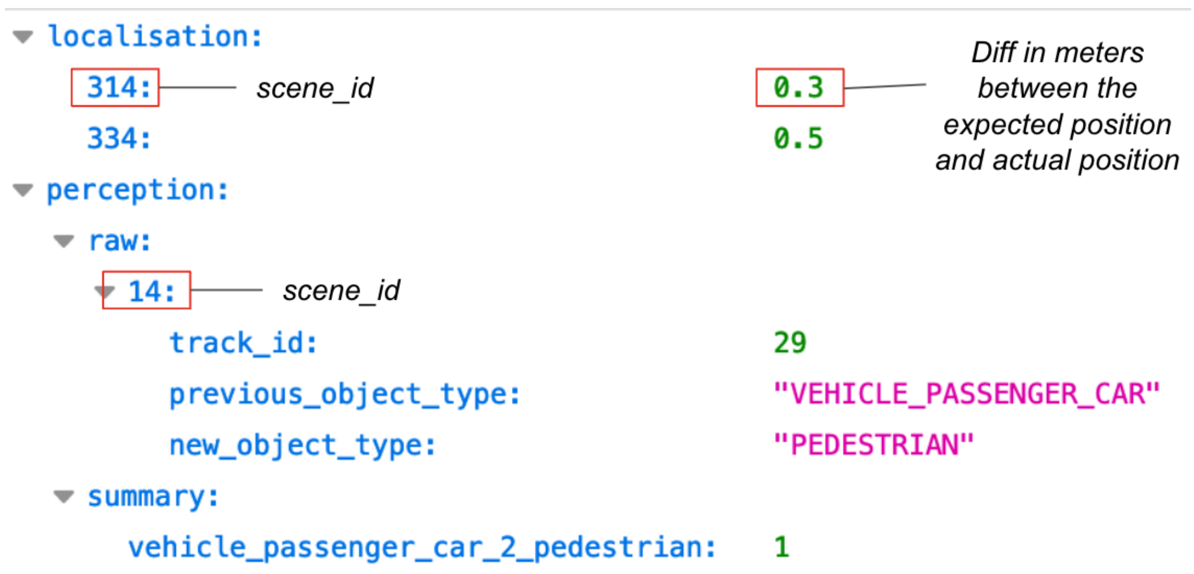
Assignment Company XX

Identifying data issues from localisation and perception data

Explanation of the software architecture

The software is organised in a repository which contains a *python package* in order to guarantee its future extensibility in a flexible and organised way.
This package includes a python module (*data_analyzer.py),* responsible for processing the input data and storing the detected anomalies into a new *json* file whose format is shown in the image below.



```
▼ localisation:
    314:                                   0.3
    334:                                   0.5
▼ perception:
  ▼ raw:
    ▼ 14:
        track_id:                          29
        previous_object_type:              "VEHICLE_PASSENGER_CAR"
        new_object_type:                   "PEDESTRIAN"
  ▼ summary:
      vehicle_passenger_car_2_pedestrian:  1
```

scene_id
Diff in meters between the expected position and actual position
scene_id

Although algorithmic details will be explained in the appropriate section here is an overview of which information is contained in the output file. As shown in the image above, the output file includes the anomalies detected in the "*localisation*" and "*perception*" system. The *localisation* category contains all the *scene_id* where the distance between the expected position and the actual position of the ego vehicle exceeds a certain threshold (specified by the user). On the other hand, "*perception*" includes in turn "*raw*" and "*summary*" categories: the first one lists all the scenes where the object type of tracked object changes, while the latter represents the sum of detected perception anomalies. To clarify with an example, in the image above the "raw" category suggests that at the *14th* frame, a tracked object that was previously classified as a car, has been classified as a pedestrian. On the other hand the summary can be read as "during all the frames, it happened once (*1*) that a tracked object initially classified as a vehicle has been subsequently classified as a pedestrian."

In order to facilitate software deployment, the repository comes with a Dockerfile that can be used to create a Docker Image and run the software in a containerized environment.

Notice that more technical instructions needed to run the software can be found in the README.md file within the repository.

Explanation of the types of issues the software can identify

This algorithm focuses on identifying issues in the localisation system using localisation data and issues in the perception system using perception data.

1. **Issues in the localisation system**: The data provided by the localisation system consist of coordinates, speed and yaw. Since the data is stored every *0.1* seconds (short time-range), we can verify that the coordinates registered by a vehicle at time *t1* are close to the expected coordinates estimated by using the ego's vehicle coordinates, speed and yaw. Even though *speed changes* and *changes of direction* should be gradual, it might happen for several reasons, such as a harsh break, that the difference between the expected coordinates and the real coordinates is large. In many cases this is the expected behaviour, for example if a pedestrian pops out from behind a parked car, and runs into the middle of the street, a harsh break is necessary to avoid a collision. On the other hand, if a pedestrian is normally waiting on the sidewalk and the car suddenly breaks in the last metres, it might suggest issues in the perception system. Overall, investigating the reason behind great differences might deliver valuable insights about the health of the whole system.
2. **Issues in the perception system**: both ADAS and AV need to be able to detect and classify tracked objects consistently, which means that, once an object is tracked by the tracking algorithm, the detected *class* shouldn't change over time. The reasons are multiple but generally speaking, a perception system unable to classify objects consistently and correctly over time might lead to wrong decisions on a planning level. Even if the specific examples strictly depend on the specifics of the *planning* software, it is possible that misidentifying a pedestrian with a vehicle could result in harsh braking. Similarly, classifying a bicycle as a pedestrian could cause wrong speed and distance estimations, increasing the risk in traffic scenarios.

