DSC 20 Project: Classes, Inheritance and Exceptions.

Total Points: 100 (10% of the Course Grade)
Submission due (SD time):

- **Checkpoint: Thursday, March 7th, 11:59pm**
- **Final submission: Thursday, March 14th, 11:59pm**

Starter Files

Download project.zip

Contents:

- `project.py`
- `image_viewer.py` A file for viewing your images
- `img/` A folder of images
- `knn_data/` A folder of sample images for the KNN

Checkpoint Submission

Earn 5 points extra credit by completing **Part 1 and Part 2** by the deadline above.

Final Submission

Submit the *`project.py`* file to gradescope.

- Only this file will be checked
- **You do not need to submit any other files**

Partners

You can work with **one** other person on this project. If working with a partner, make sure to add them after you submit. If you resubmit, please re-add your partner - Gradescope will **not** automatically relink your latest submission to your partner. You should submit only one copy per team.

Important: The lateness policy for the project is the same as all homeworks. However, both partners must have an available slip day before you submit late.

Requirements

1. **You cannot use any libraries in project.py**
2. **The project will not be graded on style**
    a. You do not need to submit your doctests but it is highly recommended to test your code with custom doctests (we will have our own tests, not just provided ones).

    b. You can add docstrings, but they will not be graded.
3. Raise exceptions when required by the question
    a. **Exception requirements will be in bold blue**
    b. **If there are no exception requirements, you can assume valid inputs**
    c. **Do not use asserts**

Project Overview:

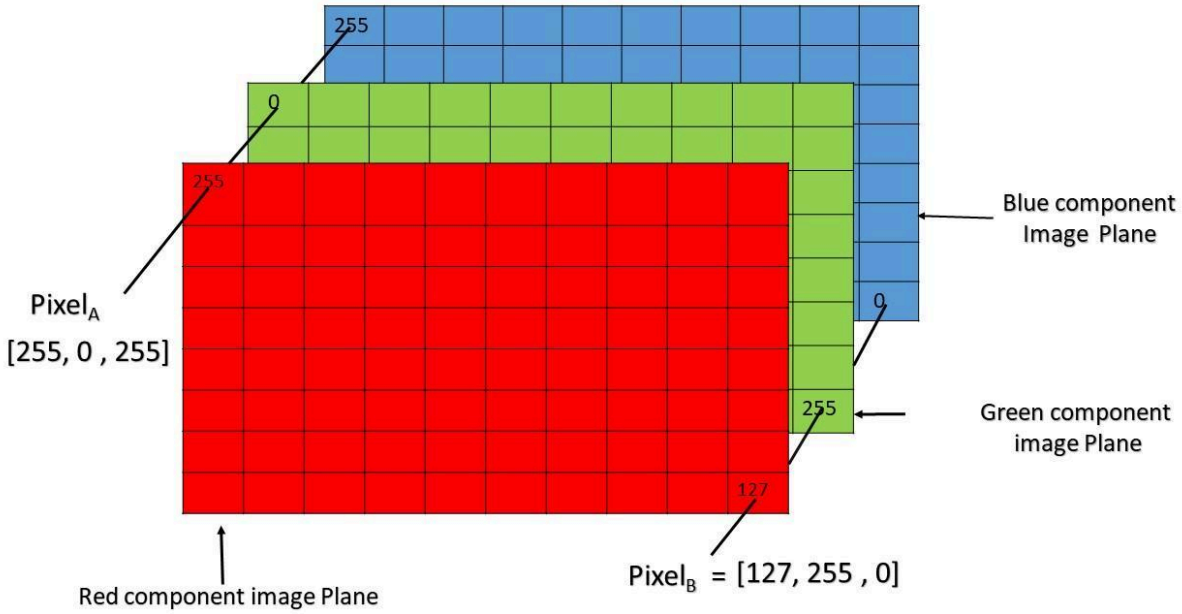
In this project, we will make a basic image processing app.

- **Part 1** covers how images are stored in code.
- **Part 2** introduces some image processing methods in a Processor Template.
- **Part 3** uses *inheritance* to simulate a monetized app.
- **Part 4** uses *inheritance* to simulate a premium app with new methods.
- **Part 5** implements a KNN classifier to predict the labels of images.

