Python Team Project Report

Felipe da Paixao, Agustin Moya Rojas, Henry Pham, Aleksander Witt

LC1: Team 15

ENGR 13300

Prof. Soudebeh Taghian

Oct 23rd, 2025

1. Project Motivation

Our motivation for this project fundamentally strands from learning important skills and techniques for image processing using python, that not only teach us problem solving in a highly technical setting, but also a potentially useful skill for future applications in our work. This is because several engineering disciplines use image processing for crucial tasks. In mechanical engineering [Henry], very detailed image processing is used for automated quality control systems and robotics in the automotive industry (*Gardel, Science Direct*), in the electrical engineering [Aleks] field it is increasingly being used, paired with trained AI models, to visually inspect client's PCBs and give feedback on how to optimize them (*Averroes*) , and in IBE [Agustin & Felipe] as it has varying relevance depending on the specific engineering discipline we end up choosing, a direct relation to business, among many others, could be the growing applications of "shelf intelligence" to optimize warehouse and inventory systems in real world retail businesses (*Trax*). It is evident that all these are direct applications of our current project for future applications we may encounter in our daily work or projects, and even if the scope varies, the experience of problem solving by trial and error and growing our teamwork skills, is of great value to us.

2. Project Overview and Methods

In this project we were tasked to develop a program for image processing and created algorithms for each step of the process including preprocessing and image extraction and finally using this extracted data along with machine learning algorithms to predict accurate and reliable new data as a result. In checkpoint 1, it was crucial to first correctly prepare the images, before extracting its features. To do this, we make sure that the image properties are consistent and correctly validated and convert them to the format of our choice (HSV, RGB, Grayscale), resize them, and finally have methods in place to handle errors.

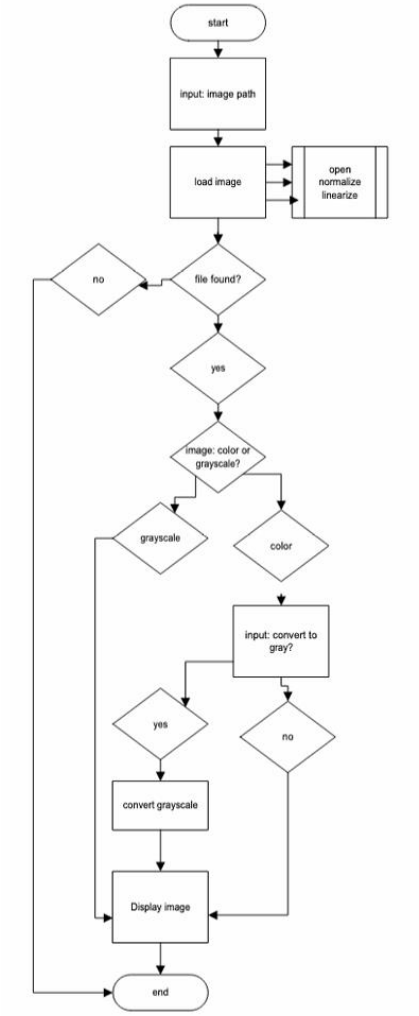
Next, for checkpoint 2 we moved on to the more practical and applicable steps, to actually extract the image's shape-based features, for example; edge detection and line detection. We used filtering techniques like a gaussian filter to make images less complex and easier to extract features from and then a Sobel filter to determine where edges were in the image based on the brightness of pixels. From there, we could then convert the images to HSV and calculate statistical values like the mean and standard deviation for the image's hue, saturation, and value. We also detected shapes by counting lines and detecting circles to give us data that could be used to classify the images.

Finally, in checkpoint 3, we used all the features extracted from previous tasks to train machine learning algorithms to classify these images. Our KNN model compared each image to others in the dataset to classify it based on the images that are most similar. The Decision Tree model splits the data into left and right branches based on feature values and whether the data met the thresholds of the feature values to predict the image classification. The Linear Regression model calculated a best fit line or curve that separated classes to determine the probability of an image fitting in that class. Our data was split into training, validation, and test sets to evaluate the model's performance.

3. Discussion of Algorithm Design

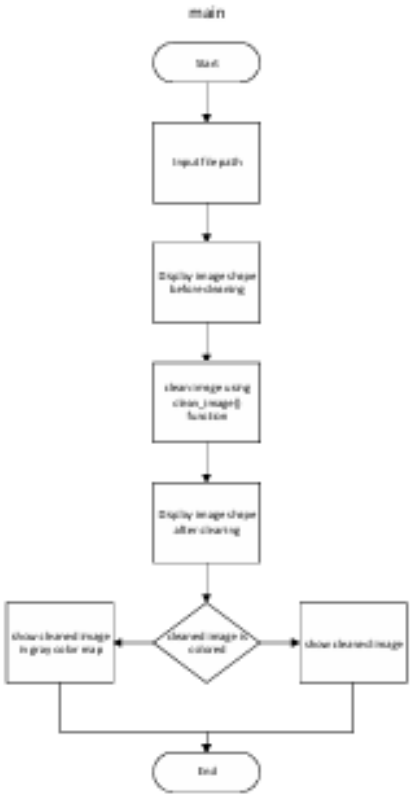
Checkpoint 1 Flowcharts:

tp1_team_1_15.pdf



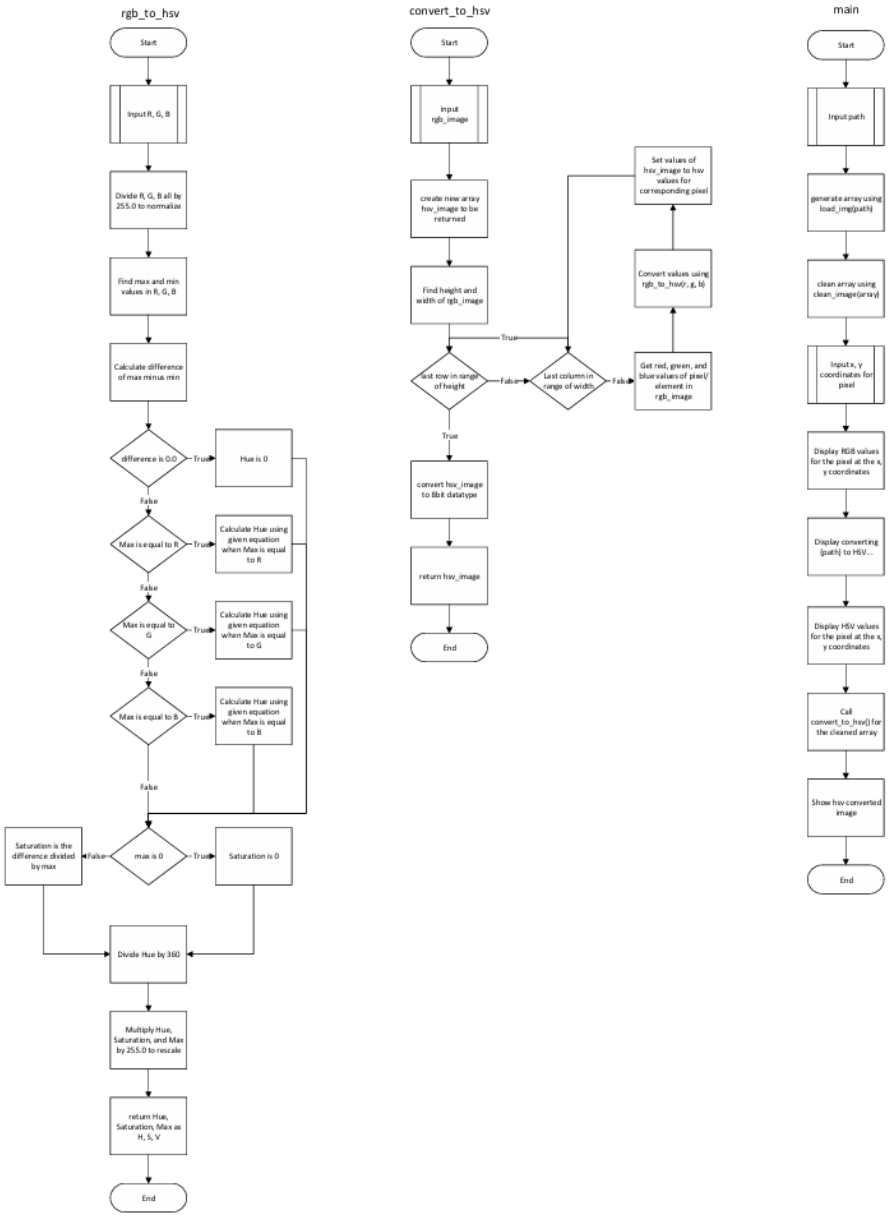
For this task, we designed a function to load images. To do so, we made a user defined function utalizing the Pillow library to load an image, convert it to RGB, then convert it to an array to work with. From there, we normalized the image and optionally converted it to greyscale. Doing so provided us with a modular and easy to use interface for loading images and converting them to arrays, which we heavily relied on later.

tp1_team_2_15.pdf



For this task, we designed a function that takes an image and then cleans and pads it. Doing so standardizes the arrays, preventing unwanted noise, outliers, or size differences between images. Doing this allows our later analysis to be more accurate, as all imported images share a size and are clean. Furthermore, defining this in a user defined function makes it easy to use later on.

