv
    Instances DrivingTypeDataset = loaderCsv.getDataSet();
    Remove removeTimeStamp = new Remove();
    // all index for attributes to keep
    int[] indexToKeep = new int[]{0,1,2,3,4,5,6};
    removeTimeStamp.setAttributeIndicesArray(indexToKeep);
    removeTimeStamp.setInvertSelection(true);
    removeTimeStamp.setInputFormat(DrivingTypeDataset);
    // Create filter dataset + create class index (slow, normal etc).
    Instances filteredDrivingTypeDataset = Filter.useFilter(DrivingTypeDataset, removeTimeStamp);
    filteredDrivingTypeDataset.setClassIndex(filteredDrivingTypeDataset.numAttributes() - 1);
    return filteredDrivingTypeDataset;
}
```

The first thing the class does is filter the driving behavior dataset. It loads the csv and gets the dataset from it. Next thing that needs to be done is to remove the timestamp from the dataset as we do not want it to affect the model in predicting driving behavior on the time of that particular journey.

To accomplish this, we set the index's we want to keep and don't include the index for the timestamp. We ensure that the same format is kept from the drivingtypedataset.

Using the filter, we combine the drivingtypedataset and the removetimestamp Remove attribute to form the final filtered dataset for the driving behavior.

Once we have filtered the dataset, we set the class index which is the attribute to indicate what type of driver the user is, e.g., slow normal aggressive etc., finally we return the filtered dataset.

```
//build a J48 decision tree
Classifier drivingTypeDecisionTree = new RandomForest();
drivingTypeDecisionTree.buildClassifier(filteredDrivingTypeDataset);
```

Next, we build our model from the classifier we chose, which is the J48 decision tree using our filtered dataset.

```
public String[] predictDriverTypeValues(ArrayList<String[]> DriverValuesTmp,Classifier DrivingTypeDecisionTreeTMP,Instances FilteredDrivingTypeDatasetTMP )
    Instance[] driverValuesInstanceTMP = new SparseInstance[DriverValuesTmp.size()];
    String[] predictDriverTypeTMP = new String[DriverValuesTmp.size()];
    int i = 0;
    while (i < DriverValuesTmp.size()){
        String[] tmpDriverValues = DriverValuesTmp.get(i);
        driverValuesInstanceTMP[i] = new SparseInstance( numAttributes: 6);
        driverValuesInstanceTMP[i].setDataset(FilteredDrivingTypeDatasetTMP);
        int ii = 0;
        while (ii < tmpDriverValues.length){
            driverValuesInstanceTMP[i].setValue(ii,Double.valueOf(tmpDriverValues[ii]));
            ii += 1;
        }
        double classIndex = DrivingTypeDecisionTreeTMP.classifyInstance(driverValuesInstanceTMP[i]);
        predictDriverTypeTMP[i] = FilteredDrivingTypeDatasetTMP.classAttribute().value((int)classIndex);
        i += 1;
    }
    return predictDriverTypeTMP;
}
```

Now we will use the decision tree, the sensor values throughout the journey and our filtered dataset to predict the type of driving behavior. First, we will initialize the array that will hold all the driving types that will be returned from the function, then we will initialize the sparse instance which will be used to classify each individual sensor array. Then the list containing all the sensor values will be looped through, tmpdrivervalues will get the first array within the drivervaluestmp array, which will be an array of each x, y and z co-ordinate of the accelerometer and gyroscope sensors. Then we will initialize the first sparse instance at index i within our instance array, setting the dataset as our filtered dataset. Then we will loop through the current sensor array, setting the value with the index and the sensor co-ordinate, looping through the size of the array to get the index and each sensor co-ordinate. We then classify the instance using the temporary sparse instance we just created and defined. This will give us the index of the class which we can get the equivalent string value of using classattribute.value(index), we then add this to our string of driver types and keep looping until there are no more driver sensor values to go through anymore.

