Finding the greenest track

From scripts to packages, collaboratively

Please read this assignment carefully.

This coursework is concerned with a data analysis pipeline for finding different types of tracks between two points in a map and apply a method to characterise their similarities.

This assignment asks you to work collaboratively within your team to create a package. For that you will need to write some code for querying, loading and analysing a dataset about different tracks in a map (by "track" in this document we mean a trajectory connecting two points). We will describe how the code must behave, but it is up to you to fill in the implementation. The package needs to follow all the good practices learnt in the course, that is, the package should: be **version controlled**; include **tests**; provide **documentation and doctests**; set up a **command line interface**; and be **installable**. Besides this, you will also need to modify an existing implementation of a provided script to make it **more readable, more efficient, and measure its performance**.

The collaboration aspect should be organised and managed using GitHub.

The exercise will be semi-automatically marked, so it is very important that your solution adheres to the correct interface, file and folder name convention and structure, as defined in the rubric below. An otherwise valid solution that doesn't work with our marking tool will not be given credit.

For this assignment, you can use the Python standard library and other libraries you may wish to use (but make sure they are clearly set as dependencies when installing your package). Your code should work with Python 3.8.

First, we set out the problem we are solving. Next, we specify the target for your solution in detail. Finally, to assist you in creating a good solution, we state the marking scheme we will use.

1 Background information

Several people in your lab are involved in a project to study a number of locations on a map. However, the topography changes every day, and each of you have to visit different locations to take some ectoplasm samples. A different research group has created a web service that provides multiple different tracks that go from one point to another on these quickly changing lands. However, that tool only gives you the tracks you can follow, but not any indication about which one you should use. This is something that's making the life of many researchers very difficult. Wouldn't it be great to provide some kind of recommendation to our colleagues? Wouldn't it be even better to give them the suggestion based on the most environmentally-friendly of them all? Or the fastest? or the shortest? What if you want to explore similar paths?

You and your group members agree this is a great tool to offer to the community and have decided to put all your brains together to come up with a simple to use python library to analyse and suggest which track to follow for your trip.

What do we know? What do we have? What do we want?

1. The research location changes every day, but we know their surface is always the same, $90\,000\,km^2$, in a 300×300 grid.

2. There's a webservice that generates tracks between two given points for the map of each day. The output is a JSON file.
3. We want to create a command line tool that gives us the coordinates, the travel time and the CO_2 emissions for the greenest trip.
4. We want also to offer a python library that enables users to analyse these tracks.
5. We have a script from a post-doc of our group that implements the so-called k-means algorithm for clustering data points. We want to include it in our library too! It will help people to choose similar tracks based in the parameters.
6. Since we are interested on making trips more efficient, we are also interested on how to make our code more efficient. In particular the k-means algorithm.
7. We want this tool to be used by any researcher, so it needs to be easy to install and use. This includes having good documentation about how to use it and how to acknowledge it in the publications they benefit from it.

Let's look at what we've got access to already:

1.1 The road-tracks webapp

The webapp is located at: https://ucl-rse-with-python.herokuapp.com/road-tracks/ and their main page provides some information about how to query this service.

The result is provided as a JSON file with the properties of the tracks found. The tracks found each time will be always different, but the underlying map is constant during the day.

The tracks are provided using a simplified version of a chaincode. A chaincode is a compressed way to represent data in image processing. Here's a simple implementation of how to create it based on a given set of coordinates.

```
def route_cc(route):
    '''
    Converts consecutive 2D coordinates from route into a Freeman chain code
          2
          |
     3  - C -  1
          |
          4

    Parameters
    ----------
    route : list of tuple of ints
        A route of coordinates at step one.

    Returns
    -------
    tuple
        Origin of the route
    str
        A simplified Freeman chaincode

    Examples
    --------

    >>> route_cc([(0, 0), (0, 1), (0, 2), (1, 2), (1, 1), (0, 1)])
    ((0, 0), '22143')

    '''
    start = route[0][:2]
    cc = []
    freeman_cc2coord = {1: (1, 0),
```

```
                    2: (0, 1),
                    3: (-1, 0),
                    4: (0, -1),}
    freeman_coord2cc = {val: key for key, val in freeman_cc2coord.items()}
    for b, a in zip(route[1:], route):
        x_step = b[0] - a[0]
        y_step = b[1] - a[1]
        cc.append(str(freeman_coord2cc[(x_step, y_step)]))
    return start, ''.join(cc)
```

For each track the webapp also provides three more parameters: the road type, the current characteristics of the terrain and the elevation for each point.

