# Image Processing review

## Introduction

### Abstract

Let us consider a professional photographer. During a photo shoot, he shall take hundreds of photos of the same setting. When the photographer takes a bad photo, of a good background and needs similar photos of the similar background, his normal photo "Tags" won't help him. Normals tags in the photo would include "location", "Time" mostly. Some softwares would even include facial recognition. But none of these are useful for the photographer as he wants photos of similar backgrounds and none of these tags can help him. So an algorithm has been developed that writes tags based on the background of all the images in the database. When the photographer inputs a query image, the algorithm will return images that have similar background. These are images that were most likely taken at the same place.

### Background

In this project, the user will create a folder called database and copy all available photos to the database folder. Then the user run the index file, which will run a custom feature selection algorithm over the entire database. Features of the all images with their imageIDs are noted down. This is the preliminary part of the code. This part needn't be repeated again unless new photos are added to the database. Then the user needs to run "Search.py" wherein he'll input a query image. The program will return 10 images that have similar background to the original image.

## Overview

### Proposed work

We propose a system based on feature selection in "select areas" of the image. Then we also apply basic feature selection techniques and image retrieval techniques.

### Requirements

In a windows laptop:

- RAM should be greater than 8GB ram.
- Core i5 processor 5$^{th}$ Gen or above
- Python 3.6+ with the numpy, OpenCv, and matplot lib, glob libraries inbuilt.
- There should be no IDE involved as that hinders the process. IDE's don't have rights to write in text files to particular folders.

The laptop OS should be specified before hand, as windows uses backward slash '\' to specify folders directories and MAC, Ubuntu and Linux use forward slash '/'. This causes the code to change quite a bit as the code uses traversal of directories a lot.

## Literature Review

Multiple process models have been considered for this project. They are listed below with their own headings.

### Search by Metadata

This is very similar to keyword based search for different images. The user inputs a query image and the algorithm returns similar images. This assessing of similarity is based on the metadata of the image. Each image has it's own tags, "location", "date and Time", " Facial Features" and "Manual Annotation", "Name" for example are some of the different tags that images mostly use. The types of tags to use varies based on the developer. When the user inputs a query image, particular tags of that image are analysed and the search is performed based on similarity of said tags. The program

searches through the database for images that have same tags, and returns those images. This model doesn't actually analyse or examine the image itself but rather examines the tags of the image. Thus as it won't actually have much of "image processing" involved, we have discarded this model. However we can use this model for future development.

### Search by example

Search by example to the contrary uses only the image. It doesn't uses tags at all, in fact it assumes that the images provided have no tags. This kind of algorithm is called CIBR, Content based Image Retrieval. This is a proper image search engine. This contains a lot of image processing model and this is the model that we consider with additive increments.

### Model considered

There is a middle ground between these two, and that is called Hybrid Approach. In this we consider both tags and analyse the images in the database. This has both the selections of the Search by example and Search by Metadata in it. Thus, it is a very powerful algorithm. But this is not the model considered for this program as it takes too long to run and requires all images to have similar metadata or "Tags".

## Implementation

There are four basic steps to each CIBR algorithm.

- **Defining the image descriptor**: We first need to define what aspect of the image needs to be considered and described. The features that can be considered are colour of the image, shape of the image, tags and so on. In this algorithm we consider the RBG colour of the image. Shape and tags are not considered. This is because the shape of the image should not determine the output. And for reasons stated above, we have dropped tag based retrieval system.
- **Indexing the dataset:** Now that the image descriptor is set up, we need to use the image descriptor and create an index file where the features of the files along with their respective ImageIDs are stored. A file reader and writer is opened in write mode and all the features of the image are written in a CSV format in a txt file to the pointed location. This location is user driven.
- **Defining the similarity metric:** We have created the index file which contains details of all the images in the dataset. To compare the features with the query image we need to have some sort of comparing algorithm. The algorithm chosen here is Chisq comparing algorithm. The chisq distance between the features of the query image and the dataset images are calculated. If the chi sq distance is low, then they are considered similar images.
- **Searching:** This is the function which performs the actual search. The user has to input the query image. The system will return multiple images of similar background

## Code

### Python files explanation

There are four files overall namely "ColorDescriptor.py", "Searcher.py", "Index.py", "Search.py". Out of these four files, we'll only run the files Index.py and Search.py. The other two files are incorporated with the index.py and search.py files.

### ColourDescriptor

This file has a class called colordescriptor. The class has many member functions and the image itself is an variable of the class. An inbuilt function is used to analyse the features of the image, and return the features to the calling function.