In the digital world, images are defined as 3-dimensional matrices: height (row), width (column), and channel (color). Each (row, col) entry is called a **pixel**. Height and width dimensions are intuitive: they define the size of the image. The channel dimension defines the color of an image.

The most commonly used color model is RGB. In this model, every color can be defined as a mixture of three primary color channels: Red, Green and Blue. Thus, the color of a pixel is digitally defined as a triplet (R, G, B). Each element in this triplet is an integer (called *intensity*) with value between 0 and 255 (both inclusive), where 0 represents no R/G/B is present and 255 means R/G/B is fully present. Thus, (0, 0, 0) represents **black** since no R/G/B are present, and (255, 255, 255) represents **white**. To better understand how the RGB color model works, you can play around the RGB value with this online color wheel.

In our project, we will use a 3-dimensional list of integers to structure the pixels. This picture shows how a *pixels* list is structured.



Pixel of an RGB image are formed from the corresponding pixel of the three component images

The first dimension is the row, starting from the top. The second dimension is the column, starting from the left. The third dimension is the color channel. In other words, len(*pixels[row][col]*) = 3, and each of the items in the *pixels[row][col]* list represents an intensity (0 - 255, both inclusive) of each color. Therefore, to index a specific intensity value at row *i*, column *j*, and channel *c* of the *pixels* list, you use `pixels[i][j][c]`.

Note that the *width* of an image is the length of the column dimension (number of columns), and the *height* of an image is the length of the row dimension (number of rows). Since in Python we conventionally consider (row, column) as the order of dimensions for 2-dimensional lists, make sure to distinguish these notions clearly.

Install Pillow and NumPy

The project uses two packages: NumPy (np) and Pillow (PIL). These are two of the most common packages to use for image processing. Although we prevent you from using these packages in your own implementation, you can still use them for testing. If you have not installed these packages before, run the following command in your terminal:

Mac/Linux: `python3 -m pip install numpy Pillow`
Windows: `py -m pip install numpy Pillow`

If you have trouble installing the packages, try updating pip first:

Mac/Linux: `python3 -m pip install --upgrade pip`
Windows: `py -m pip install --upgrade pip`

Please note: **You are <u>not</u> allowed to use numpy or PIL/Pillow methods in your code**.

Testing

We have provided basic doctests that cover simple cases to check if your code works. Note that **you will want to create more tests to check edge cases**.

The doctests use small square images between 6px and 16px in size. They also use the included expected output images in the img/exp/ directory to compare your results against.

The doctests include some exceptions, deep copy, and cost value checks. Recall the difference between deep and shallow copies.

Shallow Copy: is a concept that refers to creating a new object that shares the underlying elements with the original object. In other words, a shallow copy creates a new object but references the same nested objects or elements as the original.

Deep copy: is a concept that refers to creating a complete and independent copy of an object, including all of its nested objects or elements.

Since this project is about processing images, it makes more sense for you to visually check the output. Therefore, your main way of debugging will be to look at the images generated.

Mac/Linux: `python3 -m doctest project.py`
Windows: `py -m doctest project.py`

Using image_viewer.py

We have included a python file that allows you to view your images to check for errors.

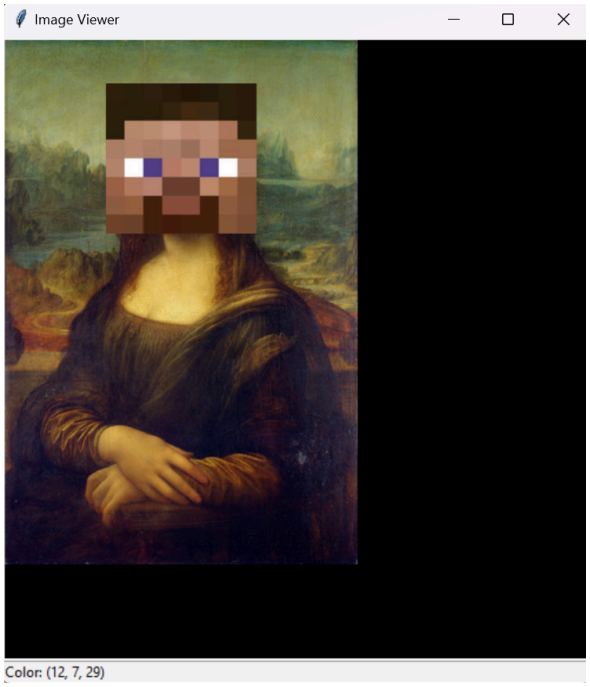
To run the file, use the following commands

Make sure to change the path in the "img/out/the_file_to_view.png" to connect to the image that you want to view.