```
public int[] getPercentageArray(String[] PredictDriverType, int PreSlow,int PreNormal, int PreAggressive, int PreTotal){
    int[] percentageArray = new int[8];
    percentageArray[0] = Collections.frequency(Arrays.asList(PredictDriverType), o: "SLOW");
    percentageArray[1] = Collections.frequency(Arrays.asList(PredictDriverType), o: "NORMAL");
    percentageArray[2] = Collections.frequency(Arrays.asList(PredictDriverType), o: "AGGRESSIVE");
    percentageArray[3] = PredictDriverType.length;
    percentageArray[4] = Collections.frequency(Arrays.asList(PredictDriverType), o: "SLOW") + PreSlow;
    percentageArray[5] = Collections.frequency(Arrays.asList(PredictDriverType), o: "NORMAL") + PreNormal;
    percentageArray[6] = Collections.frequency(Arrays.asList(PredictDriverType), o: "AGGRESSIVE") + PreAggressive;
    percentageArray[7] = PredictDriverType.length + PreTotal;
    return percentageArray;
}
```

Next, we will get the percentages of each driver type. We will pass in the previous slow, normal, aggressive and total count and the string array of all the driver types as parameters. We will get the count for the slow, normal aggressive and total counts from the current journey using collections.frequency. We will also acquire the current total counts which will be the current journey total counts and also the previous journey counts. We will then return the percentagearray which includes all these results. This percentage array will also be returned to the database manager, which will insert these counts back into the database.

5.7 SpeedLimitManager

The speed limit class does what it says on the tin, it gets the speed limit of the current road that the user is on. It returns the value to the monitorVehicleManager class. It uses this value to determine if the speed limit was exceeded, it also sends the value to the mainactivity, which displays the limit to the user. The speedlimit class needs the internet to connect to Bings Road API, so the first thing we will need to do is to be able to notify the user that they need internet connection, if they can't connect to Bings Road API.

```
class InternetCheck extends AsyncTask<Void,Void,Boolean> {

    public  InternetCheck() {
    }

    @Override protected Boolean doInBackground(Void... voids) { try {
        Socket sock = new Socket();
        sock.connect(new InetSocketAddress( hostname: "8.8.8.8",  port: 53),  timeout: 1500);
        sock.close();
        return true;
    } catch (IOException e) {
        return false;
    } }

    @Override protected void onPostExecute(Boolean internet) {  }
}
```

To acquire the information on whether or not there is internet connection, an internet check class was implemented inside the speedLimitManager class. It uses asynctask to run the class in the background, it connects to a socket, which attempts to connect to a port on the internet. There is a 1500ms delay, to ensure that it is not stuck in an infinite state trying to connect to the internet. Once it has connected it closes the socket and returns true, indicating that it has connected to the internet. If there is an IO exception aka it wasn't able to connect to the internet, it uses a catch exception and in turn returns false.

```
public static boolean isInternetConnection() throws ExecutionException, InterruptedException {
    InternetCheck internetCheck = new InternetCheck();
    return internetCheck.execute().get();
}
```

The IsInternetConnection method calls the class that we just defined, and returns the Boolean, either true or false indicating if there is an internet connection. As it is an asynctask it runs execute and get to run the class and gets the returned value.

```
boolean connected = isInternetConnection();
if(connected == true) {
    thread.start();
    thread.join();
} else{
    // unrealistic number to show that no internet
    speedLimit[0] = null;
}
```

It calls the method and assigns it to the connected boolean. If there is an internet connection it starts the thread which will determine the speed limit of the road. If there is no internet connection it returns a null value. This value will be checked for null inside the monitorVehicleManager and will be displayed to the user that there is no internet connection within the mainactivity.

```
speedLimit = speedLimitInstance.getSpeedLimit(Double.toString(Latitude), Double.toString(Longitude));
// null means no internet connection
if(speedLimit == null){
    mainActivity.setSpeedText(currentSpeed, SpeedLimit: "No internet");
    // reset speed limit value.
    speedLimit = 0;
} else{
    mainActivity.setSpeedText(currentSpeed, String.valueOf(speedLimit));
}
```

If the speed limit is null, it sets the text inside the mainactivity to display no internet, otherwise it returns the speed limit of the road the user is on.

```
public Integer speedLimitThread(String Latitude, String Longitude){
    String urlBingApi = "http://dev.virtualearth.net/REST/V1/Routes/SnapToRoad?points=" + Latitude +"," + Longitude + "&interpolate=false&includeSpeedLimit=true&in
    HttpURLConnection httpConnection = null;
    Integer speedLimitTmp = 0;
    try {
        java.net.URL url = new URL(urlBingApi);
        httpConnection = (HttpURLConnection) url.openConnection();
    } catch (Exception e) {
        System.out.println("EXCEPTION");
    }
    try {
        httpConnection.setRequestMethod("GET");
    } catch (Exception e) {
        System.out.println("EXCEPTION");
    }
```

The first part of getting the speed limit of the road is to define the URL and initialize the HTTP connection. The URL is the link to Bings Road API. It uses the

latitude and longitude values it acquires from the monitorvehiclemanager class. It then opens the connection to the URL using HTTPUrlConnection. It uses the get set request method.

