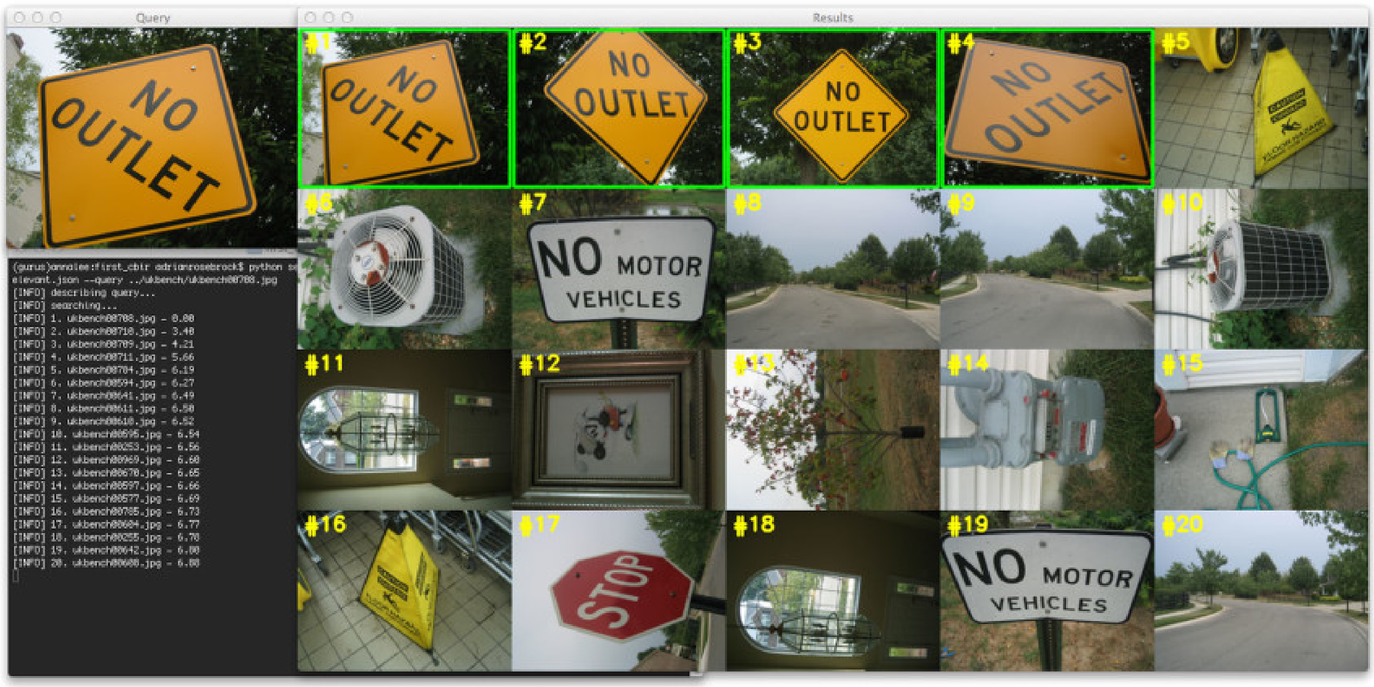
(https://gurus.pyimagesearch.com/)

PyImageSearch Gurus Course

3.2: Your first image search engine



```
(gurus)annalee:first_cbir adrianrosebrock$ python s
elevant.json --query ../ukbench/ukbench00708.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00708.jpg - 0.00
[INFO] 2. ukbench00710.jpg - 3.40
[INFO] 3. ukbench00709.jpg - 4.21
[INFO] 4. ukbench00711.jpg - 5.66
[INFO] 5. ukbench00704.jpg - 6.19
[INFO] 6. ukbench00594.jpg - 6.27
[INFO] 7. ukbench00641.jpg - 6.49
[INFO] 8. ukbench00611.jpg - 6.50
[INFO] 9. ukbench00610.jpg - 6.52
[INFO] 10. ukbench00595.jpg - 6.54
[INFO] 11. ukbench00253.jpg - 6.56
[INFO] 12. ukbench00969.jpg - 6.60
[INFO] 13. ukbench00670.jpg - 6.65
[INFO] 14. ukbench00597.jpg - 6.66
[INFO] 15. ukbench00577.jpg - 6.69
[INFO] 16. ukbench00785.jpg - 6.73
[INFO] 17. ukbench00604.jpg - 6.77
[INFO] 18. ukbench00255.jpg - 6.78
[INFO] 19. ukbench00642.jpg - 6.80
[INFO] 20. ukbench00608.jpg - 6.80
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00708.jpg)

Image search engines are honestly my favorite topic in all of computer vision. Before I started studying computer vision, I focused on Information Retrieval, which is essentially the field of researching, building, and optimizing text search engines like Google, Bing, and DuckDuckGo. Information Retrieval was my first love — and for many years, I thought I was going to focus exclusively on it during my time in college. However, once I started diving into computer vision, my direction changed entirely. I became entranced

Feedback

with computer vision, and I knew I wasn't leaving — but I brought my love of Information Retrieval along with me. Now, instead of building text-based search engines, I build image search engines. Without a doubt, I would consider CBIR to be my primary area of expertise.

You see, image search engines are a lot like text search engines — but instead of submitting *text* to our system and getting relevant documents back, we now submit an *image* and receive images in return that have *similar visual contents*.

Today, we'll build our very first image search engine, where we get to apply the four steps of building any image search engine which we'll discuss in our **next set of lessons (https://gurus.pyimagesearch.com/lessons/the-4-steps-of-building-any-image-search-engine/)**. We'll be using an extension to color histograms that can encode spatial information (i.e. where in the image the color distribution is from) to describe our images. We'll apply this image descriptor to the popular UKBench dataset and extract features from it. Next, we'll construct image search algorithms to make this collection of images visually searchable. We'll round out this lesson by putting our CBIR system to the test and viewing the results of our hard work.

Objectives:

In this lesson, we will:

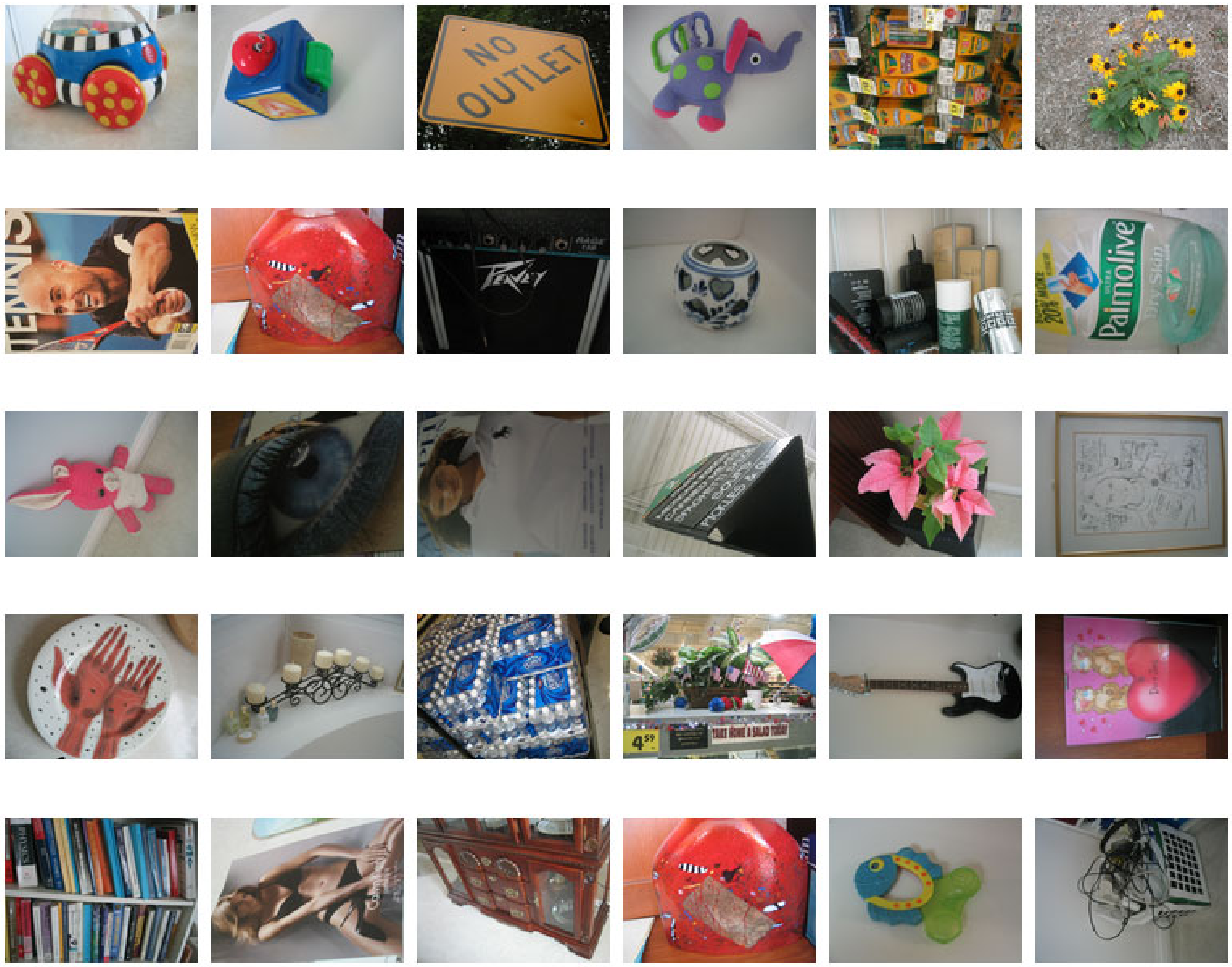
- Introduce the UKBench dataset.
- Apply the four steps of building any image search engine.
- Build our first CBIR system.

The UKBench dataset

The UK Benchmark dataset, or simply UKBench for short, is a dataset created by the Vision Group at the University of Kentucky (http://www.vis.uky.edu/~stewe/ukbench/) and is used for the evaluation of CBIR systems. This dataset has been around for some time (10 years at the time of this writing). It's well studied and many academic papers reference the dataset in their work.

Perhaps most importantly, it's a great beginner dataset — it's large enough (6,376 images) that obtaining decent accuracy is non-trivial, but still small enough that we can extract features from it and evaluate our approach in a reasonable amount of time.

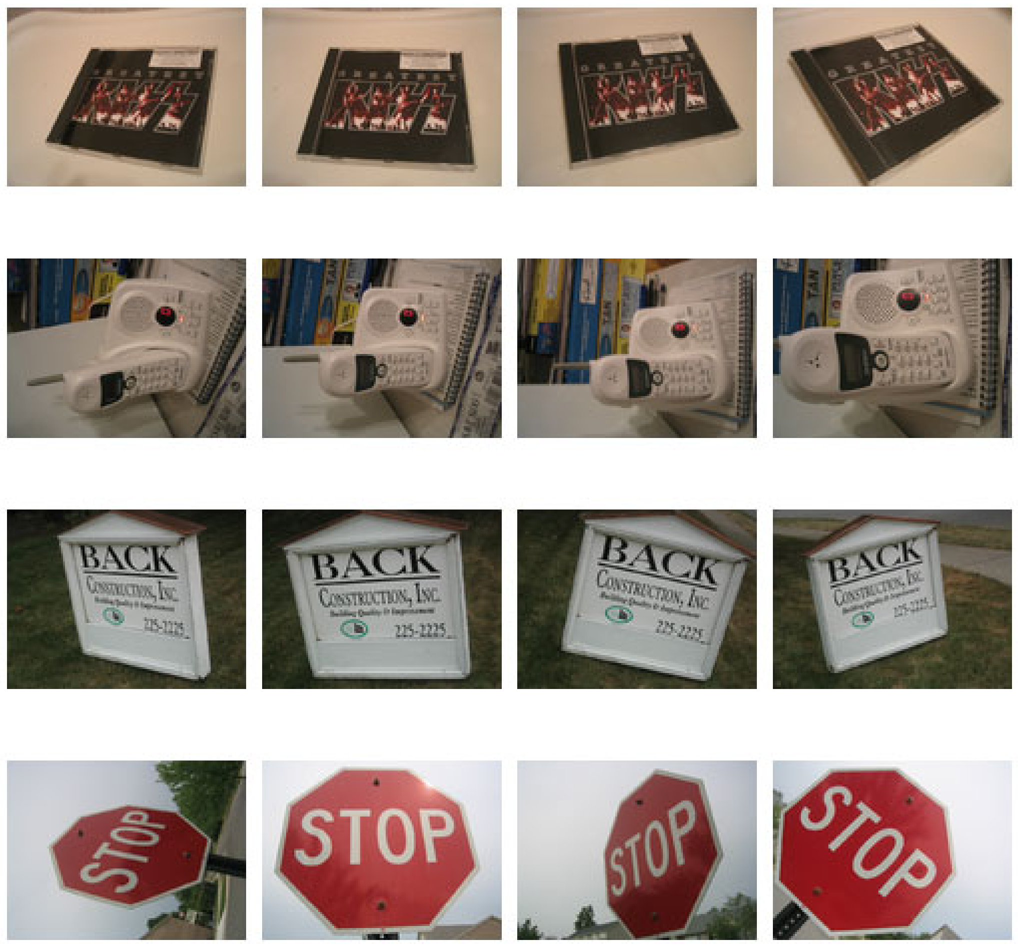
The UKBench dataset consists of both *scenes* and *objects*. Scenes can include park areas, supermarkets, residential areas, and more. Objects may include park benches, road signs, covers of CDs and books, lawn ornaments, children's play toys, etc. A few sample images of the dataset are presented below:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_sample.jpg)

FIGURE 1: A HANDFUL OF EXAMPLE IMAGES FROM THE UKBENCH DATASET. OBJECTS AND SCENES SPAN A LARGE VARIETY OF CATEGORIES.

Furthermore, each scene or object in the UKBench dataset is photographed four times:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_four_examples.jpg)

FIGURE 2: EACH OBJECT/SCENE IN THE UKBENCH DATASET IS REPRESENTED BY FOUR IMAGES.

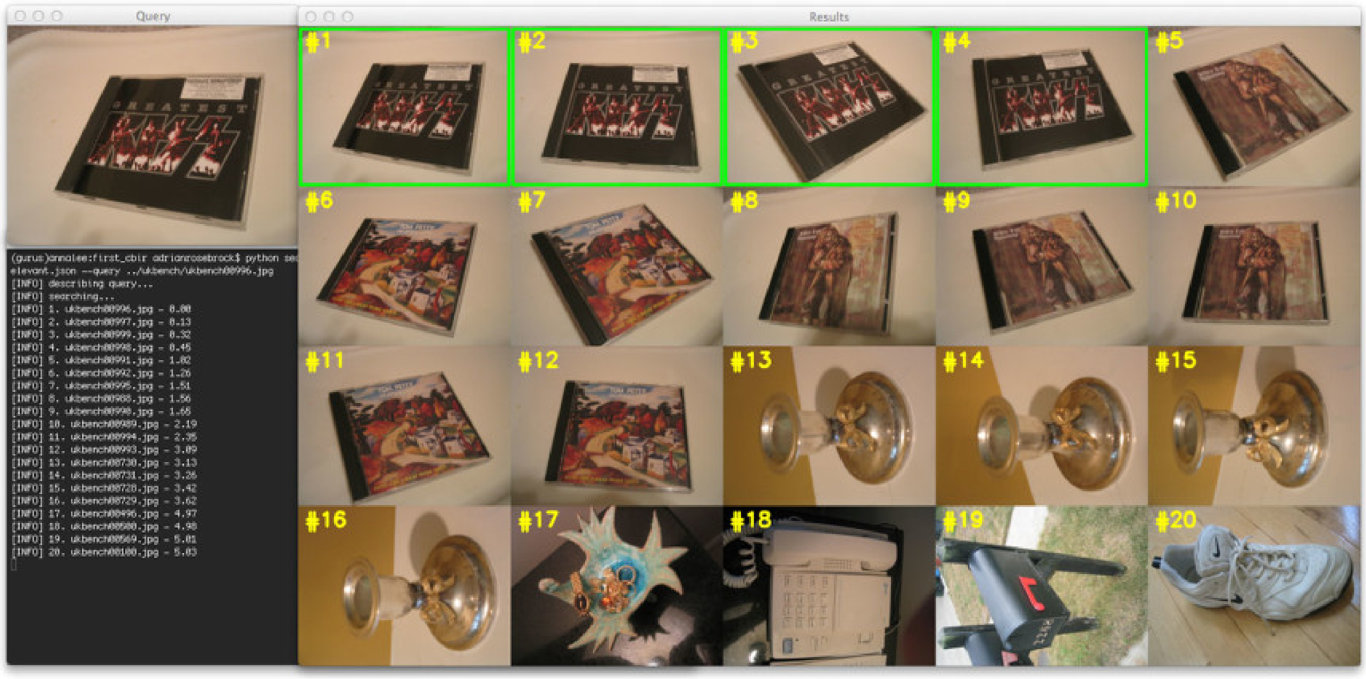
If we were to use the park bench image as our query and submit it to our CBIR system, our image search algorithms should be able to find the original park bench image along with the three other park bench images. Additionally, our algorithm should place the four park bench images in the top four results, respectively.

To get us started, I have sampled 1,000 images from the UKBench dataset. As we work through the rest of this module and learn more advanced techniques, we'll increase the size of the sample until we are eventually working with the entire dataset.

The goal of our CBIR system

Our goal here is to build our first image search engine. Given our dataset of photos, we want to make this dataset "search-able" by creating a "more like this" functionality — this will be a "search by example" image search engine. For instance, if I submit a photo of a park bench, our image search engine should be able to find and retrieve all images that contain the park bench. Furthermore, according to the UKBench dataset, our CBIR system should *also* place the park bench images at the top of the search results list.

Take a look at the example below, where I have submitted a photo of a CD cover and have found relevant images in dataset:



