Improving the geology group's code

From scripts to packages, collaboratively

This assignment asks you to work in a team on an existing piece of code, making improvements and adding features to it.

1 Background

Several people in your lab are involved in a project focused on comparing geological samples from different areas. A former postdoc has written some code to automate the analysis and reporting of results.

The data is stored in CSV files, each holding samples from a single location. Each sample is represented by a set of numerical features, all in the same row. In other words, a file containing N rows with M columns represents N samples with M features each.

The existing analysis compares the same number of samples from two locations. The samples are stored in order in the files, so that the first sample from location 1 is compared to the first sample from location 2, and so on. The comparison takes all features into account, but with different weights. For two samples with features $s_1, s_2, \ldots, s_M$ and $s'_1, s'_2, \ldots, s'_M$, if the weight of feature i is w_i, then the comparison algorithm computes a **sample difference**

$$d = \sum_{i=1}^{M} w_i \left| s_i - s'_i \right|$$

The above analysis method is called **Algorithm X**, and it is repeated for every pair of samples. The output is a vector (a list, in the current Python implementation) of differences, which can be summarised and reported in different ways.

One way of reporting the output is the **d-index**, which is the average of the vector across sample pairs. Another is the **criticality**, which is the number of elements in the vector that exceed a chosen threshold value.

The former postdoc has left two files implementing computational workflows involving Algorithm X. The files read the sample data and weights from CSV files, then report back either the d-index or the criticality.

The code is written in Python 3 and runs correctly. However, with the project growing, more people are starting to use it and require new features. It is up to you and your teammates to tidy up the code, extend it, and make it more robust and easy to use for the rest of your lab – and the rest of the geological community.

You can run the existing code with `python workflow1.py` (or `workflow2.py`). With the sample files provided, you should see the following messages printed for workflow 1:

```
criticality: 1 result above 5
```

and workflow 2:

```
d-index: 4.7
```

2 Your tasks

2.1 Refactoring

The code you have been given stands to be improved a lot. To decide among the many possible changes you can make, pay particular attentions to the following considerations:

- how easy the code is to read and understand, especially by someone new to it
- whether repetition can be eliminated
- where to place code so that it is easy to find and modify

You may want to think about **changes to both the structure and the style** of the code for this. However, the existing behaviour should be preserved, subject to the notes in the following sections, so make sure that your updates do not introduce changes by accident. The extensions described below may give you more ideas for how to better structure the code.

2.2 Adding features

2.2.1 Flexibility

At the moment, many assumptions are hardcoded into the scripts. You should **make the code more generic** to allow your colleagues to customise it according to their needs. Users should at least be able to specify:

- the names of the sample files
- the name of the file holding the weights
- the threshold used in the criticality calculation

See the following sections (especially Interfaces and packaging) for how users can specify these parameters.

2.2.2 Missing values

The original code assumed that all features should be taken into account, but sometimes a sample may be missing some features due to how the data was collected. This is recorded in the file as `nan` or `NaN`. Your colleauges would like you to change the code so that these **missing values are not considered at all** – *i.e.*, the code should treat that sample and its pair as if that feature didn't exist when computing their difference.

For example, assume that the files generally have 5 features, with weights $(1, 10, 0.5, 2, 2)$. Consider two matching samples which are each missing one feature, represented as $(1, \text{nan}, 2, 3, 2.6)$ and $(1.1, 3.5, 2, 5, \text{nan})$. To calculate the difference for these two samples, the missing features should be ignored. This gives a difference of

$$d = 1 \cdot |1 - 1.1| + 0.5 \cdot |2 - 2| + 2 \cdot |3 - 5| = 4.1$$

2.2.3 More analyses

It is becoming apparent that Algorithm X is not always a good choice for comparing the samples, and that other approaches may be more appropriate. Some colleagues prefer to use the following **Algorithm Y**:

- Assume two samples $s_1, s_2, \dots, s_M$ and $s'_1, s'_2, \dots, s'_M$, and weights $w_1, w_2, \dots, w_M$.
- Compute the discrepancy
$$d = \sqrt{\sum_{i=1}^{M} \left(w_i \left(s_i - s'_i \right)^2 \right)}$$
- Repeat the same for all pairs of samples and return the vector of discrepancies.

Adapt the code so that it also implements Algorithm Y, and allows users to choose which of the two approaches to use. The implementation should read the features and weights from CSV files, like in the original code.

This is only one of many possible alternatives to Algorithm X. It is likely that others will need to be implemented soon. A very good solution will allow other algorithms to be incorporated easily without requiring many changes to the existing code, or even allow users to implement their own analysis algorithm and "pass" it to this framework (when invoking it as a library – see below).

2.3 Improving performance

We want to see whether the code can be made to run faster. There are several approaches we can take, but for this exercise we will look at using `numpy`. **Use the `numpy` library** to rewrite as much of the code as you can.

We also want to **measure the performance** of the code for different inputs. Generate input files of varying sizes, and run the final version of your code on them. Make two plots with the results for these scenarios:

- number of samples ranging between 10–10,000, and 10 features
- 100 samples, and number of features ranging between 10–10,000

Make sure to label the plots appropriately and clearly, including the axes. The files should be saved as PNG images named `time_samples.png` and `time_features.png`, and included in your submission. You should also include the code that generates these results, in a file called `performance.py`.