Mac/Linux: `python3 image_viewer.py "img/out/the_file_to_view.png"`
Windows: `py image_viewer.py "img/out/the_file_to_view.png"`

If you get an error like "ModuleNotFoundError: No module named 'tkinter'", then run the command `python3 -m pip install tk`. If this doesn't work, you can come to office hours and we can help install it for you (involves reinstalling python).

The script takes in a file path that points to an image, and displays it for you to view



You can zoom with your scroll wheel and pan around with your mouse. When you hover over a pixel, it will display the current color value at the bottom.

You can use this to check where your code is creating the wrong output values. You can also use it to manually compare your output against the expected output.

Video Guide:

This is a setup video from a previous quarter with a similar project: Link
We recommend that you watch this video while working on the project to better understand all of its components.

Understanding Quiz:

There is an optional quiz on canvas to check your understanding of the basic ideas behind this project, it is not required but we recommend taking it to make sure you understand the writeup before creating your code: Not ready yet, come back later

Efficiency and Runtime for Autograder

Because we are dealing with massive lists of lists, your code will need to be efficient.

Your code must run for less than 10 minutes to pass the autograder. Each test in the autograder is limited to 30 seconds. The solution takes around 70 seconds to run the runner file (this time is beatable).

Part 1: RGB Image

In this part, you will implement the **RGBImage** class

__init__ (self, pixels)

A constructor for an RGBImage instance

The argument `pixels` is a 3D list, where `pixels[i][j][c]` indicates the intensity value at position (`i`, `j`) in channel `c`. You simply need to assign this list to **`self.pixels`**.

You also need to set the values of **`self.num_rows`** and **`self.num_cols`** accordingly.

Exceptions:
If the argument `pixels` is not a 3-dimensional list, raise a `TypeError()`. The conditions you need to check for are the following:

- `pixels` is a list, and has at least one element
- All elements (rows) in `pixels` are also lists, and have at least one element
- All rows are the same length
- All elements (pixels) in each row are also lists
- All pixels are a list of length 3

Then, if any pixel has an intensity value other than an integer from 0-255, raise a `ValueError()`

size (self)

A getter method that returns the size of the image, where size is defined as a tuple of (number of rows, number of columns).

get_pixels (self)

A getter method that returns a **DEEP COPY** of the pixels matrix. This matrix of pixels is exactly the same *`pixels`* passed to the constructor.

Note:
It is very important to create a deep copy, as otherwise the ImageProcessing methods will break.

Recall that we can create a deep copy of a 1D-list with a list comprehension (`[elem for elem in x]`) or with (`list(x)`). However, this will not work on `pixels` since it is a 3D-list. The outer list would be reconstructed, but the inner lists are the same object in memory for the original and the copy.

copy (self)

A method that returns a **COPY** of the `RGBImage` instance. You should create a new `RGBImage` instance using a **deep** copy of the `pixels` matrix (you can utilize the get_pixels function) and return this new instance.

get_pixel (self, row, col)

A getter method that returns the color of the pixel at position (`row`, `col`). The color should be returned as a 3-element **tuple**: (red intensity, green intensity, blue intensity) at that position.

Note that the rows and columns are 0-indexed, so `get_pixel(0, 0)` is valid.

Exceptions:
If row and col are not integers, raise a `TypeError()`

If row and col are not valid indices in the pixels matrix, raise a `ValueError()`. Negative

indexing is considered an invalid index, so a `ValueError()` should be raised in that case.

set_pixel (self, row, col, new_color)

A setter method that updates the color of the pixel at position (`row`, `col`) to the new_color. The argument `new_color` is a 3-element tuple (red intensity, green intensity, blue intensity).

We want this function to have the ability to modify one color channel, while leaving the rest unchanged. Therefore, **if the given color channel is any negative number, then you should not modify that channel**, but you should still modify the other channels. Consider this example:

```
img = [
        [[255, 128,  64], [ 32,  16,   8]],
        [[  1,   2,   4], [  8,  16,  32]]
]
new_color = (-1, 0, 0)
new_row = 0
new_col = 0
img.set_pixel(new_row, new_col, new_color)
```

Modified image matrix:

```
img = [
        [[255,   0,   0], [ 32,  16,   8]],
        [[  1,   2,   4], [  8,  16,  32]]
]
```

You can see that the R value of the 0,0 pixel was not changed, while the G and B values were both set to 0.

