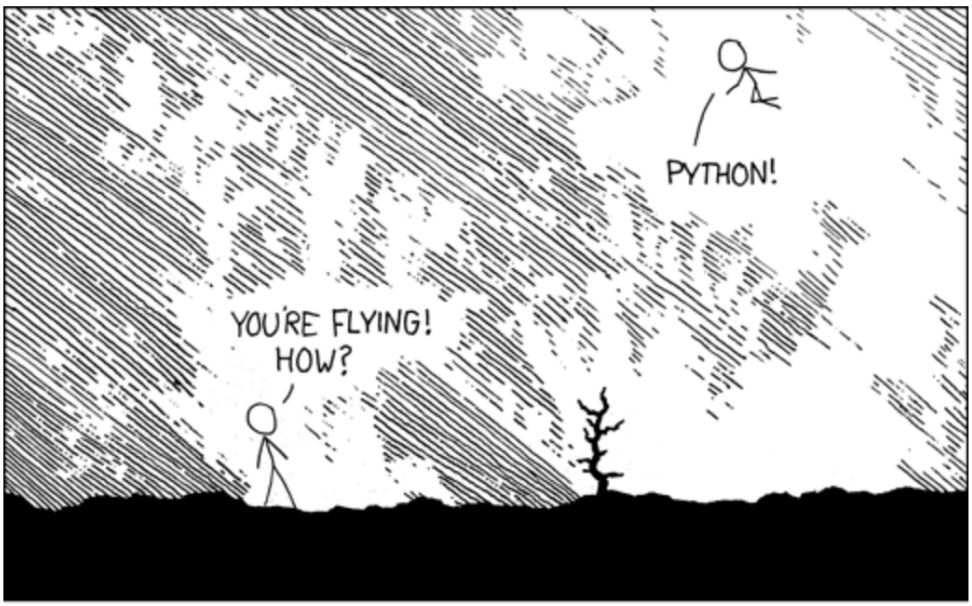
CS251 Outlab 4: ComputationalPython

CS 251 Outlab 4 : Python, Numpy, Scipy, Pandas and Matplotlib

Please refer to the submission guidelines at the end of this document before submitting.

General Instructions

- Make sure you know what you write, you might be asked to explain your code at a later point in time.
- Unless otherwise stated, you can't use any kind of loops (directly or indirectly).
- Parts of this lab will be graded automatically, so stick to the naming conventions strictly. We won't tolerate any kind of wrong submissions. You will get 0 for wrong submission.
- By matrix, we mean two dimensional numpy arrays.
- If we mention array, we mean an n-dimensional numpy array.
- The deadline for this lab is **12th September, 11:55 pm**.



Source: xkcd

Task 1 - (20 Marks (4 + 8 + 8))

● Write a function `fn_plot1d` which takes a single argument function `fn`, along with a finite continuous domain (in the form of a range) and a filename and plots that function and saves it into a file. You can assume that `fn` is well behaved (doesn't go to infinity or oscillates too fast) in the given domain.

The signature of `fn_plot1d` should be:

```
def fn_plot1d(fn, x_min, x_max, filename):
        pass
        # write your code here
```

● Use `fn_plot1d` to plot `b(x)`:

$$b(x) = g(|x|)$$

where:

