Assignment no. 1

Programming in Python II

Scenario: Preparing data for your ML project

In this assignment you will prepare the dataset for your ML project using Python. This starts with collecting the data (exercise 1). Afterwards you will perform a clean-up of the data (exercise 2). As last step, you will implement a first analysis and preprocessing of your dataset (exercise 3).

Exercise 1 [6 points]

In this exercise you will collect data for the dataset. For this, you have to take 100 pictures, e.g. with your cellphone.

You can choose which scenes/objects you take pictures of, as long as you follow these rules: The pictures must

- not exceed 850 kB per picture (a script for decreasing image size is provided in the supplements),
- not show humans or parts of humans,
- not show personal/private information (license plates, door bell nameplate, private documents, etc.),
- be unique with a maximum of 5 pictures of the same object/scene,
- be JPEG files (`https://en.wikipedia.org/wiki/JPEG`),
- have a minimum size of 64x64 pixels (other than that there is no specific or consistent layout required),
- include color information (no grayscale images),
- be taken by you (don't steal images from the internet!), and
- not include watermarks (deactivate watermarks for Huawei phones).

Create a `.zip` archive containing the 100 JPEG files. The name of the `.zip` file should be your student ID starting with the "k", as used for your KUSSS login (e.g. `k12345678.zip`). Upload this `.zip` file to our cloud server: `https://cloud.ml.jku.at/s/HBF59PJBdEsp57K` (the password for the upload is `PyThOn.2`).

The JPEG files from all students will be pooled into one large dataset, which we will use to train our model. This dataset will be made available to all students in this course. GPS information and other metadata will be removed before the dataset is made available to students.

The upload will be open until April 14th, 23:55. Please keep in mind that the server might be slow when a lot of students upload at the same time, so make sure to upload in time.

Exercise 2 [17 points]

In this exercise we will clean up the raw dataset. For this, you should write a Python function `ex2` that takes 3 keyword arguments as input:

- `inp_dir`: The input directory as string in which your function should look for files.
- `out_dir`: The directory as string in which your function should place the output files.
- `logfile`: The file path of the logfile as string.

Your function should scan `inp_dir` for files* recursively and sort this list of file names. The files should then be processed in the order that they appear in the sorted list of file names. Files should be copied to `out_dir` if they are valid. The name of the copied filename should be `xxxxxxx.jpg`, where `xxxxxxx` is the serial number. The serial number is an integer starting at 0, zero-padded to 7 digits, and being increased with each file that has been copied. For example, the first copied file should be placed in `out_dir` with the filename `0000000.jpg`, the second copied file with `0000001.jpg`, then `0000002.jpg`, and so on.

Files are considered valid if the following rules are met:

1. The file name ends with `.jpg`, `.JPG`, `.JPEG`, or `.jpeg`.
2. The file size is larger than `10kB`.
3. The file can be read as image (i.e. the `PIL`/`pillow` module does not raise an exception when reading the file).
4. The image data has shape (H, W), with H and W larger or equal to 100 pixels. Note that this excludes image data with color channels and shape (H, W, C).
5. The image data does have variance> 0, i.e. there is not just 1 value in the image data.
6. The same image data has not been copied already.

File names of invalid files should be written to `logfile`. The format of `logfile` should be as follows: Each line should contain the filename of the invalid file, followed by a semicolon, an error code, and a newline character. The error code is an integer with 1 digit, corresponding to the list of rules for file validity. Only one error code per file should be written and the rules should be checked in the order 1, 2, 3, 4, 5, 6. Each file name should only contain the relative file path starting from `inp_dir`.

The function should return the number of valid files that were copied as integer.

If `out_dir` or the directory that `logfile` should be in does not exist, your function should create it.

*) Since we are using Ubuntu (Linux) as reference operating system, directories are considered "files". Using `glob.glob(...)` to search for all contents of a folder recursively should return a list of all file paths, which also includes the directory paths. Therefore, you can apply the check for the rules on the sorted output of `glob.glob(...)` call without further processing.

Hint: You can store the hashes of all copied files to check if the current file has already been copied.