tp1_team_3_15.pdf



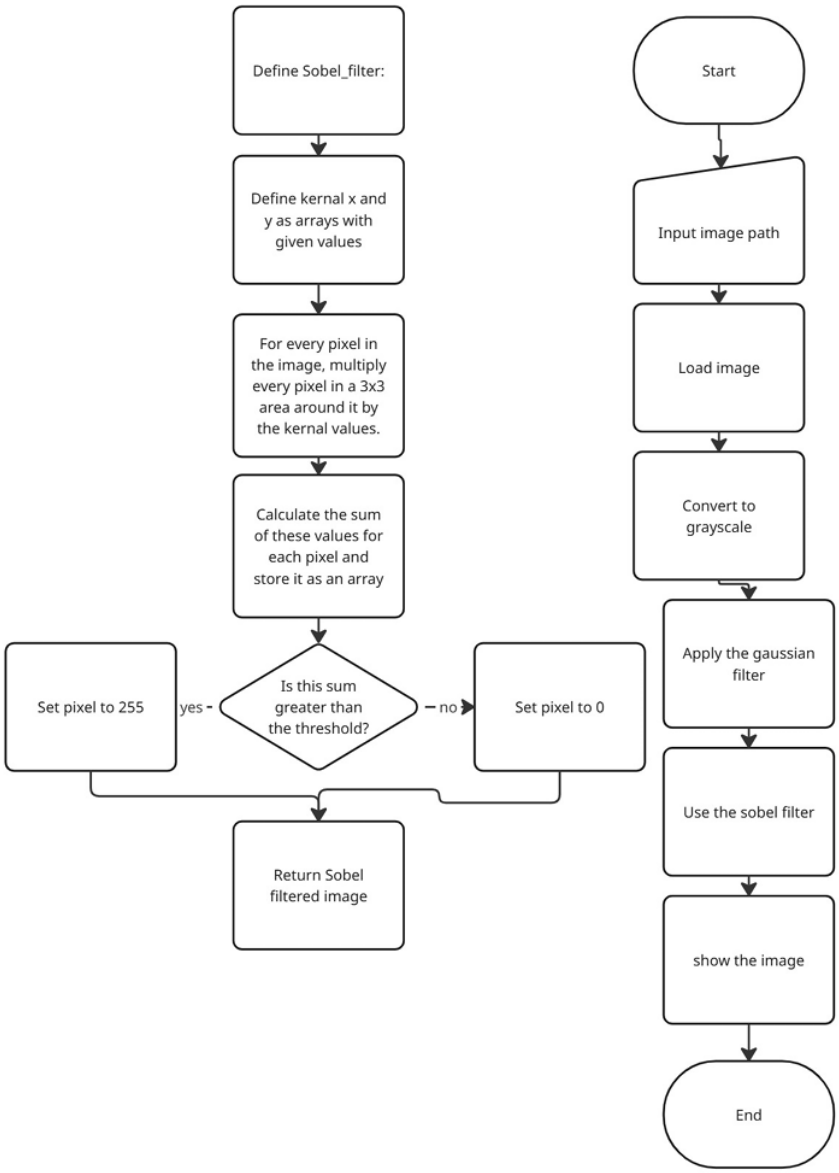
In this assignment, we created two functions in order to convert a whole image into HSV from RGB. First, we designed a function that iterates over a single pixel, determines the RGB values, then converts to HSV for that single pixel. From there, we created another function that iterates over every pixel in a cleaned and padded image, using the previous function to convert the entire image to HSV. By doing so, we make color detection much easier, as HSV images are far easier to detect changes in color than RGB ones.

Checkpoint 2 Flowcharts:

tp2_team_1_15.pdf

In this task, we designed a function to apply a gaussian filter to a greyscale image. To do so, we used the functions from previous tasks to load an image, pad it, and convert it to grayscale. From there, we defined a Gaussian filter kernel which we then applied to every pixel in the image, returning an array of a filtered image. Doing this blurs the image, making later detection of edges far easier and reducing noise. Therefore, the Gaussian filter is a vital part of many image processing applications.

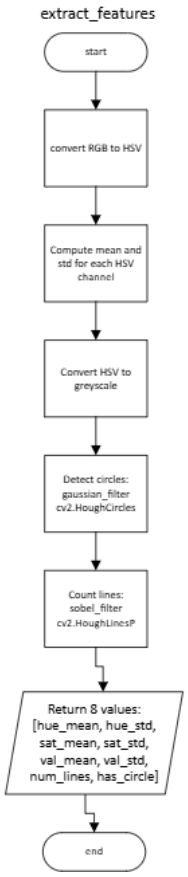
tp2_team_2_15.pdf



For this task, we designed a function that applies a Sobel filter to the image processed with a Gaussian process prior. To do so, we initialized a kernel for both the x and y gradients in the image, then used those values to find the Sobel filtered array for every pixel in a 3x3 area of each iterated pixel, eventually producing a new image with only the edges visible. Doing so allows us to easily analyze images based on the edges, as the function clearly defines them without any noise. Because of this fact, Sobel filters are necessary when trying to analyze the edges of multiple images.

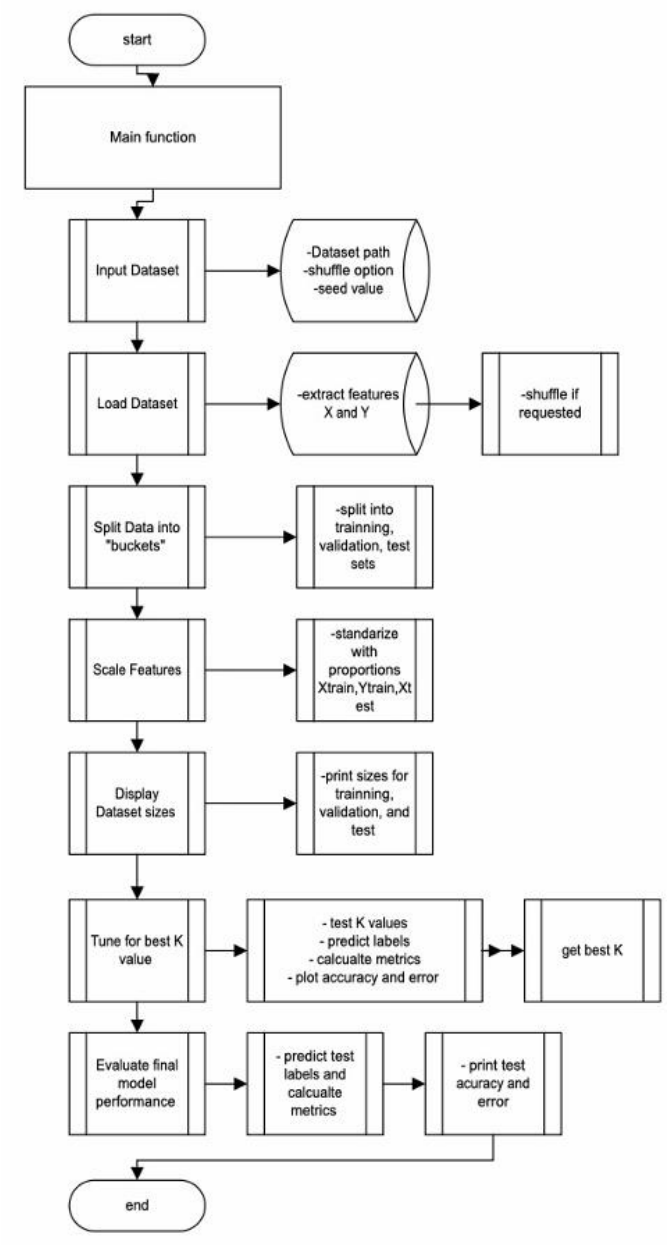
tp2_team_3_15.pdf

Unfortunately, our team was unable to complete task three. Because of this, we later used the provided CSV for our data. The following flow chart explains how the extract features function, which we got working works.



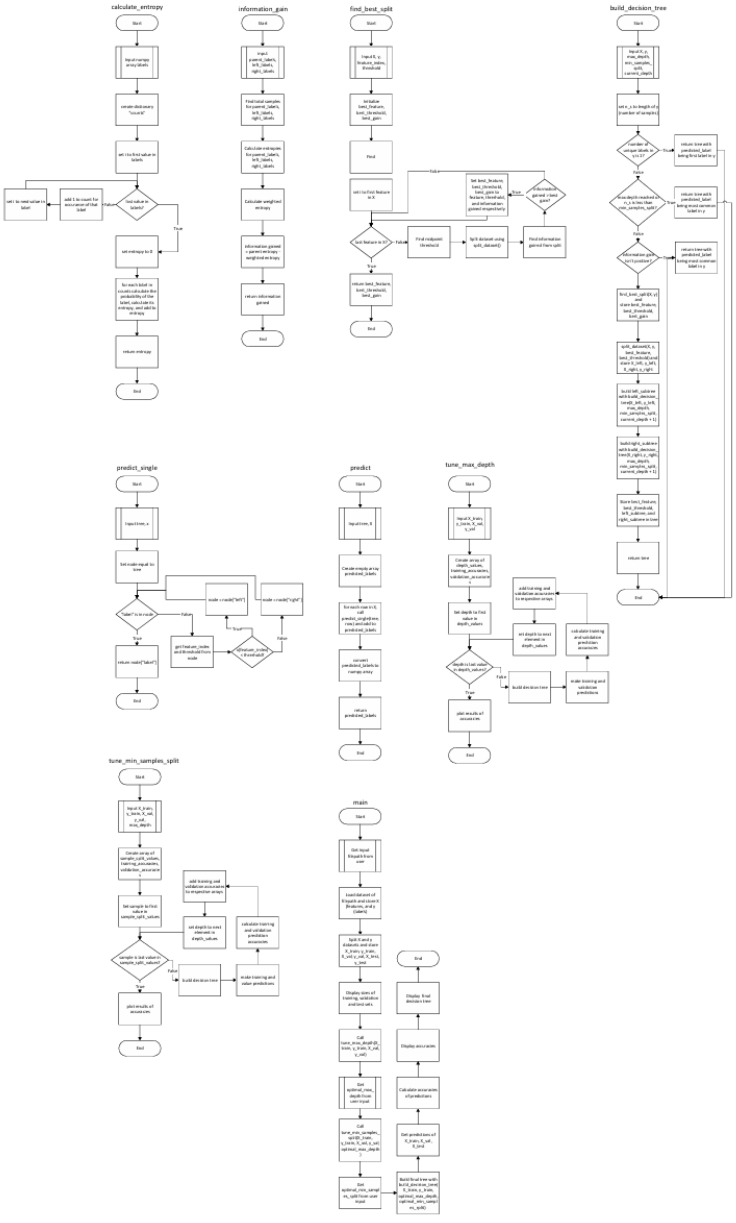
Checkpoint 3 Flowcharts:

tp3_team_1_15.pdf



For this assignment, we bult a KNN (K-Nearest Classifier) in order to use our model and detect stop signs. To do so, we made functions that loaded a dataset, trained the dataset, and predicted the results in order to analyze our data. In doing so, we received promising results. We received a training accuracy of 91 percent, which meant that our model was successful. However, this method of classification was not as accurate as the logistic regression seen below.