Explanation of the algorithms in the software

The algorithm is implemented using Object-Oriented Programming principles to facilitate both its extensibility and to check that independent features are working as expected with specific unit-tests.
The algorithm contains *3* methods that are responsible to process the input file and return the results within a *json* file at a given path.
The 3 methods are *run*, *diff_location* and *tracked_object_type_consistency*.

- ***run*:** this method iterates over the whole *json* file frame by frame and internally invokes the other two methods listed above. Once the iteration is completed, the results are stored in a *json* file which is stored in a path specified by the user.
- ***diff_location:*** this method is responsible to detect anomalies within the localisation data. For each frame, it controls that the gap between the coordinates at the current frame and the coordinates estimated from the previous frame is below a given threshold (decided by the user). For example, if we wanted to filter all the timesteps where the difference between the expected coordinates and the real coordinates exceeds *1* metre, we should set the *localisation_max_diff* class argument to *1*. Notice that the estimation of the coordinates of the next frame is computed using trigonometry: since *distance = speed * time*, the movement across the *x* and *y* axis can be estimated with the cosine and sine of the yaw. Subsequently, by summing

together the initial coordinates and the expected movement across each axis, we estimate the future position of the ego vehicle.

- **tracked_object_type_consistency:** controls that the class (i.e. pedestrian, car, etc.) of tracked objects is consistent over time. In order to perform this comparison, the method internally uses a hash-map that is updated over time to record objects (*track ids*) currently tracked, delete lost objects and add newly detected objects.
  To clarify how this methods works imagine that at a given frame the perception system detects and tracks *two* objects:
  1. object *0* which is a *pedestrian*
  2. object *1* which is a *car*.

  In the next scene the perception system detects 3 objects:
  1. object *0* which is a *pedestrian*.
  2. object *1* which is a *bicycle*.
  3. object *2* which is a *traffic light*.

  After processing the first frame, the hash-map assigns *track id 1* to a car. However, in the following frame, the object class associated with *track id 1* is classified as a bicycle and this anomaly will be recorded in the output file for further analysis.
  Another great feature of this algorithm is that it can handle object occlusion: the user can specify the maximum number of consecutive frames that an unseen tracked object is held in the hash-map before being deleted.

  Notice that this algorithm currently works if the perception data is registered using modern tracking algorithms where track_id reassignment is delayed. For example, if object 0 would be lost at a given frame, it is expected that the track_id 0 will not be reassigned in the right next frame.

Trade-off analysis on the functionality of the software and possible improvements

Currently the analysis performed on the localisation data is already using all the provided information (the coordinates, the speed and the yaw of the ego vehicle) and therefore this feature can be considered as completed.
On the other hand, the algorithm responsible for detecting anomalies within the perception data is taking into account only the *track id* and the *object type* and the remaining information of detected objects (coordinates, speed, yaw, dimensions) could be used to make the analysis more robust. For example, in order to perform the analysis on data recorded with an old tracking system (that cannot handle object occlusion and automatically reassign a given track id in the very next frame), we should take into account the coordinates in order to avoid consider anomalies tracking ids that have just been reassigned. This kind of limit however is not severe since modern tracking algorithms such as Deep Sort, can handle object occlusion and therefore *tracking ids* are set free only after a specified number of frames since the object was last detected. Another potential improvement of the *tracked_object_type_consistency* algorithm would be to filter the analysis to objects that lay within a certain distance with respect to the ego vehicle and this distance should be set by the user. The reason is that the misclassification of an object that is very far from the ego

vehicle is much less serious compared to a misclassification of an object that is close to the ego vehicle.

Potential future extensions

This software, which is already able to deliver meaningful insights, could be extended with many additional features that could turn it into a real product.
In fact, the data provided with this assignment could be used both to expand the analysis of the perception data and also to run a crossed analysis between localisation data and perception data.

1. ***perception features***: Besides the analysis on the object class consistency already integrated in the software, several others could be added. For example, it could be possible to analyse for each tracked object, the distance between its expected future coordinates and the actual ones by using the same trigonometry formula already adopted by the localisation algorithm. Secondly, it could be interesting to estimate the consistency of the width and height of the detected objects in relation to the distance between them and the ego vehicle.
2. ***planning system***: In this case, two analyses could be carried out. The first analysis could be delivered within a short time and it would consist of storing all the frames where the distance between the ego's vehicle and the detected objects is below a certain security threshold. This analysis would give insights of how the AV is able to distantiate itself safely from external objects. The second analysis would be more ambitious and would aim to investigate how the ego vehicle reacts to the perception data by running a trajectory analysis of both the ego vehicle and the detected objects. This way we could assess, for example, how safely and responsively the vehicle reacts to unexpected trajectory shifts. Despite this proposal might seem quite ambitious, it could be performed by modifying the already implemented data-structure (*hash-map*) to include information on the coordinates, speed and yaw of the detected objects.

How I spent 7 hours of development

I started by taking some time to understand what kind of design was more suitable for the task and I analysed whether functional programming was enough or the problem required an Object Oriented Design approach. After identifying OOD as the most suitable practice, I spent most of the time on developing the algorithms (4-5 hours) and designing the unit-tests necessary to evaluate the correctness of the algorithms (1-2 hours).
Finally, I spent around extra 2 hours to complete the following activities:

- Wrapping the repository inside a python package.
- Writing the Dockerfile and checking that containers were working correctly.
- Writing documentation.