Leaf Classification Project

Daniel Christodoulou

1690083

June 2022

1 Introduction

The aim of this project is to create a program that can correctly classify different species of leaves using only images of these leaves. This task will be done without the use of any learning algorithms. Instead we use image manipulation techniques to classify the leaves from a given dataset.

2 Methodology

2.1 Algorithms Used

The algorithm at the heart of this project is template matching (the function *matchTemplate* was used from the OpenCV library). Template matching is a technique that aims to find a match between a smaller template image within a larger image (source image). The way this algorithm works is by sliding the template image over the source image, pixel by pixel, a metric is then calculated at each pixel to determine how similar the template is to the region around that particular pixel. That metric is then stored in an array of the same size as the source image, named the "metric matrix". The metric matrix will return a gray image. This matrix will return a image that either displays varying brighter regions (or higher intensity values) to indicate a closer match verses the darker regions, or vice versa depending on which method is used to calculate the metric matrix.

The method which was used in this project was *TM_CCOEFF*. This method was chosen after brief testing each of the different methods on several test images, proving to be slightly more accurate against some of the other methods or largely more accurate in some other cases. The metric matrix R is calculated as follows.

$$R(x,y) = \sum_{x',y'} (T'(x',y') \cdot I'(x+x', y+y')) \tag{1}$$

Where

$$\begin{aligned} T'(x',y') &= T(x',y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} T(x'',y'') \\ I'(x+x',y+y') &= I(x+x',y+y') - 1/(w \cdot h) \cdot \textstyle\sum_{x'',y''} I(x+x'',y+y'') \end{aligned} \tag{2}$$

The resulting array is then passed to the *minMaxLoc* function, also found within the OpenCV library. This function then returns the global minimum and maximum of the an array as well as the location of these values. These values correspond to the location of highest match between the template and the source image.

This algorithm works fairly well with testing images from the database. However, the algorithm fails when testing the algorithm with images from outside the database. There are two main reasons for this. Firstly,

images from outside the database have different lighting and backgrounds to the images from the database creating different regions and thus affecting the metric calculation. This can be seen by Figure 3 in the appendix, there are two images that have vastly different backgrounds or differing lighting. The method also fails if the input image has a leaf that has a different orientation to that of the templates, which mostly have a vertical orientation.

2.2 Pre-processing

The template matching algorithm requires a database of templates in order to identify a match. The way in which these templates were generated was through the function *cropObject*. This function aims to create a template by detecting the boundaries of the leaf in the image and then cropping the leaf. [*Please see the appendix for the source code*].

The function takes in a RGB image as an input, converts the image into gray-scale and subsequently to a binary where the leaf (the darker object) becomes the foreground pixels. This was done using the *adaptiveThreshold* function, the image is divided using the *ADAPTIVE_THRESH_GAUSSIAN_C*. This was chosen through from a variety of tests done on different images. It was sometimes the case that the gray background had similar values to the leaf. Thus the most effective method of thresholding was the adaptive thresholding method as opposed to global thresholding methods. This can be seen in Figure 2 of the appendix, where the *cropObject* function was used with global thresholding.

Subsequently to the thresholding, an opening followed by a closing operation was performed to eliminate salt noise that was affecting the template sizes. A structuring element of a 4 by 4 box of ones was used to eliminate the salt noise. This worked for 99% of cases, however, there are a few cases in which not all the salt was not completely removed and thus the template was larger than would have been ideal. The *cropObject* function then finds the bounds of the leaf using the binary image and uses those coordinates to crops the gray input image. It does this by scanning from each edge of the image and stops when it finds a white pixel. The reason for choosing to create templates of a gray-scale image were to reduce the time taken to apply the template matching algorithm to each template. An example showing the different stages of image manipulation in order to create the template can be seen in Figure 1 of the appendix. The first image is the gray image of the input, the second is once it has been converted to a binary image. The third image is one in which the salt noise has been removed and the final image is the created template.

The function *createTemplate* is a function that puts into use the *cropObject* function. This function takes, as the input, the string of the directory to the database of RGB images. It will then pass each image of each species of leaf to *cropObject* and creates the gray template as discussed, by the *cropObject* function. Each template is then saved under a folder with the name of that species and will be used for the template matching implementation.

2.3 Implementation

The implementation of the the program is the culmination of the ideas and methods that have been discussed thus far.

