VISUAL INFORMATION RETRIEVAL

# OVERVIEW

This project is made for searching images in Oxford Building (5K) dataset. This project uses Python for backend, HTML and JavaScript for web page UI. Opencv is used for all the image processing including read and extract features from images. Flask is used for getting and responding requests from web page.

## System overview

### UI

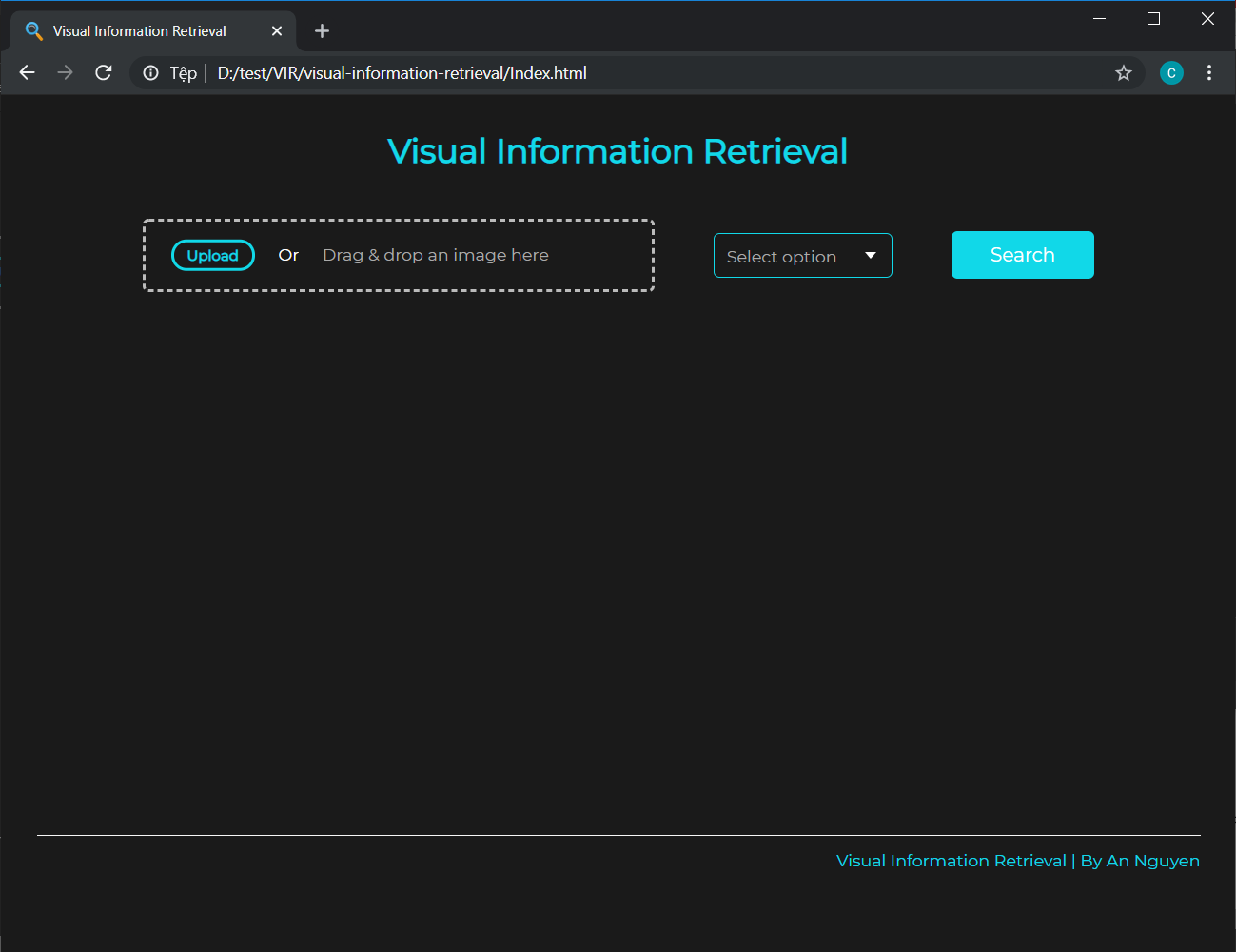
The web page is built with HTML and JavaScript. On the home page,