```
(gurus)annalee:first_cbir adrianrosebrock$ python se
elevant.json --query ../ukbench/ukbench00996.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00996.jpg - 0.00
[INFO] 2. ukbench00997.jpg - 0.13
[INFO] 3. ukbench00999.jpg - 0.32
[INFO] 4. ukbench00998.jpg - 0.45
[INFO] 5. ukbench00991.jpg - 1.02
[INFO] 6. ukbench00992.jpg - 1.26
[INFO] 7. ukbench00995.jpg - 1.51
[INFO] 8. ukbench00988.jpg - 1.56
[INFO] 9. ukbench00990.jpg - 1.65
[INFO] 10. ukbench00989.jpg - 2.19
[INFO] 11. ukbench00994.jpg - 2.35
[INFO] 12. ukbench00993.jpg - 3.09
[INFO] 13. ukbench00730.jpg - 3.13
[INFO] 14. ukbench00731.jpg - 3.26
[INFO] 15. ukbench00728.jpg - 3.42
[INFO] 16. ukbench00729.jpg - 3.62
[INFO] 17. ukbench00496.jpg - 4.97
[INFO] 18. ukbench00500.jpg - 4.98
[INFO] 19. ukbench00569.jpg - 5.01
[INFO] 20. ukbench00100.jpg - 5.03
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00996.jpg)

FIGURE 3: THE GOAL OF OUR CBIR SYSTEM IS TO SUBMIT A QUERY IMAGE AND THEN RETRIEVE THE FOUR RELEVANT IMAGES FROM THE UKBENCH DATASET. FURTHERMORE, THE FOUR RELEVANT IMAGES SHOULD BE PLACED IN THE TOP FOUR RESULTS.

In order to build this system, we'll be using a simple, yet effective image descriptor: **the color histogram**.

By utilizing a color histogram as our image descriptor, we'll be relying on the *color distribution* of the image. Because of this, we have to make an important assumption regarding our image search engine:

Assumption: Images that have similar color distributions will be considered relevant to each other. Even if images have dramatically different contents, they will still be considered "similar" provided that their color distributions are similar as well.

This is a really important assumption, but is normally a fair and reasonable assumption to make when using color histograms as image descriptors.

Our project structure

Feedback

Before we dive into implementing the four steps of building an image search engine, let's quickly review the structure of our project:

```
Directory structure of our first CBIR system                                          Shell
|--- pyimagesearch
|    |--- __init__.py
|    |--- cbir
|    |    |---- __init__.py
|    |    |--- dists.py
|    |    |--- hsvdescriptor.py
|    |    |--- resultsmontage.py
|    |    |--- searcher.py
|--- index.py
|--- search.py
```

The project structure itself is quite simple. We'll create the `pyimagesearch` module and the `cbir` sub-module to keep our code organized.

Inside the `cbir` sub-module, we'll find a `dists.py` file which will contain our distance metric/similarity function used to compare two images for similarity.

As the name suggests, the `hsvdescriptor` will implement our color descriptor used to extract feature vectors from our images.

The `resultsmontage` contains a utility class that is used to display the results of a search to our screen. We won't be reviewing this code inside this lesson, as it's very basic code used to construct a large "montage" of result images that can be conveniently displayed to us; however, I do encourage you to download the code and play with the class as it's a good exercise to see how methods like these are implemented.

Next, we have `searcher` , which will encapsulate our `Searcher` class used to perform an actual search.

Finally, we have two driver files, `index.py` , which is used to extract features from our UKBench dataset, followed by `search.py` , which will accept a query image, call the `Searcher` , and then display the results to our screen.