The function *classifyLeaf* takes an RGB image as an input. The function then applies the template matching algorithm with each image generated in the template database. Since the *matchTemplate* function returns a metric matrix, we need to extract the regions of the source image where the greatest match was found. This is done with the *minMaxLoc* function described under section 2.1. The maximum location, in this case, is all that is require to splice out the rectangular section of the source image that has the highest match with the template. The resulting rectangle is the same size as the template image. The rectangle is then, along with the template image, converted into a binary image. These two images are then subtracted from one another and passed to the *aveIntensity*, this function then calculates the probability that the two images

that were subtracted from one another are a close enough match, in order to be accepted. If we subtract this probability from 1, we obtain the confidence level that image has found a correct match. All templates that have a confidence level greater than 99% are then stored in a python dictionary where the species is recorded along with the corresponding confidence level. The species from the dictionary that has the highest confidence level is then returned. If no suitable match was found then the function returns a string indicating that it could not classify the given image.

3 Results

A significant downside to the current implementation is time taken in order to run. It takes roughly two and a half minutes for a single classification. This also made testing a large enough sample more difficult. Testing was done using the *testing* function. This function takes as an input, the directory to the images and an integer n, n images are then picked randomly from the database and tested. The percentage of correct classifications are then returned. Testing the program with 50 sample produced an accuracy of 72% (see source code in the appendix). Although there is roughly 400 images in total, testing only 12.5% of the database is far from ideal. The total runtime for this test was roughly two hours and so testing a greater sample size could not be feasibly done with the time restrictions. A way in which a future implementation could be done by using the *cropObject* function to split the templates along a line of symmetry in cases where there exists symmetrical leaves in order to reduce the runtime of the program.

4 Conclusion

Template matching is a useful too to identify objects within larger images, however, it is not very useful when trying to generalise the algorithm to images outside of the database. This is case because the metric used to calculate the similarities between template and source image is not able to generalise across different orientations and different lighting etc. An implementation could be adopted in which the background of the objects can be ignored by the metric calculation. There is no more available time to try this implementation.

Appendices

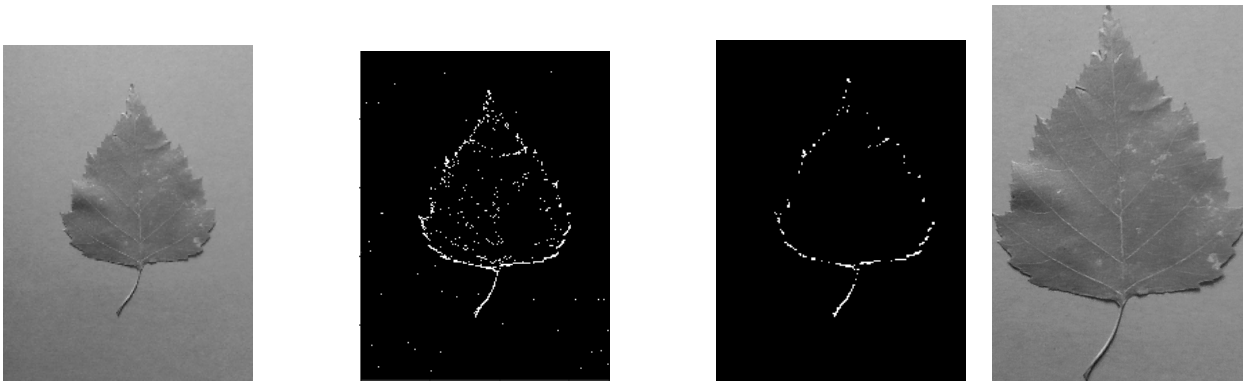

Figure 1: Example the *cropObject* function using adaptive thresholding and removing salt