Make sure that you only measure the time taken for analysing the data and computing the result, without including the time to read in files. Also take steps to report a more reliable result, and reduce the variability when measuring performance time.

2.4 Interfaces and packaging

The original code contains two scripts that can be executed, but we want the final version to be more useable. Furthermore, the final code should be in the form of a Python package that other users can install. This should include documentation about the package and its contents.

2.4.1 Packaging

The final code should be in the form of a **package** called `rocksamples`. Someone should be able to install it 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!

2.4.2 Command line interface

The package should expose a **command-line interface** through the `comparesamples` command. The user should be able to specify the details of the workflow they want to run in two ways:

2.4.2.1 Command line arguments: Passing the algorithm name and filenames when invoking the command using this syntax:

```
comparesamples <sample file 1> <sample file 2> [--summary <measure>]
               [--analysis <algorithm>] [--weights <weights file>]
```

The acceptable options for `--summary` are `d` and `criticality`, indicating that the program should report the d-index or the criticality, respectively. The acceptable options for `--analysis` are `x` and `y`, indicating Algorithms X and Y, respectively. The `analysis`, `summary` and `weights` arguments are optional. If not given, the program should default to Algorithm X, d-index, and weights of 1 for each feature, respectively. The sample and weights files can be located anywhere, and are identified through the path to them, such as:

```
comparesamples new_samples/file1.csv /Users/home/sam/backup/samples/file2.csv
               --summary d
```

2.4.2.2 YAML configuration: Specifying these parameters in a YAML file. The file should be structured as in the following example:

```yaml
workflow_name:
    algorithm: x
    summary: criticality
    samples:
    - new_samples/file1.csv
    - /Users/home/sam/backup/samples/file2.csv
    weights: my_weights.csv
```

Here, the main key (`workflow_name` in the example) can be any name you choose to give the workflow. The meaning and acceptable values of the parameters is the same as when passing them on the command line. In this case, the user only specifies the path to the YAML file using the `--config` option:

```
comparesamples --config my_config.yaml
```

(if other command line arguments such as filenames are passed, you should ignore them)

With this interface, the user can provide multiple workflows to be run at the same time. This is done by having multiple segments like the above, one after the other (with different workflow names). The output from each workflow should be in its own line.

2.4.3 Library-style interface

Once the package is installed, it should expose an **`analyse` function** which takes the following arguments:

- a list of sample data, with one element for each location (the samples from each location should be a list or array)
- the feature weights, as a list or array
- `analysis`: the analysis algorithm (`"x"` or `"y"`)
- `summary`: the summary measure (`"d"` or `"criticality"`)

The last two arguments are optional and their default values are the same as in the command line interface. The function should not print any text, but should return the requested measure as its output, as a numerical value. The function should be callable as in this example:

```python
from rocksamples import analyse

input_files = ['file1.csv', 'file2.csv']
input_data = []

# FIXME read data from input files and weights file
#       into input_data and weight_data

value = analyse(input_data, weight_data, analysis='x', summary='criticality')
```

2.4.4 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 <your_package/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)

2.5 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 compare two locations with different numbers of samples
- a user tries to compare two locations with different numbers of features
- the number of weights doesn't match the number of features
- a weight is negative
- a user specifies an invalid number of sample files (*i.e.*, not 2)
- a user specifies an unrecognised analysis method or summary measure

As you make changes to the code, we want you to add checks that ensure that it behaves correctly, for both the new and old functionality. This will be primarily achieved by **unit tests** for the `pytest` framework. The final code should include:

- tests for the original and new functionality (algorithms X and Y);
- negative tests, where that makes sense;
- 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 Travis CI for this, but you are free to choose another platform if you prefer. However, keep in mind that most platforms have a limit for how long you can use them for free, particularly for private repositories (UCL has a paid plan for Travis which enables you to use it for free).

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.

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).

3.4 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 zcvf filename.tar.gz directoryName
```

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_16`), so that on running

```
tar zxvf filename.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 for the package, tests and documentation>
│   ├── benchmark/
│   │   ├── performance.py
│   │   ├── times_samples.png
│   │   └── times_features.png
│   └── <other files you think are required>
├── issues.png
├── pr.png
└── pr_link.txt
```

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.

4.2 Marking scheme

Total: 100 marks

- **Structure and style (12%)**
  - Code readability (6 marks)
  - Good structure, avoiding repetition (6 marks)
- **Performance (16%)**
  - Good use of numpy (8 marks)
  - Performance plots and script (8 marks)
- **Additional functionality (12%)**
  - Generalisations to running code (3 marks)
  - Ignoring missing features (5 marks)
  - Incorporation of Algorithm Y (4 marks)

- **Validation and robustness (20%)**
  - Error messages (4 marks)
  - Unit tests (10 marks)
  - Continuous integration (6 marks)
- **Packaging and interfaces (20%)**
  - Installable package (2 marks)
  - Metadata files (2 marks)
  - Command-line interface with arguments passed (4 marks)
  - Command-line interface with YAML configuration (6 marks)
  - Sphinx documentation (6 marks)
- **Ways of working (20%)**
  - Git commits of reasonable size with meaningful messages (4 marks)
  - Consistent use of issues (8 marks) and pull requests (8 marks)