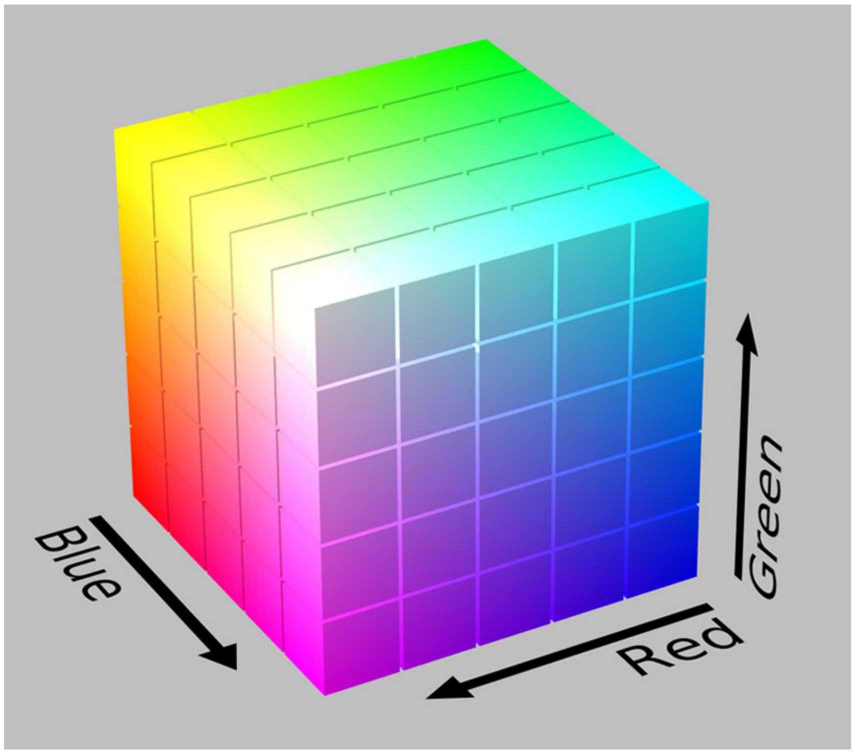
Now that we have a handle on the directory structure of the project, let's get coding.

Step 1: Defining our Image Descriptor

Instead of using a standard color histogram, we are going to apply a few tricks and make it a little more robust and powerful.

Feedback

Our image descriptor will be a 3D color histogram in the HSV color space (Hue, Saturation, Value). Typically, images are represented as a 3-tuple of Red, Green, and Blue (RGB). We often think of the RGB color space as "cube", as shown below:

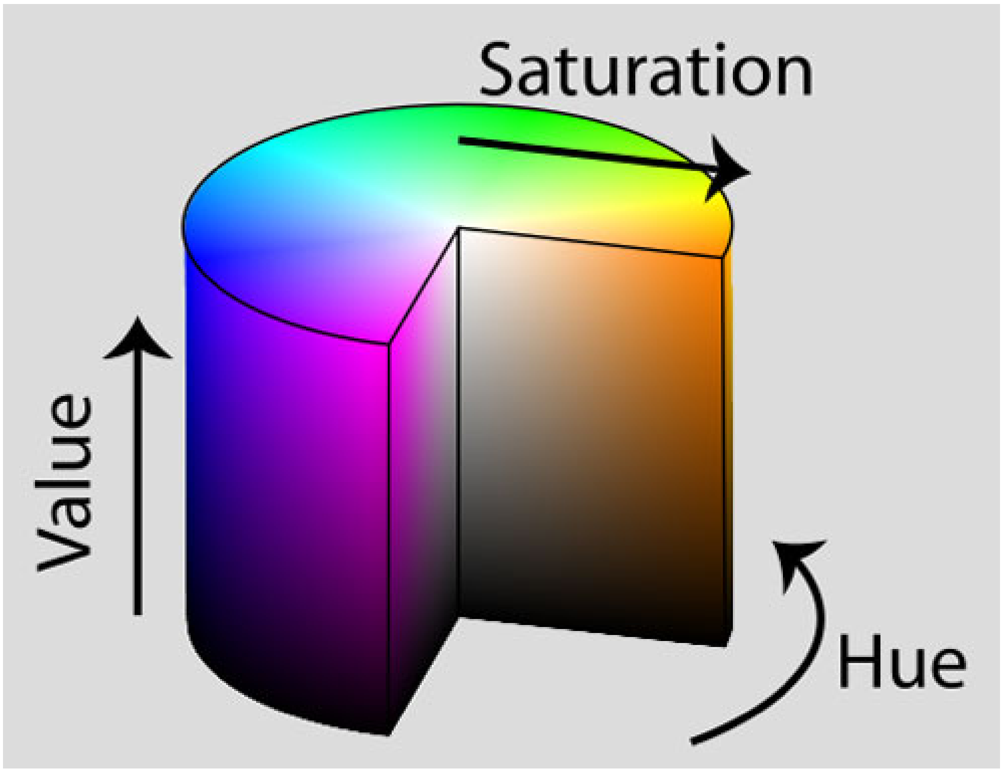


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/color_spaces_rgb_color_cube.jpg)

FIGURE 4: EXAMPLE OF THE RGB CUBE.

Feedback

However, while RGB values are simple to understand, the RGB color space fails to mimic how humans perceive color. Instead, we are going to use the HSV color space which maps pixel intensities into a cylinder:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/color_spaces_hsv_color_cylinder.jpg)

FIGURE 5: EXAMPLE OF THE HSV COLOR CYLINDER.

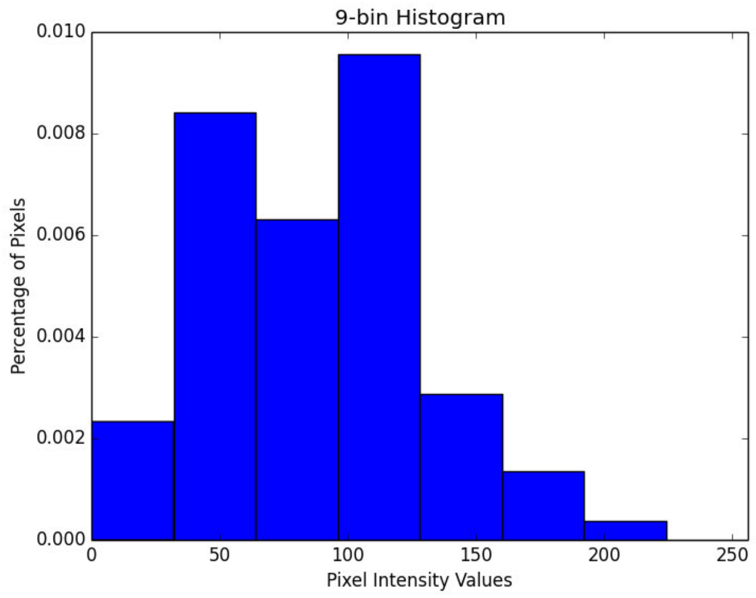
Feedback

As we already know from the **lighting and color spaces lesson (https://gurus.pyimagesearch.com/lessons/lighting-and-color-spaces/)**, there are other color spaces that do an even better job at mimicking how humans perceive color, but let's keep our color model relatively simple for our first image search engine implementation.

So now that we have selected a color space, we now need to define the number of **bins** for our histogram. Histograms are used to give a (rough) sense of the density of pixel intensities in an image. Essentially, our histogram will estimate the probability density of the underlying function, or in this case, the probability *P* of a pixel color *C* occurring in our image *I*.

It's important to note that there is a trade-off with the number of bins you select for your histogram. If you select *too few bins*, then your histogram will have less components and be unable to disambiguate between images with substantially different color distributions. Likewise, if you use *too many bins*, your histogram will have many components, and images with very similar contents may be regarded as "not similar" when, in reality, they are.

Here's an example of a histogram with only a few bins:

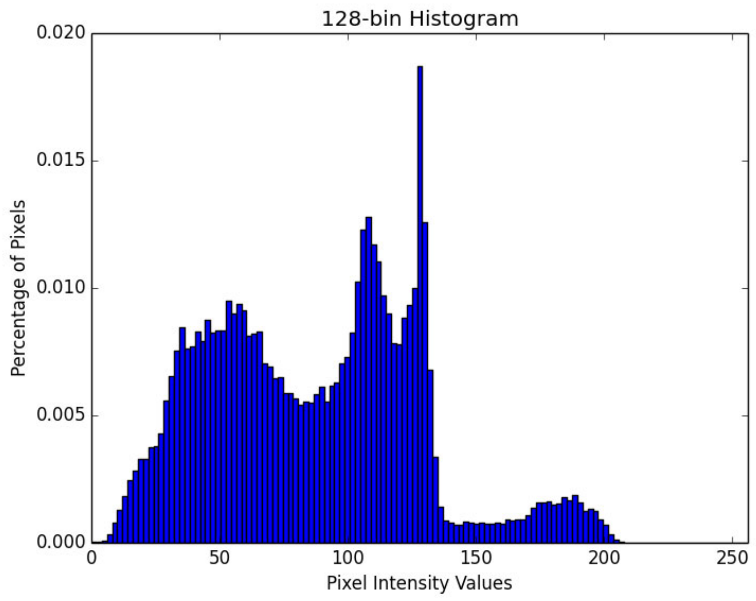


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_histogram_few_bins.jpg)

FIGURE 6: AN EXAMPLE OF A 9-BIN HISTOGRAM. NOTICE HOW THERE ARE VERY FEW BINS FOR A GIVEN PIXEL TO BE PLACED INTO.

Notice how there are very few bins that a pixel can be placed into.

And here's an example of a histogram with lots of bins:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_histogram_many_bins.jpg)

FIGURE 7: AN EXAMPLE OF A 128-BIN HISTOGRAM. NOTICE HOW THERE ARE MANY BINS THAT A GIVEN PIXEL CAN BE PLACED IN.

In the above example, you can see that many bins are utilized, but with the larger number of bins, you lose your ability to "generalize" between images with similar perceptual content, since all of the peaks and valleys of the histogram will have to match in order for two images to be considered "similar".

Personally, I like an iterative, experimental approach to tuning the number of bins. This iterative approach is normally based on the size of my dataset. The smaller that my dataset is, the less bins I use. And if my dataset is large, I use more bins, making my histograms larger and more discriminative.

In general, you'll want to experiment with the number of bins for your color histogram descriptor, as it is dependent on (1) the size of your dataset and (2) how similar the color distributions in your dataset are to each other.

For our UKBench image search engine, we'll be utilizing a 3D color histogram in the HSV color space with 4 bins for the Hue channel, 6 bins for the saturation channel, and 3 bins for the value channel, yielding a total feature vector of dimension *4 x 6 x 3 = 72*.

This means that for every image in our dataset, no matter if the image is *36 x 36* pixels or *2000 x 1800* pixels, all images will be abstractly represented and quantified using only a list of *72* floating point numbers. If you need a quick refresher on histograms, I would suggest reading the **introduction to color histograms lesson (https://gurus.pyimagesearch.com/lessons/histograms/)**, followed by our lesson on **using color histograms as features (https://gurus.pyimagesearch.com/lessons/color-histograms/)**.