$$g(x) = \frac{h(2-x)}{h(2-x) + h(x-1)}$$

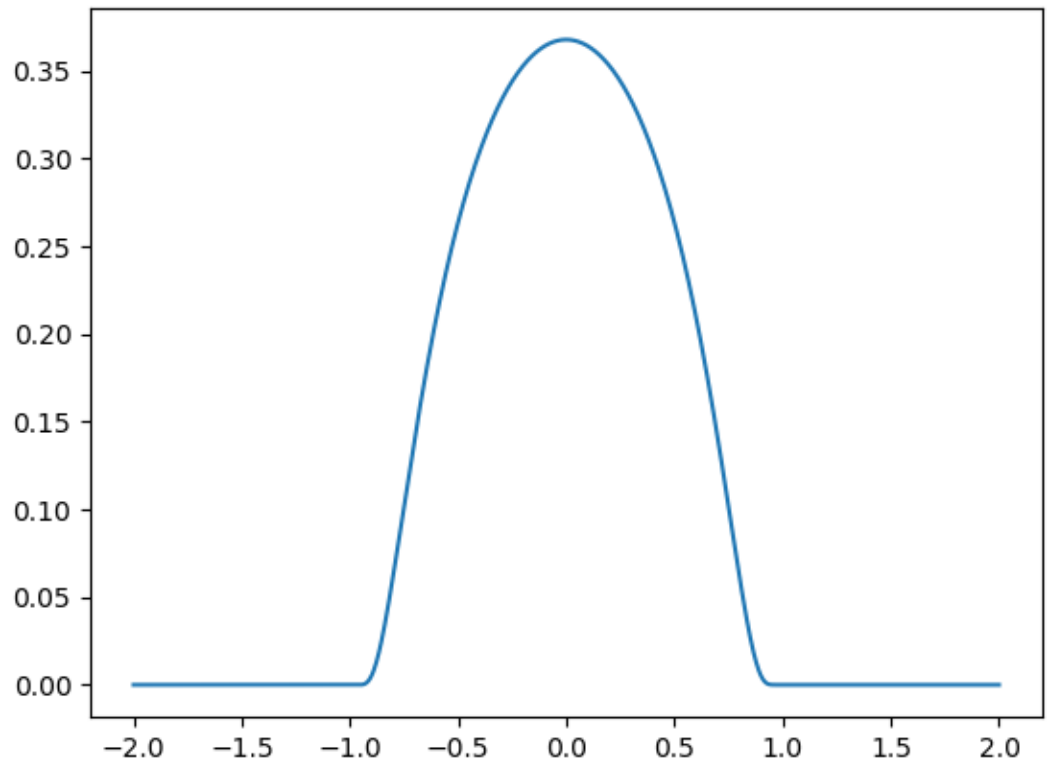
where,

$$h(x) = e^{\frac{-1}{x^2}}, if\ x > 0$$

$$0\ otherwise$$

in domain [-2,2]. Save it into file `fn1plot.png`

Here is a sample plot of a function plotted by fn_plot1d:



(No, this isn't the plot of `b(x)`).

● Similarly write `fn_plot2d` which takes a two-argument function `fn`, along with a rectangle and a filename and creates a surface plot of that function and saves it into a file. Again, assume that `fn` is well behaved in the given domain.

The signature of `fn_plot2d` should be:

```
def fn_plot2d(fn, x_min, x_max, y_min, y_max, filename):
        pass
        # write your code here
```

- Use `fn_plot2d` to plot the 2d sinc function:

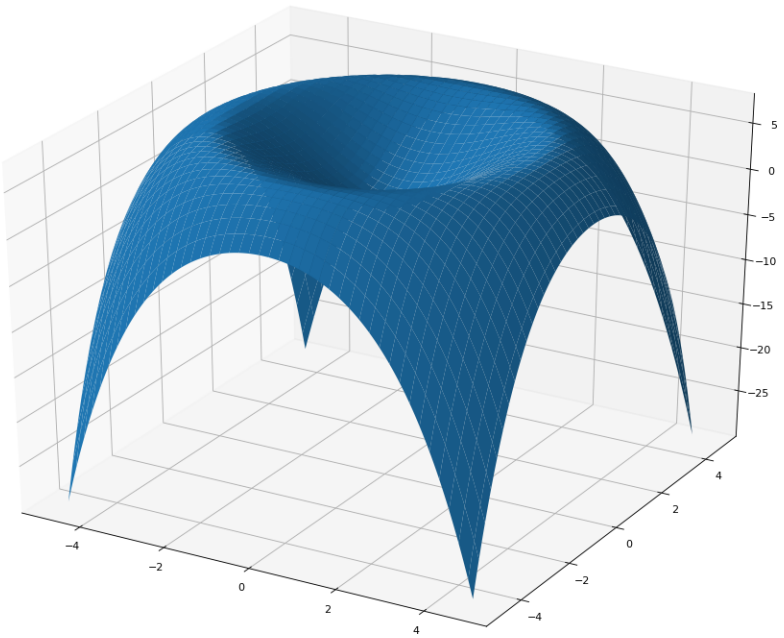
$$sinc(x,y) = \frac{sin(\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}} \text{ if } \sqrt{x^2+y^2} > 0$$
$$1 otherwise$$

In domain $x,y \epsilon [-1.5\pi, 1.5\pi]$. Save it into file `fn2plot.png`

Note: you **can't** use loops to implement both `fn_plot1d` and `fn_plot2d`. You **CAN** use vectorize.