```
try {
    BufferedReader httpBingInput = new BufferedReader(
            new InputStreamReader(httpConnection.getInputStream()));
    String inputLine;
    StringBuffer bingResponse = new StringBuffer();
    while ((inputLine = httpBingInput.readLine()) != null) {
        bingResponse.append(inputLine);
    }

    httpBingInput.close();

    speedLimitTmp = parseJSon(bingResponse.toString());

} catch (Exception e) {
    //return 0;
}
return speedLimitTmp;
```

It gets the response in Json form from using a get request to the URL. It then attempts to parse the Json from the response using a separate method.

```
// Parses the JSON data received from the API
private int parseJSon(String data) throws JSONException {


    if (data == null) {
        return 0;
    }

    JSONObject jsonData = new JSONObject(data);


    String speedLimitString = (jsonData.getJSONArray("resourceSets").getJSONObject(0).getJSONArray("resources").getJSONObject(0).getJSONArray("snappedPoints").getJSONObject(0).get("speedLimit")).toString();



    return Integer.parseInt(speedLimitString);
}
```

The function converts the original response string into a Json object. It gets the Json array resource sets, the first object associated with that array, then within that object the Json array resources and its first object. Then it gets the snappedpoints array, it then gets the first object associated with that array, inside

that object is the speed limit of the road that the user is currently on which is the value we are looking for, finally it returns the speed limit.

5.8 VoiceManager

Originally, we initialized the voiceservice within the mainactivity, but after some consideration we realized we would be using it for detecting objects as well, so we decided to put the voiceservice within its own class, so the code does not have to be duplicated.

```
public void onInit(int status) {
    // setup voice service
    voiceSpeed = new TextToSpeech(activityContext.getApplicationContext(), statusnew -> {
        System.out.println("status...b");
        if (statusnew == TextToSpeech.SUCCESS) {
            int result = voiceSpeed.setLanguage(Locale.ENGLISH);
            System.out.println("Result " + result);
            if (result == TextToSpeech.LANG_MISSING_DATA
                    || result == TextToSpeech.LANG_NOT_SUPPORTED) {
                Log.e( tag: "TTS",  msg: "Language not supported");
            }
        } else {
            Log.e( tag: "TTS",  msg: "Initialization failed");
        }
    });
}

public void voiceSpeak(String VoiceOutput){
    voiceSpeed.speak(VoiceOutput, TextToSpeech.QUEUE_FLUSH,  params: null);
}
```

The voiceService is initialized using the onInit method. This will be called whenever the voiceservice class is initialized using the oninit method inside the mainactivity and detectionactivity. The voice emits what it is told to say using the speak method on the voice object that was just created.

```
public void TestVoiceParameters(String Language,float Pitch, float SpeechRate){
    TextToSpeech tmpVoiceSpeed;
    tmpVoiceSpeed = voiceSpeed;
    Locale tmpLocale = GetLocale(Language);
    tmpVoiceSpeed.setLanguage(tmpLocale);
    tmpVoiceSpeed.setSpeechRate(SpeechRate);
    tmpVoiceSpeed.setPitch(Pitch);
    tmpVoiceSpeed.speak( text: "This is a test",TextToSpeech.QUEUE_FLUSH,  params: null);
}

public void SetVoiceParameters(String Language,float Pitch, float SpeechRate){
    voiceSpeed.setSpeechRate(SpeechRate);
    voiceSpeed.setPitch(Pitch);
    Locale setLocale = GetLocale(Language);
    voiceSpeed.setLanguage(setLocale);

}
```

The testVoiceParameter function is used inside the mainactivity when the user wants to test a new voice locale, pitch and speech rate. It then emits “this is a test” using the parameters initialized by the user.

The testvoiceparameters sets the parameters of the current voice object using the parameters initialized by the user. These are the speechrate, pitch and locale set by the user. Now anytime the voice speaks it will use the parameters set by the user.

5.9 DetectObjects

For the detectObjects class, we initialise the objectdetector using the setupObjectDetector method.

```
Conor Marsh *
private fun setupObjectDetector() {
    // set score treshold, max amount of objects detected
    val optionsBuilder =
        ObjectDetector.ObjectDetectorOptions.builder()
            .setScoreThreshold(0.4f)
            .setMaxResults(3)

    // number of used threads running
    val baseOptionsBuilder = BaseOptions.builder().setNumThreads(2)

    // hardware for running the model. Default to CPU
    when (currentDelegate) {
        DELEGATE_CPU -> {
            // Default
        }
        DELEGATE_GPU -> {
            if (CompatibilityList().isDelegateSupportedOnThisDevice) {
                baseOptionsBuilder.useGpu()
            } else {
                objectDetectorListener?.onError( error: "GPU is not supported on this device")
            }
        }
        DELEGATE_NNAPI -> {
            baseOptionsBuilder.useNnapi()
        }
    }

    optionsBuilder.setBaseOptions(baseOptionsBuilder.build())
```

We use tensor flows object detector, setting the options for the object detector, such as the score threshold that has to be met, which is 0.6. That means that the model has to be at least 60% sure that the object is either a traffic light or a car. We set the max results to three. This is because when we are detecting cars, there can be multiple cars on the road that we are trying to detect. However, we only really want to detect the car in front of us, so we have to include more than one result. Then we set the number of threads that will be used by the object detector. We then allow the object detector to use the gpu inside the phone.