Hint: `os.path.getsize()` will report the size of a file.
Hint: You can use `shutil.copy` to copy the files.
Hint: `inp_dir` might be an absolute or relative path. You can use `os.path.abspath(inp_dir)` to get the absolute path of `inp_dir`, which might come in handy when you need the relative file path of a file that is located in `inp_dir`.

Exercise 3 [17 points]

In this exercise we will load and normalize each image by the global mean and standard deviation.

For this, you should write a class `ImageNormalizer`. This class should provide 3 methods: `__init__`, `analyze_images`, and `get_images_data`. You may add more methods to the class but these 3 methods are required with their functionalities as described below.

The `__init__` method of this class should:

1. Take one keyword argument, `input_dir`, which is the path to an input directory as string. This can be an absolute or relative path.
2. Scan this input directory recursively for files ending in `.jpg`. Remove the `input_dir` path from the found file paths to get the relative file paths of the `.jpg` files located in `input_dir`. (See last hint in exercise 2.)
3. Sort this list of relative file paths alphabetically using the `sort()` or `sorted()` function.
4. Store the sorted list of file paths as list of strings in a class attribute `self.file_paths`.
5. Store number of file paths as integer in a class attribute `self.n_file_paths`.
6. Create a class attribute `self.mean` with value `None`.
7. Create a class attribute `self.std` with value `None`.

   Note: Since `self.file_paths` only contains the relative paths starting at `input_dir`, you might need to create another attribute that let's you get the full path names in the other methods.

The `analyze_images` method of this class should:

1. Take no arguments.
2. Compute the mean values and standard deviations of all images in the list `self.file_paths`. Make sure to convert the image values to `np.float64` before computing the mean and standard deviation to have sufficient precision.
3. Store the average over the mean values of all images in class attribute `self.mean`. This value should be a scalar numpy `np.float64` object, as it is for example returned by using `np.mean()` on an array of datatype `np.float64`.
4. Store the average over the standard deviations of all images in class attribute `self.std`. This value should be a scalar numpy `np.float64` object, as it is for example returned by using `np.std()` on an array of datatype `np.float64`.
5. Return a tuple `(mean, std)`, where `mean` holds the value also stored in `self.mean` and `std` holds the value also stored in `self.std`. Both values should be returned as `np.float64` scalars.

Note: You may compute the average over mean and standard deviations on-the-fly or by storing the mean and standard deviation values in an array or list before you compute the averages.

The `get_images_data` method of this class should:

1. Take no arguments.
2. Raise a `ValueError` if `self.mean` or `self.std` are `None`.
3. Yield the (pixel) data of each image in the order that the image files appear in `self.file_paths`. For this, in each yield-iteration, the method should:
   - Load the image pixel data from the image file path using `PIL`.
   - Store this image data in a 2D numpy array of datatype `np.float32`.
   - Normalize the image data using the mean and standard deviation in `self.mean` and `self.std`. To do this, subtract the `self.mean` from the pixel values and subsequently divide the pixel values by `self.std`.
   - Yield the resulting normalized image data as 2D numpy array of datatype `np.float32`.

     Note: Perform the normalization operations in-place for less RAM consumption and faster processing.

   Note: Make sure to load the image data one-by-one in the yield iteration. (We are pretending that the complete dataset is too large to be loaded into the RAM at once.)

- You can assume that all files with file names ending in `.jpg` are valid grayscale image files.
- Small deviations due to float datatype (see Programming in Python I, Unit00 and slides) are to be expected and allowed. Make sure to use the specified datatype for computations, or the deviations will be too large.
- You may use multiprocessing (e.g. `multiprocessing.Pool`) in your methods to speed up your computations. However, please make sure that the number of worker processes used in your submission does not exceed 3.
- How exactly you design the solution to this exercise is up to you, as long as it fulfills the requirements stated in the exercise text.

Submission: electronically via Moodle unless stated otherwise:

`https://moodle.jku.at/`

Deadline: For deadlines see individual Moodle exercises unless stated otherwise.
Follow the **instructions for submitting homework** stated on the Moodle page!

Copyright statement:

This material, no matter whether in printed or electronic form, may be used for personal and non-commercial educational use only. Any reproduction of this material, no matter whether as a whole or in parts, no matter whether in printed or in electronic form, requires explicit prior acceptance of the authors.