Exceptions:
If `row` and `col` are not integers, raise a `TypeError()`

If `row` and `col` are not valid indices in the pixels matrix, raise a `ValueError()`

If `new_color` is not all of the following, raise a `TypeError()`

- a tuple
- of length 3
- with all integers

If `new_color` has an intensity value greater than 255, raise a `ValueError()`. Note that you should allow negative values, per the example above.

Part 2: Image Processing Template Class

In this part, you will implement several image processing methods in the **ImageProcessingTemplate** class. This will be the parent class that both the premium and standard version of your app will inherit from.

Note:

- All methods in this class must return a **new `RGBImage` instance** with the processed pixels matrix. After calling any of these methods, the original image passed in should not be modified.

You need to implement the following methods:

__init__ (self)
A constructor that initializes an ImageProcessingTemplate instance and necessary instance variables. For this, you simply need to set `self.cost` to 0. This variable will track the total cost incurred by the user.
get_cost (self)
A getter method that returns the current total incurred cost. (For the template class this will always be 0, the child classes will change the cost)

negate (self, image)
A method that returns the negative of the given `image`. To produce a negative image, all pixel values must be inverted. Specifically, for each pixel's channel with intensity *`val`*, this method should create a new image with `(255 - val)`. **Requirements:** **No explicit for/while loops.** You can use list comprehensions or `map()` instead. **Example:**

image	negate (image)
	

grayscale (self, image)
A method that converts the given image to grayscale. For each pixel (R, G, B) in the pixels matrix, calculate the average (R + G + B) / 3 and update all channels with this average, i.e. (R, G, B) -> ((R + G + B) / 3, (R + G + B) / 3, (R + G + B) / 3).

Note that since intensity values must be an integer, you should use integer division (100 // 3 = 33)

Requirements:
No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	grayscale(image)
	

rotate_180 (self, image)

A method that rotates the image 180 degrees.

Tip:
There are multiple different ways to do this. Since the pixel values do not change, you only need to consider the logic for a 2d matrix. Draw out an input array and an output array and figure how pixels move.

Requirements:
No explicit for/while loops. You can use list comprehensions or `map()` instead.

Example:

image	rotate_180 (image)
	

get_average_brightness (self, image)

A method that returns the average brightness of the image.

The brightness of a pixel is defined to be the average of its 3 RGB values. For example the brightness of the pixel [100, 200, 150] is 150.

```
pix_avg = (100 + 101 + 101) // 3 = 100
```

The brightness of an image is defined to be the average brightness of all of its pixels.

```
all_avg = sum of pixels / number of pixels
```

Use **floor division** when calculating the averages

Requirements:
No explicit for/while loops. You can use list comprehensions or `map()` instead.

adjust_brightness (self, image, intensity)

A method that alters the brightness of an image by `intensity`.

Alter the brightness by changing the R, G, and B values of each pixel by `intensity`. For example, when called with an `intensity` of 25 on an image with the pixel [100, 200, 150], the pixel would become [125, 225, 175].

Remember that the R, G, and B values should not go above 255 or below 0.

So given an image with a pixel [0, 10, 0] and an `intensity` of -50, the pixel would become [0, 0, 0]

Exceptions:
If `intensity` is not an integer, raise a `TypeError()`

If `intensity` is greater than 255 or less than -255, raise a `ValueError()`

Requirements:
No explicit for/while loops. You can use list comprehensions or `map()`/`filter()` instead.

Example:

image	adjust_brightness (image, 75)
	

blur (self, image)

A method that blurs the image. To blur an image, make the RGB values of each pixel the average of itself and its 8 neighbors.

For pixels on the edge, only average neighbors that exist. For example, a pixel on the corner will have RGB values corresponding to the average of itself and its 3 neighbors.

Rather than round, use floor division. Take the averages for each color channel individually.