```
val modelName =
    when (currentModel) {
        MODEL_EFFICIENTDETV0 -> "VehicleDetection.tflite"
        else -> "VehicleDetection.tflite"
    }

try {
    objectDetector =
        ObjectDetector.createFromFileAndOptions(context, modelName, optionsBuilder.build())
} catch (e: IllegalStateException) {
    objectDetectorListener?.onError(
        error: "Object detector failed to initialize. See error logs for details"
    )
    Log.e( tag: "Test",  msg: "TFLite failed to load model with error: " + e.message)
}
```

We set the model that the object detector is going to use which is the tflite vehicledetection model that we created earlier using google collab. We define the model's name and begin to build the object detector. We create the detector using the current context, the model's name and the options that we just defined above.

```
fun detect(image: Bitmap, imageRotation: Int) {
    if (objectDetector == null) {
        setupObjectDetector()
    }

    var inferenceTime = SystemClock.uptimeMillis()

    // Create preprocessor for the image.
    val imageProcessor =
        ImageProcessor.Builder() ImageProcessor.Builder
            .add(Rot90Op( k: -imageRotation / 90)) ImageProcessor.Builder!
            .build()

    // convert it into a TensorImage for detection.
    val tensorImage = imageProcessor.process(TensorImage.fromBitmap(image))

    val results = objectDetector?.detect(tensorImage)
    inferenceTime = SystemClock.uptimeMillis() - inferenceTime
    objectDetectorListener?.onResults(
        results,
        inferenceTime,
        tensorImage.height,
        tensorImage.width)
}
```

We define the detect method, which takes in the image that is obtained using the camera's phone. Which we will use to detect objects within the image. This function is used to first process the image using an imageProcessor. We then use the objectdetector that we defined earlier and use it to detect the image that we just processed. We then define the onresults listener which will listen for results that are passed through.

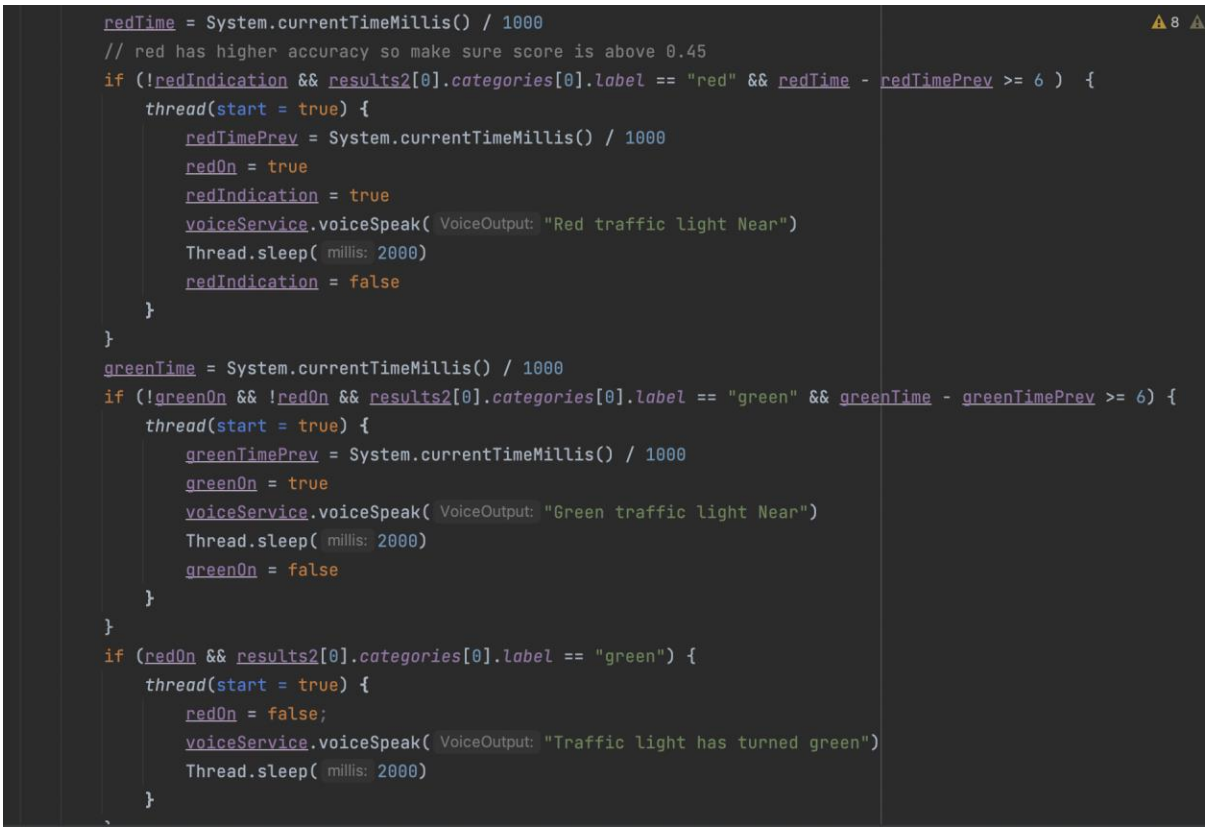
5.10 DetectionActivity