Road type and terrain properties are provided as a list of characters as the chain code (abbreviated as shown below). Note, that if a track has N points, the chaincode is formed of the starting coordinate and $N-1$ steps. The road types and terrain properties are also $N-1$ elements as they show what connects these points. The elevation, however, is given for each point (therefore, N points).

The units for the step size on the plane and the elevation are given as metadata in the JSON output. A sample file (`short_tracks.json`) is provided with 10 different short routes.

1.1.1 Road properties and their effect

We need to make some assumptions for our first version of our package. We are ignoring many factors that normally play an important role in measuring efficiency on a car. For example, we are ignoring the effects of any extra weight our equipment puts on the car, the aerodynamic changes produced by carrying extra hauling cargo on the roof, the weather effects (dry/wet conditions, heat/cold temperatures, wind direction), use of car accessories like the air conditioning, or any traffic conditions (idling time, change of speeds, curves, …)

For this first version we will only account for three properties that are constant. The road type, the road surface or terrain and the slope (elevation change).

Also, we are only considering the properties for our group car, the ecto-π, a Fiat Tipo. In normal conditions (*i.e.*, factor 1), the car consumes 5.4 litres per 100 kilometres.

1.1.1.1 Road type There are three types of road, for our purpose they affect the speed at which we can go through them. Our car's consumption (and CO2 emission) varies depending on the speed. This table summarise these properties:

road type	abbreviation	average speed (km/h)	consumption factor
residential	r	30	1.40
local	l	80	1
motorway	m	120	1.25

1.1.1.2 Terrain We've got three types of terrain that affect the efficiency of our car. They are summarised below:

terrain type	abbreviation	consumption factor
dirt	d	2.5
gravel	g	1.25
paved	p	1

You would never expect to find gravel in a motorway, right? Not in this world!

1.1.1.3 Slope The slope is measured as the ratio between the vertical increment and the horizontal distance travelled. To obtain the percentage we multiply that by 100.

slope (%)	consumption factor
-8	0.16
-4	0.45
0	1
4	1.3
8	2.35
12	2.90

In this first version, we are going to approximate each slope by the closest of the values provided in the table above. i.e.,for slope in ranges [-2, 2] we will assume it is 0, for (2, 6] it's 4, for (6, 10] it's 8, and for larger than 10 it's 12. Similarly for the negative side.

1.1.1.4 Combining all factors together To combine all the factors together, they need to be multiplied. Knowing that a litre of diesel will produce $2.6391\,kg$ of carbon dioxide, then we can calculate our total emissions.

For example, if we drive for $1\,km$ in a flat local paved road, then we will have consumed 0.054 l and emitted 0.1425 kg of CO2 in 45 seconds.

Because, this is a **paved** ($f_{terrain} = 1$) road, it's **flat** (*i.e.,* its slope is 0; $f_{slope} = 1$), and is a **local** road where we drive at 80 km/h ($f_{road} = 1$), we travel 1 km in 45 seconds

$t = \frac{1\,km}{80\,km/h} = \frac{6}{8}\,min = 45\,s$

and the total consumption is calculated as

$5.4\frac{l}{100\,km} \cdot f_{slope} \cdot f_{road} \cdot f_{terrain} \cdot d.$

This gives us $0.054\,l$ which we can convert into $CO_2 = 0.054 \cdot 2.6391\,kg/l = 0.1425\,kg$.

However, if we go downhill (from 466 to 416 m; -5%) in a residential dirt road for one kilometer on the map we will consume 0.085 l and emitted 0.2243 kg of CO2 in 2 minutes.

In this case, as the slope is not 0, the distance travelled is a bit more:

$d = \sqrt{\Delta h^2 + \Delta x^2}$, i.e., $d = 1\,001.2492\,m$.