Interesting fact: the function `b(x)` has compact support and it is a $C^{[?]?[?]}$ function.

Here is a sample surface plot of a 2d function:



- Create a function nth_derivative_plotter which takes following arguments:
  - `fn`: a well behaved function in the given domain
  - `n`: a positive integer. fn should be n-times differentiable.
  - `xmin`: lower limit of domain
  - `xmax`: upper limit of domain
  - `filename`: name of the output file

`nth_derivative_plotter` should plot the n^{th} derivative of `fn` in the given range and save it into a file with name `filename`. It should also add a suitable title to the plots! You have to **implement it without any kind of loops**. Again, vectorize is allowed.

- Use `nth_derivative_plotter` to plot the first 10 derivatives of $b(x)$ in [-2,2]. Save nth derivative with name `bd_<n>.png`. For example, plot of 1st derivative should be saved as `bd_1.png`. You **can** use loop for this part.

Write the above functions in **task1.py**.

Task 2 - (35 Marks)

Images are basically matrices. Grayscale (black and white) images can be represented using 2d numpy arrays. The value at each position of the matrix denotes the light intensity at that pixel in the image. We shall work with grayscale images, but, you can easily extend it to colored images.

Usually, each element is an 8-bit integer and its value ranges from 0 (black) to 255 (white). But, you can use default 64-bit integers for this assignment.

Just FYI, color images are 3 dimensional numpy arrays, in which, third dimension is of size 3. This third dimension stores colour. You can imagine that it is a 2d array of triplets. Each element of a triplet gives the intensity of Red, Green and Blue colour respectively.

Here is some background. Images often contain noise. One such noise is **salt and pepper** noise. Here, some pixels get corrupted with probability p, into either black (0) or white (255).
Here is an example of an image which has been corrupted with salt and pepper noise:

Here is another:

So, here is the deal: we will give you multiple (possibly overlapping) patches (along with their locations) of an image and you have to reconstruct the original image, while minimising the noise. We will also give you the overall size of image.
Understand that each patch carries pixel values of only few locations. But, for each pixel in the (to be) reconstructed image, you will get many values (likely to be distinct), from different patches. Now, the challenge is to decide which value do we put inside that pixel location.

Here is an algo that you shall follow **at each pixel** of (to be) reconstructed image:
Let i_1, i_2… i_m be 'm' values that you get for a pixel. We will scan them sequentially (this is equivalent to saying that we will see patches sequentially) and maintain **4** values. Let's call them **`black_count`**, **`mid_count`**, **`white_count`**, **`mid_total`**. Suppose that we have scanned till i_t. At this point:

- **`black_count`** is the number of times, we encountered a 0.
- **`white_count`** is number of times we encountered 255.
- **`mid_count`** is the number of times we encountered something between 0 and 255 (both exclusive).
- **`mid_total`** is the sum of such values.

Once you have scanned all the patches, if **`mid_count`** is non-zero, the final value shall be the (**`mid_total`** / **`mid_count`**) (rounded to nearest non-negative int < 256). Otherwise, it will be 0 or 255, depending on whether **`black_count`** is > **`white_count`** (which implies, if **`black_count`** <= **`white_count`**, put 255, else, put 0).

You have to do this at every pixel location in an image. There might be some pixels where no patches fall! Put 0 (black) there.

You should write a function named **`reconstruct_from_noisy_patches`** and save it into a file named **task 2.py**.
The signature of `reconstruct_from_noisy_patches` should be:

```
def reconstruct_from_noisy_patches(input_dict, shape):
        """
      input_dict:
      key: 4-tuple: (topleft_row, topleft_col, bottomright_row,
      bottomright_col): location of the patch in the original image.
      topleft_row, topleft_col are inclusive but bottomright_row,
      bottomright_col are exclusive. i.e. if M is the reconstructed matrix.
```

```
    M[topleft_row:bottomright_row, topleft_col:bottomright_col] will give the
    patch.

    value: 2d numpy array: the image patch.

    shape: shape of the original matrix.
    """
    # return the reconstructed image
```