```
private fun bindPreview(cameraProvider: ProcessCameraProvider){

    val preview = Preview.Builder().build()


    val cameraSelector = CameraSelector.Builder().requireLensFacing(CameraSelector.LENS_FACING_BACK).build()

    preview.setSurfaceProvider(binding.previewView.surfaceProvider)

    val imageAnalysis = ImageAnalysis.Builder().setTargetResolution(Size( width: 1280, height: 720))
        .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
        .setOutputImageFormat(OUTPUT_IMAGE_FORMAT_RGBA_8888)
        .build()

    imageAnalysis.setAnalyzer(ContextCompat.getMainExecutor( context: this)) { image ->
        if (!::bitmapBuffer.isInitialized)
        {
            bitmapBuffer = Bitmap.createBitmap( image.width, image.height, Bitmap.Config.ARGB_8888 )
        }
        detectObjects(image)
    }

    cameraProvider.bindToLifecycle(this as LifecycleOwner, cameraSelector, imageAnalysis, preview)
```

The detectionActivity builds a preview of the camera using the backwards facing camera. It then analyses the current image from the camera, using a target resolution while setting the output image format. It then creates a bitmap of the current image and calls the detectObjects function of the image.

```
private fun detectObjects(image: ImageProxy) {
    image.use { bitmapBuffer.copyPixelsFromBuffer(image.planes[0].buffer) }

    val imageRotation = image.imageInfo.rotationDegrees
    // Pass Bitmap and rotation to the object detector helper for processing and detection
    DetectObjectsScr.detect(bitmapBuffer,imageRotation)


}
```

This function calls the detectobjects script and specifically the detect function that we defined earlier. Here it passes the image in the form of a bitmap buffer and passes in the image rotation of the image.