It's a **residential** road where we drive at 30 km/h ($f_{road} = 1.40$), and the terrain is classified as **dirt** ($f_{terrain} = 2.5$). The calculation of the CO2 emitted is as:

$5.4\frac{l}{100\,km} \cdot f_{slope} \cdot f_{road} \cdot f_{terrain} \cdot d,$

therefore:

$5.4\frac{l}{100\,km} \cdot 0.45 \cdot 1.40 \cdot 2.5 \cdot 1\,001.2492\,m = 0.0851\,l$

which is then converted into $CO_2 = 0.0851 \cdot 2.6391\,kg/l = 0.2247\,kg$.

Note: though the numbers above have been truncated to the 4th decimal place, you should not do so when calculating the consumption. this was only done here in order to keep the explanation readable.

1.2 The k-means algorithm

This is a brief description of how the algorithm given by our post-doc works, but note that you do not need to understand it in detail: the focus of the exercise is on how the code and data are structured. Importantly, **you are not expected to (in fact, should not) make changes to the algorithm itself!**

The goal of the algorithm is to separate data points into groups. In the original version we got (`clustering.py`), each point is represented as a tuple of its coordinates. The algorithm then proceeds to form three clusters of nearby points by following these steps:

1. Pick three points randomly to be the initial centres of the clusters (line 10).
2. Assign each data point to a cluster. This is done by computing its distance from all cluster centres, and assign it to the nearest centre (lines 18-21).
3. Update the centre of each cluster by setting it to the average of all points assigned to the cluster (lines 23-25).
4. Repeat steps 2-3 for the desired number of times.

In the end, each cluster will (ideally) contain points that are close to each other, and far from the other clusters. The code then prints some basic statistics about the resulting clusters.

Note: You may find it useful or interesting to visualise the output of the algorithm, that is, how the points are clustered. To do that, you can use this or a similar piece of code at the end of the code provided:

```
from matplotlib import pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

for i in range(3):
 alloc_ps = [p for j, p in enumerate(ps) if alloc[j]==i]
 ax.scatter([a[0] for a in alloc_ps],[a[1] for a in alloc_ps],[a[2] for a in alloc_ps])

plt.show()
```

This will plot the three clusters in different colours. Naturally, you will need to adjust that code as you make changes to the clustering code if you want it to keep working!. This is purely for your own benefit – **the version you submit should not perform any plotting**.

(An extra note for those interested: this version of the code is non-deterministic; that is, every time you run it, you may get different results. If you want to remove this randomness – for example, to test that it still behaves as expected as you make changes –, you can think about setting the seed of the random generator in the beginning of the file. Note that this is in no way required for this exercise, and in fact, would change the behaviour of the code, so **don't include changes like this** in your final submission!)

2 Your tasks

2.1 Team work

This is a collaborative effort. It's up to you how to distribute the work between the team. Read more at How to work and Marking sections below to understand how this part would be evaluated.

2.2 Interfaces and packaging

Your final product should be in the form of a Python package that other users can install. This should include documentation about the package and its contents.

2.2.1 Packaging

The final code should be in the form of a **package** called `tracknaliser`. Someone should be able to install it (and its dependencies) by navigating to the directory containing the package code and running

```
pip install .
```

Note: Do **NOT** upload your package to PyPI or any other public package repository!

2.2.2 Command line interface

The package should expose a **command-line interface** through the `greentrack` command. The user should be able to specify the details of the trip they plan to do and request a simple or verbose output:

2.2.2.1 Command line arguments: The program should query for 50 tracks to the service and only accept values that are in a 300x300 grid. The user should be able to input the properties using this syntax:

```
greentrack --start  <x coord> <y coord> --end <x coord> <y coord>
           [--verbose]
```

The acceptable inputs for `--start` and `--end` are `x` and `y` coordinates for the start and end of the travel planned. The program should return three properties for the greenest path: - the coordinates pairs (x, y) of the path, i.e., the start, end and corners of the trip, - the total CO2 emission in kg, and - the total time of travel (as `HH:MM:SS`).

If the `--verbose` flag is passed, then that indicates that the program should report the journey in a human readable way, as shown in the example below.