Figure 1 - Home page

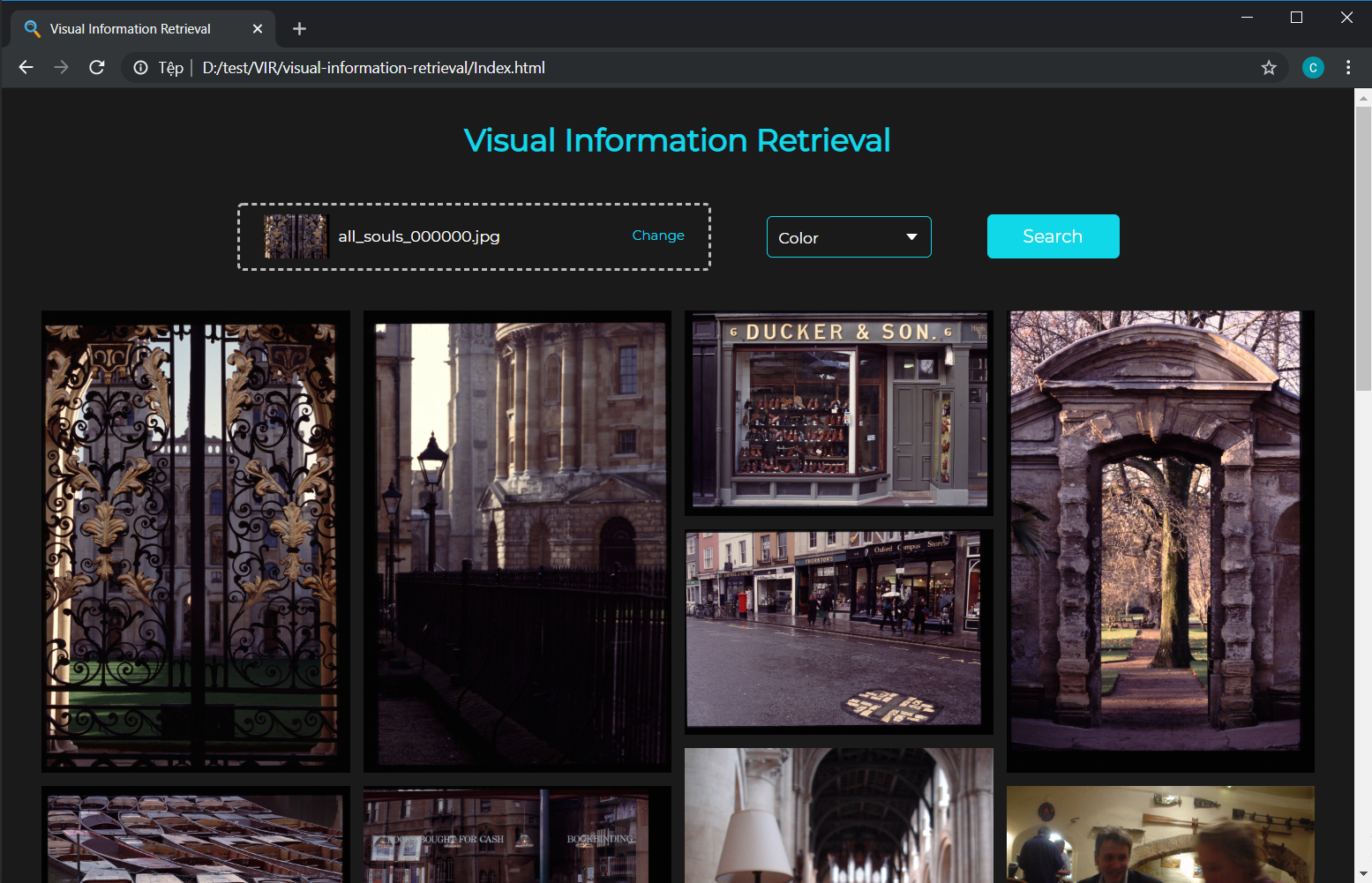


Figure 2 - Result page

### Backend

Backend of this project is built as RESTful API for easy to use and deploy purpose. There are 2 features extraction methods at the moment: color and edge features extraction methods.

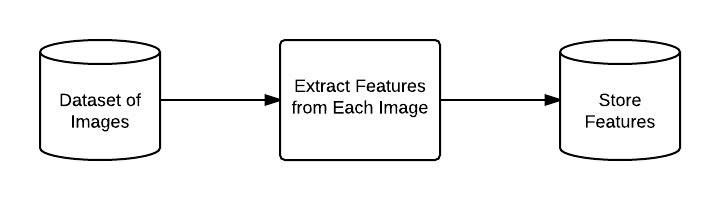
[](https://www.pyimagesearch.com/wp-content/uploads/2014/11/preprocessing_and_indexing.jpg)Before the API can run, all the data features are extracted and saved into csv files to use later.