Feedback

Anyway, enough talk. Let's get into some code, starting with the `hsvdescriptor.py` file:

hsvdescriptor.py Python

```python
1  # import the necessary packages
2  import numpy as np
3  import cv2
4  import imutils
5
6  class HSVDescriptor:
7      def __init__(self, bins):
8          # store the number of bins for the histogram
9          self.bins = bins
10
11     def describe(self, image):
12         # convert the image to the HSV color space and initialize
13         # the features used to quantify the image
14         image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
15         features = []
16
17         # grab the dimensions and compute the center of the image
18         (h, w) = image.shape[:2]
19         (cX, cY) = (int(w * 0.5), int(h * 0.5))
```

We then define our `HSVDescriptor` class on **Line 6**. This class will encapsulate all the necessary logic to extract our 3D HSV color histogram from our images.

The `__init__` method of the `HSVDescriptor` takes only a single argument, `bins` , which is the number of bins for our color histogram.

We can then define our `describe` method on **Line 11**. This method requires an `image` , which is the image we want to describe.

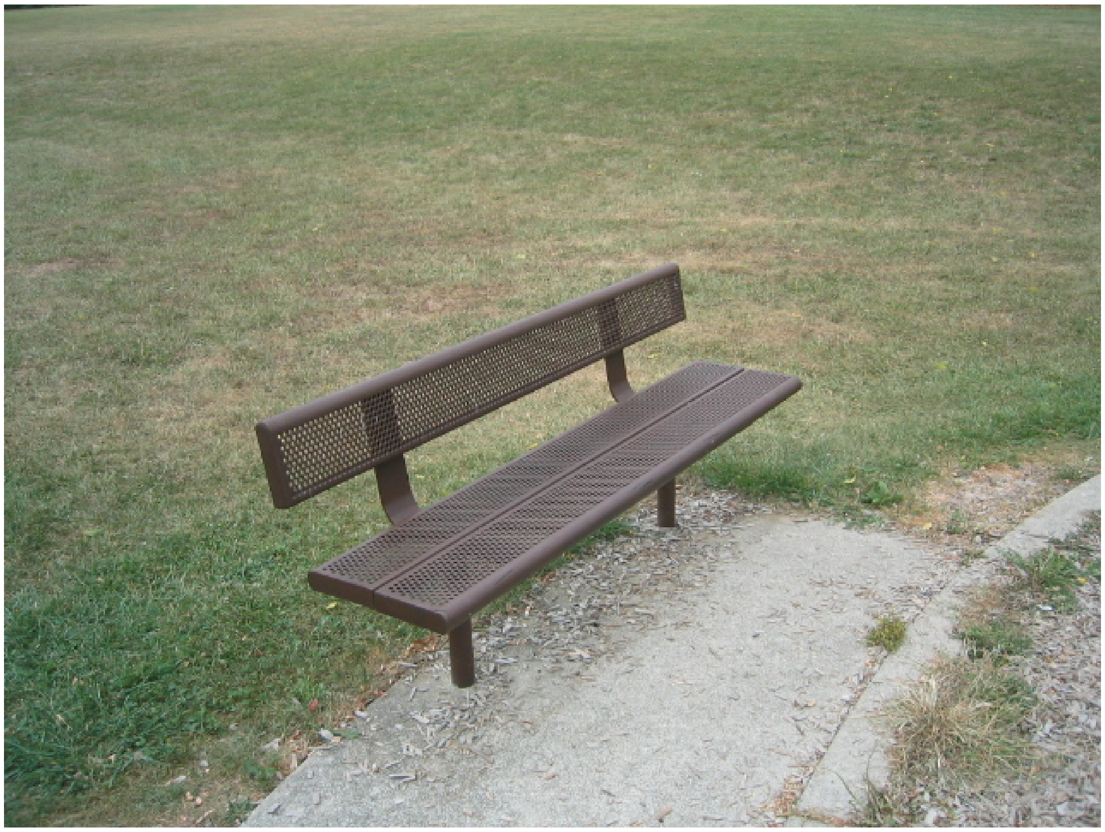
Inside of our `describe` method, we'll convert from the RGB color space (or rather, the BGR color space; OpenCV represents RGB images as NumPy arrays, but in reverse order) to the HSV color space, followed by initializing our list of `features` to quantify and represent our `image` .

Lines 18 and 19 simply grab the dimensions of the image and compute the center *(x, y)* coordinates.

So now the hard work starts.

Instead of computing a 3D HSV color histogram for the ***entire*** image, let's instead compute a 3D HSV color histogram for different ***regions*** of the image.

Using ***regions***-based histograms rather than ***global***-histograms allows us to simulate locality in a color distribution. For example, take a look at this image below:



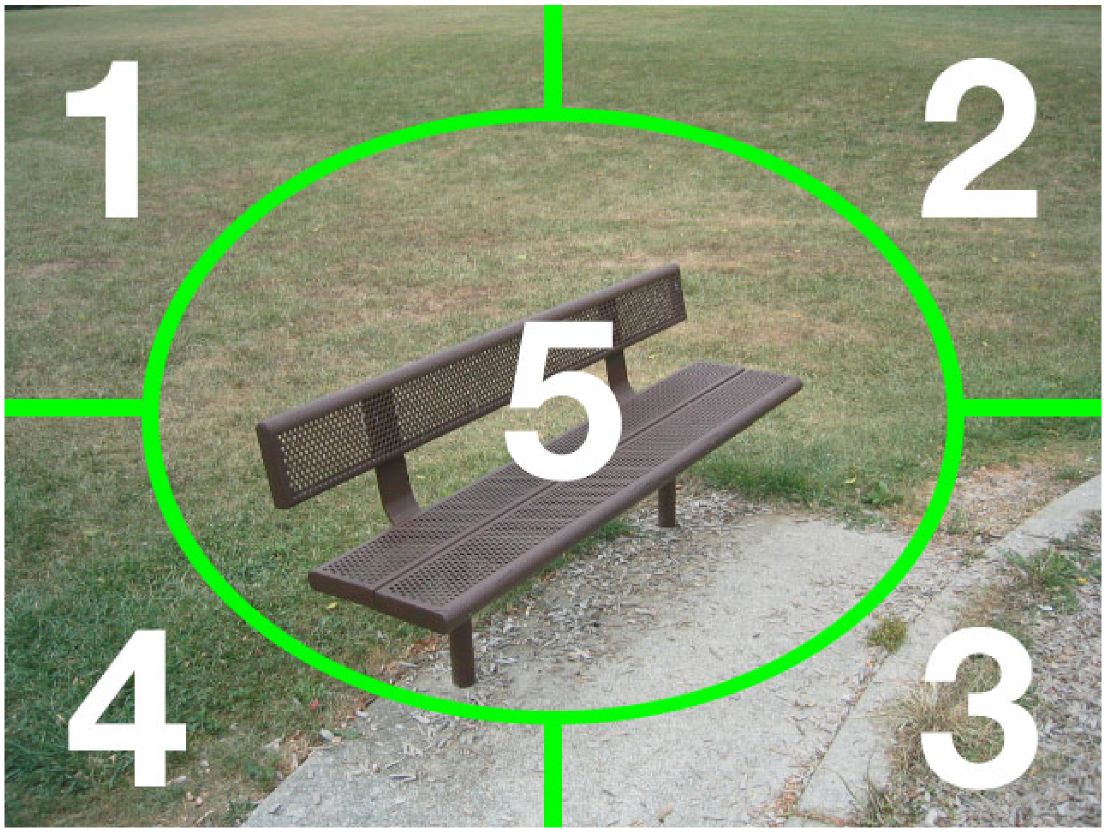
[(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_example_query.jpg)](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_example_query.jpg)

FIGURE 8: OUR EXAMPLE QUERY IMAGE.

Feedback

In this photo, we can clearly see a brown park bench at the center of the image and green grass/gray concrete surrounding it. Using a global histogram, we would be unable to determine *where* in the image the “green” grass occurs or “gray” concrete occurs and where the “brown” bench occurs. Instead, we would just know that there exists some percentage of green, some percentage of brown, and some percentage of gray.

To remedy this problem, we can compute color histograms in regions of the image:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_segments.jpg)

FIGURE 9: EXAMPLE OF DIVIDING OUR IMAGE INTO FIVE DIFFERENT SEGMENTS.

Feedback

As the above figure demonstrates, we are going to divide our image into five different regions: (1) the top-left corner, (2) the top-right corner, (3) the bottom-right corner, (4) the bottom-left corner, and finally (5) the center of the image.

By utilizing these regions, we'll be able to mimic a crude form of localization, being able to represent our image as having shades of green in the top-left, top-right, bottom-left, and bottom-right corners, while having a brown park bench in the center.

That all said, here is the code to create our region-based color descriptor:

hsvdescriptor.py Python

```
21        # divide the image into four rectangles/segments (top-left,
22        # top-right, bottom-right, bottom-left)
23        segments = [(0, cX, 0, cY), (cX, w, 0, cY), (cX, w, cY, h),
24            (0, cX, cY, h)]
25
26        # construct an elliptical mask representing the center of the
27        # image
28        (axesX, axesY) = (int(w * 0.75) // 2, int(h * 0.75) // 2)
29        ellipMask = np.zeros(image.shape[:2], dtype="uint8")
30        cv2.ellipse(ellipMask, (cX, cY), (axesX, axesY), 0, 0, 360, 255, -1)
31
32        # loop over the segments
33        for (startX, endX, startY, endY) in segments:
34            # construct a mask for each corner of the image, subtracting
35            # the elliptical center from it
36            cornerMask = np.zeros(image.shape[:2], dtype="uint8")
37            cv2.rectangle(cornerMask, (startX, startY), (endX, endY), 255, -1)
38            cornerMask = cv2.subtract(cornerMask, ellipMask)
39
40            # extract a color histogram from the image, then update the
41            # feature vector
42            hist = self.histogram(image, cornerMask)
43            features.extend(hist)
44
45        # extract a color histogram from the elliptical region and
46        # update the feature vector
47        hist = self.histogram(image, ellipMask)
48        features.extend(hist)
49
50        # return the feature vector
51        return np.array(features)
```

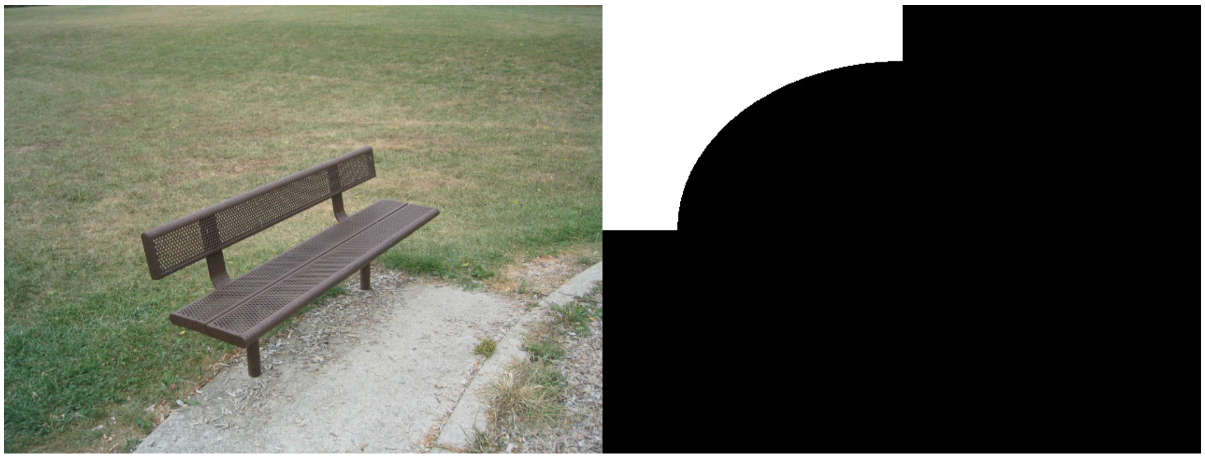
Lines 23 and 24 start by defining the indexes of our top-left, top-right, bottom-right, and bottom-left regions, respectively.