The command line interface should: - check the validity of the inputs before making the query (i.e, coordinates should be in the range from 0 up to 300) and produce helpful messages if they are invalid. - produce a meaningful error message if there's no internet connection or the webapp is inaccessible

2.2.2.2 Example Simple:

```
$ greentrack --start 12 15 --end 25 46
Path: (12, 15), (12, 46), (25, 46)
CO2: 2.31 kg
Time: 0:23:00
```

Verbose:

```
$ greentrack --start 12 15 --end 25 46 --verbose
Path:
- Start from (12, 15)
- Go north for 31 km, turn right at (12, 46)
- Go east for 13 km,
- reach your estination at (25, 46)
CO2: 2.31 kg
Time: 0:23:00
```

2.2.3 Library-style interface

Once the package is installed, it should expose a `load_tracksfile` and `query_tracks` functions which return a `Tracks` object.

The `load_tracksfile` function takes only a `Path` to a JSON filename. The function needs to check the validity of the path (i.e., that it exists) and that the file follows the structure expected (see sample json provided).

The `query_tracks` function takes as many arguments as needed to query the webapp and an optional `save` argument that if set to `True` will save the obtained data as a JSON file with the following pattern `tracks_{date}_{n_tracks}_{start}_{end}.json`. For example,

```
from tracknaliser import query_tracks

tracks = query_tracks(start=(12, 15), end=(25, 46),
                      min_steps_straight=1, max_steps_straight=40,
                      n_tracks=30, save=True)
```

will create a tracks object and save this file in the current directory:

`tracks_20211218T150212_30_12_15_25_46.json`.

Where the `date` is obtained in the query result but the `ntracks` should be using the one from the argument, not the number of tracks obtained.

The defaults for `query_tracks` should be the same than the ones offered by the webapp.

A `Tracks` object should provide the following attributes and methods:

- `len(tracks)` should return how many tracks are in the object.
- `tracks.greenest()`, `tracks.fastest()` and `tracks.shortest()` should return a `SingleTrack` object for the greenest (less CO2 emission), fastest and shortest tracks respectively.
- `tracks.kmeans()` should return as many lists as clusters required, each containing the indices of the tracks for each group. It may accept the `clusters` and `iterations` arguments, using 3 and 10 respectively by default if they are not provided.
- `tracks.get_track(x)` will return a `SingleTrack` object for the track number `x` (where `x` is smaller than `len(tracks)`).
- `tracks.start`, `tracks.end`, `tracks.map_size` and `tracks.date` should return the start and end point of the tracks, the size of the map for that set and the date as a `DateTime` object of when the query was run.
- `print(tracks)` should produce an output such as

```
>>> print(tracks)
<Tracks: {n_tracks} from (x0,y0) to (x1,y1)>
```

  where `n_tracks` should show the number of tracks in the class and, `(x0,y0)` and `(x1,y1)` should show the start and end points.

A `SingleTrack` object should provide the following attributes and methods:

- `len(a_track)` should return how many coordinates are in that track, including the starting and end point (i.e., N)
- `a_track.corners()` will return a list with the computed `(x,y)` coordinates of only the corners along the track, including starting and end points.
- `a_track.visualise()` should visualise/save a graph with a distance vs elevation plot on the left and the coordinates of the path followed on the right (as shown in the figure below). By default, the `visualise()` method should show the graph on the screen. If `show=False` is passed, it should not show it on the screen, and instead save it on disk as `track.png`. If a `filename="my_track.png"` is passed, then that filename should be used.
- `a_track.co2()`, `a_track.distance()` and `a_track.time()` should be methods that calculate and return the emission (CO2 kg), length (km) and time (hours) for that track. (values should not be rounded).
- `a_track.start`, `a_track.cc`, `a_track.road`, `a_track.terrain` and `a_track.elevation` should return the starting `(x,y)` point, the chaincode, the road type and the terrain as a string, and the elevation as a list of heights (all these properties as given by the webapp).
- As with the `Tracks`, `print(a_track)` should produce an output such as

```
>>> print(a_track)
<SingleTrack: starts at (x0,y0) - {steps} steps>
```

  where `steps` indicates the number of steps for that track, and `(x0,y0)` shows the coordinates of the start point.