Figure 3 - Process of extracting features from each image in dataset

In general, while running, the web page will get the image and send it to backend to extract features and search for similar images within the dataset. After getting results, backend will send a list of images paths to web page for it to read those images and show them to user.

Figure 4- General flow of the API

Description of each step:

* Step 1: Because the API is built as a local RESTful API, the API will get image and extraction mode using Flask framework with “POST” method.
* Step 2: After getting the image and mode, it will proceed to read the image using Opencv before processing and extracting features from it using the method user chose.
* Step 3: After getting the features, the API will load the extracted features of the dataset to calculate the distance of the query image with each image in the dataset to find the similar ones.
* Step 4: After getting all the similar images from the dataset, the API will send their relative path to web page to take and show on the UI.

## Extraction methods

### Color Histogram

In this mode, image will be converted to HSV format. But instead of turning the whole image into a feature vector, we chose to split the image into 5 small parts to extract features and then concatenate them to make a single feature vector. Because of using color histogram, we couldn’t determine where this color is or which part of the image has more of this color than the others. This information is important for finding similarities in images. So to remedy for this problem, we compute the color histogram in regions of image.

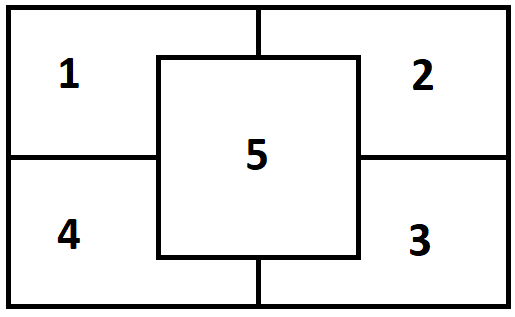


Figure 5 - Dividing image into 5 segments

We construct a rectangle to represent the center region that used 75% of height and width of the image. Then we make a mask to cover other regions, only left 1 region for extracting features from that region until finish extracting every region on the image. By using mask, we can safely extract all the features from each region.

After the histogram of the image is achieved, it will be normalized to be scale invariant. With this normalization, if we computed 2 identical images with different size, the color histograms from those 2 images would be nearly identical. And also, performing this normalization will ensure that images with similar content but dramatically different dimensions will still be “similar” once we apply our similarity function.

The similarity function, we used the chi-squared distance since we are comparing color histograms, which are by definition probability distributions, the chi-squared function is a suitable choice.

### Edge features

In this mode, we’re using the normal image without changing its format like in color feature. Like on color feature, we also split the image into multiple segment before extracting feature from it. But this time, we slice it into many slices with size of 10x10 on all channels of the image and use a kernel with shape [5, 2, 2] and do convolution with stride (1, 1) on each slice.

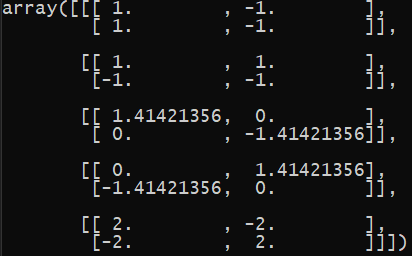


Figure 6- Kernel used in edge feature

At first, the image will be split into many segments with the size of 10x10, we will call them slices. The number of slices is vary depend on input image’s size.

After having those slices, we use the predefined kernel to convolve it with each slice to get the histogram. Then normalize them before put them into an array.

Last step is to flatten the array into a feature vector before saving it or use it to find similarities.

For this feature, we tried chi-square function but got bad results so we changed to cosine and it got better. So the similarity function we are using in this mode is cosine function.

# RESULTS

Queries used for benchmark are provided by <http://www.robots.ox.ac.uk/~vgg/data/oxbuildings/>, there are 55 images used for queries for each extraction mode. Each of the query result is used to compute the average precision using C++ code and ground-truth data come with the dataset. The ground-truth data is split into 3 types:

* Good - A nice, clear picture of the object/building.
* OK - More than 25% of the object is clearly visible.
* Junk - Less than 25% of the object is visible, or there are very high levels of occlusion or distortion.

For each query, the system will output 25 images result and that result will be calculated using the provided ground-truth and get the average precision.

