**Lucene Based Java Swing Application for Similar Image Search**

Purpose of this project is to find similar images from the large databases (or from internet). For an uploaded candidate image, visually similar images are displayed as a result. Developed a Java swing application to retrieve similar images for an uploaded image from a dataset of multiple images indexed using Lucene. OpenCV library with Python is used to process images to extract features. Dataset of images called CalTech 256 is used and 2000 images from it are indexed. Used Lucene’s query parser and ranged queries to compare and query using the features of a candidate image. Once the images with similar average color are retrieved by querying on multiple indices (R G B distribution), those images are again one to one compared to decide the similarity using SIFT detection. Images are sorted as per the decreasing order of their similarity. Python code to preprocess each image is executed at runtime using ProcesBuilder in Java.

Following are the detail steps followed to build this project.

1. **Defining Image Descriptor**

Any one of Color, Texture and shape or combination of three can be used to define a feature vector for an image.

Calculated average color [R G B] vector using OpenCV library in Python as follows.

import cv2

import numpy as np

import sys

img = cv2.imread(sys.argv[1], -1)

average\_color\_per\_row = np.average(img, axis=0)

average\_color = np.average(average\_color\_per\_row, axis=0)

print(average\_color)

This gives an output vector [R G B] where R, G, and B represents average number of red pixels in image, G represents average number of green pixels and B represents average number of blue pixels.( All are float values)

A vector for shape property of an image can be calculated using following Python code.

img = cv2.imread(sys.argv[1], -1)

shapeArray=img.shape

shapeString=""

for digit in shapeArray:

shapeString += str(digit)+" "

print(shapeString)

This gives an output vector [ht w bpp] where ht is height , w is width and bpp is bits per pixel value.

1. **Indexing your dataset**

Lucene is a full-text search library in Java which makes it easy to add search functionality to an application or website. It does so by adding content to a full-text index. It then allows you to perform queries on this index. This type of index is called an **inverted index**, because it inverts a page-centric data structure (page->words) to a keyword-centric data structure (word->pages).

Here built an index over six float values which are nothing but each vector value in 3 dimensional average color vector and a shape vector.

doc.add(**new** FloatField("shapeA", shapeA , Field.Store.***NO***));

doc.add(**new** FloatField("shapeB", shapeB , Field.Store.***NO***));

doc.add(**new** FloatField("shapeC", shapeC , Field.Store.***NO***));

doc.add(**new** FloatField("green",green,Field.Store.***NO***));

doc.add(**new** FloatField("red",red,Field.Store.***NO***));

doc.add(**new** FloatField("blue",blue,Field.Store.***NO***));

Indexing involves adding Documents to an IndexWriter

writer.updateDocument(**new** Term("path", file.toString()), doc);

1. **Building Queries**

As we have 6 float indices which are used to query the dataset. I have used a RangeQuery which matches documents with terms between beginning and ending terms, including or excluding the end points.Searching involves retrieving Documents from an index via an IndexSearcher

BooleanQuery finalQuery = **new** ~~BooleanQuery~~();

finalQuery.~~add~~(redQuery, Occur.***MUST***);

//MUST implies that the keyword must occur.

finalQuery.~~add~~(greenQuery, Occur.***MUST***);

//Using all "MUST" occurs is equivalent to "AND" operator.

finalQuery.~~add~~(blueQuery, Occur.***MUST***);

finalQuery.~~add~~(shapeAQuery, Occur.***SHOULD***);

//MUST implies that the keyword must occur.

finalQuery.~~add~~(shapeBQuery, Occur.***SHOULD***);

//Using all "MUST" occurs is equivalent to "AND" operator.

finalQuery.~~add~~(shapeCQuery, Occur.***SHOULD***);

TopDocs hits = searcher.search(finalQuery, 200);

Once an image is uploaded first it is preprocessed to calculate average R,G,B and height, wodth and bpp values. Once they are calculated indexed data set is searched in a range greater than or less that the calculated values as follows.