Example:

```
[
 [[215, 209, 223], [108, 185, 135], [ 54, 135,  36]],
 [[184,  20, 245], [ 32,  52, 249], [243, 155,  96]],
 [[108,  66, 116], [ 65, 200,  34], [  2, 213, 238]]
]
```

To calculate the new pixel at `row=0`, `col=0`, find the averages of each channel:

```
r_avg = (215 + 108 + 184 +  32) // 4 = 134
b_avg = (209 + 185 +  20 +  52) // 4 = 116
g_avg = (223 + 135 + 245 + 249) // 4 = 213
```

Here is the calculation for `row=1`, `col=1`:

```
r_avg = (215 + 108 + 54 + 184 + 32 + 243 + 108 + 65 + 2) // 9 = 112
b_avg = (209 + 185 + 135 + 20 + 52 + 155 + 66 + 200 + 213) // 9 = 137
g_avg = (223 + 135 + 36 + 245 + 249 + 96 + 116 + 34 + 238) // 9 = 152
```

Requirements:
None. You can use explicit loops.

Example:

image	blur (image)
	

Checkpoint submission ends here.

Part 3: Standard Image Processing Class

In this part, you will create a monetized version of the template class. Because the investment money is running out, you will need to make money off of your users, so each method will add a certain amount to the cost each time it is called.

__init__ (self)

A constructor that initializes a `StandardImageProcessing` instance and necessary instance variables.

For this, you simply need to set `self.cost` to 0. This variable will track the total cost incurred by the user.

You can add your own instance variables to help implement the methods below.

negate (self, image)

Same as negate in the template, but should add +5 cost for each use.

Requirements:
You must use inheritance, do not copy paste code.
You must use `super()`, do not use the class name.

grayscale (self, image)

Same as grayscale in the template, but should add +6 cost for each use.

Requirements:
You must use inheritance, do not copy paste code.
You must use `super()`, do not use the class name.

rotate_180 (self, image)

Same as rotate_180 in the template, but should add +10 cost for each use.

Requirements:
You must use inheritance, do not copy paste code.
You must use `super()`, do not use the class name.

get_average_brightness (self, image)

Since this method is the same as the template, you do not need to write anything here.

adjust_brightness (self, image, intensity)

Same as adjust_brightness in the template, but should add +1 cost for each use.

Requirements:
You must use inheritance, do not copy paste code.
You must use `super()`, do not use the class name.

blur (self, image)

Same as blur in the template, but should add +5 cost for each use.

Requirements:
You must use inheritance, do not copy paste code.
You must use `super()`, do not use the class name.

redeem_coupon(self, amount)

Makes the next `amount` image processing method calls free, as in they add 0 to the cost. If it is called again the `amount` of free method calls is added to however many the user currently has.

Example:
`redeem_coupon(5)` -> next 5 method calls are free
`redeem_coupon(1)` -> next 6 (5+1) method calls are free
`grayscale(img)` -> cost was not changed, next 5 calls are free

Exceptions:
If `amount` is not an integer, raise a `TypeError()`

If `amount` is negative or 0, raise a `ValueError()`

Hint:
You will want to use an instance variable to track this

Part 4: Premium Image Processing Class

Premium versions are all the rage with the youth nowadays, so you decide to make a premium image processing app. This version of the app should cost 50 dollars up front, but every method call after that will be free. Additionally, to draw in premium users it will have 2 new methods: `chroma_key` and `sticker`.

__init__ (self)

A constructor that initializes a `PremiumImageProcessing` instance and necessary instance variables.

Since this is a premium version, it should have a fixed cost of 50. This will not change when the methods are called.

chroma_key (self, chroma_image, background_image, color)

A method that returns a new image where pixels with the specified `color` in the `chroma_image` are replaced by the corresponding pixel in the `background_image`.

This is the same process as a green screen, except that the color can vary. The `chroma_image` is in the foreground, while the background is replaced by `background_image`.

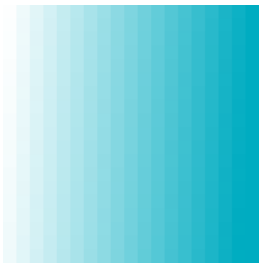
Exceptions:
If either `chroma_image` or `background_image` are not RGBImage instances, raise a `TypeError()`

If the images are not the same size, raise a `ValueError()`

You can assume that *`color`* is a valid (R, G, B) tuple.

Requirements:
None. You can use explicit loops.