```
redTime = System.currentTimeMillis() / 1000
// red has higher accuracy so make sure score is above 0.45
if (!redIndication && results2[0].categories[0].label == "red" && redTime - redTimePrev >= 6 )  {
    thread(start = true) {
        redTimePrev = System.currentTimeMillis() / 1000
        redOn = true
        redIndication = true
        voiceService.voiceSpeak( VoiceOutput: "Red traffic light Near")
        Thread.sleep( millis: 2000)
        redIndication = false
    }
}
greenTime = System.currentTimeMillis() / 1000
if (!greenOn && !redOn && results2[0].categories[0].label == "green" && greenTime - greenTimePrev >= 6) {
    thread(start = true) {
        greenTimePrev = System.currentTimeMillis() / 1000
        greenOn = true
        voiceService.voiceSpeak( VoiceOutput: "Green traffic light Near")
        Thread.sleep( millis: 2000)
        greenOn = false
    }
}
if (redOn && results2[0].categories[0].label == "green") {
    thread(start = true) {
        redOn = false;
        voiceService.voiceSpeak( VoiceOutput: "Traffic light has turned green")
        Thread.sleep( millis: 2000)
    }
```

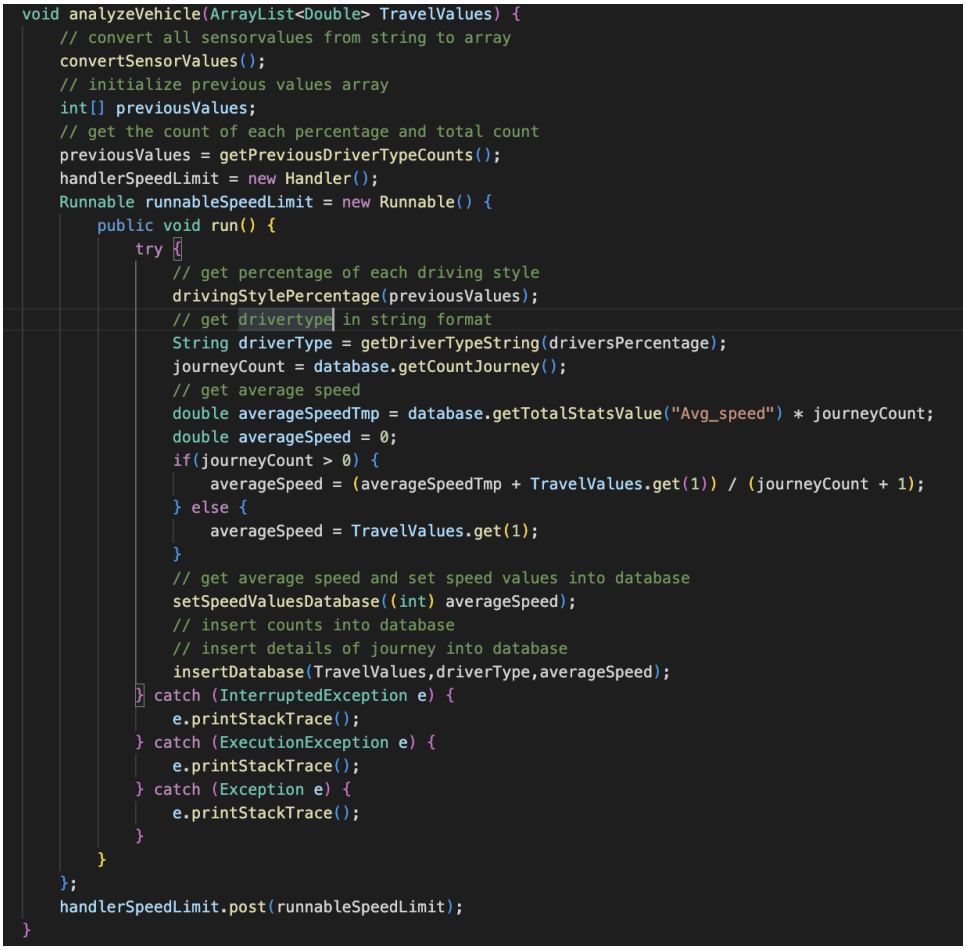
Inside the detection activity, the onresults function listens for results that are being passed through and detected by the model. This piece of code determines if the result is a traffic light. We get the current time using currentTimeMillis. We then determine if the current Time minus the previous time is greater than six seconds. This is used to ensure that the voice is not constantly repeating and speaking. If the class label of the result is red, then it will say a red traffic light is near. It will also indicate that the last traffic light seen is red, which will be used later. For the green traffic light, if the traffic light is green and red hasn't been on and six seconds have passed the voice will say that a green traffic light is near. If the last traffic light seen was red and the traffic light is green, then the voice will notify the user that the traffic light has turned green.

```
private fun detectCars(resultsCars: List<Detection> = LinkedList<Detection>()){
    if(resultsCars[0].categories[0].label == "car" && resultsCars[0].categories[0].score >= 0.65 ) {
        if ((resultsCars[0].boundingBox.right + resultsCars[0].boundingBox.left).toInt() in 2500..3500 ) {
            // get average of distance estimate
            totalDistance += 1
            if (totalDistance == 1) {
                distance1 =
                    (resultsCars[0].boundingBox.bottom - resultsCars[0].boundingBox.top).toInt()
            }
            if (totalDistance == 2) {
                distance2 =
                    (resultsCars[0].boundingBox.bottom - resultsCars[0].boundingBox.top).toInt()
                // will now get average distance
                averageDistance = (distance1 + distance2 ) / 2
                // get stopping distance with speed + meter estimate