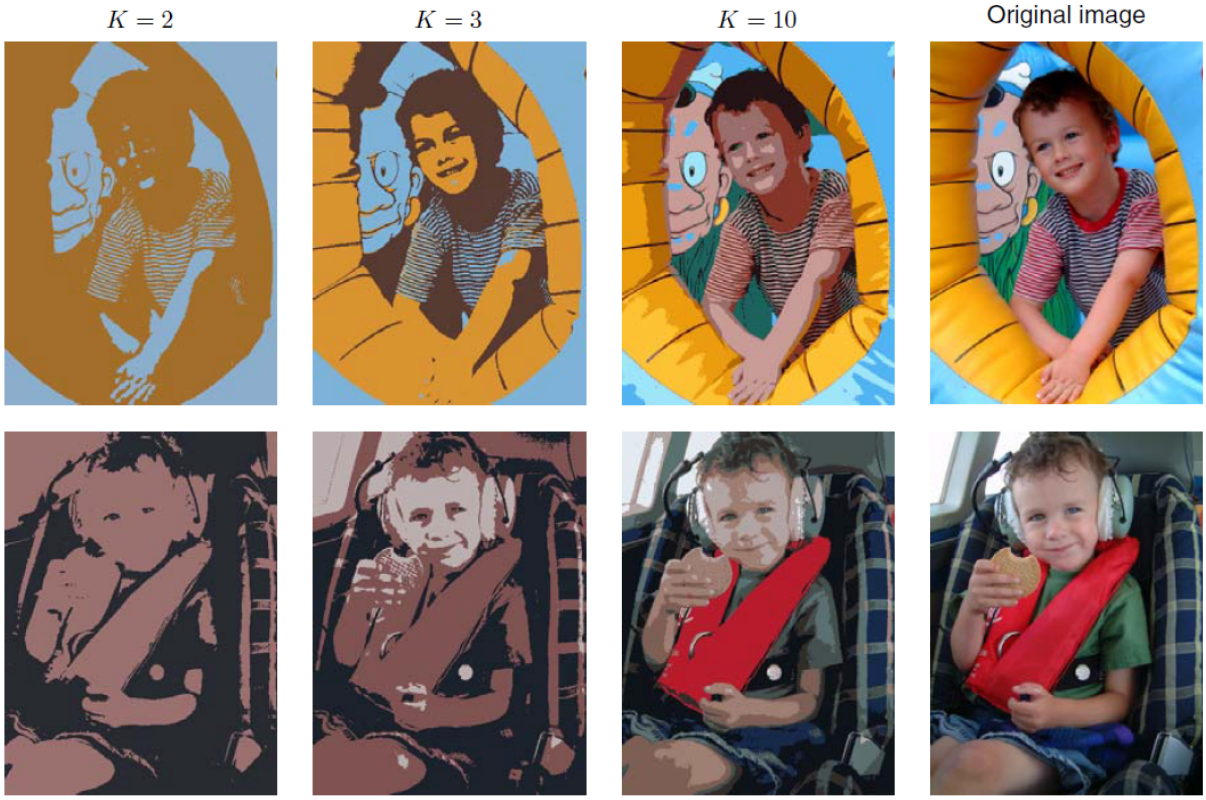
Note: you can use loops only to iterate over different patches.

Task 3 - (25 Marks)

Background:

We can represent colours fairly accurately as a combination of Red, Green and Blue. In a grayscale image, we use a 2d matrix to represent the brightness of the pixel at different locations. Similarly, we can use 3 different 2d matrices to denote intensities of R, G and B colours respectively, to represent a colourful image. These 3 matrices are called colour channels, or simply, channels. These channels are stacked over each other to make a 3d array. So, basically, a colourful image can be represented as a 3d matrix, whose 3rd dimension denotes colours. In the case of a grayscale image, each pixel is a scalar intensity. But, **for coloured images, each pixel is a 3d vector**, whose components denote the intensities of RGB colours.

In this task, we would group colours to get 'flatter' but cool images. We would take advantage of the fact that colours are nothing but vectors.
There are various algorithms, which can be used to group vectors. We shall use KMeans++. You need to give it the whole set of vectors of an image along with a number **k**, and it will return you the **k** groups of these vectors. You don't need to know how it works. Just use it! Look up: scipy.cluster.vq.kmeans2.



source: Pattern Recognition and Machine Learning, Christopher M. Bishop

Problem Statement

Here is the concrete problem statement:
Write a script called task3.py, which takes following commands:

--input in, where in is the path of input image.
--k k, k in kmeans++. It's an int, ranging from 1 to 50.
--output out, where out is the path of output image

It will take the image, apply k-means++ algorithm on its pixels and use replace each pixel by the centroid of the group of its pixel and save the constructed image back.
It's all that matters. **You can use whatever you want to complete this task (loops are also allowed)**. We will just see the output image. Don't bother about checking for invalid inputs. (For example, we won't give negative numbers or floats as k).