Example:

chroma image	background image	color = (255, 255, 255)	color = (255, 205, 210)
		 (white background replaced)	 (pink font color replaced)

sticker (self, sticker_image, background_image, x_pos, y_pos)

A method where the top left corner of the sticker is placed at position x_pos, y_pos on the given background image.

Note that we are using x-position and y-position instead of row and column. Think about how to convert between the two. The x and y positions are still 0-indexed at the top left corner (0,0 is at the top left corner)

Example for converting x_pos, y_pos

Given the following 4x4 matrix:

```
[
    [ 1,  2,  3,  4],
    [ 5,  6,  7,  8],
    [ 9, 10, 11, 12],
    [13, 14, 15, 16]
]
```

x_pos, y_pos = (1,0) would yield the value 2.

x_pos, y_pos = (2,2) would yield the value 11.
x_pos, y_pos = (3,2) would yield the value 12.

We define the x and y axis as the following: X increases from left to right, and y increases from top to bottom. See the following (not to scale) image:

Exceptions:
If either `sticker_image` or `background_image` are not RGBImage instances, raise a TypeError()

If the sticker is wider or taller than the background, raise a ValueError()

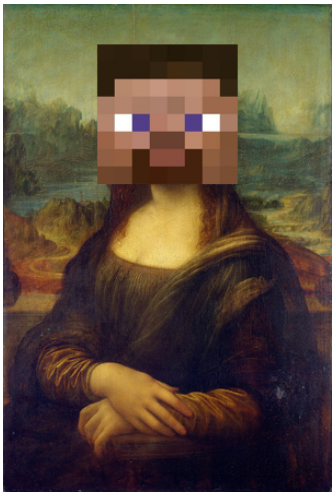
If either `x_pos` or `y_pos` is not an integer, raise a TypeError()

If the `sticker_image` will not fit at the given position, raise a ValueError()

Requirements:
None. You can use explicit loops.

Example:

sticker image 128x128	background image 300x447	x_pos=86, y_pos=37



edge_highlight (self, image)

A method that highlights the edges in an image.

To highlight edges, we will want to ignore all color data. Therefore, your first step should be to convert all pixels into single values.

For the following RGB Image

```
[
 [[215, 209, 223], [108, 185, 135], [ 54, 135,  36]],
 [[184,  20, 245], [ 32,  52, 249], [243, 155,  96]],
 [[108,  66, 116], [ 65, 200,  34], [  2, 213, 238]]
]
```

This would become (use floor division when calculating average)

```
[
 [215, 142,  75],
 [149, 111, 164],
 [ 96,  99, 151]
]
```

The next step is to apply something called a **kernel**. It is a type of 2D array that acts like a mask, being applied to every pixel. It looks like this

```
[
 [-1, -1, -1],
 [-1,  8, -1],
 [-1, -1, -1]
]
```

To calculate the output for the pixel at `row=0`, `col=0`, you multiply all of the mask's values with all of the pixel values, then add them together.

Since the mask is centered at the top left, you can ignore the weights outside of the image. The colored values are the relevant weights that we will use for this single operation.

```
[
 [-1, -1, -1],
 [-1,  8, -1],
 [-1, -1, -1]
]
```

Then, centered at 0,0 we multiply each weight by the relevant value

```
[
 [215 *  8, 142 * -1, …],
 [149 * -1, 111 * -1, …],
 [       …,        …, …]
]
```

masked_value = 215*8 + 142*-1 + 149*-1 + 111*-1 = 1318 -> 255

Note that the masked_value needs to be between 0 and 255 in your code.

Next, we will show the calculation for the value at `row=1`, `col=1`. This will use the full mask.

```
[
 [215 * -1, 142 * -1,  75 * -1],
 [149 * -1, 111 *  8, 164 * -1],
 [ 96 * -1,  99 * -1, 151 * -1]
]
```

masked_value = 215*-1 + 142*-1 + 75*-1 + 149*-1 + 111*8 + 164*-1 + 96*-1 + 99*-1 + 151*-1 = -203 -> 0

This is what the two calculations we have done would look like in the output matrix. Note that it is certainly possible to have values between 0 and 255 in the output, but that these examples did not yield that.

```
[
 [255,   …,   …],
 [  …,   0,   …],
 [  …,   …,   …]
]
```

Finally, you will want to convert this into an RGBImage object. Simply use the single intensity value for all 3 channels.

```
[
 [[255,255,255],             …,   …],
 [            …, [  0,  0,  0],   …],
 [            …,             …,   …]
]
```

This process of applying a mask to all of the pixels in an image is called convolution, and is the technique that Convolutional Neural Networks use.