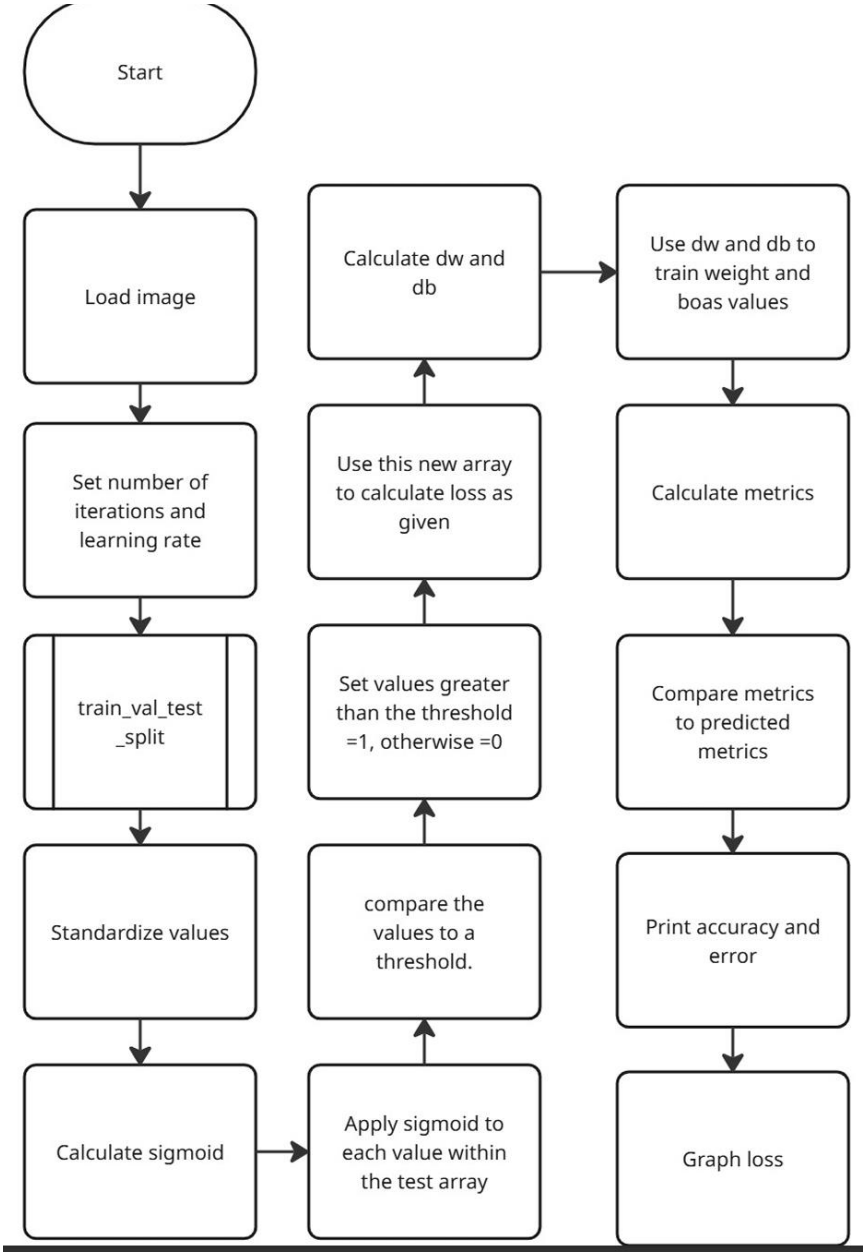
tp3_team_2_15.pdf



For this assignment, we used a decision tree classifier to analyze our data. Simply put, we defined functions to calculate the entropy of our data, the information gain from each split, and building the decision tree itself. The specifics of our design can be seen in the flowchart above. Ultimately, this

method gave us similar results to the KNN, just with a different method. This exposed us to the fact that there are many ways to solve the same problem within image processing.

tp3_team_3_15.pdf



Finally, we created a logistic regression program in order to analyze our data in a third way. This time, we calculated the probability of each value occurring, predicted the labels, calculated our loss and gradients, then used this data to train a model and compare results. Once again, this model gave us similar results to the other two. While similar, receiving similar results with different methods of classification revealed that there are many approaches to solving the same problem, simply using different methods for different use cases.

4. References

- Gardel, A., Molina, J., Garcia, I., and Bravo, J.L. "Applications of Image Processing in Robotics and Instrumentation." *Mechanical Systems and Signal Processing*, vol. 124, 2019, pp. 142-169. Elsevier/ScienceDirect. https://doi.org/10.1016/j.ymssp.2018.05.048
- Averroes AI. "AI Visual Inspection for PCB & Electronics Manufacturing." *Averroes Technologies*, 2024. https://averroes.ai/industry/ai-electronics. Accessed 23 Oct. 2025.
- Trax Retail. "Revolutionizing Retail Through AI-Driven Data and Consumer Engagement." *Trax Retail*, 2025. https://traxretail.com/. Accessed 23 Oct. 2025.

5. Appendix

1. User manual

Checkpoint 1:

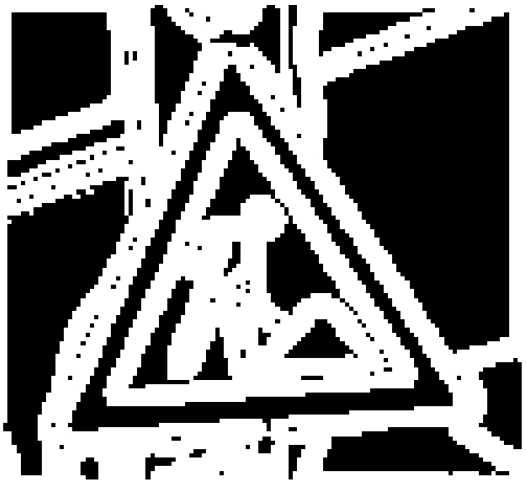
- ***Task 1-input an image to load and confirm if you want it to be converted to grayscale***
- 
-
```
PS C:\Users\henry\OneDrive\ENGR_133\ENGR 133\TP_1> & C:/Users/henry/AppData/Local/Microsoft/WindowsApps/python3.11.exe
 "c:/Users/henry/OneDrive/ENGR_133/ENGR 133/TP_1/tp1_team_1_15.py"
Enter the path of the image you want to load: ref_col.png
Would you like to convert to grayscale?
yes
PS C:\Users\henry\OneDrive\ENGR_133\ENGR 133\TP_1>
```
- ***Task 2 – input an image to clean and resize***

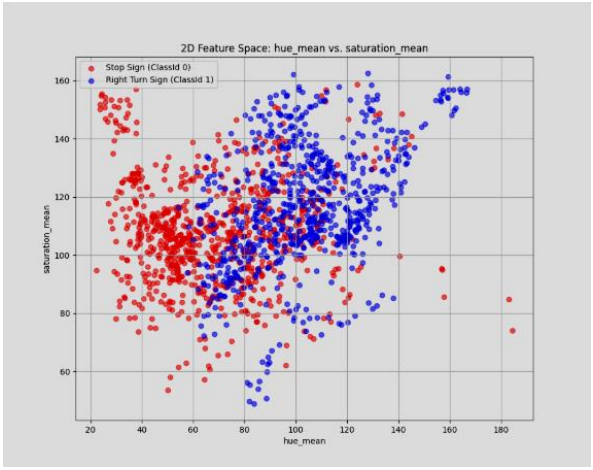
- 
-
```
Enter the path of the image you want to clean: ref_col_raw0.png
Image shape before cleaning: (112, 121, 3)
Resized image to: (92, 100)
Image shape after cleaning: (100, 100, 3)
```

- ***Task 3 – input an RGB image to be converted to HSV and input XY coordinates of a pixel to be inspected***

- 
-
```
Enter the path of the RGB image you want to convert to hsv: ref_col_raw4.png
Resized image to: (92, 100)
Enter the x and y coordinates of the pixel you want inspect: 50, 50
RGB values of the (50, 50) pixel: R=255, G=255, B=255
Converting ref_col_raw4.png to HSV...
HSV values of the (50, 50) pixel: H=0, S=0, V=255
```

Checkpoint 2:

- ***Task 1 – input an image to be blurred using a gaussian filter***
- 
- ```
  Enter the path to the image file: ref_col.png
  ```
- ***Task 2 – input an image to define its edges using a sobel filter***
- 
- ```
  Enter the path to the image file: ref_col.png
  ```
- ***Task 3 – input a dataset folder, metadata file, csv feature file, and x and y axis features***