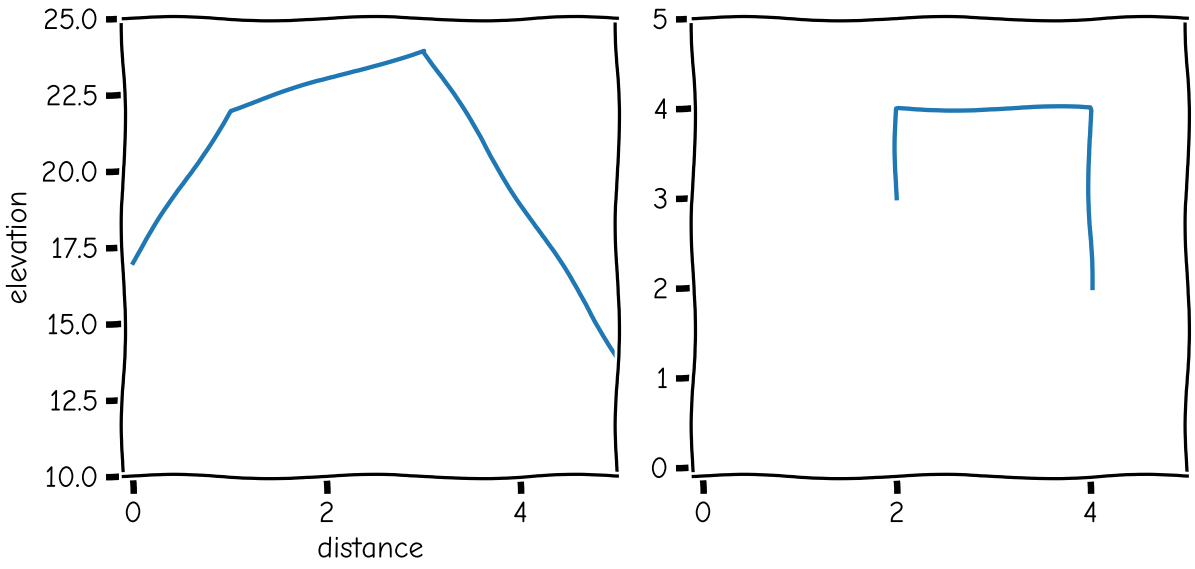

Figure 1: Example of the `visualise` method

2.2.4 Documentation

The code should contain enough information that will explain to users what it does, how to run it, and any other important details. This information will come in a variety of formats.

Firstly, the code should have **docstrings** that explain, for example, what functions do and what arguments they take. The docstrings should be in the numpy format. You should also use **comments** to clarify any particular points in the code that you feel require more explanation. The package you create should contain any **metadata files** that you find appropriate (as also discussed in the course notes).

Finally, the submission should include the sources to generate documentation pages using the **Sphinx** framework. These should be in a directory named `docs`. We will run the following commands to build and check your documentation:

```
cd repository/docs
make html
python -m http.server -d build/html 8080
```

(after the above, your documentation should be viewable in a browser at http://localhost:8080)

2.3 Validation and robustness

The new code should include checks on the validity of the inputs, and **raise appropriate errors** if the users give inputs that don't make sense. At a minimum, the following situations should cause an error:

- a user tries to query for one or multiple negative coordinates
- a user tries to load a JSON file with the wrong schema
- a user tries to make a query when they don't have a working internet connection
- a user tries to load a JSON file where some tracks have illegal values on the chaincodes or road properties.

As you create the code, we want you to add checks that ensure that it behaves correctly. This will be primarily achieved by **unit tests** in the `pytest` framework. The final code should include:

- tests for converting the `chaincode` to corner coordinates;

- tests for any of the functions and public methods listed above;
- negative tests, where that makes sense;
- tests mocking services that require internet connection;
- usage examples in documentation strings that can be run using `doctest`.

You should also set up these tests to run automatically when you push to GitHub or open a pull request, using a **Continuous Integration** platform. You can use GitHub Actions for this. However, keep in mind that there's a finite number of credits, approximately 100 min per group and per month, so use them wisely - see how many more credits testing on Mac costs compared with Linux. If you abuse the system you will be affecting other groups and you may be penalised for that (We will be monitoring the usage so we can warn you before it's too late).

Also think about what other measures you can take that help you check that the code does what is expected and handles user input sensibly.