*Code*

```
# import the necessary packages
# import the necessary packages
import numpy as np
import cv2

class ColorDescriptor:
	def __init__(self, bins):
		# store the number of bins for the 3D histogram
		self.bins = bins

	def describe(self, image):
		# convert the image to the HSV color space and initialize
		# the features used to quantify the image

		image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
		features = []

		# grab the dimensions and compute the center of the image
		(h, w) = image.shape[:2]
		(cX, cY) = (int(w * 0.5), int(h * 0.5))

		# divide the image into four rectangles/segments (top-left,
		# top-right, bottom-right, bottom-left)
		segments = [(0, cX, 0, cY), (cX, w, 0, cY), (cX, w, cY, h),
			(0, cX, cY, h)]

		# construct an elliptical mask representing the center of the
		# image
		(axesX, axesY) = (int(w * 0.75 / 2), int(h * 0.75 / 2))
		ellipMask = np.zeros(image.shape[:2], dtype = "uint8")
		cv2.ellipse(ellipMask, (cX, cY), (axesX, axesY), 0, 0, 360, 255, -1)

		# loop over the segments
		for (startX, endX, startY, endY) in segments:
			# construct a mask for each corner of the image, subtracting
			# the elliptical center from it
			cornerMask = np.zeros(image.shape[:2], dtype = "uint8")
			cv2.rectangle(cornerMask, (startX, startY), (endX, endY), 255, -1)
			cornerMask = cv2.subtract(cornerMask, ellipMask)

			# extract a color histogram from the image, then update the
			# feature vector
			hist = self.histogram(image, cornerMask)
			features.extend(hist)

		# extract a color histogram from the elliptical region and
		# update the feature vector
		hist = self.histogram(image, ellipMask)
		features.extend(hist)
```

```
        # return the feature vector
        return features

    def histogram(self, image, mask):
        # extract a 3D color histogram from the masked region of the
        # image, using the supplied number of bins per channel; then
        # normalize the histogram
        hist = cv2.calcHist([image], [0, 1, 2], mask, self.bins,
            [0, 180, 0, 256, 0, 256])

        #hist = cv2.normalize(hist).flatten()
        cv2.normalize(hist, hist)
        hist = hist.flatten()
        # return the histogram
        return hist
```

## Searcher

This file is used to search for other images based on the features. This file takes in the query image, calls the ColourDescriptor file, gets the relevant features of the query image. Then, it has a chisq distance calculating function which will compare the features of the query image and the inmates in the dataset and then return 10 images that have similar background to the original image.

### *Code*

```
# import the necessary packages
import numpy as np
import csv

class Searcher:
    def __init__(self, indexPath):
        # store our index path
        self.indexPath = indexPath

    def search(self, queryFeatures, limit = 10):
        # initialize our dictionary of results
        results = {}

        # open the index file for reading
        with open(self.indexPath) as f:
            # initialize the CSV reader
            reader = csv.reader(f)

            # loop over the rows in the index
            for row in reader:
                # parse out the image ID and features, then compute the
                # chi-squared distance between the features in our index
                # and our query features
                features = [float(x) for x in row[1:]]
                d = self.chi2_distance(features, queryFeatures)

                # now that we have the distance between the two feature
                # vectors, we can udpate the results dictionary -- the
```

```
                        # key is the current image ID in the index and the
                        # value is the distance we just computed, representing
                        # how 'similar' the image in the index is to our query
                        results[row[0]] = d

                # close the reader
                f.close()

        # sort our results, so that the smaller distances (i.e. the
        # more relevant images are at the front of the list)
        results = sorted([(v, k) for (k, v) in results.items()])
        print(limit)
        # return our (limited) results
        return results[:limit]


def chi2_distance(self, histA, histB, eps = 1e-10):
        # compute the chi-squared distance
        d = 0.5 * np.sum([((a - b) ** 2) / (a + b + eps)
                for (a, b) in zip(histA, histB)])

        # return the chi-squared distance
        return d
```

## Index

This is one of the file that can be run by the user. Before running the file, the user must first create a folder called “database” with all the images stored. Then, he needs to run the “index.py” file. This file will create a file called “index.txt” in the running folder. This txt file will have all information about the features of the images stored in it.
The index functions gets receives all the images in the dataset and runs the colour descriptor class functions to get the features of all the images. Then, it writes them into a txt file of type CSV.