Query redQuery = NumericRangeQuery.*newFloatRange*("red", red-15.03f, red+15.10f, **true**, **true**);

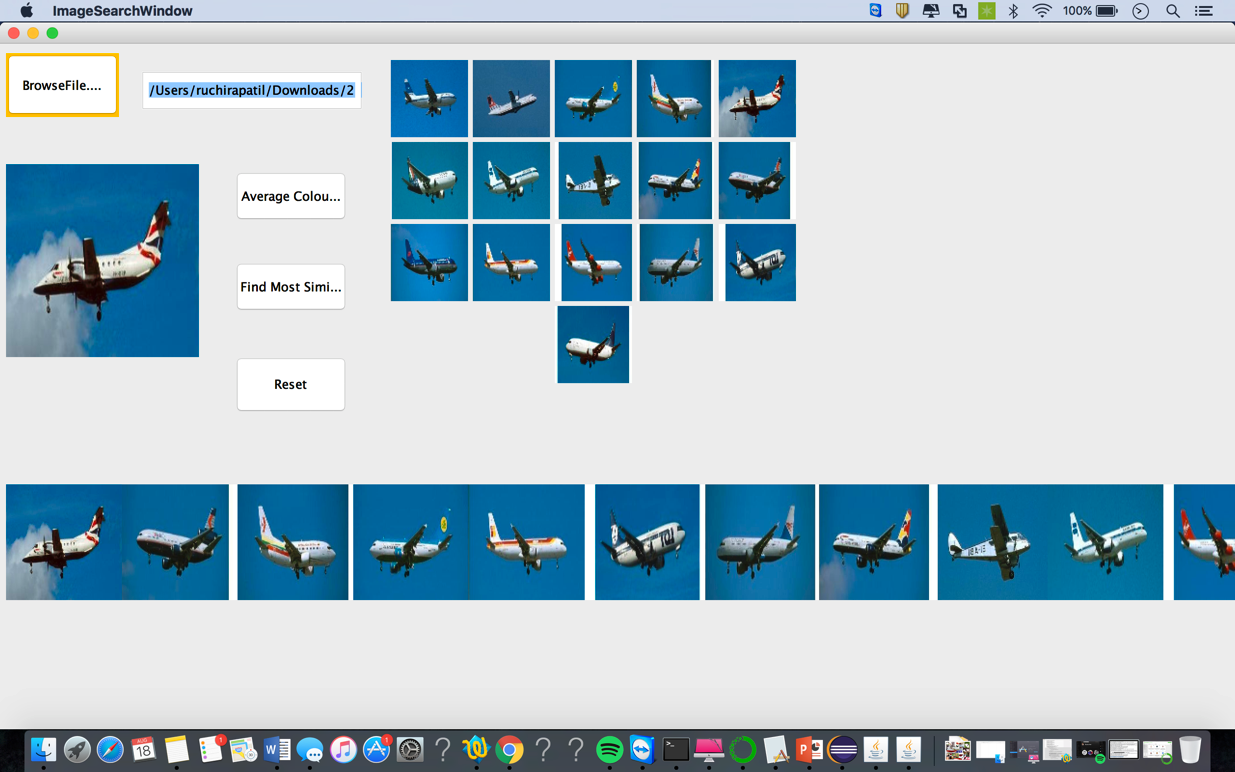
Query greenQuery = NumericRangeQuery.*newFloatRange*("green", green-15.03f, green+15.10f, **true**, **true**);

Query blueQuery = NumericRangeQuery.*newFloatRange*("blue", blue-15.03f, blue+15.10f, **true**, **true**);

Query shapeAQuery = NumericRangeQuery.*newFloatRange*("shapeA", shapeA-20.03f, shapeA+20.10f, **true**, **true**);

Query shapeBQuery = NumericRangeQuery.*newFloatRange*("shapeB", shapeB-20.03f, shapeB+25.10f, **true**, **true**);

Query shapeCQuery = NumericRangeQuery.*newFloatRange*("shapeC", shapeC-20.03f, shapeC+20.10f, **true**, **true**);



1. **Calculating a similarity score and sorting.**

Images are then sorted from left to right from Maximum similarity to minimum. Similarity between the images obtained after querying over indexed images considering average RGB color distribution and shape. And then they are sorted in the Ascending order of their similarity score which is calculated using SIFT detector and BFMatcher in OPenCV in python.

SIFT: **Distinctive Image Features from Scale-Invariant Keypoints**, which extract keypoints and compute its descriptors. Brute-Force matcher is simple. It takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned.