From there, we'll need to construct an ellipse to represent the center region of the image. We'll do this by defining an ellipse radius that is 75% of the width and height of the image on **Line 28**.

We then initialize a blank image (filled with zeros to represent a black background) with the same dimensions of the image we want to describe on **Line 29**. Finally, let's draw the actual ellipse on **Line 30** using the `cv2.ellipse` function.

We then allocate memory for each corner mask on **Line 36**, draw a white rectangle representing the corner of the image on **Line 37**, and then subtract the center ellipse from the rectangle on **Line 38**.

If we were to animate this process of looping over the corner segments, it would look something like this:



[(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_mask_segments.gif)](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_mask_segments.gif)

FIGURE 10: CONSTRUCTING MASKS FOR EACH REGION OF THE IMAGE WE WANT TO EXTRACT FEATURES FROM.

As this animation shows, we are examining each of the corner segments individually, removing the center of the ellipse from the rectangle at each iteration.

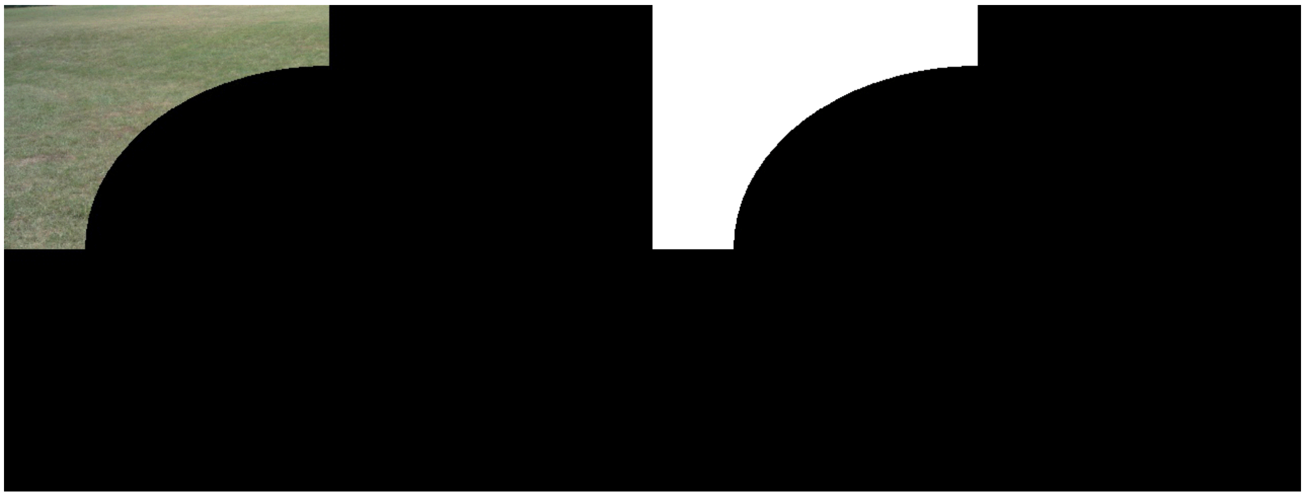
So you may be wondering, *"Aren't we supposed to be extracting color histograms from our image? Why are we doing all this 'masking' business?"*

Great question.

The reason is we need the mask to instruct the OpenCV histogram function where to extract the color histogram from.

Remember, our goal is to describe each of these segments *individually*. The most efficient way of representing each of these segments is to use a *mask*. Only *(x, y)* coordinates in the image that have a corresponding *(x, y)* location in the mask with a white (255) pixel value will be included in the histogram calculation. If the pixel value for an *(x, y)* coordinate in the mask has a value of black (0), it will be ignored.

To reiterate this concept of only including pixels in the histogram with a corresponding mask value of white, take a look at the following animation:



[(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_applied_mask_segments.gif)](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_applied_mask_segments.gif)

FIGURE 11: APPLYING THE MASKED REGIONS TO THE IMAGE. NOTICE HOW ONLY THE PIXELS IN THE *LEFT* IMAGE ARE SHOWN IF THEY HAVE A CORRESPONDING WHITE MASK VALUE IN THE IMAGE ON THE *RIGHT*.

As you can see, only pixels in the masked region of the image will be included in the histogram calculation.

Makes sense now, right?

So now for each of our segments, we make a call to the `histogram` method on **Line 42**, extract the color histogram by using the `image` we want to compute features from as the first argument and the `mask` representing the region we want to describe as the second argument.

The `histogram` method then returns an HSV color histogram representing the current region, which we append to our `features` list.

Lines 47 and 48 extract a color histogram for the center (ellipse) region and update the `features` list a well.

Finally, **Line 51** returns our feature vector to the calling function.

Now, let's quickly look at our actual `histogram` method:

hsvdescriptor.py Python

```
53      def histogram(self, image, mask=None):
54          # extract a 3D color histogram from the masked region of the
55          # image, using the supplied number of bins per channel; then
56          # normalize the histogram
57          hist = cv2.calcHist([image], [0, 1, 2], mask, self.bins,
58              [0, 180, 0, 256, 0, 256])
59
60          # handle if we are using OpenCV 2.4
61          if imutils.is_cv2():
62              hist = cv2.normalize(hist).flatten()
63
64          # otherwise handle for OpenCV 3+
65          else:
66              hist = cv2.normalize(hist, hist).flatten()
67
68          # return the histogram
69          return hist
```

Our `histogram` method requires two arguments: the first is the `image` that we want to describe, and the second is the `mask` that represents the ***region*** of the image we want to describe.

Calculating the histogram of the masked region of the image is handled on **Lines 57 and 58** by making a call to `cv2.calcHist` using the supplied number of `bins` from our constructor.

Our color histogram is normalized on **Line 62 or 66** (depending on OpenCV version) to obtain scale invariance. This means that if we computed a color histogram for two identical images where one was 50% larger than the other, our color histograms would be (nearly) identical. It is ***very important*** that you normalize your color histograms, so each histogram is represented by the relative **percentage** counts for a particular bin and not the **integer** counts for each bin. Again, performing this normalization will ensure that images with similar content but dramatically different dimensions will still be "similar" once we apply our similarity function.

Finally, the normalized, 3D HSV color histogram is returned to the calling function on **Line 69**.

Step 2: Extracting Features from Our Dataset

Now that we have our image descriptor defined, we can move on to Step 2, and extract features (i.e. color histograms) from each image in our dataset. The process of extracting feature vectors, storing them, and optimizing for efficient comparison is commonly called "indexing". We'll only be performing feature extraction and storage in this lesson, but in future lessons, we'll look into specialized data structures that can greatly increase our search speed.

Let's get started coding `index.py` :

```python
1  # import the necessary packages
2  from __future__ import print_function
3  from pyimagesearch.cbir import HSVDescriptor
4  from imutils import paths
5  import progressbar
6  import argparse
7  import cv2
8  
9  # construct the argument parser and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-d", "--dataset", required=True,
12     help = "Path to the directory that contains the images to be indexed")
13 ap.add_argument("-i", "--index", required=True,
14     help = "Path to where the features index will be stored")
15 args = vars(ap.parse_args())
16 
17 # initialize the color descriptor and open the output index file for writing
18 desc = HSVDescriptor((4, 6, 3))
19 output = open(args["index"], "w")
```

We start off by importing the packages we'll need on **Lines 2-7**. These packages should all look familiar, except for `progressbar`. There is a new package that I'm introducing in this lesson and one that we'll be using later in the course. The `progressbar` package is totally unrelated to computer vision, but as the name suggests, it allows us to display a nicely formatted progress bar in our terminal, detailing the completion percentage, along with an ETA.

If you don't have progressbar (https://pypi.python.org/pypi/progressbar2) installed on your system, let pip install it for you:

How to install progressbar Shell

```shell
1 $ pip install progressbar2
```

We'll need two command line arguments to run `index.py`. The first is `--dataset`, which will point to the directory containing the UKBench images. We'll also need an `--index`, or the output `.csv` file where our extracted feature vectors will be stored.

Finally, we initialize our `HSVDescriptor` using 4 bins for the Hue channel, 6 bins for Saturation, and 3 bins for Value. We also open our output index file for writing.

index.py Python

```python
21 # grab the list of image paths and initialize the progress bar
22 imagePaths = list(paths.list_images(args["dataset"]))
23 widgets = ["Indexing: ", progressbar.Percentage(), " ", progressbar.Bar(), " ", progressbar.ETA(
24 pbar = progressbar.ProgressBar(maxval=len(imagePaths), widgets=widgets)
25 pbar.start()
26 
27 # loop over the image paths in the dataset directory
28 for (i, imagePath) in enumerate(sorted(imagePaths)):
29     # extract the image filename (i.e. the unique image ID) from the image
30     # path, then load the image itself
31     filename = imagePath[imagePath.rfind("/") + 1:]
32     image = cv2.imread(imagePath)
33 
34     # describe the image
35     features = desc.describe(image)
36 
37     # write the features to our index file
38     features = [str(x) for x in features]
39     output.write("{},{}\n".format(filename, ",".join(features)))
40     pbar.update(i)
41 
42 # close the output index file
43 pbar.finish()
44 print("[INFO] indexed {} images".format(len(imagePaths)))
45 output.close()
```

Lines 22-24 grab the paths to the UKBench images, followed by initializing our progress bar, so we can keep track of the feature extraction progress.