Here is a sequence of **recommended steps to do this task**. Using some other way is also fine.

1. The array elements will be integers in range [0,255]. You should convert it into float array.
2. Convert the array into a 2d-array of shape (n, 3). $n\ (= nr \times nc)$ is the total number of pixels in the image. nr is number of rows in the image and nc is the number of columns in the image. Let's call this matrix M. Each row of M is a vector representing a pixel in an image. Again, the vector contains the RGB values at that pixel.
3. Pass this matrix to kmeans2 with the argument **minit = '++'**.
4. Kmeans++ will give you two return values:
    a. Centroid: an array of shape (k,3). i^{th} is the centroid i^{th} cluster.
    b. Label: label[i] is the index of the centroid which is closest to i^{th} row of M.
5. Use this to replace each pixel by its centroid.
6. Save the image back.

Task 4 - (15 marks)

1. Write a function `mean_filter` to implement a mean filter of 1-d array of shape (n,) and kernel size $2k+1$. Output should be a float array. Output size should be same as input size. Assume zero padding to achieve this.
2. Write another function `generate_sin_wave` which takes argument (`period`, `range_`, `num`) and returns an array of shape (`num`,). The contents of array are samples of the function $sin\ (2\pi x\ /\ period)$ with given period in the given range. `range_` is a 2-tuple (`xmin`, `xmax`).
3. Write a function `noisify(array, var)` which adds gaussian noise with mean `0` and variance `var` to array and returns it. It shouldn't modify the input array.
4. Create a function `driver` which does the following using above defined functions:
    a. Create an array (let's call it `clean_sin`) of shape (1000,) containing a sin wave of period 2 in the range $[-2,8]$ using `generate_sin_wave` and plot it.
    b. Make another array (let's call it `dirty_sin`) by adding a gaussian noise of **standard deviation** 0.05. Plot it! Of course, it's shape will be same as that of `clean_sin`.
    c. Now, use `mean_filter` with kernel size 3 (k = 1) to clean `dirty_sin` to create an array called `cleaned_sin`. Plot it!
    d. Save the plots with respective names as: 'clean_sin.png', 'dirty_sin.png' and 'cleaned_sin.png'.

`driver` should be kept in a file driver.py. Other functions should be kept in a file task4.py

This task should be accomplished without any kind of loops, comprehensions or functions like np.vectorize, etc. Just like inlab.

Task 5 - (15 marks)

The file sml.csv contains 1 header + 7350 rows of data (total 7351 lines). It has 7 columns (as mentioned in the header): instance, algorithm, randomSeed, epsilon, horizon, REG.
You will generate three plots: one for each instance. The plot will have horizon on the x axis and regret on y axis. Use log scale on both axes. It will contain 7 lines: one for each algorithm (counting epsilon-greedy as three algorithms). For other algos, ignore epsilon. Each point will give the average regret from the fifty random runs at the particular horizon for the algorithm. Make sure you provide a clear key (legend) so the plot is easy to follow. Your plots should contain titles on both axes as well as on the main plot. **You are free to use literally anything in python.**
Name your script as task5.py. We should be able to call it as:

```
python3 task5.py --data sml.csv
--data: the path of data file
```

Your script should produce three plots:
instance1.png, instance2.png and instance3.png.
We will run it over different files. These will be different only in terms of horizons and randomSeed. Other things (instances, algorithms, epsilon) will be same.

ProTip: Pandas will be really helpful here. See pandas.DataFrame.groupby.

Submission Instructions

After creating your directory, package it into a tarball
outlab4-<team_name>.tar.gz
Submit once only per team from the moodle account of the smallest roll number.
The directory structure should be as follows (nothing more nothing less)

```
outlab4-<team_name>/
├── references.txt
├── Task1
│   └── task1.py
├── Task2
│   ├── utils.py
│   └── task2.py
├── Task3
│   └── task3.py
├── Task4
│   ├── driver.py
│   └── task4.py
└── Task5
    └── task5.py
```