Requirements:

None. You can use explicit loops.

Example:

image	edge_highlight (image)
	

Part 5: Image KNN Classifier

Now you want to indulge in some data science, maybe classifying images will give you more good business ideas?

Classification

Classification is one of the major tasks in machine learning, which aims to predict the label of a piece of unknown data by learning a pattern from a known collection of data. To train a classifier, you need to fit the classifier with training data, meaning add to the model a correct collection of (data, label) pairs to base predictions off of. The classifier will apply the training data to its own algorithm. After training, you can provide a piece of known or unknown data, and the classifier will try to predict a label. Through this process we can extract essential information from pieces of complicated data.

Example

For example, you want to train your model to recognize whether it is day or night. You might start by loading a dataset of daytime images and nighttime images to your model. To match the label 'daytime' with an image of the day and 'nighttime' with images of the night, we can use a `KNN Classifier`.

Our task: given an image we want to predict if it's **day** or **night**. We start by calculating a statistic of how similar our image is to the day/night images in our dataset using the features of the image, like pixels. Then, you can take k-nearest neighbors, (k is a number of most similar images), and count up the number of neighbors that are labeled as 'daytime' and 'nighttime'. If there are more neighbors with the label 'daytime', then our image is probably a showing daytime. If there are more neighbors with the label 'nighttime', then our image is probably showing 'nighttime'.

Algorithm

In this part of the project, you will implement a K-nearest Neighbors (KNN) classifier for the RGB images. Given an image, this algorithm will predict the label by finding the most popular labels in a collection (with size *k*) of **nearest** training data.

How could we evaluate how similar two images are? We look at "how far away" they are from each other. For that, we need a definition of a **distance** between images. With this definition, we can find the nearest training data by finding the shortest distances. Here, we use the Euclidean distance as our definition of distance. Since images are represented as 3-dimensional matrices, we can first flatten them to 1-dimensional vectors, then apply the Euclidean distance formula. For image *a* and *b*, we define their Euclidean distance as:

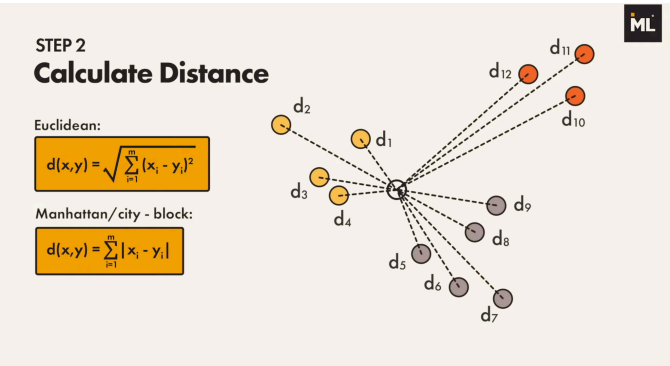
$$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

Where a_i and b_i (`1 <= i <= n` based on the above equation) are the intensity values at the same position of two image matrices, and n is the count of individual intensity values in the image matrices, which equals to (`number of rows` × `number of columns` × `number of channels` (which is 3 for RGB)). Note that **to calculate the distance, two images must have the same size.** Can you figure out why?

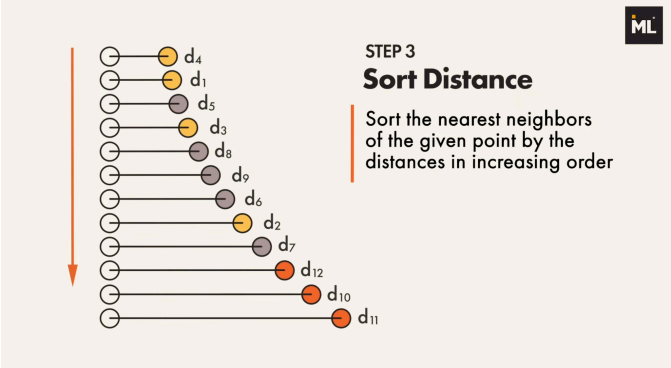
Video

Here is a handy video on KNN that explains the KNN classifier process step by step: https://www.youtube.com/watch?v=0p0o5cmgLdE