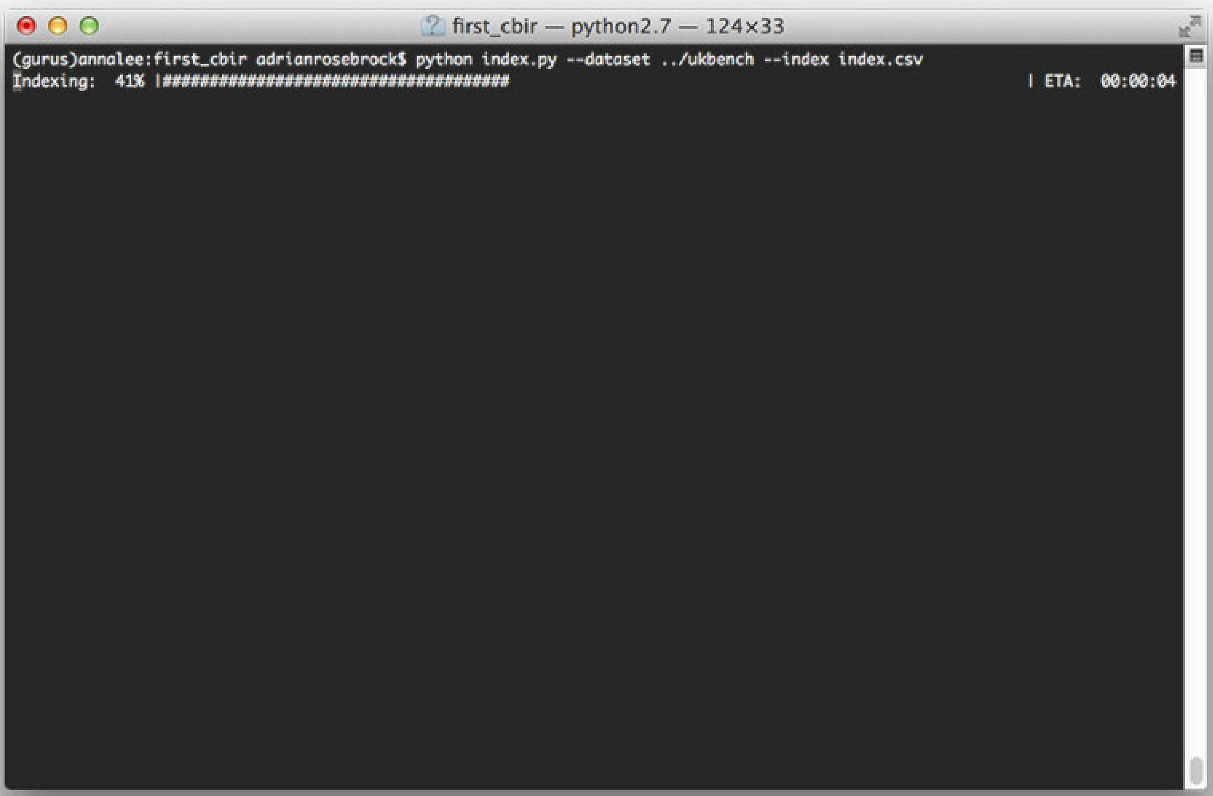
We start looping over each of the images in our dataset on **Line 28** and extract features on **Line 35**. This feature vector contains representations for each of the 5 image regions we described in **Step 1**. Each section is represented by a histogram with *4 x 6 x 3 = 72* entries. Given 5 entries, our overall feature vector is *5 x 72 = 360* dimensionality. Thus, each image is quantified and represented using 360 numbers.

We then write the features to our output file on **Line 39**. Notice how we are including *both* the filename (which is assumed to be unique) of the input image along with the feature vector.

To execute our script, just issue the following command:

```
index.py                                                                 Shell
$ python index.py --dataset ../ukbench --index index.csv
```

Which will kick off the feature extraction process. Below, you can see an example of the script executing on my system:



```
(gurus)annalee:first_cbir adrianrosebrock$ python index.py --dataset ../ukbench --index index.csv
Indexing:  41% |#######################################                                    | ETA:  00:00:04
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_indexing.jpg)

FIGURE 12: EXTRACTING COLOR FEATURES FROM THE UKBENCH DATASET.

When the script finishes, you'll see you have a new file named `index.csv` in your current working directory. Doing a line count on the `index.csv` file will reveal that we have extracted color histograms from our 1,000 image UKBench dataset sample:

Indexing the UKBench dataset — Shell

```
$ wc -l index.csv
    1000 index.csv
```

Step 3: The Searcher class

Now that we've extracted features from our dataset, we need a method to *compare* these features for similarity. That's where Step 3 comes in — we are now ready to create a class that will handle comparing feature vectors, and thus the actual similarity of two images:

searcher.py — Python

```python
1  # import the necessary packages
2  from . import dists
3  import csv
4
5  class Searcher:
6      def __init__(self, dbPath):
7          # store the database path
8          self.dbPath = dbPath
9
10     def search(self, queryFeatures, numResults=10):
11         # initialize the results dictionary
12         results = {}
```

We start off by importing two packages, `dists`, which contains our similarity metric (which we'll define later in this section), and `csv`, which is a built-in Python module used to parse `.csv` files.

From there, let's define our `Searcher` class on **Line 5**. The constructor for our `Searcher` will only require a single argument, `dbPath`, which is the path to where our `index.csv` file resides on disk.

To actually perform a search, we'll be making a call to the `search` method on **Line 10**. This method will take two parameters, the `queryFeatures` extracted from the query image (i.e. the image we'll be submitting to our CBIR system and asking for similar images to), and `numResults`, which is the maximum number of results to return.

Finally, we initialize our `results` dictionary on **Line 12**. A dictionary is a good data-type in this situation, as it will allow us to use the (unique) `imageID` for a given image as the key and the similarity to the query as the value.

searcher.py Python

```python
14          # open the database for reading
15          with open(self.dbPath) as f:
16              # initialize the CSV reader
17              reader = csv.reader(f)
18
19              # loop over the rows in the index
20              for row in reader:
21                  # parse out the image ID and features, then compute the chi-squared
22                  # distance between the features in our database and the query features
23                  features = [float(x) for x in row[1:]]
24                  d = dists.chi2_distance(features, queryFeatures)
25
26                  # now that we have the distance between the two feature vectors, we
27                  # can update the results dictionary -- the key is the current image
28                  # ID in the database and the value is the distance we just computed,
29                  # representing how 'similar' the image in the database is to our query
30                  results[row[0]] = d
31
32              # close the reader
33              f.close()
34
35          # sort our results, so that the smaller distances (i.e. the more relevant images)
36          # are at the front of the list)
37          results = sorted([(v, k) for (k, v) in results.items()])
38
39          # return the results
40          return results[:numResults]
```

We open up our `index.csv` file on **Line 15**, grab a handle to our CSV reader on **Line 17**, and start looping over each row of the `index.csv` file on **Line 20**.

For each row, we grab the color histograms associated with the indexed image and then compare it to the query image features using the `chi2_distance` (**Line 24**), which I'll define in a second.

Our `results` dictionary is updated on **Line 30** using the unique image filename as the key and the similarity of the query image to the indexed image as the value.

Last, all we have to do is sort the results dictionary according to the similarity value in ascending order.

Images that have a chi-squared similarity of 0 will be deemed to be ***identical*** to each other. As the chi-squared similarity value increases, the images are considered to be ***less similar*** to each other.

Let's go ahead and define the `chi2_distance` function in the `dists.py` file:

```
dists.py                                                                Python
```

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

def chi2_distance(histA, histB, eps=1e-10):
    # compute the chi-squared distance
    d = 0.5 * np.sum(((histA - histB) ** 2) / (histA + histB + eps))

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

Our `chi2_distance` function requires two arguments, which are the two histograms we want to compare for similarity. An optional `eps` value is used to prevent division-by-zero errors.

The function gets its name from the Pearson's chi-squared test statistic which is used to compare discrete probability distributions.

Since we are comparing color histograms, which are by definition probability distributions, the chi-squared function is an excellent choice.

When it comes to comparing histograms, it's often the case that the difference between large bins vs. small bins is less important and should be weighted as such — and this is exactly what the chi-squared distance function does when dividing the squared difference of the histograms by the magnitude of the combined histograms.

Feedback

Step 4: Performing a search

Performing the search is actually the easiest part — we have already defined our image descriptor, extracted feature vectors, and defined a method to compare images for similarity. All we need now is a driver that imports of all the packages we defined earlier and use them in conjunction with each other to build a full-fledged Content-based Image Retrieval System.

Let's work on a final file, `search.py`, and put all the pieces together:

search.py Python

```python
1  # import the necessary packages
2  from __future__ import print_function
3  from pyimagesearch.cbir import ResultsMontage
4  from pyimagesearch.cbir import HSVDescriptor
5  from pyimagesearch.cbir import Searcher
6  import argparse
7  import imutils
8  import json
9  import cv2
10 
11 # construct the argument parser and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-i", "--index", required=True, help="Path to where the features index will be s
14 ap.add_argument("-q", "--query", required=True, help="Path to the query image")
15 ap.add_argument("-d", "--dataset", required=True, help="Path to the original dataset directory")
16 ap.add_argument("-r", "--relevant", required=True, help = "Path to relevant dictionary")
17 args = vars(ap.parse_args())
```

Lines 2-9 import the classes we have defined in Step 1 and Step 3 of this lesson. We'll also import the `ResultsMontage` class that is used to take our search results and display them in a nice formatted way on our screen.

Lines 12-17 parse our command line arguments. The first argument is `--index`, which is the path to our `index.csv` file that stores our extracted feature vectors. The `--query` switch is the path to our query image we are going to submit to our CBIR system. We'll define the `--dataset` switch, which is the path to the UKBench dataset directory, so we can load the result images from disk after performing a search. Finally, we define a `--relevant` switch that will point to a JSON file indicating which images contain the same scene/object. This JSON file is named `relevant.json` and is included in the download of this lesson.

It's very important that we have this `relevant.json` file — mainly so we can (visually) evaluate the performance of our CBIR system. Remember how in the **UKBench dataset** section of this lesson, I mentioned there are four images of each scene/object in the dataset? Well, this `.json` file contains mappings for each of the images in the dataset, specifying which other images are relevant.

For example, take a look at the first few entries in the `relevant.json` file:

relevant.json JavaScript
```javascript
{
    "ukbench00682.jpg": ["ukbench00680.jpg", "ukbench00681.jpg", "ukbench00682.jpg", "ukbench0068
    "ukbench00740.jpg": ["ukbench00740.jpg", "ukbench00741.jpg", "ukbench00742.jpg", "ukbench0074
    "ukbench00891.jpg": ["ukbench00888.jpg", "ukbench00889.jpg", "ukbench00890.jpg", "ukbench0089
    ...
}
```

The `relevant.json` file simply defines a dictionary where the key is the image filename (i.e. the query), and the value is a list of the four relevant images in the UKBench dataset. Since an image is relevant to itself, it's also included in the list. We'll be using this file to highlight which images are relevant to our query in our search results.

Let's continue on:

search.py Python