                // speed 30-40km and average distance is 12m
                if (speed in 30..40 && (resultsCars[0].boundingBox.bottom - resultsCars[0].boundingBox.top) > 480) {
                    voiceService.voiceSpeak("Maintain Distance! Stopping distance reached")
                }
                // speed 40-50km and average distance is 18m
                if (speed in 40..50 && (resultsCars[0].boundingBox.bottom - resultsCars[0].boundingBox.top) > 320) {
                    voiceService.voiceSpeak("Maintain Distance! Stopping distance reached")
                }
                // speed 50-60km and average distance is 25m
                if (speed in 50..60 && (resultsCars[0].boundingBox.bottom - resultsCars[0].boundingBox.top) > 240) {
                    voiceService.voiceSpeak("Maintain Distance! Stopping distance reached")
                }
                // speed 60-80km and average distance is 33m.
                if (speed in 60..80 && (resultsCars[0].boundingBox.bottom - resultsCars[0].boundingBox.top) > 110) {
                    voiceService.voiceSpeak("Maintain Distance! Stopping distance reached")
                }
                totalDistance = 0
            }
        }
```

The detectCars function is used to detect if the stopping distance has been breached. First it checks if the result is a car and if the prediction accuracy is above 65%. It will get the last two results to get an average to be 100% certain the stopping distance has been breached. We tested using a car and a measuring tap to determine the bottom-top values with regards to each stopping distance. We found that after 12m, the bottom-top value averages around 450, we decided to give the user a small margin of error to make sure the stopping distance has been 100% breached. We also used the location service to pass in the speed value to the detection activity. If the speed is between 30 and 40 and the distance is above 480, the stopping distance has been reached, and we notify the user. The values for bottom top have been defined to coincide with the speed value as defined by the garda's handbook. When two distances have been checked, the variable to determine the previous two car distances is reset.

5.11 Database Manager

The database manager class analyses the output from the vehicle journey and uses these values to ensure that the data is saved correctly within the database.



```
void analyzeVehicle(ArrayList<Double> TravelValues) {
    // convert all sensorvalues from string to array
    convertSensorValues();
    // initialize previous values array
    int[] previousValues;
    // get the count of each percentage and total count
    previousValues = getPreviousDriverTypeCounts();
    handlerSpeedLimit = new Handler();
    Runnable runnableSpeedLimit = new Runnable() {
        public void run() {
            try {
                // get percentage of each driving style
                drivingStylePercentage(previousValues);
                // get drivertype in string format
                String driverType = getDriverTypeString(driversPercentage);
                journeyCount = database.getCountJourney();
                // get average speed
                double averageSpeedTmp = database.getTotalStatsValue("Avg_speed") * journeyCount;
                double averageSpeed = 0;
                if(journeyCount > 0) {
                    averageSpeed = (averageSpeedTmp + TravelValues.get(1)) / (journeyCount + 1);
                } else {
                    averageSpeed = TravelValues.get(1);
                }
                // get average speed and set speed values into database
                setSpeedValuesDatabase((int) averageSpeed);
                // insert counts into database
                // insert details of journey into database
                insertDatabase(TravelValues,driverType,averageSpeed);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
    handlerSpeedLimit.post(runnableSpeedLimit);
}
```

The analysevehicle method is called when the journey is finished. It runs in the background to ensure that the app is not frozen for a period when the journey is finished. It gets the driver percentages of each driver type, and the main driver type of the journey. It uses the previous total average speed value and uses the average speed of the current journey alongisde the previous total to create a new total average speed. It then calls the insertdatabase method with the travelvalues(which are average speed and total distance of current journey), the driver type and the average speed.

```
void convertSensorValues(){
    // get sensor values from the database
    Cursor sensorCursor = database.getSensorValues();
    while(sensorCursor.moveToNext()){
        // sensor value as string
        String sensorValue = sensorCursor.getString( i: 0);
        // 6 string values converted to array by separator ","
        if(sensorValue != null) {
            String[] driverValues = convertStringToArray(sensorValue);
            // add to array of arrays of sensor values
            driverList.add(driverValues);
        }
    }
}
```

The convertsensorvalues method is called inside the Analyse vehicle method above and is used to convert the sensorvalues of the journey into a string array. It loops through each value inside the database which will be in the form of a string and separates it using a delimiter, which will be used when getting driver percentages.

```
int[] drivingStylePercentage(int[] PreviousValues) throws Exception {
    // initalize python object
    InputStream targetStream = null;
    try {
        targetStream = mainActivity.getAssets().open("test_motion_data.csv");
    } catch (IOException e) {
        e.printStackTrace();
    }
    DrivingTypeModel model = new DrivingTypeModel();
    driversPercentage = model.PredictDriverType(targetStream,driverList,PreviousValues[0], PreviousValues[1], PreviousValues[2], PreviousValues[3]);
    return driversPercentage;
}
```

This function uses the drivingtypemodel that we defined earlier to predict the driving type percentages of the current journey using the driving type behavior dataset as an input stream, the previous values to get the new updated values and the driver list, which is a list of all the sensor values that were in the current journey. This is then assigned to a driver's percentage list which will tell us the count of each driver type for the current journey plus total journey.