- 


- ```
Enter the name of the dataset folder:
ML_Images
Enter the name of the metadata file:
img_metadata.csv
Enter the name of output csv features file:
img_features.csv

Training Metadata DataFrame:
   Width Height ClassId   Path
0   26   29   1 1_00015_00000.png
1   27   30   1 1_00015_00001.png
2   27   30   1 1_00015_00002.png
3   28   31   1 1_00015_00003.png
4   28   30   1 1_00015_00004.png

Features have been extracted and saved to img_features.csv

Feature Dataset Shape:
(1469, 10)

Feature Dataset Head:
   hue_mean   hue_std saturation_mean saturation_std value_mean value_std num_lines has_circle   Path ClassId
0   77.5901 74.895838   87.5351   76.634594   16.0337   17.254430   0.0   0.0 1_00015_00000.png   1
1   71.8692 73.134396   91.1982   74.631321   16.4933   17.784649   0.0   0.0 1_00015_00001.png   1
2   78.3621 73.230186   91.6578   76.665620   16.7615   17.559311   0.0   0.0 1_00015_00002.png   1
3   77.3715 71.860282   92.6498   75.498783   16.5275   17.430635   0.0   0.0 1_00015_00003.png   1
4   81.7215 70.873320   101.1194   77.180979   17.8706   18.428946   0.0   0.0 1_00015_00004.png   1

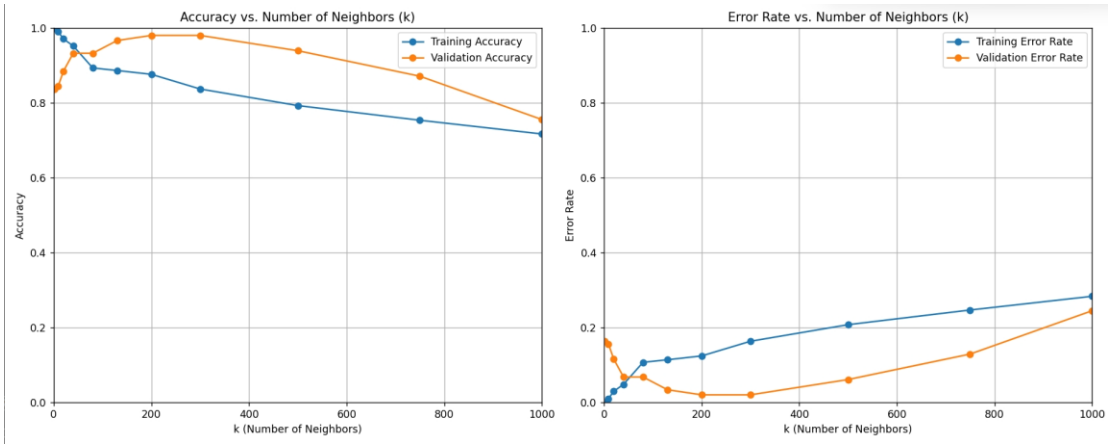
Enter the column to use as the x-axis feature:
hue_mean

Enter the column to use as the y-axis feature:
saturation_mean

Scatter plot saved to KNN_feature_space.png
```

***Checkpoint 3:***

- ***Task 1 – input csv file, whether to shuffle, and seed number***
```

- 

-
```
Enter the path to the feature dataset: img_features.csv
Shuffle the dataset? (yes/no): no
Enter a seed for loading the dataset: 70

Data loaded and split into

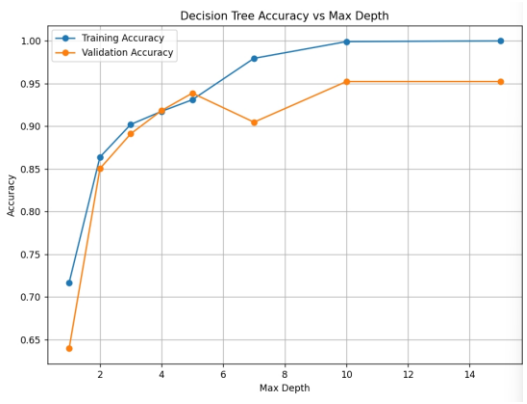
Training set:
  size: 1175
Validation set:
  size: 147
Test set:
  size: 147

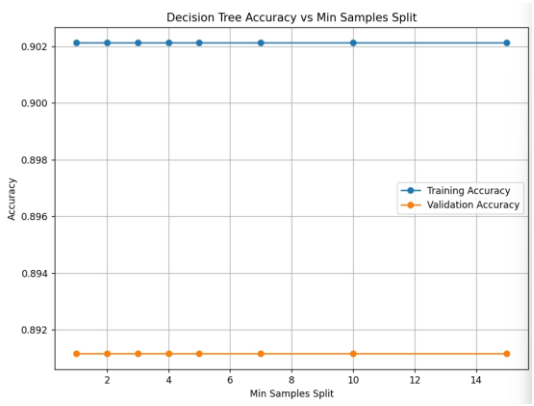
Based on the plots, the best k appears to be: 200

Evaluating final model on test set with k = 200...

--- Final Model Performance ---
Test Set Accuracy: 0.6667
Test Set Error Rate: 0.3333
```

- ***Task 2 – input feature dataset, optimal max depth, and optimal min samples***

- 
Decision Tree Accuracy vs Max Depth
Training Accuracy
Validation Accuracy
Accuracy
Max Depth
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
2
4
6
8
10
12
14


Decision Tree Accuracy vs Min Samples Split
Training Accuracy
Validation Accuracy
Accuracy
Min Samples Split
0.902
0.900
0.898
0.896
0.894
0.892
2
4
6
8
10
12
14

```
Enter the path to the feature dataset: img_features.csv

Data loaded and split:
Training set:  size: 1175
Validation set:  size: 147
Test set:  size: 147

Finding optimal tree max depth and min samples split...

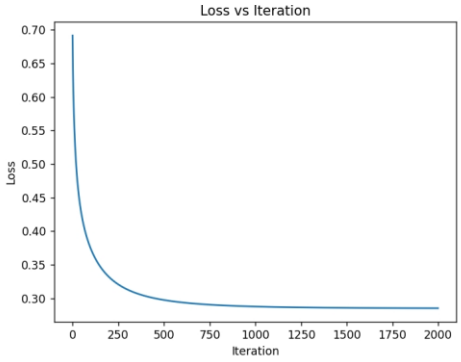
Enter the optimal max depth according to your interpretation of the plot:
3

Enter the optimal min samples split according to your interpretation of the plot:
30
Evaluating final model on test set with the chosen hyperparameters

--- Final Model Performance ---
Test Set Accuracy:       0.8912
Validation Set Accuracy: 0.8912
Training Set Accuracy:   0.9021

--- Final Decision Tree Rules ---
If hue_mean < 61.62:
  If hue_mean < 58.24:
    Predict: Stop Sign
  Else (hue_mean > 58.24)
    If value_mean < 43.09:
      Predict: Stop Sign
    Else (value_mean > 43.09)
      Predict: Stop Sign
Else (hue_mean > 61.62)
  If hue_std < 82.89:
    If hue_mean < 70.80:
      Predict: Right Turn Sign
    Else (hue_mean > 70.80)
      Predict: Right Turn Sign
  Else (hue_std > 82.89)
    If num_lines < 15.50:
      Predict: Right Turn Sign
    Else (num_lines > 15.50)
      Predict: Stop Sign
```

- ***Task 3 – outputs logistic regression model***



```
Iteration 0, Loss: 0.6916
Iteration 100, Loss: 0.3735
Iteration 200, Loss: 0.3314
Iteration 300, Loss: 0.3129
Iteration 400, Loss: 0.3030
Iteration 500, Loss: 0.2973
Iteration 600, Loss: 0.2936
Iteration 700, Loss: 0.2912
Iteration 800, Loss: 0.2895
Iteration 900, Loss: 0.2883
Iteration 1000, Loss: 0.2875
Iteration 1100, Loss: 0.2869
Iteration 1200, Loss: 0.2864
Iteration 1300, Loss: 0.2860
Iteration 1400, Loss: 0.2858
Iteration 1500, Loss: 0.2856
Iteration 1600, Loss: 0.2854
Iteration 1700, Loss: 0.2853
Iteration 1800, Loss: 0.2852
Iteration 1900, Loss: 0.2851
Validation accuracy: 0.9048
Test accuracy 0.9184
Validation error 0.0952
Test error 0.0816
```

-

2. **Project management plan** *(*Team members' contribution*)*

Our methods for collaboration during this team project included organizing the tasks we needed to get done by the weekly deadlines and continuously communicating in our group chat to keep us on track, and check if anyone needed support. By working on our tasks together, we ultimately supported each other in every step of the way and learned from each other's skills and insights as we worked. As we worked on tasks together, this involved delegating certain small components of the task between members and continuously checking on their progress and understanding of it, to then bring all components together. The general contribution of each team member was the following; Henry focused on debugging and communicating task progression with the team. He led the coding on the

decision tree model and the gaussian filter implementation. One improvement for Henry would be to add more thorough commenting on his code to help others understand its functions. Agustin led the coding for Checkpoint 1 Task 1 along with Checkpoint 3 Task 1 and worked closely to answer project report questions. One improvement for Agustin would be to be more detailed when creating flowcharts to convey the right processes to users. Aleks led the coding for Checkpoint 1 Task 3, Checkpoint 2 Task 2, and Checkpoint 3 Task 3. Additionally, Aleks significantly contributed to checkpoint 3 task 1 after errors with our code. One improvement for Aleks would be to write cleaner and more standard code, allowing team members to read it easier. Felipe led the coding for Checkpoint 2 Task 3, helped with Checkpoint 3 Task 1, and worked closely with team members on other smaller pieces of code in other tasks. One improvement for Felipe would be to work farther away from the deadline to have enough time to debug and get help from teammates if needed.

3. **Discussion of design process** *(Approach to the design process)*

Our design process fundamentally involves working together to assess the general project, idea, and specific tasks assigned, and then following a design-oriented process to complete this project. This process first included talking about our approach and how we would tackle each task, and how do the tasks complement each other, as in both checkpoint 1 and 2, many of the latter tasks like #2, #3, included using functions completed in task #1, and continued to improve on them, or modify their usage. This is why our approach had to be sequential or progressive, rather than recur to delegation and separation of tasks. After this, it was important to create flowcharts for each task, to have a conceptual understanding of what we had to do, and the functions we had to use, and after deliberation on our approach, we got to work. This part was the bulk of the project, coding our specific

functions and main functions, and after we had a rough working draft, we also stepped back and thought if this was the intended way to do it, or if there was an easier, more efficient way too.

4. Code

tp1_team_1_15.py