```python
19 # initialize the image descriptor and results montage
20 desc = HSVDescriptor((4, 6, 3))
21 montage = ResultsMontage((240, 320), 5, 20)
22 relevant = json.loads(open(args["relevant"]).read())
23
24 # load the relevant queries dictionary and look up the relevant results for the
25 # query image
26 queryFilename = args["query"][args["query"].rfind("/") + 1:]
27 queryRelevant = relevant[queryFilename]
```

Line 20 defines our `HSVDescriptor` using the same number of bins as in **Step 2**.

We'll define our `ResultsMontage` on **Line 21**. The `ResultsMontage` requires three arguments. The first is the target output size of each entry in the results montage. Here, we specify that each result image will have a width of 320 pixels and a height of 240 pixels, where we ignore the aspect ratio. The second parameter is the number of images per row in our montage — we'll specify a value of 5 indicating five images per row. Finally, the last parameter is the number of results to include in the montage, where we'll display the top-20 results.

Line 22 loads our relevant image mappings from disk, and **Lines 26 and 27** use the `relevant` dictionary to determine the four relevant images for the input query image.

Now, we can perform the actual search:

search.py Python

```
29 # load the query image, display it, and describe it
30 print("[INFO] describing query...")
31 query = cv2.imread(args["query"])
32 cv2.imshow("Query", imutils.resize(query, width=320))
33 features = desc.describe(query)
34
35 # perform the search
36 print("[INFO] searching...")
37 searcher = Searcher(args["index"])
38 results = searcher.search(features, numResults=20)
39
40 # loop over the results
41 for (i, (score, resultID)) in enumerate(results):
42     # load the result image and display it
43     print("[INFO] {result_num}. {result} - {score:.2f}".format(result_num=i + 1, result=resultID
44         score=score))
45     result = cv2.imread("{}/{}".format(args["dataset"], resultID))
46     montage.addResult(result, text="#{}".format(i + 1), highlight=resultID in queryRelevant)
47
48 # show the output image of results
49 cv2.imshow("Results", imutils.resize(montage.montage, height=700))
50 cv2.waitKey(0)
```

Before we can perform a search, we need to first load the query image from disk and extract features from it (**Lines 31-33**).

To perform the actual search, we use our `Searcher` class on **Lines 37 and 38**. The `Searcher` class accepts the path to our `index.csv` file, while the `search` method of the `Searcher` class compares the query features to the set of features in the `index.csv` file, returning the set of relevant `results`.

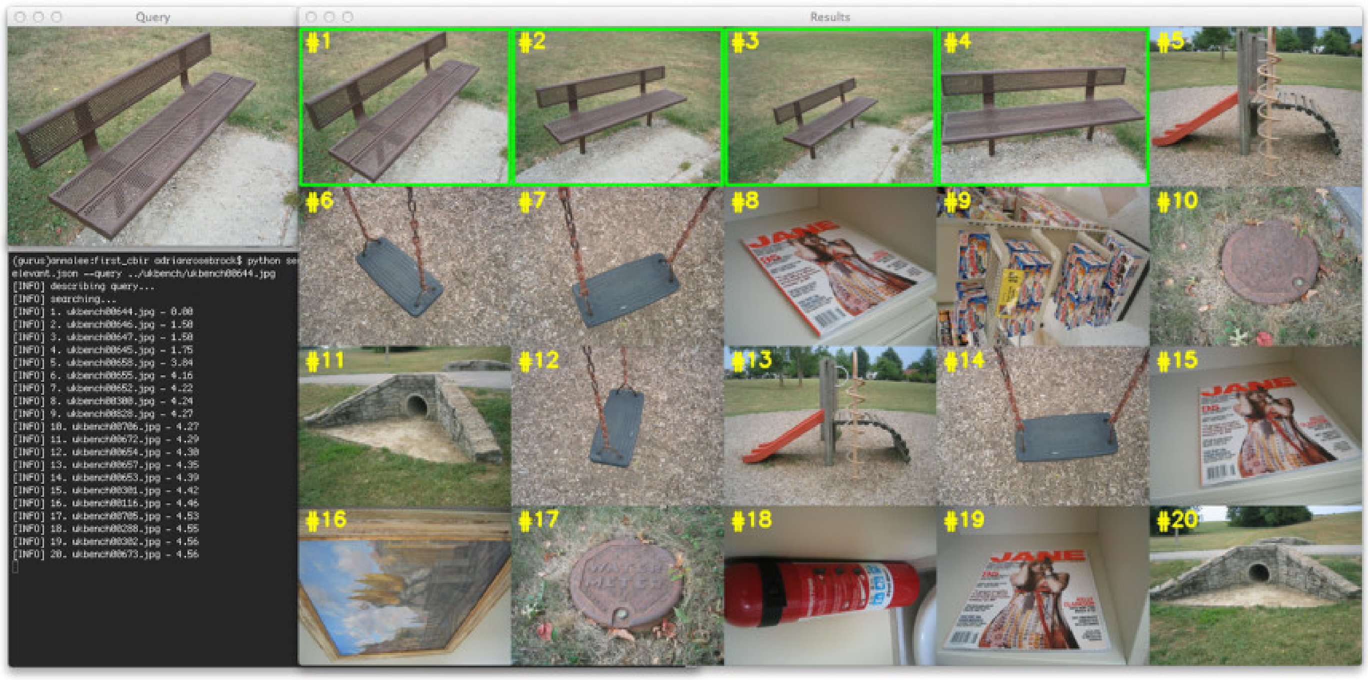
We loop over each of the `results` individually on **Line 41**, adding the result image to the montage, including the result number, and whether or not the image should be highlighted (i.e. the result is part of the relevant images for the query).

Finally, **Lines 49 and 50** displays the nicely formatted results montage on our screen.

Our CBIR system in action

After all this hard work, let's see what our CBIR system can do. Open up a terminal and issue the following command:

```
search.py                                                                        Python
$ python search.py --index index.csv --dataset ../ukbench --relevant ../ukbench/relevant.json \
    --query ../ukbench/ukbench00644.jpg
```



```
(gurus)annalee:first_cbir adrianrosebrock$ python se
elevant.json --query ../ukbench/ukbench00644.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00644.jpg - 0.00
[INFO] 2. ukbench00646.jpg - 1.50
[INFO] 3. ukbench00647.jpg - 1.50
[INFO] 4. ukbench00645.jpg - 1.75
[INFO] 5. ukbench00658.jpg - 3.84
[INFO] 6. ukbench00655.jpg - 4.16
[INFO] 7. ukbench00652.jpg - 4.22
[INFO] 8. ukbench00308.jpg - 4.24
[INFO] 9. ukbench00828.jpg - 4.27
[INFO] 10. ukbench00706.jpg - 4.27
[INFO] 11. ukbench00672.jpg - 4.29
[INFO] 12. ukbench00654.jpg - 4.30
[INFO] 13. ukbench00657.jpg - 4.35
[INFO] 14. ukbench00653.jpg - 4.39
[INFO] 15. ukbench00301.jpg - 4.42
[INFO] 16. ukbench00116.jpg - 4.46
[INFO] 17. ukbench00705.jpg - 4.53
[INFO] 18. ukbench00288.jpg - 4.55
[INFO] 19. ukbench00302.jpg - 4.56
[INFO] 20. ukbench00673.jpg - 4.56
```

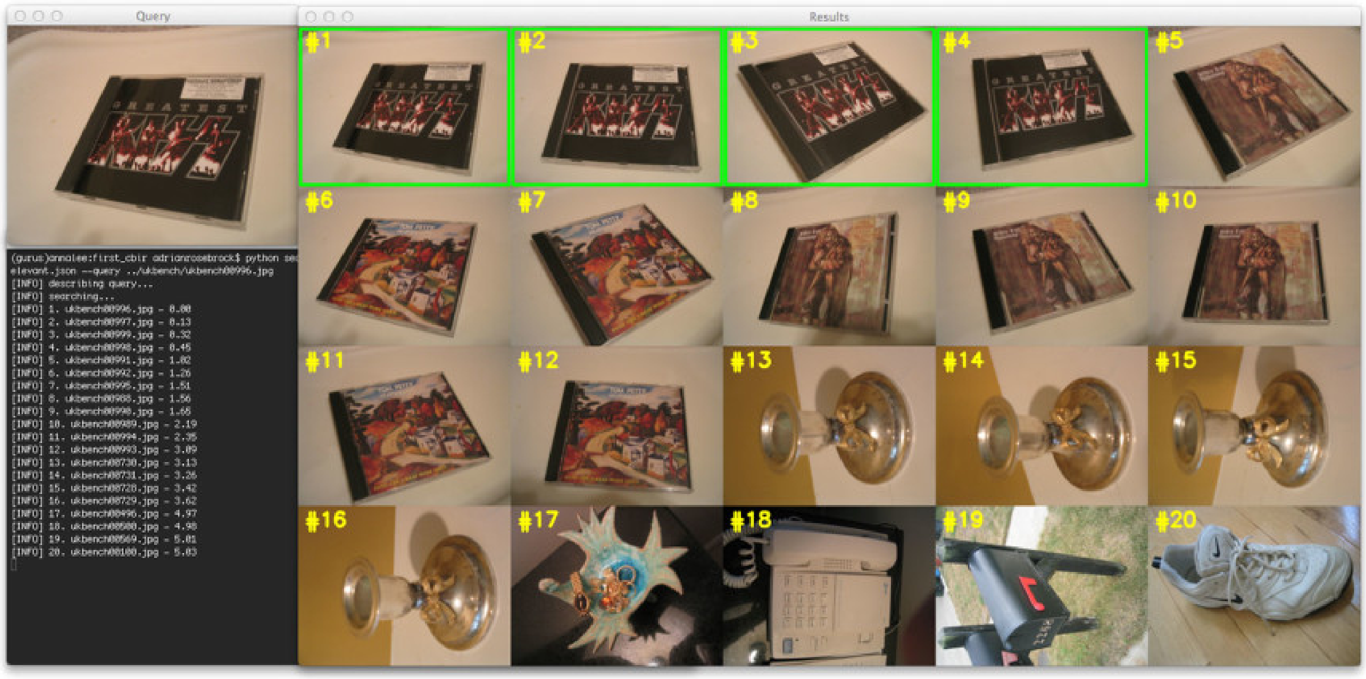
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00644.jpg)

FIGURE 13: ON THE *LEFT*, WE HAVE OUR INPUT QUERY IMAGE. ON THE *RIGHT*, WE HAVE OUR SEARCH RESULTS MONTAGE. NOTICE HOW OUR CBIR SYSTEM HAS RETURNED THE CORRESPONDING PARK BENCH IMAGES IN THE TOP-4 RESULTS.

On the *left*, we have our input query image of our park bench — and on the *right*, we have our search results. The corresponding park bench images in the UKBench dataset have been highlighted in green. Notice how our CBIR system has ranked the park bench images in the top-4 results.

Let's give another image a try:

```
search.py                                                                      Shell
$ python search.py --index index.csv --dataset ../ukbench --relevant ../ukbench/relevant.json \
    --query ../ukbench/ukbench00996.jpg
```