2.4 Incorporating kmeans

2.4.1 Refactoring kmeans for readability

The kmeans that we have provided you with is not very clearly written. There are various changes that would make it more readable, as discussed in class and in the notes. We want you to pick **five** of those changes (or more if you feel like it!) and apply them to the code. Each change should be accompanied by a commit with a relevant message. It is fine if a change is split over multiple commits, but do not mix different types of changes in the same commit.

To keep easier track of these changes create an issue on your given repository with a checklist with a meaningful explanation of the changes you intent to do. Each time you apply one of these changes to the whole file, commit and add the commit number(s) to the item of the list that refers to.

For example, you can create a checklist in markdown as:

```
- [x] Modify something to make it readable; a9d018253655b3eeb5107b6cd0576ea44c5e5b8b
- [ ] Change this unknown thing to something known;
```

Note that you can edit an issue as many times you need to by clicking the three dots button on the top right corner of the issue comment.

For the submission, take a screenshot of that issue and name it `refactoring_steps.png`.

Do not introduce external libraries like `numpy` at this stage; focus on readability over performance. Remember not to change the algorithm itself, although you are free to improve, for instance, how the data is read, as long as the input and output remain the same.

As part of these changes, we want to make sure that the code is easy to run. Specifically, we want to be able to call the code in the file in two ways. Firstly, the file should include a function called `cluster`, which should take a list of points (as tuples) and, optionally, a number of iterations, and perform the analysis. Secondly, it should have a command-line interface (created using `argparse`) so that a user can call it by specifying a file and, optionally, a number of iterations, like this:

```
python clustering.py samples.csv [--iters 20]
```

In either case, if the number of iterations is not specified, it should be set to 10. Creating these interfaces does not count as one of the five changes you should make.

Tip: Read through the code first before making any changes. Pick one possible improvement, apply it, then commit your changes before starting on another type of improvement. This will help ensure that your code still works as you change things, and keep your history clearer.

Note: Do not set any precision when returning or printing the centres obtained.

2.5 Using NumPy

The initial code is written in plain Python, without using external libraries. One way of improving its performance is using a dedicated library for numerical computations, and this is what this part is concerned with.

Create a new file called `clustering_numpy.py`. Starting from the plain Python version (either the version we gave you or your cleaned up version), introduce `numpy` structures and functions to make the code more concise and (hopefully!) faster. Your final version of the file should use `numpy` as much as possible, avoiding, for example, Python lists or `for` loops (where possible).

The new file should also expose the same interface as before: a command-line interface, and a `cluster` function. The user should be able to run it like this:

```
python clustering_numpy.py my_samples.csv [--iters 20]
```

2.6 Integration with tracknaliser

One of both versions (either with and without numpy) should be available to be called from the `Tracks` class via the `kmeans()` method. Note that the output of that function should not return the canters, but a lists with the track indices that belong to each group.

2.7 Measuring performance

We would like to see how the two versions of the code (with and without `numpy`) compare in terms of performance, especially as the input grows in size. For this part, create a script `performance.py` that runs the two versions on different input files which contain an increasing number of points, ranging from 100 to 10,000. Plot the time required against the size of the input (number of points), using a single plot for both versions. Include the script and the plot in your final submission, in a file called `performance.png`. Remember to label your axes and lines/points clearly, and make sure you commit the final plot to the repository!

(The execution time also depends on how many times we run the main loop of the algorithm. For this analysis, keep the number of iterations of the assign/recentre loop fixed to 10, as in the original version.)

Tip: You may find it useful to write a script that calls your code for the different input files, so that you can run the whole analysis with a single command. Some versions of the code may take a long time to run, so you may want to include some progress monitoring, such as printing a message when a file has been processed. In any case, make sure to not include this or other input/output when measuring time!

3 How to work

3.1 Collaboration

You will work in groups of 4-5 people to accomplish the tasks above. How you split the work within the group is entirely up to you. You may want to assign one aspect of the work (*e.g.*, tests) to each person, and have them be responsible for it throughout the project. Alternatively, you can decide to split the total work into smaller units ("sprints"), and within each of those allocate some of the smaller tasks to each person. Or you can come up with a different scheme!

Similarly, how you communicate is up to you. You can use some of the tools and practices we have mentioned in class (such as issue tracking), over whatever platform is convenient.