### *Code*

```
# import the necessary packages
from colordescriptor import ColorDescriptor
import argparse
import glob
import cv2

# construct the argument parser and parse the arguments
#ap = argparse.ArgumentParser()
#ap.add_argument("-d", "--dataset", required = True,
#       help = "Path to the directory that contains the images to be indexed")
#ap.add_argument("-i", "--index", required = True,
#       help = "Path to where the computed index will be stored")
#args = vars(ap.parse_args())

args ={
        'index' : "index.txt",
        'dataset' : "dataset",
}
```

```
# initialize the color descriptor
cd = ColorDescriptor((8, 12, 3))

# open the output index file for writing
output = open(args['index'], "w")

#print(glob.glob(args['dataset'] + "/*.jpg"))

# use glob to grab the image paths and loop over them
for imagePath in glob.glob(args['dataset'] + "/*.jpg"):
        # extract the image ID (i.e. the unique filename) from the image
        # path and load the image itsel

        imageID = imagePath[imagePath.rfind("dataset") + 8:]
        image = cv2.imread(imagePath)
        print(imagePath)
        # describe the image
        features = cd.describe(image)
        # write the features to file
        features = [str(f) for f in features]
        print(imageID)
        output.write("%s,%s\n" % (imageID, ",".join(features)))

# close the index file
output.close()
```

## Search

This is the final file of the program. This file will accept a query image from the user, and then run the algorithm to find the images similar to it. It calls the searcher functions which returns the ImageIDs of the images that are similar to the original image. Then on received the imageIDs of the result images, it'll display the images using the imshow function.

### *Code*

```
# import the necessary packages
from colordescriptor import ColorDescriptor
from searcher import Searcher
import argparse
import cv2
import time

# construct the argument parser and parse the arguments
#ap = argparse.ArgumentParser()
#ap.add_argument("-i", "--index", required = True,
#       help = "Path to where the computed index will be stored")
#ap.add_argument("-q", "--query", required = True,
#       help = "Path to the query image")
#ap.add_argument("-r", "--result-path", required = True,
#       help = "Path to the result path")
#args = vars(ap.parse_args())
```

```
# initialize the image descriptor
cd = ColorDescriptor((8, 12, 3))

# load the query image and describe it

query = cv2.imread('images.jpg')
features = cd.describe(query)


# perform the search
searcher = Searcher("index.txt")
#result path should be the same as the dataset path.
results = searcher.search(features)
print(len(results))

# display the query
cv2.imshow("Query", query)
cv2.waitKey(0)


# loop over the results
for (score, resultID) in results:
        # load the result image and display it
        #result = cv2.imread('/dataset' + '/' + resultID)
        print(resultID)
        #cv2.imshow("Result", result)
        #time.sleep(1000)
```

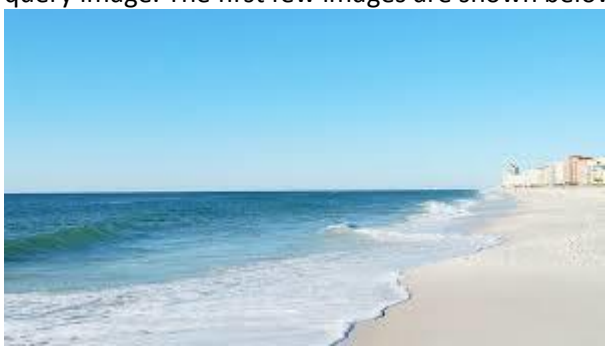
## Output

This is the query image that was given to the function searcher.

The folder “database” has multiple files that are similar to the this. These were found in the google Search ‘Beaches’.

```
D:\Image Processing>python search.py
10
10
images (3).jpg
images (6).jpg
download (3).jpg
images.jpg
download (1).jpg
download.jpg
download (2).jpg
images (4).jpg
images (2).jpg
images (1).jpg
```

In this picture we see that the code has returned the names of ten images that are similar to the query image. The first few images are shown below.



Image(6).jpg

Image(3).jpg

Download(3).jpg

In all images we see that the output produces is valid as the sand (brown) colour is always on the right and the sea (blue) colour is always on the left. This is indicative that the code has worked as expected.

## Results and conclusion

The image search retrieval system worked well with bright and dark background, with an overdose of certain colours (Red, green, blue and yellow). On most test cases, it worked well. However one out of ten photos would be incorrect and would not match the scenario.

## Future work

For future work, we shall incorporate face recognition and machine learning algorithms to make the search even more accurate. Based on a given input and a proper predefined outputs, we'll run a machine learning algorithm and store it's results in the same directory.We'll use those results along with the feature selection techniques to get better and more accurate results.