```
import numpy as np

from PIL import Image

import matplotlib.pyplot as plt



# load image

def load_img(path):

  try: #Try function to check for errors; looked up method to check for errors.

    img = Image.open(path)

  except FileNotFoundError: #Says that if a file not found error is detected to print an error.

    print(f"error {path} not found.")

    return None

  # remove 4th channel

  if img.mode == 'RGBA':

    img=img.convert('RGB')
```

```
    #convert to array

    pixel_array = np.array(img)

    # normalize

    normalized_array = pixel_array/255.0

    # Linearizing the Image

    img_arr = np.where(normalized_array <= 0.04045,normalized_array /
12.92,((normalized_array + 0.055) / 1.055) ** 2.4)

    #unnormalizes and makes sure it's 8bit data

    img_array=(img_arr*255).astype(np.uint8)

    return img_array



# grayscale conversion

def rgb_to_grayscale(img_array):

    # check for color image

    if img_array.ndim != 3:

        print("Error")

        return img_array

    #gets color channels

    red= img_array[:, :, 0]
```

```
    green= img_array[:, :, 1]
    blue = img_array[:, :, 2]
    #appends array of gray values
    gray_arr = (0.2126 * red) + (0.7152 * green) + (0.0722 * blue)
    #makes sure it's 8bit
    gray_array=(gray_arr).astype(np.uint8)
    return gray_array


# main
def main():
    path = input("Enter the path of the image you want to load: ")
    loaded_img = load_img(path)
    final_img = loaded_img
    # if color, convert?
    if loaded_img.ndim == 3: #so a color one will have 3 channels
        choice = input("Would you like to convert to grayscale?\n")
        if choice == 'yes':
            final_img = rgb_to_grayscale(loaded_img)
```

```
    #cmap for gray
    if final_img.ndim == 2: # so gray
        plt.imshow(final_img, cmap='gray')
    else:
        plt.imshow(final_img)
    plt.axis('off')
    plt.show()


if __name__ == "__main__":
    main()
```

tp1_team_2_15.py

```
from PIL import Image, ImageOps
import numpy as np
from tp1_team_1_15 import load_img
import matplotlib.pyplot as plt


def clean_image(arr):
    img = Image.fromarray(arr) #convert array back to image
```

```
        width, height = img.size



        aspect_ratio = width/height

        #resize image based on aspect ratio

        if aspect_ratio > 1: #wider than tall

            new_w = 100

            new_h = int(100/aspect_ratio)

        elif aspect_ratio < 1: #taller than wide

            new_w = int(100*aspect_ratio)

            new_h = 100

        else: #square

            new_w = 100

            new_h = 100

        img = img.resize((new_w, new_h), Image.BILINEAR) #resizes image

        print(f"Resized image to: {(img.size[1], img.size[0])}") #lists height then width to match
samples

        if img.mode == "RGB": #creates fill color for color images or grayscale images

            fill_color = (0, 0 , 0)

        else:
```

```
            fill_color = 0

        img_padded = ImageOps.pad(img, (100, 100), method = Image.BILINEAR, color = fill_color)
#ImageOps.pad pads the image to the given dimensions with given color

        new_img = np.array(img_padded) #convert back to array

        return new_img #returns cleaned array




    def main():

        path = input("Enter the path of the image you want to clean: ") #takes input

        img_array = load_img(path)

        print(f"Image shape before cleaning: {img_array.shape}") #dimensions of array

        cleaned_img = clean_image(img_array) #cleans image

        print(f"Image shape after cleaning: {cleaned_img.shape}")

        if cleaned_img.ndim == 3: #display image for color or grayscale

            plt.imshow(cleaned_img)

        else:

            plt.imshow(cleaned_img, cmap = "gray")

        plt.axis('off')

        plt.show()
```

```
if __name__ == "__main__":
  main()
```

tp1_team_3_15.py

```
from pathlib import Path
from PIL import Image # Importing libraries
import numpy as np
import matplotlib.pyplot as plt
from tp1_team_1_15 import load_img
from tp1_team_2_15 import clean_image


def rgb_to_hsv(r,g,b):
  r2=r/255.0 #Added .0 to every number to try to fix floating point differences.
  b2=b/255.0 #Normalizing values
  g2=g/255.0
  cmax=max(r2,g2,b2) #Finding the max and minimum RGB value and their difference
  cmin=min(r2,g2,b2)
  difference=cmax-cmin
  if difference == 0.0:
```

```
    H2=0.0
  elif cmax==r2: #Defining the Hue based on the most prominent RGB value
    H2=(60.0*(((g2-b2)/difference))%360.0) #Added 360 to account for negatives.
  elif cmax==g2:
    H2=((60.0*((b2-r2)/difference))+120.0)%360.0
  elif cmax==b2:
    H2=((60.0*((r2-g2)/difference))+240.0)%360.0
  if cmax == 0:
    S2=0.0 #Defining Saturation (0 if black)
  else:
    S2 = (difference/cmax)
  V2=cmax #The value is just the max rgb value
  H=(H2/360.0)*255.0 #Not entirely sure why H2 must be divided by 360 but the assignment says so.
  S=S2*255.0 #Scaling the values back up.
  V=V2*255.0
  return H,S,V


def convert_to_hsv(rgb_image):
```

```
    hsv_im=rgb_image.astype(float).copy() #Copying the rgb array to a new array to work on;
converting to float from unassigned integer.

    shape=hsv_im.shape #Finding the shape

    h=shape[0]

    w=shape[1]

    for i in range(h): #For loop to change every pixel (done in previous assignments)

        for k in range(w):

            r= rgb_image[i,k,0]

            g= rgb_image[i,k,1]

            b= rgb_image[i,k,2]

            H,S,V=rgb_to_hsv(r,g,b) #Actually converting

            hsv_im[i,k,0]=H

            hsv_im[i,k,1]=S #Actually converting each one to HSV

            hsv_im[i,k,2]=V

    hsv_image=(hsv_im).astype(np.uint8) #Converting back to uint8

    return hsv_image
```

```
def main():

  path = input("Enter the path of the RGB image you want to convert to hsv: ") #Defining file

  arr=load_img(path)

  clean_img=clean_image(arr) #Running the udfs

  inp=(input("Enter the x and y coordinates of the pixel you want inspect: "))

  x,y=inp.split(',') #Splitting the input to seperate x and y. Did this before in the assignment where we had to split up the arrays. Pretask something i'm pretty sure.

  R,G,B=clean_img[int(x),int(y)].astype(int)

  print(f'RGB values of the ({x},{y}) pixel: R={R}, G={G}, B={B}')

  print(f'Converting {path} to HSV...') #All self explanatory.

  H,S,V=rgb_to_hsv(R,G,B)

  print(f'HSV values of the ({x},{y}) pixel: H={int(H)}, S={int(S)}, V={int(V)}') #Used int instead of :.0f to get autograder to work.

  hsv_img=convert_to_hsv(clean_img) #Converting to HSV

  plt.imshow(hsv_img)

  plt.axis('off')

  plt.show()
```

```
if __name__ == "__main__":

  main()
```

tp2_team_1_15.py

```
import numpy as np

from PIL import Image

import math

from tp1_team_1_15 import load_img

from tp1_team_1_15 import rgb_to_grayscale

import matplotlib.pyplot as plt



def gaussian_filter(gry_arr, std_dev):

  gry_arr = gry_arr.astype(np.float64)

  kernel_size = int(2*math.ceil(3*std_dev) + 1) #calculate kernel size

  pad_size = int((kernel_size - 1)/2) #calculate size to pad image by



  x, y = np.meshgrid(np.arange(-(kernel_size//2), kernel_size//2 + 1), np.arange(-
(kernel_size//2), kernel_size//2 + 1)) #creates grid for x and y values

  kernel = (np.exp(-(x**2+y**2)/(2 * std_dev**2))) #uses gaussian function that gives weights
for each pixel in kernel
```

```
    kernel = kernel/np.sum(kernel, dtype = np.float64) #normalize values to be within 0, 1


    padded_arr = np.pad(gry_arr, pad_size, mode = "constant")#creates padded array with zeros
    height, width = gry_arr.shape
    output = np.zeros((height, width), dtype = float) #creates blank output array


    #iterates through padded array
    for r in range(height):
       for c in range(width):
          region = padded_arr[r:r + kernel_size, c:c + kernel_size] #gets 7x7 region of pixels
          output[r, c] = np.sum(region * kernel) #applies weighted pixel to output array


    return (np.clip(output, 0, 255)).astype(np.uint8) #makes sure it's 8bit


def main():
    path = input("Enter the path to the image file: ")
    img = load_img(path)
    gry_arr = rgb_to_grayscale(img)
```

```
    std_dev = 1

    filtered_arr = gaussian_filter(gry_arr, std_dev) #blurs image



    plt.imshow(filtered_arr, cmap = "gray") #show image

    plt.axis('off')

    plt.show()




if __name__ == "__main__":

    main()
```

tp2_team_2_15.py

```
from pathlib import Path

from PIL import Image # Importing libraries

import numpy as np

from tp1_team_1_15 import load_img

from tp1_team_1_15 import rgb_to_grayscale #Importing all the files this thing needs

import matplotlib.pyplot as plt
```

```
from tp2_team_1_15 import gaussian_filter



def sobel_filter (gaussian_filter):
  blur_pad=np.pad(gaussian_filter, (1,1)) #Using the pad function to pad the filtered image.
  kernalx=np.array([[-1,0,1],[-2,0,2],[-1,0,1]]) #Defining the two kernals
  kernaly=np.array([[-1,-2,-1],[0,0,0],[1,2,1]])
  shape=gaussian_filter.shape
  h=shape[0] #Getting the shape of the array and using that to define the height and width
  w=shape[1]
  i=0 #Setting the indices to 0
  k=0
  gx=np.zeros((h,w)) #Forming two zero arrays so I don't have to overwrite existing arrays
  gy=np.zeros((h,w))
  for i in range(h):
    for k in range(w): #For loops that go over every pixel in the image
      sum1=0 #Reseting sums for each pixel
      sum2=0
      for m in range(3):
```