We will ask each team to meet with one instructor approximately halfway through the exercise, to see how the collaboration is going.

3.2 Suggestions for successful team work

1. Introduce yourself, either from the start or as opportunities arise share your strengths, weakness, and your values (what's important for you and why)
2. Define a set of policy/rules about how to interact and what's expected and what's unacceptable from the group. You can adopt a code of conduct (contributor covenant, Carpentries, Python Software Foundation's, …)
3. Define roles for activities. These roles could be for the duration of the project, for a fragment of it, or changed daily. Some roles could be easier to transfer from day to day, for example in each of your meetings you could have a facilitator, gatekeeper, timekeeper and a note-taker and that won't disrupt the evolution of the project, whereas other roles, like a lead-developer, may need some global knowledge or skills that may not be easily and quickly transferable to change them with a high frequency.
4. Decide the set of tools to use to keep the whole team in the same page. Remember, the power is not in the tool you choose, but how effectively you use them (is it the right tool for the task?).
    1. Be aware of what is needed to run that tool (do you need to create a new account? Is it accessible for everyone? Are there some concerns such as privacy, political or moral of using that tool?)
    2. Spend time explaining how to effectively use that tool to everyone in the team in case it is new for them or they are unaware of certain features. Provide some resources for future references.
    3. Keep discussions (and most importantly decisions!) accessible to everyone. If possible use a common place to record decisions and track tasks. For example, having to scroll up and down through an endless chat or forum to find who is doing what is not very efficient.
5. Establish a methodology for reviewing and collaborating on the code that your team will produce (branch naming convention, branching strategy, who merges and when).
6. Communicate, communicate and communicate… and be careful with assuming that you or others have understood what has been said! Express what you've understood to get confirmation that your understanding is correct.

3.3 Version control

You are expected to use `git` throughout the project, and work on the GitHub repository that we will provide you with.

You should use the GitHub issue tracker to record planned work and issues that come up. Changes to the code should be made through pull requests rather than committing directly to the main branch. Make sure that pull requests are only merged after being reviewed and approved first. In your submission, include files that evidence your use of issues and pull requests. Specifically:

- `issues.png`: a screenshot of open and closed issues in your repository;
- `pr.png`: a screenshot of open and closed pull requests in your repository;
- `pr_link.txt`: a text file containing the URL of a pull request that you consider representative of your work.

To get a list of open and closed issues on the same page, go to the issues page of your repository and filter with only: `is:issue` (similarly use `is:pr` on the pull requests page).

3.3.1 Where is the repository for my group?

You should have received an invitation to join a GitHub repository named `tracknaliser-Working-Group-X` where `X` is your group number, under the UCL-MPHY0021-21-22 organisation. The repository is initially empty. If you don't see the invitation (check your emails or GitHub notifications), or it has expired, e-mail the teaching team (rits-teaching@ucl.ac.uk) with your GitHub username. You can also find your invite by going to: `https://github.com/UCL-MPHY0021-21-22/tracknaliser-Working-Group-X` changing `X` for your group number.

3.3.2 Can we change the settings of our repository?

By default, you have not enough permissions to alter the settings of your repository. This is to avoid the possibility that, for example, someone deletes it. A member of a group can be provided with increased ("admin") permissions to have full access to the repository settings. Agree between your groups who should have these elevated permissions and email the teaching team, including in CC all the other members of your group. Tell us who you want to be given admin access and their github username.

Alternatively, if you want a setting changed, email us and we can make the change for you.

3.4 Deliverables

You must submit your exercise solution to Moodle as a single uploaded gzip format archive. (You must use only the tar.gz, not any other archiver, such as .zip or .rar. If we cannot extract the files from the submitted file with gzip, you will receive zero marks.)

To create a tar.gz file you need to run the following command on a bash terminal:

```
tar zcf filename.tar.gz directoryName
```

For example, if you group is working group 31, then you'll create your file as:

```
tar zcf working_group_31.tar.gz working_group_31
```

The folder structure inside your tar.gz archive must have a single top-level folder, whose folder name is your team name (*e.g.*, `working_group_31`), so that on running

```
tar zxf working_group_31.tar.gz
```

this folder appears. This top level folder must contain all the parts of your solution. Specifically, it should include a directory named `repository` with the following:

- the final code for the package, its tests and documentation sources
- the `performance.py` script and two performance plots, as described above, within a `benchmark` directory.