```
void insertDatabase(ArrayList<Double> TravelValues, String DriverType, double AverageSpeed){
    // insert the total count + total percentage count to database
    database.insertDriverTypeCounts(driversPercentage[0], driversPercentage[1],driversPercentage[2],driversPercentage[3]);
    double currentTotalDist = database.getTotalStatsValue("total_dist");
    // add to total count of values overall
    database.updateTotalCounts(driversPercentage[4], driversPercentage[5],driversPercentage[6],driversPercentage[7],currentTotalDist + TravelValues.get(0),  AverageSpeed);
    //get total count of journeys
    int journeyCount = database.getCountJourney();
    String journeyName = "Journey" + (journeyCount + 1);
    // insert journeydetails
    database.insertJourneyDetails(journeyName,TravelValues.get(0),TravelValues.get(1),DriverType);
}
```

This function updates the total driver counts inside the database using the driverspercentage array we defined above. It inserts the driver type counts for the current journey as well as updating the total counts of every journey. It also inserts the current journeys details, which will be the name of the journey, the total distance travelled of the journey as well as the average speed of the current journey.

5.12 Login/Create Account

The page to create an account and login to an account is on the same page. We wanted the login functionality to be as simple as possible to ease the user into our app. As our database is offline and we wanted to keep the size of the database file down, we elected to permit the user to use only one account. Now if the user wants to create an account, they will lose all their data. We also created a splash screen and a stay logged in feature. When the app is open the splash screen will determine if the stay logged in feature was set and log the user into the app otherwise the user is sent to the login screen.

6. Problems solved

The problems that we solved in this application was how to detect objects, how to implement machine learning to solve a practical problem and how to implement a lightweight database in a phone application while it still being able to display relevant data to the user. To solve the problem of detecting objects using a phone we implemented a tflite model as well as implementing tflite's object detection libraries to implement it in java. We built the model using a dataset full of images

of the objects that we wanted to be detected. We then trained a tflite model to be able to predict images of unseen data and accurately identify the objects we wanted to be detected. This in turn helped to solve the problem of helping drivers using technology to prevent accidents / help their overall driving knowledge / experience. We solved the problem of implementing machine learning using the weka library and forest trees classifier to be able to make predictions on a dataset containing driving behavior. This helped solve a practical problem that is improving users driving style/ skills by being able to determine the users driving style. This has helped us understand that applying machine learning in the real world can be beneficial. To implement a lightweight database within a phone application we used SQLite. This is a lightweight database. We implemented a threshold of 25 journeys to ensure that the application is not taking up too much space on the user's phone. To implement this functionality, we had to be able to let the user delete any journeys that they wanted to. The app implements a lightweight database that conveys to the user any details they might need in a manner that is concise as possible.

7. Results

After lots of testing with the app we felt that the driving behavior worked as intended and gave the user a good insight into the type of driver they were, as we ran a test advising the driver to drive slow and then to drive normal, for the first journey they got a driver type of slow and then for the second driver type it was aggressive. We also felt like the speed limit worked well as we tested it on national, regional and urban roads and they matched the national speed limits that were set. We also thought that the total distance worked well as we tested the functionality on a journey from point x to point y. The app gave us a total distance of 1.48km, we placed the same points x and y on google maps and google maps returned with a total distance of 1.5km, which they round up. We also felt measuring the speed of the vehicle worked as intended as we tested it alongside various speedometer apps, and they were both giving similar results. We believed detecting the traffic lights also worked the way we intended. We did, however, feel like it could have worked better at detecting traffic lights at a further distance away to give the user more notice. We felt that the application

was good at detecting cars, also our implementation of detecting cars in front of the vehicle instead of oncoming traffic worked well. Considering there is not really much technology surrounding distance measurement in phones and you need a special camera we felt that the app worked as well as it could have to detect the distance of cars in front of us. Still similar to the traffic light situation it was unable to measure a distance of longer than 30-40 meters, which meant our stopping distance was limited to 80km/h and worked best in the first 60km/h range.

8. Future Work

For future work we believe the app can be developed further with refined features and new features to improve the app. If we built our own dataset of Irish traffic lights, we feel that the traffic light detection could have worked better at further distances. The dataset we used were from traffic lights that were not Irish, so the accuracy would therefore be much higher if the traffic light dataset was solely based on Irish traffic lights. Also, if we built our dataset of Irish cars that are in front of a car, so the back of a car, the app would work better at detecting cars at a distance that are in front. We could also create a dataset of various distance measurements with various cars, and then use machine learning to predict the bottom-top measurement value that the app used with the dataset to get an optimal measurement. We felt that while the app was good at predicting driver behavior it didn’t give much insight as to whether the driving style was optimal or not. To do this we would have to ask for our users' data and ask for permission to use it, then we could compare the users driving style with the average of all users driving styles to compare and see how similar they are to the way most people drive or if they are an outlier. To do this we would have to use a centralized online database that would store all the user's data.