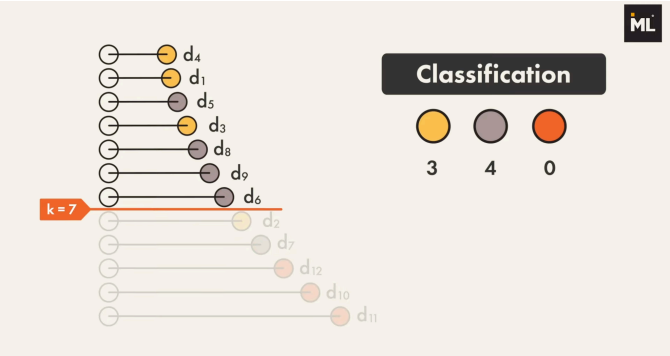
In the `fitting` (training) stage, all you need to do is to store the training data for later.



In the `distance` stage, the algorithm will find the distance between the provided image and all images in the training data.



In the next step, we sort the distances from shortest to longest. This gives us the nearest neighbors.



Last is the `predict` stage. The K value is the number of neighbors that we check. In this example, we check the labels of the top 7 nearest neighbors. The most popular is gray, so we will classify this point as gray.

__init__ (self, k_neighbors)
A constructor that initializes a `ImageKNNClassifier` instance and instance variables. **Parameters:** `k_neighbors`: integer of how many neighbors to consider when predicting the label
fit (self, data)
Fit the classifier by storing all training data in the classifier instance. You can assume *`data`* is a list of (*`image`*, *`label`*) tuples, where *`image`* is a `RGBImage` instance and *`label`* is a `string`.

Parameters:
`data`: A list of tuples, where every tuple is a (*`image`*, *`label`*). In a tuple *image* is a RGBImage instance, and labels is a string.

Exceptions:
If the given number of data points is less than the number of neighbors, raise a `ValueError()`

Requirements:
Make sure that the name of the instance attribute that you store the training data in is `data`

distance (image1, image2)

Description:
A method to calculate the Euclidean distance between RGB image *`image1`* and *`image2`*.

Parameters:
`Image1`: RGBImage instance

`Image2`: RGBImage instance

Exceptions:
Make sure that both arguments are RGBImage instances with the same size.

If either (or both) are not RGBImage instances, raise a `TypeError()`.

If they are not the same size, raise a `ValueError()`.

Hint:
To calculate the Euclidean distance, for the value at each position (row, column, channel) in the *pixels* of both images, calculate the **squared difference** between two values. Then, add all these values of squared difference together. The Euclidean distance between two images is the **square root** of this sum. You can refer to

$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$ if you prefer a more formal definition.

Where a_i and b_i (`1 <= i <= n` based on the above equation) are the intensity values at the same position of two image matrices, and n is the count of individual intensity values in the image matrices, which equals to (`number of rows` × `number of columns` × `number of channels` (which is 3 for RGB)).

Requirements:
No explicit for/while loops. You can use list comprehensions or `map()` instead.
Do not use np.sqrt(...). You will be deducted points for this.

vote (candidates)

Description:
Find the most popular label from a `list` of `candidates`. If there is a tie when determining the majority label, you can return **any** of them.

You will give this function a list of labels, and it will return the most frequent one.

Parameters:
`candidates`: a list of `k_neighbors` numbers of labels, as strings, that are nearest neighbors.

predict (self, image)

Description:
You need to put everything together in this function. Use previous functions as helper functions to implement aspects of the KNN classification algorithm: predict and execute it from beginning to end. In this function, your task is to predict the label of the given *`image`* and the data stored in the `ImageKNNClassifier` instance using the KNN classification algorithm. Refer to the introduction and helper video if you're unsure about the process.

Parameters:
`image`: an RGBImage instance

Exceptions:
Make sure that the training data is present in the classifier instance. Raise a `ValueError()` if the `fit()` method has not been called before calling this method.

Hint: You should use the `vote()` method to make the prediction from the nearest neighbors.

Requirements:
No explicit for/while loops. You can use list comprehensions or `map()` instead.

Final Submission

Submit the *project.py* file to gradescope

- Only this file will be checked
- One submission per group. **If working with a partner, make sure to add them after you submit. If you resubmit, please re-add your partner - Gradescope will not automatically relink your latest submission to your partner.**
- You do not need to submit any other files