```
                for l in range(3): #Setting another loop that goes over kernals

                    sum1+=blur_pad[i+m,k+l]*kernalx[m,l] #Multiplying each pixel in a 3x3 area of the target pixel by the kernal

                    sum2+=blur_pad[i+m,k+l]*kernaly[m,l] #Same here

            gx[i,k]=sum1 #Summing up all the values to find the gradient

            gy[i,k]=sum2

        gmag=np.sqrt(gx**2+gy**2) #Self explanatory - gives the magnitude of the gradient at each pixel.

        threshold=50 #Manual threshold setting

        for i in range(h):

          for k in range(w): #Loop that checks every pixel

            if gmag[i,k]>=threshold:

                gmag[i,k]=255 #Loop that says that if gmag is greater than the threshold, the pixel has max brightness. Otherwise it is black.

            else:

                gmag[i,k]=0

        return gmag.astype(np.uint8) #Returning the sobel filtered image.
```

```
def main():
  path = input("Enter the path to the image file: ") #Self explanatory.
  img = load_img(path)
  gry_arr = rgb_to_grayscale(img)
  std_dev = 1
  filtered_arr = gaussian_filter(gry_arr, std_dev) #All of this has been done in other parts.
  sobel=sobel_filter(filtered_arr)
  plt.imshow(sobel, cmap="gray")
  plt.axis('off')
  plt.show()


if __name__ == "__main__":
  main()
```

tp2_team_3_15.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import cv2

from tp1_team_1_15 import load_img

from tp1_team_1_15 import rgb_to_grayscale

from tp1_team_2_15 import clean_image

from tp1_team_3_15 import rgb_to_hsv

from tp1_team_3_15 import convert_to_hsv

from tp2_team_1_15 import gaussian_filter

from tp2_team_2_15 import sobel_filter



# extract features function (HSV mean and sd)

def extract_features(new_img):

  # converts the input RGB image to the HSV color space

  new_hsv = convert_to_hsv(new_img)



  # calculates the mean and standard deviation for each of the three channels of the HSV image

  hue_mean = np.mean(new_hsv[:,:,0])

  hue_std = np.std(new_hsv[:,:,0])

  saturation_mean = np.mean(new_hsv[:,:,1])
```

```
saturation_std = np.std(new_hsv[:,:,1])

value_mean = np.mean(new_hsv[:,:,2])

value_std = np.std(new_hsv[:,:,2])



# convert the rgb image to grayscale

gray_img = rgb_to_grayscale(new_img)



def detect_circles(gray_img):

   """Detects if a large circle is present. Returns 1 if found, 0 otherwise."""

   # Hough Circles works best on a grayscale, slightly blurred image

   # Apply a Gaussian blur

   blurred_img = np.array(gaussian_filter(gray_img, sigma=1.5))



    # Detect circles

   circles = cv2.HoughCircles(

       blurred_img,

      cv2.HOUGH_GRADIENT,

      dp=1.2,  # Inverse ratio of accumulator resolution
```

```
        minDist=100,  # Minimum distance between centers of detected circles
        param1=150,  # Upper threshold for the internal Canny edge detector
        param2=50,  # Threshold for center detection
        minRadius=20,  # Minimum circle radius to detect
        maxRadius=50  # Maximum circle radius to detect
    )

    return 1 if circles is not None else 0


def count_lines(gray_img):
    """Counts the number of lines in an image using Hough Line Transform."""
    # Edge detection via sobel filtering is a prerequisite for Hough Lines
    edges = sobel_filter(gray_img)

    # Detect lines using the edge map
    lines = cv2.HoughLinesP(edges, 1, np.pi / 180, threshold=50, minLineLength=30,
maxLineGap=10)

    return len(lines) if lines is not None else 0
```

```
    # runs UDFs to get the number of lines and detect the shape of the grayscale image

    number_lines = count_lines(gray_img)

    has_circles = detect_circles(gray_img)



    return hue_mean, hue_std, saturation_mean, saturation_std, value_mean, value_std,
number_lines, has_circles



def main():

    # get inputs from the user

    folder = input("Enter the name of the dataset folder: ")

    name = input("Enter the name of the metadata file: ")

    output = input("Enter the name of output csv features file: ")



    # load the metadata csv -- used Claude AI

    metadata_path = f"{folder}/{name}"



    try:

        df = pd.read_csv(metadata_path)
```

```
    except:
        print(f"Error: Could not find file at '{metadata_path}'")
        print(f"Please make sure the folder '{folder}' exists and contains '{name}'")
        return


print("Training Metadata DataFrame:")
print(df.head())
print()


def process_row(row, folder):
    # creates the image path for each row
    image_path = f"{folder}/{row['Path']}"

    # loads the image
    loaded_img = load_img(image_path)

    # cleans the image
    cleaned_img = clean_image(image_path)
```

```
    # extracts the 8 features from the cleaned image

    hue_mean, hue_std, sat_mean, sat_std, val_mean, val_std, num_lines, has_circles =
extract_features(cleaned_img)



    # returns the 10 values (2 from the original row and 8 extracted features)

    return pd.Series({

      'Path': row['Path'],

      'ClassId': row['ClassId'],

      'hue_mean': hue_mean,

      'hue_std': hue_std,

      'saturation_mean': sat_mean,

      'saturation_std': sat_std,

      'value_mean': val_mean,

      'value_std': val_std,

      'number_lines': num_lines,

      'has_circles': has_circles

    })
```

```
# applies the process row function to each row

features_df = df.apply(process_row, axis=1)


# saves to csv

features_df.to_csv(output, index=False)

print(f"Features have been extracted and saved to {output}")


# prints the shape and head of final dataset

print("\nFinal Dataset Shape:", features_df.shape)

print("\nFinal Dataset Head:")

print(features_df.head())

print()


# initialize variables for scatter plot from user input

#x = input("Enter the column to use as the x-axis feature: ")
```

```
    #y = input("Enter the column to use as the y-axis feature: ")



    # creates the scatter plot

    #plt.scatter(x,y, color='blue', marker='o', s=100, alpha=0.7, edgecolors='black')

    #plt.scatter(x,y, color='blue', marker='o', s=100, alpha=0.7, edgecolors='black')




    # adds title and axis labels

    #plt.title(f"2D Feature Space: {x} vs. {y}")

    #plt.xlabel(f"{x}")

    #plt.ylabel(f"{y}")

    #plt.legend()

    #plt.grid(True)

    #plt.show



if __name__ == "__main__":

  main()
```