Similarity score is stored as a value for image name as key and the this hashmap is sorted considering keys.

img1 = cv2.imread(sys.argv[1],-1) # queryImage

img2 = cv2.imread(sys.argv[2],-1) # trainImage

# Initiate SIFT detector

sift =cv2.xfeatures2d.SIFT\_create()

# find the keypoints and descriptors with SIFT

kp1, des1 = sift.detectAndCompute(img1,None)

kp2, des2 = sift.detectAndCompute(img2,None)

# BFMatcher with default params

bf = cv2.BFMatcher()

matches = bf.knnMatch(des1,des2, k=2)

# Apply ratio test

good = []

for m,n in matches:

if m.distance < 0.75\*n.distance:

good.append([m])

# cv2.drawMatchesKnn expects list of lists as matches.

print(len(good))

1. **Accuracy Analysis**

Accuracy of searching application can be measured in terms of precision, recall and f1 score.

precision=((**float**)totalRetrivedRelevantInstances)/((**float**)totalRetrivedInstances);

recall=(**float**)totalRetrivedRelevantInstances/(**float**)totalRelevan Instances;

f1Score=(precision\*recall)/(precision+recall);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Precision** | **Recall** | **F1 Score** | **Time(ms)** | **Images** |
| 0.14285715 | 0.13402061 | 0.069148935 | 3309 | 013\_0001.jpg |
| 0.75 | 0.0075 | 0.007425742 | 3071 | 251\_0002.jpg |
| 1 | 0.02 | 0.019607844 | 3142 | 251\_0001.jpg |
| 1 | 0.017857144 | 0.01754386 | 3355 | 024\_0009.jpg |
| 1 | 0.02 | 0.019607844 | 3142 | 251\_0330.jpg |
| 0.8 | 0.040816326 | 0.03883495 | 3115 | 002\_0001.jpg |
| 0.65909094 | 0.03625 | 0.03436019 | 3036 | 251\_0001.jpg |
| 0.23809524 | 0.05102041 | 0.042016808 | 3093 | 002\_0002.jpg |
| 0.51428574 | 0.18367347 | 0.13533837 | 3032 | 001\_0003.jpg |
| 0.12087912 | 0.09821428 | 0.05418719 | 3078 | 024\_0045.jpg |

Accuracy for varying R, G,B range for single image.

|  |  |  |
| --- | --- | --- |
| F1 Score | Time | Range difference |
| 0.01010101 | 3599 | 5 |
| 0.01010101 | 3363 | 10 |
| 0.03883495 | 3731 | 15 |
| 0.06666667 | 3666 | 20 |
| 0.07428571 | 3491 | 25 |
| 0.064516135 | 3571 | 30 |
| 0.08389261 | 3563 | 35 |

1. **Dynamically fitting all the result images in a square of 400X400 square**

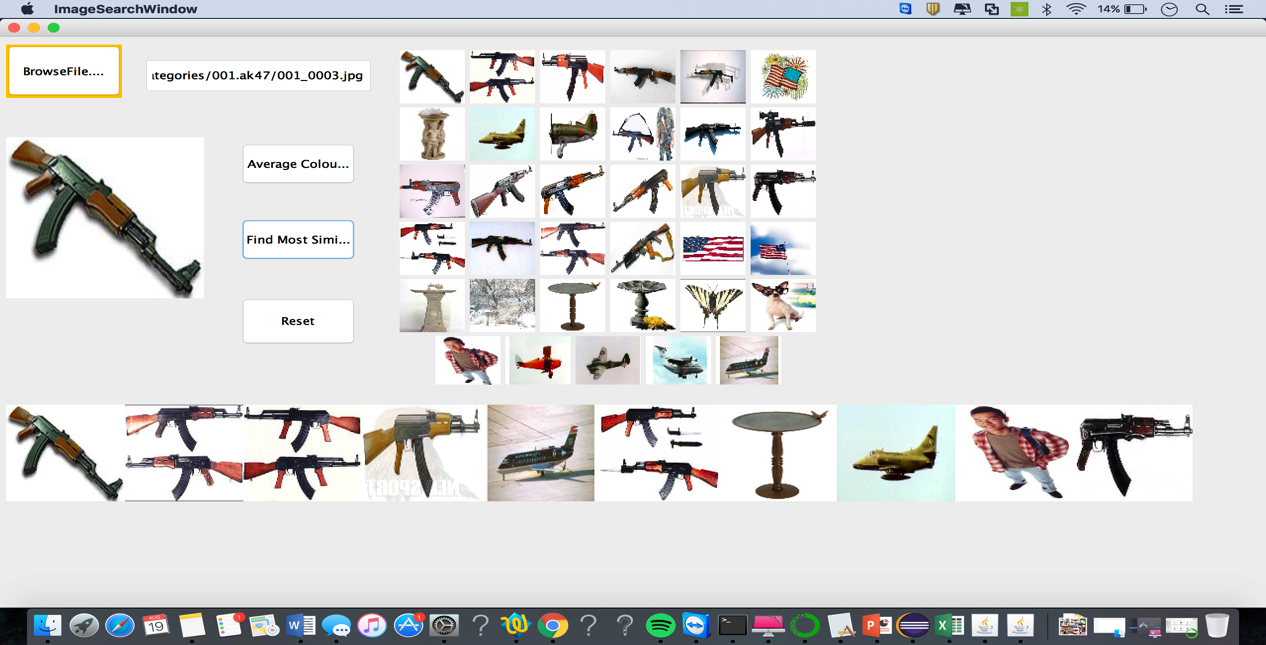
Each time number of image search vary. So in order that all images fit inside the 400x400 matrix, a simple algorithm is used as follows:-