Note: We will only mark the `main` branch of the repository you submit!

In summary, your directory structure as extracted from the `working_group_xx.tar.gz` file should look like this:

```
working_group_xx/
├── repository/
│   ├── .git/
│   ├── <files and directories for the package, tests and documentation>
│   ├── benchmark/
│   │   ├── performance.py
│   │   └── performance.png
│   └── <other files you think are required>
├── refactoring_steps.png
├── issues.png
├── pr.png
└── pr_link.txt
```

Because this is a group assignment, only one member of the team needs to submit the the exercise solution to Moodle. Make sure you agree on who submits!

4 Marking

4.1 IPAC

Your submitted assignment will be marked as a single project for the whole group. As part of your final submission, you will also be required to assess the rest of your team. These two factors (group mark and peer evaluation) will determine your personal grade, using the IPAC methodology (Individual Peer Assessment of Contribution to group work).

We will ask you to evaluate your group members (and yourself) on the following criteria:

- communicating and sharing knowledge
- good team-working skills (such as respect, listening, leadership)
- quality of research and application of skills
- time and effort contributed
- overall value to the team's success

Note that the purpose of the scheme is not to set students against each other. Due to how IPAC works, falsely claiming that you have done most of the work and giving poor evaluations to your fellow group member is unlikely to artificially raise your own grade.

Once the group work is submitted, and soon after the submission deadline, the IPAC will be available on Moodle. This will need to be submitted individually by each member of the team. You'll have a week to fill it up.

4.2 Marking scheme

Total: 100 marks

- **Packaging and interfaces (20%)**
  - Installable package (2 marks)
  - Appropriate metadata (including version number and other properties) (1 mark)
  - Which packages code (but not tests), correctly. (1 mark)
  - Which specifies library dependencies (1 mark)
  - Which points to the entry point function (1 mark)
  - Which allows to import `load_tracksfile` and `query_tracks` from the library (1 mark)
  - Contains three other metadata files. Hint: How to use the package, how to reference it, who can copy it (3 marks)
  - Command-line interface correctly installed from the package (1 mark)
  - Command-line interface correctly passes the arguments (1 mark)
  - Command-line interface provides the different output when used with or without `--verbose`. (3 marks)
  - Sphinx documentation (builds and provides appropriate documentation for all the functions and classes (5 marks)
- **Code Structure, Style and Functionality (25%)**
  - Code readability (6 marks)
  - Clear code and file structure for all the functions and classes definitions (4 marks)
  - Avoids repetition through out all the package (3 marks)
  - Accepts passing the arguments required for each of the requirements asked (6 marks)
  - Objects are represented with the correct information (4 marks)
  - File and images produced are properly named and annotated (e.g., file names, axis labels, structure) (2 marks)
- **Validation and robustness (20%)**
  - Appropriate input checks meaningful error messages (3 marks)

  - Meaningful error messages for wrong inputs or external problems (e.g., no-internet connection) (3 marks)
  - Unit tests (possitive and/or negative) for at least 12 functions or methods (6 marks)
  - Mock tests (4 marks)
  - Doctests (2 marks)
  - Continuous integration (2 marks)

- **Refactoring and Performance (25%)**
  - At least 5 changes to the `clustering.py` provided code keeping its functionality (5 marks)
  - Exposing cluster function as described and provide a command line interface (4 marks)
  - Conversion to NumPy (works correctly, uses arrays, avoids loops, …) (8 marks)
  - Performance plots and script (8 marks)

- **Ways of working (10%)**
  - Git commits of reasonable size with meaningful messages (4 marks)
  - Consistent use of issues and pull requests (6 marks)

Appendix: Additional information about the data

The data in this assignment is based on multiple sources:

- Speed and fuel efficiency from mpgforspeed and fueleconomy;
- Terrain and its effect on consumption from leisruewheels and people experiences - (reedit);
- Slope factors on consumption is based on the Effective Slope for Fuel Consumption Calculation US Patent Application 20120109512; and
- CO2 emissions by type of car is provided by comcar.co.uk.