tp3_team_1_15.py

```
from pathlib import Path

from PIL import Image # Importing libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt



def load_dataset(file_path, feature_cols, label_col, shuffle, seed=70):

df = pd.read_csv(file_path)

X = df[feature_cols].to_numpy()

y = df[label_col].to_numpy()



if shuffle:

   np.random.seed(seed)

   shuffled_indices = np.random.permutation(len(y)) #This function is taken directly from the examples

   X = X[shuffled_indices]

   y = y[shuffled_indices]



return X, y
```

```
def train_val_test_split(X, y, train_ratio=0.8, val_ratio=0.1, test_ratio=0.1):
  shape=X.shape #Defines the shape of the X matrix
  samples=shape[0]
  X_train=[]
  y_train=[]
  X_val=[] #Setting up empty arrays to later iterate upon
  y_val=[]
  X_test=[]
  y_test=[]
  trainrange=round(samples*train_ratio) #Calculating the range for the training set using the ratio. Using round cause it didn't match with int
  valrange=trainrange+round((val_ratio*samples)) #Same as above
  X_train=X[0:trainrange]
  y_train=y[0:trainrange]
  X_val=X[trainrange:valrange] #Slicing the matrices using the ranges set above
  y_val=y[trainrange:valrange]
  X_test=X[valrange:samples]
  y_test=y[valrange:samples]
```

```
    return X_train, y_train, X_val, y_val, X_test, y_test



def scale_features(X_train,X_val,X_test):

    mean=np.mean(X_train,axis=0) #Finding mean + stddev

    std=np.std(X_train,axis=0)

    X_train_scaled=(X_train-mean)/std #Scaling with the same values for each

    X_val_scaled=(X_val-mean)/std

    X_test_scaled=(X_test-mean)/std

    return X_train_scaled,X_val_scaled,X_test_scaled



def calculate_metrics (predicted_labels,true_labels):

    acc=np.mean(predicted_labels==true_labels) #Calculating accuracy as the mean amount of
labels that equal the true labels

    err=1-acc #Self explanatory

    return acc, err



def knn_single_prediction(new_example,X_train,y_train,k):

    difference=X_train-new_example #Creating a difference matrix to then use linalg.norm on

    dist=[]
```

```
    for i in range(len(X_train)):

        dist.append((np.linalg.norm(difference[i]), y_train[i])) #Adding each linalg.norm value for each element in the array. Had to look up what linalg.norm did

    dist.sort() #Sorting the distances

    closest=dist[0:k] #Saying the closest ones the ones with distances from 0 to k

    labels=[label for(_, label) in closest] #Had help from groupmembers for this one

    values,count=np.unique(labels, return_counts=True) #Finding unique values and counts

    max_count=max(count)

    for i in range(len(count)):

        if count[i]==max_count: #Saying that if the count is the max count, then the predicted label is the value at i. Couldn't figure out another way to do this even though its strange

            predicted_label=values[i]

            break

    return predicted_label



def predict_labels_knn(X_new,X_train,y_train,k):

    predicted_labels=[]

    for i in range(len(X_new)):

        new_example=X_new[i]
```

```
    label=knn_single_prediction(new_example,X_train,y_train,k) #Iterating the label values
across a new array
    predicted_labels.append(label)
  return predicted_labels


def tune_k_values(k_values,X_train,y_train,X_val,y_val,is_shuffle):
  metrics = {
    "acc": {"train_acc": [], "val_acc": []},
    "error": {"train_error": [], "val_error": []} #Ripped this straight from the instructions
  }
  best_val=0
  for k in k_values:
    train_label=predict_labels_knn(X_train,X_train,y_train,k) #Getting the labels for each set
    val_label=predict_labels_knn(X_val,X_train,y_train,k)
    train_acc,train_err=calculate_metrics(train_label,y_train) #Calculating error for each set
    val_acc,val_err=calculate_metrics(val_label,y_val)
    metrics["acc"]["train_acc"].append(train_acc)
    metrics["acc"]["val_acc"].append(val_acc) #Appending the metrics
    metrics["error"]["train_error"].append(train_err)
```

```
    metrics["error"]["val_error"].append(val_err)

    if val_acc>best_val:

      k_good=k #Saying that for each iteration, if the current accuracy is better than the last, then the new k value is better

      best_val=val_acc #Same as before but with accuracies

  plot_knn_performance(metrics,k_values) #Plotting here cause I already defined the metrics in this function

  return k_good



def plot_knn_performance(metrics,k_values):

  fig,axes = plt.subplots(1, 2, figsize=(15, 6))

  axes[0].plot(k_values, metrics["acc"]["train_acc"], label="Training Accuracy", marker='o')

  axes[0].plot(k_values, metrics["acc"]["val_acc"], label="Validation Accuracy", marker='o') #Whole bunch of plot stuff

  axes[0].set_title("Accuracy vs. Number of Neighbors (k)")

  axes[0].set_xlabel("k (Number of Neighbors)")

  axes[0].set_ylabel("Accuracy")

  axes[0].legend()

  axes[0].grid(True)

  axes[1].plot(k_values, metrics["error"]["train_error"], label="Training Error Rate", marker='o') #All self explanatory
```

```
    axes[1].plot(k_values, metrics["error"]["val_error"], label="Validation Error Rate", marker='o')
    axes[1].set_title("Error Rate vs. Number of Neighbors (k)")
    axes[1].set_xlabel("k (Number of Neighbors)")
    axes[1].set_ylabel("Error Rate")
    axes[1].legend()
    axes[1].grid(True)
    axes[0].axis([0,1000,0,1.0])
    axes[1].axis([0,1000,0,1.0])
    plt.tight_layout() #To make it cleaner when it pops up
    plt.show()


def main():
    path=input("Enter the path to the feature dataset: ") #Importing
    shuffle=input("Shuffle the dataset? (yes/no): ")
    if shuffle =="yes": #Shuffle by using BOO! leans (Cause its halloween)
        shuff=True
    else:
        shuff=False
```

```
  seeds=int(input("Enter a seed for loading the dataset: ")) #Defining the seed

  features =
["hue_mean","hue_std","saturation_mean","saturation_std","value_mean","value_std","num_l
ines","has_circle"] #Defining the feature array

  label="ClassId" #Setting the label as ClassId

  X,y=load_dataset(path,features,label,shuff,seed=seeds) #Loading dataset

  X_train,y_train,X_val,y_val,X_test,y_test=train_val_test_split(X,y)

  X_train_scaled, X_val_scaled, X_test_scaled = scale_features(X_train, X_val, X_test) #Scaling
and getting matrices

  print("\nData loaded and split into")

  print("\n")

  print("Training set\n size:", X_train_scaled.shape[0])

  print("Validation set\n size:", X_val_scaled.shape[0]) #Whole lot of printing

  print("Test set\n size:", X_test_scaled.shape[0])

  k_values=[1, 9, 20, 40, 80, 130, 200, 300, 500, 750, 1000] #Initializing k values

  best_k=tune_k_values(k_values, X_train_scaled, y_train, X_val_scaled, y_val, shuff) #Tuning

  print(f"\nBased on the plots, the best k appears to be: {best_k:.0f}")

  print(f"\nEvaluating final model on test set with k = {best_k:.0f}...") #Same as before

  test_label=predict_labels_knn(X_test_scaled, X_train_scaled, y_train, best_k) #Labels

  test_acc, test_err = calculate_metrics(test_label, y_test) #Testing metrics
```

```
    print("\n--- Final Model Performance ---")
    print(f"Test Set Accuracy: {test_acc:.4f}")
    print(f"Test Set Error Rate: {test_err:.4f}") #Displaying results.


if __name__ == "__main__":
    main()
```

tp3_team_2_15.py

```
import numpy as np
import math as m
import pandas as pd
import matplotlib.pyplot as plt
from tp3_team_1_15 import load_dataset
from tp3_team_1_15 import train_val_test_split


def calculate_entropy(labels):
    counts = {} #dictionary to count occurances of labels
    for i in labels:
        counts[i] = counts.get(i, 0) + 1 #adds 1 for each occurance of i in count
```

```
    entropy = 0
    for count in counts.values(): #for every element in counts
        prob = count / len(labels) #probability of the label
        entropy += -prob*m.log2(prob) #calculates entropy of that label
    return entropy


def information_gain(parent_labels, left_labels, right_labels):


    n_s = len(parent_labels) #total samples
    n_L = len(left_labels) #total left samples
    n_R = len(right_labels) #total right samples


    #calculate entropies
    parent_entropy = calculate_entropy(parent_labels)
    left_entropy = calculate_entropy(left_labels)
    right_entropy = calculate_entropy(right_labels)
```

```
    weighted_entropy = (n_L/n_s)*left_entropy + (n_R/n_s)*right_entropy #calculated weighted
entropies



    info_gain = parent_entropy - weighted_entropy #info gain from reducing entropy



    return info_gain



#function given by task

def split_dataset(X, y, feature_index, threshold):



    """

    Splits a dataset based on a feature and threshold.

    """

    X_left, y_left, X_right, y_right = [], [], [], []



    for i, example in enumerate(X):

        if example[feature_index] < threshold:

            X_left.append(example)

            y_left.append(y[i])
```

```
        else:
            X_right.append(example)
            y_right.append(y[i])

    return np.array(X_left), np.array(y_left), np.array(X_right), np.array(y_right)

def find_best_split(X, y):

    best_feature = 0 #split parameters
    best_threshold = 0.0
    best_gain = -1

    num_features = X.shape[1] #number of features

    for feat_index in range(num_features): #for every feature
        features = X[:, feat_index]
        unique_features = np.unique(features) #finds all unique features
```

```
        for i in range(len(unique_features) - 1): #for every unique feature

            threshold = (unique_features[i] + unique_features[i+1])/2 #finds threshold via midpoint



            X_left, y_left, X_right, y_right = split_dataset(X, y, feat_index, threshold) #split dataset



            info_gain = information_gain(y, y_left, y_right) #calculate info gained from Y split



            if info_gain > best_gain: #finds split that gives greatest info gain

                best_gain = info_gain

                best_feature = feat_index

                best_threshold = threshold



    return best_feature, best_threshold, best_gain



def build_decision_tree(X, y, max_depth, min_samples_split, current_depth):



    n_s = len(y) #number of samples

    if len(np.unique(y)) == 1: #if theres only 1 unique label aka all same
```

```
        predicted_label = y[0]

        tree = {"label": predicted_label} #becomes leaf node

        return tree



    if current_depth >= max_depth or n_s < min_samples_split: #if max depth reached or not
enough samples



        counts = {} #dictionary to count occurances of labels

        for i in y:

            counts[i] = counts.get(i, 0) + 1 #adds 1 for each occurance of i in count



        common_label = 0

        count = 0



        for label, count2 in counts.items(): #finds label with max count aka most common



            if count2 > count:

                common_label  = label

                count = count2
```

```
        tree = {"label": common_label} #becomes leaf node

        return tree



    best_feature_index, best_threshold_value, best_gain = find_best_split(X, y) #finds conditions
to split



    if best_gain <= 0: #if no positive gain

        counts = {} #dictionary to count occurances of labels

        for i in y:

            counts[i] = counts.get(i, 0) + 1 #adds 1 for each occurance of i in count



        common_label = 0

        count = 0



        for label, count2 in counts.items(): #iterates through labels and their counts



            if count2 > count: #if current count is greater than previous count, assign new most
common label
```

```
            common_label  = label

            count = count2



        tree = {"label": common_label} #becomes leaf node

        return tree



    X_left, y_left, X_right, y_right = split_dataset(X, y, best_feature_index, best_threshold_value)
#splits the dataset



    left_subtree =  build_decision_tree(X_left, y_left, max_depth, min_samples_split,
current_depth + 1) #builds tree for left split

    right_subtree = build_decision_tree(X_right, y_right, max_depth, min_samples_split,
current_depth + 1) #builds tree for right split



    tree = {

    "feature": best_feature_index,

    "threshold": best_threshold_value,

    "left": left_subtree,

    "right": right_subtree

    } #saves parameters for each split
```

```
    return tree


#helper
def predict_single(tree, x):

    node = tree #top node of the tree

    while True: #continues until a leaf
        if "label" in node: #if leaf is reached
            return node["label"]

        feature_index = node["feature"] #get feature index and threshold
        threshold = node["threshold"]

        if x[feature_index] < threshold: #predict split
            node = node["left"]
        else:
```

```
            node = node["right"]


def predict(tree, X):
    predicted_labels = [] #array of prediction label
    for feature in X: #loops through rows
        prediction = predict_single(tree, feature) #finds feature label in row
        predicted_labels.append(prediction) #adds to array
    predicted_labels = np.array(predicted_labels) #converts to numpy array
    return predicted_labels


def calculate_accuracy(y_true, y_pred):
    accuracy  = sum(y_true == y_pred) / len(y_true) #proportion of correct predictions to total
predictions
    return accuracy


#function for plotting data
def plot_dt_performance(train_accs, val_accs, param_values, param_name):


    fig, axes = plt.subplots(1, 1, figsize=(8, 6))
```

```
    axes.plot(param_values, train_accs, marker='o', label='Training Accuracy')

    axes.plot(param_values, val_accs, marker='o', label='Validation Accuracy')

    axes.set_xlabel(param_name)

    axes.set_ylabel('Accuracy')

    plt.title(f"Decision Tree Accuracy vs {param_name}")

    axes.legend()

    axes.grid(True)



    plt.tight_layout()

    plt.show()



def tune_max_depth(X_train, y_train, X_val, y_val):

    depth_values = [1, 2, 3, 4, 5, 7, 10, 15] #range of depth values

    train_accuracies = []

    val_accuracies = []



    for depth in depth_values: #for each depth value

        tree = build_decision_tree(X_train, y_train, depth, min_samples_split = 2) #builds tree at
depth
```

```
    train_predictions = predict(tree, X_train) #makes predictions on training and validation sets
    val_predictions = predict(tree, X_val)

    train_acc = calculate_accuracy(y_train, train_predictions) #calculate prediction accuracies
    val_acc = calculate_accuracy(y_val, val_predictions)

    train_accuracies.append(train_acc) #adds accuracies to arrays
    val_accuracies.append(val_acc)

  plot_dt_performance(train_accuracies, val_accuracies, depth_values, "Max Depth") #plots accuracy

def tune_min_samples_split(X_train, y_train, X_val, y_val, max_depth):
  sample_split_vals = [1, 2, 3, 4, 5, 7, 10, 15] #range of split values
  train_accuracies = []
  val_accuracies = []

  for sample in sample_split_vals:
```

```
        tree = build_decision_tree(X_train, y_train, max_depth, sample) #builds tree with split value



        train_predictions = predict(tree, X_train) #makes predictions

        val_predictions = predict(tree, X_val)



        train_acc = calculate_accuracy(y_train, train_predictions) #accuracies of prediction

        val_acc = calculate_accuracy(y_val, val_predictions)



        train_accuracies.append(train_acc) #stores accuracies

        val_accuracies.append(val_acc)



    plot_dt_performance(train_accuracies, val_accuracies, sample_split_vals, "Min Samples Split")
#plots accuracies



#function to print tree out

def print_tree(tree, feature_names, indent=""):

    """

    Recursively prints the decision tree in a readable format.

    """
```

```
    # Base case: we have reached a leaf node
    if "label" in tree:
        # Map the numeric label to a meaningful name
        label_name = "Stop Sign" if tree["label"] == 0 else "Right Turn Sign"
        print(f"{indent}Predict: {label_name}")
        return

    # Extract information for the current decision node
    feature_index = tree["feature"]
    feature_name = feature_names[feature_index]
    threshold = tree["threshold"]

    rule = f"{feature_name} < {threshold:.2f}"

    # Recursively print the 'true' branch (left)
    print(f"{indent}If {rule}:")
    print_tree(tree["left"], feature_names, indent + "  ")
```

```
    # Recursively print the 'false' branch (right)

    print(f"{indent}Else ({rule.replace('<', '>')})")

    print_tree(tree["right"], feature_names, indent + "  ")




def main():

    filepath = input("Enter the path to the feature dataset: ")



    feature_columns = ['hue_mean', 'hue_std', 'saturation_mean', 'saturation_std', 'value_mean',
'value_std', 'has_circle', 'num_lines'] #names of all feature columns



    label_column = "ClassId" #name of label col



    X, y = load_dataset(filepath, feature_columns, label_column, shuffle = True, seed=70) #loads
dataset



    X_train, y_train, X_val, y_val, X_test, y_test = train_val_test_split(X, y) #splits datasets



    # prints sizes
```

```
    print("\nData loaded and split:")

    print(f"Training set:  size: {len(X_train)}")

    print(f"Validation set:  size: {len(X_val)}")

    print(f"Test set:  size: {len(X_test)}")



    print("\nFinding optimal tree max depth and min samples split...")



    tune_max_depth(X_train, y_train, X_val, y_val) #tests max depths



    optimal_max_depth = int(input("\nEnter the optimal max depth according to your
interpretation of the plot:\n"))



    tune_min_samples_split(X_train, y_train, X_val, y_val, optimal_max_depth)#tests sample
splits with optimal max_depth



    optimal_min_samples_split = int(input("\nEnter the optimal min samples split according to
your interpretation of the plot:\n"))



    print("Evaluating final model on test set with the chosen hyperparameters")
```

```
    tree = build_decision_tree(X_train, y_train, optimal_max_depth, optimal_min_samples_split)
#build final tree



    #make predictions

    y_train_pred = predict(tree, X_train)

    y_val_pred = predict(tree, X_val)

    y_test_pred = predict(tree, X_test)



    #calculate accuracies

    train_acc = calculate_accuracy(y_train, y_train_pred)

    val_acc = calculate_accuracy(y_val, y_val_pred)

    test_acc = calculate_accuracy(y_test, y_test_pred)



    #print results

    print("\n--- Final Model Performance ---")

    print(f"Test Set Accuracy:      {test_acc:.4f}")

    print(f"Validation Set Accuracy: {val_acc:.4f}")

    print(f"Training Set Accuracy:   {train_acc:.4f}")
```

```
    #print tree results

    print("\n--- Final Decision Tree Rules ---")

    print_tree(tree, feature_columns)




if __name__ == "__main__":

    main()
```