Figure 2: Example of the *cropObject* using global thresholding

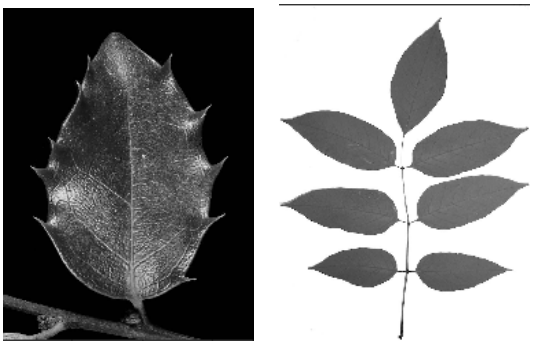

Figure 3: Gray images from outside the database

Leaf Classification - Daniel Christodoulou - 1690083

```
import numpy as np
import os, os.path
from os import listdir
from skimage import io
from scipy.ndimage import binary_opening,binary_closing
from skimage.filters import threshold_mean
import cv2 as cv
import random as r

def cropObject(im):
    n,m = im.shape[::-1]
    # Convert image to binary
    im_2 =
cv.adaptiveThreshold(im,1,cv.ADAPTIVE_THRESH_GAUSSIAN_C,cv.THRESH_BINA
RY,41,10)-1
    # Remove salt noise
    SE = np.ones((4,4))
    im_2 = binary_opening(im_2,SE)
    im_2 = binary_closing(im_2,SE)
    # Locate bounds of object (leaf)
    for i in range(m):
        for j in range(n):
            if im_2[i,j]==1:
                ymax = i
                break

    for i in range(n):
        for j in range(m):
            if im_2[j,i]==1:
                xmax = i
                break

    for i in range(m-1,0,-1):
        for j in range(n-1,0,-1):
            if im_2[i,j]==1:
                ymin = i
                break

    for i in range(n-1,0,-1):
        for j in range(m-1,0,-1):
            if im_2[j,i]==1:
                xmin = i
                break

    a = (xmax - xmin)
    b = (ymax - ymin)
```

```
    output = im[ymin:ymin + b,xmin:xmin + a]
    return output

# Create Template
def createTemplates(path):
    database = [f for f in listdir(path)]
    for i in database:
        if not i.startswith('.'):
            leaves = [p for p in listdir(path+"/"+i)]
            for j in leaves:
                if not j.startswith('.'):
                    image = cv.imread(path+"/"+i+"/"+j,0)
                    template = cropObject(image)
                    save_folder =
"/Users/danielchristodoulou/Documents/Wits -
University/Honours/Digital Image
Processing/Project/leaf/Templates_Gray"
                    cv.imwrite(os.path.join(save_folder+"/"+i,
'template_'+j), template)

createTemplates("leaf/RGB")

# Gets the probability of white pixels in an image
def aveIntensity(img):
    m,n = img.shape
    sum = 0
    for i in range(m):
        for j in range(n):
            sum += img[i,j]/255
    return sum/(m*n)

# Functon that classifies images of leaves with a level of confidence
def classifyLeaf(img):
    img2 = img.copy()
    database = [f for f in listdir("leaf/Templates_Gray")]
    storage = {}
    # Begins the loop to enter the database
    for i in database:
        if not i.startswith('.'):
            counter = 0
            Templates = [p for p in listdir("leaf/Templates_Gray/"+i)]
            # Begins the loop to enter each template in the databse
            for j in Templates:
                if not j.startswith('.'):
                    template =
cv.imread("leaf/Templates_Gray/"+i+"/"+j,0)
                    w,h = template.shape[::-1]
                    img = img2.copy()
                    # Template matching method
                    method = eval("cv.TM_CCOEFF")
                    # Applying template matching
```

```
                    res = cv.matchTemplate(img,template,method)
                    # Applying minMaxLoc
                    min_val, max_val, min_loc, max_loc =
cv.minMaxLoc(res)
                    # Splice out rectangle
                    crop_img =
img[max_loc[1]:max_loc[1]+h,max_loc[0]:max_loc[0]+w]
                    # Convert to binary
                    temp_bi =
cv.threshold(template,threshold_mean(template),1,cv.THRESH_BINARY)[1]
                    test_bi =
cv.threshold(crop_img,threshold_mean(crop_img),1,cv.THRESH_BINARY)[1]
                    diff = test_bi - temp_bi
                    confidence = 1 - aveIntensity(diff)
                    if confidence > 0.99:
                        genus = i.split(" ")[1]
                        species = i.split(" ")[2]
                        storage[genus+" "+species] = confidence
    if bool(storage):
        output = max(storage)
    else:
        output = "Could not classify the leaf in this image"
    return output

# Function to test accuracy of implementation
def testing(path,n):
    correctCounter = 0
    for t in range(n):
        # Chooses a random directory
        species = r.choice(os.listdir(path))
        if not species.startswith("."):
            # Chooses a radnom image within the directory
            im_Name = r.choice(os.listdir(path+"/"+species))
            if not im_Name.startswith("."):
                im = cv.imread("leaf/RGB/"+species+"/"+im_Name,0)
                test = classifyLeaf(im)
                spec = species.split(" ")
                ref = spec[1]+" "+spec[2]
                # Tests the result of the classifyLeaf
                if test == ref:
                    correctCounter += 1
    return correctCounter/n
path = "/Users/danielchristodoulou/Documents/Wits -
University/Honours/Digital Image Processing/Project/leaf/RGB"
print("The percentage accuracy of testing 50 images from
database:"+str(testing(path,50)))
```

```
The percentage accuracy of testing 50 images from database:0.72
```