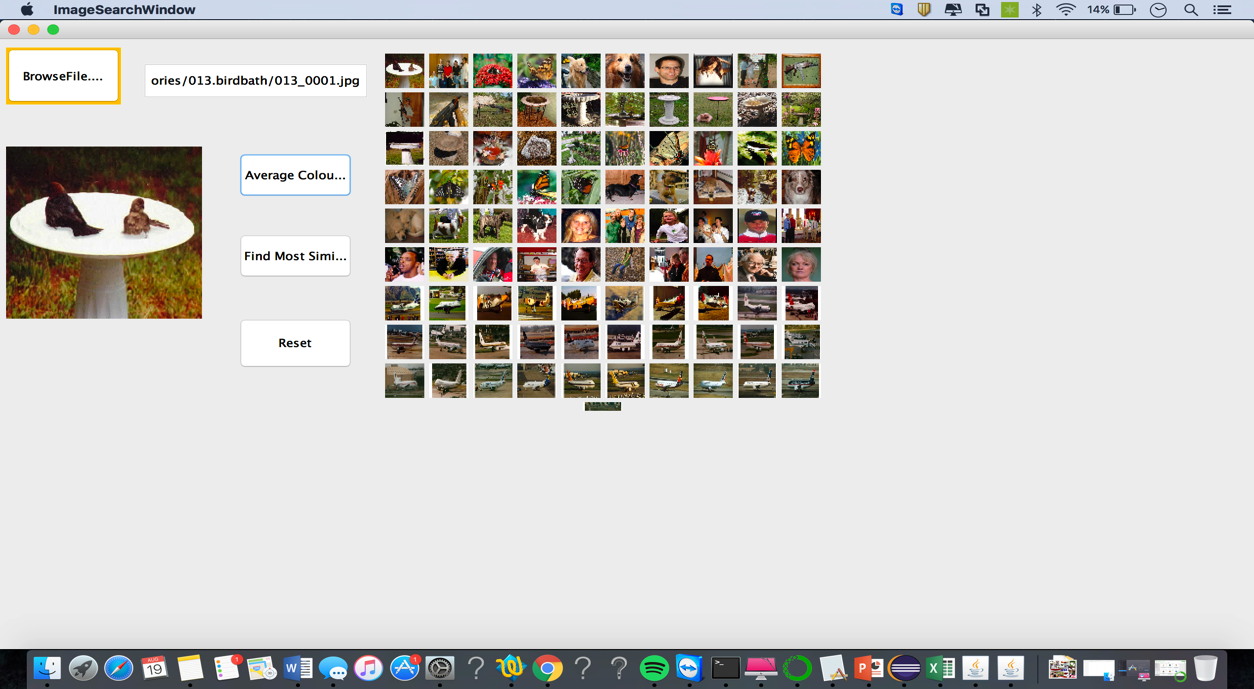
Height=400/square root of no. of result images;

numberOfSquaresInOneRow=Math.*sqrt*(totalDocumentResults);

absoluteNumberOfSquaresInOneRow = 1+Math.*abs*(numberOfSquaresInOneRow);

height=400/absoluteNumberOfSquaresInOneRow;





1. **Steps to Run the Project :-**

1)Unzip the jar and save java project on your machine directory.

2)Install Python 3.3 with Anaconda and know the python path with command ‘which python’

3)Install OpenCV by executing command pip install opencv-contrib-python

4)In a findProperty method in LuceneReadIndexFromFileExample class of project. In ProcessBuilder argument mention Python path as well as the path where the python scripts are saved. Currently I have kept all scripts in project root folder.

ProcessBuilder("/Users/ruchirapatil/anaconda/bin/python","/Users/ruchirapatil/Downloads/LuceneExamples/"+fileName,fileLoc);

4)All image file in InputFiles2 folder are already indexed. So no need to index again.

6)Directly run Java swing application by running ImageSearchWindow class.

7)If wish to index all files all over again. Update the Python path and path of the folder where all Python scripts are in Process builder of findProperty method of WriteIndexFile class.

8)Images in folder inputFiles2 are indexed.