tp3_team_3_15.py

```
from pathlib import Path

from PIL import Image # Importing libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from tp3_team_1_15 import train_val_test_split, load_dataset #Importing load dataset and
training stuff from task 1.



def calculate_sigmoid(z):

    si=np.clip(z,-400,400) #Clipping the values of Z to prevent overly large values that will mess up
data; range is clipped to (-400,400)
```

```
  sigma= 1/(1+np.exp(-si)+(1*10**-15)) #Calculating sigmoid value; epsilon vlaue is added in order to prevent division by 0

  return sigma



def predict_proba(X,w,b):

  z=(np.dot(X,w))+b #Calculating z as the dot product between x and the weight, with the bias added

  prob_array = calculate_sigmoid(z)

  return prob_array



def predict_labels(X,w,b,threshold=0.5):

  prob_array2=predict_proba(X,w,b)

  i=0

  labels=np.zeros(len(prob_array2)) #Forming a zero array to iterate over

  for i in range(len(prob_array2)):

    if prob_array2[i] >= threshold: #Saying that if the probability value is above the threshold we write it as one, otherwise we write it as 0

      labels[i]=1

    else:

      labels[i]=0
```

```
    return labels.astype(int) #Returning this array of 1s and 0s



def compute_loss_and_grads(X,y,w,b):

    m=len(X) #Defining the number of values

    prob_array=predict_proba(X,w,b)

    epsilon_prob=prob_array+(1*10**-15) #Adding an epsilon to prevent trying to find the log of 0

    loss=(-1/m)*np.sum(y*np.log(epsilon_prob)+(1-y)*np.log(1-epsilon_prob))

    dw=(1/m)*np.dot(X.T,(prob_array-y)) #Formula given in the instructions; using X.T to find the
transpose of X

    db=(1/m)*np.sum((prob_array-y)) #loss,dw,db are calculated as given in the instructions using
hte np.sum function

    return loss,dw,db



def train_logistic_regression(X_train, y_train, learn, num):

    shape=X_train.shape #Defining the height of the data

    h=shape[1]

    lost=np.zeros(num) #Making a new empty array

    weight=np.random.randn(h)*0.01 #Defining the initial weight as a random array with the
same dimensions as X_train

    b=0 #Setting bias as 0
```

```
  for i in range(num):

    loss,dw,db=compute_loss_and_grads(X_train,y_train,weight,b)  #A loop that first uses the function to calculate loss and the gradients

    weight=weight-learn*dw #Changing the weight as training goes on using the gradient

    b=b-learn*db #Same as above but for bias

    lost[i]=loss #Writing each loss value to the lost array

    if i % 100 == 0:

      print(f"Iteration {i}, Loss: {loss:.4f}") #Printing every 100th iteration (Took this from the description)

  return weight,b,lost



def calculate_metrics(predicted_labels,true_labels):

  acc=np.mean(predicted_labels==true_labels) #Calculating accuracy as the mean amount of labels that equal the true labels

  error=1-acc

  return acc, error



def evaluate_logistic_regression(X,y,w,b):

  predicted=predict_labels(X,w,b) #Applying functions used above to evaluate

  accuracy,error=calculate_metrics(predicted,y)
```

```
    return accuracy,error



def main():

    learn=0.4 #Defining the learning rate and number of iterations

    num=2000

    features =
["hue_mean","hue_std","saturation_mean","saturation_std","value_mean","value_std","num_li
nes","has_circle"] #Defining the feature array

    label="ClassId" #Setting the label as ClassId

    X,y=load_dataset("img_features.csv",features,label,shuffle=True,seed=70) #Loading dataset

    X_train,y_train,X_val,y_val,X_test,y_test=train_val_test_split(X,y)

    featmean=np.mean(X_train) #Using the feature mean and standard deviation to standardize
the data

    featstd=np.std(X_train)

    X_train_std=(X_train-featmean)/featstd

    X_val_std=(X_val-featmean)/featstd #Standardizing as given in the description

    X_test_std=(X_test-featmean)/featstd

    weight,b,lost=train_logistic_regression(X_train_std,y_train,learn,num) #Training

    accval,errval=evaluate_logistic_regression(X_val_std,y_val,weight,b)

    acctest,errtest=evaluate_logistic_regression(X_test_std,y_test,weight,b) #Evaluating the
validation and test sets
```

```
    print(f"Validation accuracy: {accval:.4f}")

    print(f"Test accuracy {acctest:.4f}")

    print(f"Validation error {errval:.4f}") #Printing the results

    print(f"Test error {errtest:.4f}")

    plt.plot(lost)

    plt.plot() #Plotting everything

    plt.title("Loss vs Iteration")

    plt.xlabel("Iteration")

    plt.ylabel("Loss")

    plt.show()


if __name__ == "__main__":

    main()
```