The average precision of the current system for each query:

|  |  |  |
| --- | --- | --- |
| queries | color score | edge score |
| all\_souls\_1\_query | 1.59% | 1.46% |
| all\_souls\_2\_query | 3.64% | 1.37% |
| all\_souls\_3\_query | 2.33% | 0.00% |
| all\_souls\_4\_query | 11.33% | 1.28% |
| all\_souls\_5\_query | 2.56% | 0.11% |
| ashmolean\_1\_query | 8.02% | 4.00% |
| ashmolean\_2\_query | 5.55% | 4.00% |
| ashmolean\_3\_query | 5.30% | 4.87% |
| ashmolean\_4\_query | 6.82% | 4.00% |
| ashmolean\_5\_query | 4.90% | 4.00% |
| balliol\_1\_query | 16.67% | 8.33% |
| balliol\_2\_query | 8.33% | 0.00% |
| balliol\_3\_query | 8.33% | 0.00% |
| balliol\_4\_query | 8.33% | 0.19% |
| balliol\_5\_query | 8.33% | 0.00% |
| bodleian\_1\_query | 4.17% | 0.00% |
| bodleian\_2\_query | 8.13% | 0.52% |
| bodleian\_3\_query | 10.46% | 4.17% |
| bodleian\_4\_query | 4.17% | 4.17% |
| bodleian\_5\_query | 4.17% | 4.17% |
| christ\_church\_1\_query | 1.48% | 1.28% |
| christ\_church\_2\_query | 1.76% | 0.00% |
| christ\_church\_3\_query | 2.97% | 1.82% |
| christ\_church\_4\_query | 1.28% | 0.00% |
| christ\_church\_5\_query | 1.46% | 0.00% |
| cornmarket\_1\_query | 25.12% | 0.00% |
| cornmarket\_2\_query | 11.11% | 0.00% |
| cornmarket\_3\_query | 13.76% | 0.00% |
| cornmarket\_4\_query | 22.22% | 0.00% |
| cornmarket\_5\_query | 17.59% | 11.11% |
| hertford\_1\_query | 3.92% | 1.85% |
| hertford\_2\_query | 2.14% | 0.06% |
| hertford\_3\_query | 2.27% | 1.07% |
| hertford\_4\_query | 5.56% | 1.96% |
| hertford\_5\_query | 2.62% | 0.00% |
| keble\_1\_query | 22.62% | 0.00% |
| keble\_2\_query | 17.09% | 0.00% |
| keble\_3\_query | 14.29% | 0.00% |
| keble\_4\_query | 14.29% | 0.00% |
| keble\_5\_query | 14.29% | 0.00% |
| magdalen\_1\_query | 1.85% | 0.00% |
| magdalen\_2\_query | 2.04% | 0.00% |
| magdalen\_3\_query | 1.85% | 0.00% |
| magdalen\_4\_query | 4.41% | 0.00% |
| magdalen\_5\_query | 1.85% | 1.85% |
| pitt\_rivers\_1\_query | 16.67% | 0.00% |
| pitt\_rivers\_2\_query | 16.67% | 0.00% |
| pitt\_rivers\_3\_query | 16.67% | 0.00% |
| pitt\_rivers\_4\_query | 16.67% | 0.00% |
| pitt\_rivers\_5\_query | 16.67% | 0.00% |
| radcliffe\_camera\_1\_query | 5.53% | 0.00% |
| radcliffe\_camera\_2\_query | 4.33% | 0.01% |
| radcliffe\_camera\_3\_query | 2.44% | 0.00% |
| radcliffe\_camera\_4\_query | 0.75% | 0.00% |
| radcliffe\_camera\_5\_query | 1.77% | 0.49% |

## Conclusion

The precision of the system is low on both modes compare to ground-truth data, the reason is that the features extractors cannot get the correct unique features of an image for similarity search.

The reasons make color feature not giving good results:

* The dataset being used is the building dataset, most of them have similar color despite not the same building, this leads to getting wrong data
* The way we constructed the color feature also makes it hard to for matching data. Since an image is split into 5 segments and extract color feature in those separated regions will make 2 pictures of 1 building taken from different angles have totally different histograms.

The reasons make edge feature not giving good results:

* Since this feature only takes generic edges of the whole image instead of certain region or takes certain types of edge like corners or curves, which will make it better at recognize objects for checking similarity.

The accuracy can be increased further by changing to a different way to extract features like: using deep neural networks to get high level features, using SIFT or others high level features will make the results much better than using low level features like this.