```
(gurus)annalee:first_cbir adrianrosebrock$ python se
elevant.json --query ../ukbench/ukbench00996.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00996.jpg - 0.00
[INFO] 2. ukbench00997.jpg - 0.13
[INFO] 3. ukbench00999.jpg - 0.32
[INFO] 4. ukbench00998.jpg - 0.45
[INFO] 5. ukbench00991.jpg - 1.02
[INFO] 6. ukbench00992.jpg - 1.26
[INFO] 7. ukbench00995.jpg - 1.51
[INFO] 8. ukbench00988.jpg - 1.56
[INFO] 9. ukbench00990.jpg - 1.65
[INFO] 10. ukbench00989.jpg - 2.19
[INFO] 11. ukbench00994.jpg - 2.35
[INFO] 12. ukbench00993.jpg - 3.09
[INFO] 13. ukbench00730.jpg - 3.13
[INFO] 14. ukbench00731.jpg - 3.26
[INFO] 15. ukbench00728.jpg - 3.42
[INFO] 16. ukbench00729.jpg - 3.62
[INFO] 17. ukbench00496.jpg - 4.97
[INFO] 18. ukbench00500.jpg - 4.98
[INFO] 19. ukbench00569.jpg - 5.01
[INFO] 20. ukbench00100.jpg - 5.03
```

[(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00996.jpg)](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00996.jpg)

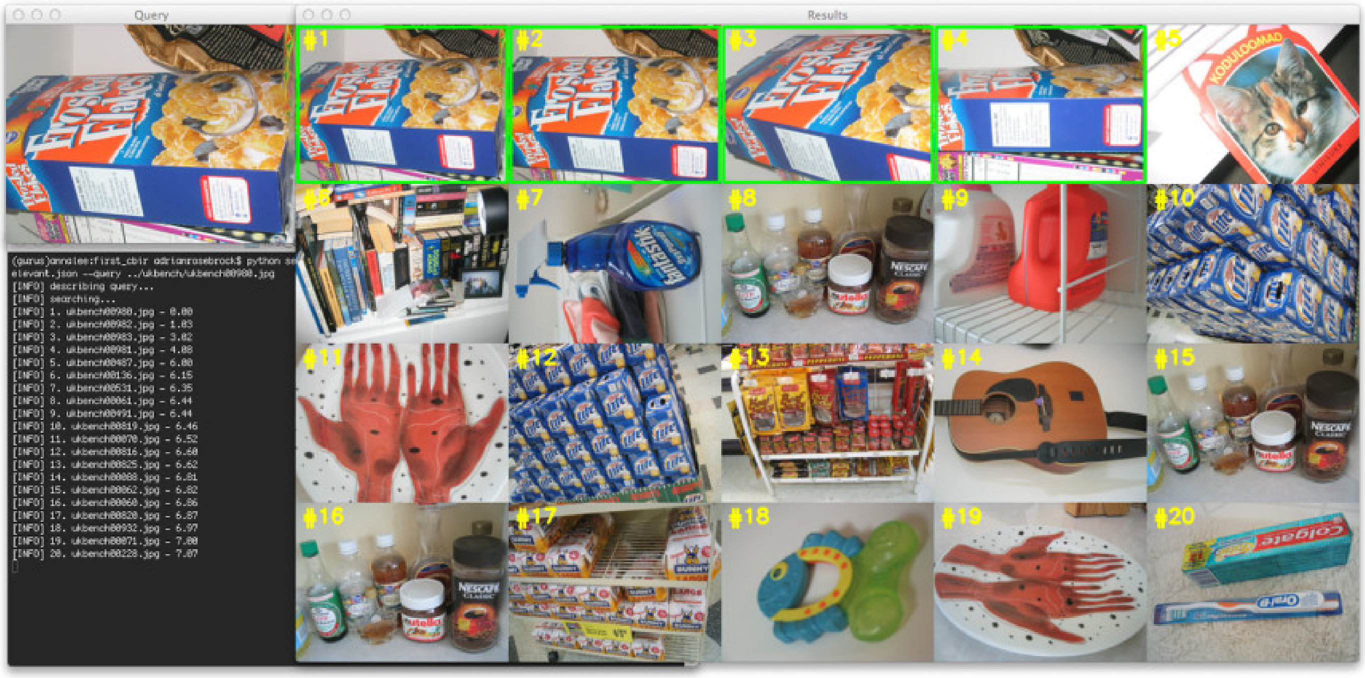
FIGURE 14: THIS TIME WE SUBMIT A KISS ALBUM COVER TO OUR CBIR SYSTEM.

Again, our CBIR system is able to find the corresponding four KISS album covers in the dataset and place them in the top-4 search results.

Feedback

The same is true for this query image of *Frosted Flakes:*

search.py Python
```
$ python search.py --index index.csv --dataset ../ukbench --relevant ../ukbench/relevant.json \
    --query ../ukbench/ukbench00980.jpg
```



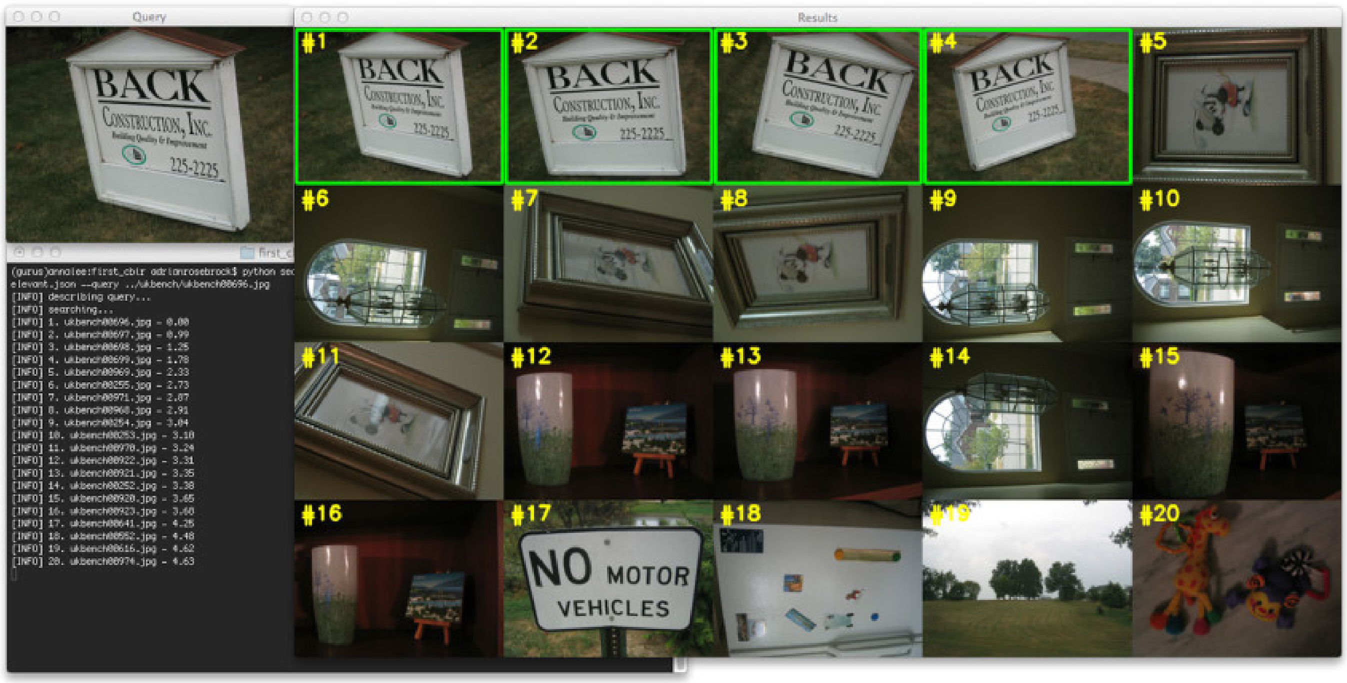
```
(gurus)annalee:first_cbir adrianrosebrock$ python se
elevant.json --query ../ukbench/ukbench00980.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00980.jpg - 0.00
[INFO] 2. ukbench00982.jpg - 1.03
[INFO] 3. ukbench00983.jpg - 3.02
[INFO] 4. ukbench00981.jpg - 4.08
[INFO] 5. ukbench00487.jpg - 6.00
[INFO] 6. ukbench00136.jpg - 6.15
[INFO] 7. ukbench00531.jpg - 6.35
[INFO] 8. ukbench00061.jpg - 6.44
[INFO] 9. ukbench00491.jpg - 6.44
[INFO] 10. ukbench00819.jpg - 6.46
[INFO] 11. ukbench00070.jpg - 6.52
[INFO] 12. ukbench00816.jpg - 6.60
[INFO] 13. ukbench00825.jpg - 6.62
[INFO] 14. ukbench00088.jpg - 6.81
[INFO] 15. ukbench00062.jpg - 6.82
[INFO] 16. ukbench00060.jpg - 6.86
[INFO] 17. ukbench00820.jpg - 6.87
[INFO] 18. ukbench00932.jpg - 6.97
[INFO] 19. ukbench00071.jpg - 7.00
[INFO] 20. ukbench00228.jpg - 7.07
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00980.jpg)

FIGURE 15: EVEN UNDER VARYING VIEWPOINTS, OUR CBIR SYSTEM IS ABLE TO RETURN THE CORRESPONDING IMAGES FROM THE DATASET.

And for this query image of an architectural firm:

search.py Python
```
$ python search.py --index index.csv --dataset ../ukbench --relevant ../ukbench/relevant.json \
    --query ../ukbench/ukbench00696.jpg
```



```
(gurus)annalee:first_cbir adrianrosebrock$ python sea
elevant.json --query ../ukbench/ukbench00696.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00696.jpg - 0.00
[INFO] 2. ukbench00697.jpg - 0.99
[INFO] 3. ukbench00698.jpg - 1.25
[INFO] 4. ukbench00699.jpg - 1.78
[INFO] 5. ukbench00969.jpg - 2.33
[INFO] 6. ukbench00255.jpg - 2.73
[INFO] 7. ukbench00971.jpg - 2.87
[INFO] 8. ukbench00968.jpg - 2.91
[INFO] 9. ukbench00254.jpg - 3.04
[INFO] 10. ukbench00253.jpg - 3.10
[INFO] 11. ukbench00970.jpg - 3.24
[INFO] 12. ukbench00922.jpg - 3.31
[INFO] 13. ukbench00921.jpg - 3.35
[INFO] 14. ukbench00252.jpg - 3.38
[INFO] 15. ukbench00920.jpg - 3.65
[INFO] 16. ukbench00923.jpg - 3.60
[INFO] 17. ukbench00641.jpg - 4.25
[INFO] 18. ukbench00552.jpg - 4.48
[INFO] 19. ukbench00616.jpg - 4.62
[INFO] 20. ukbench00974.jpg - 4.63
```

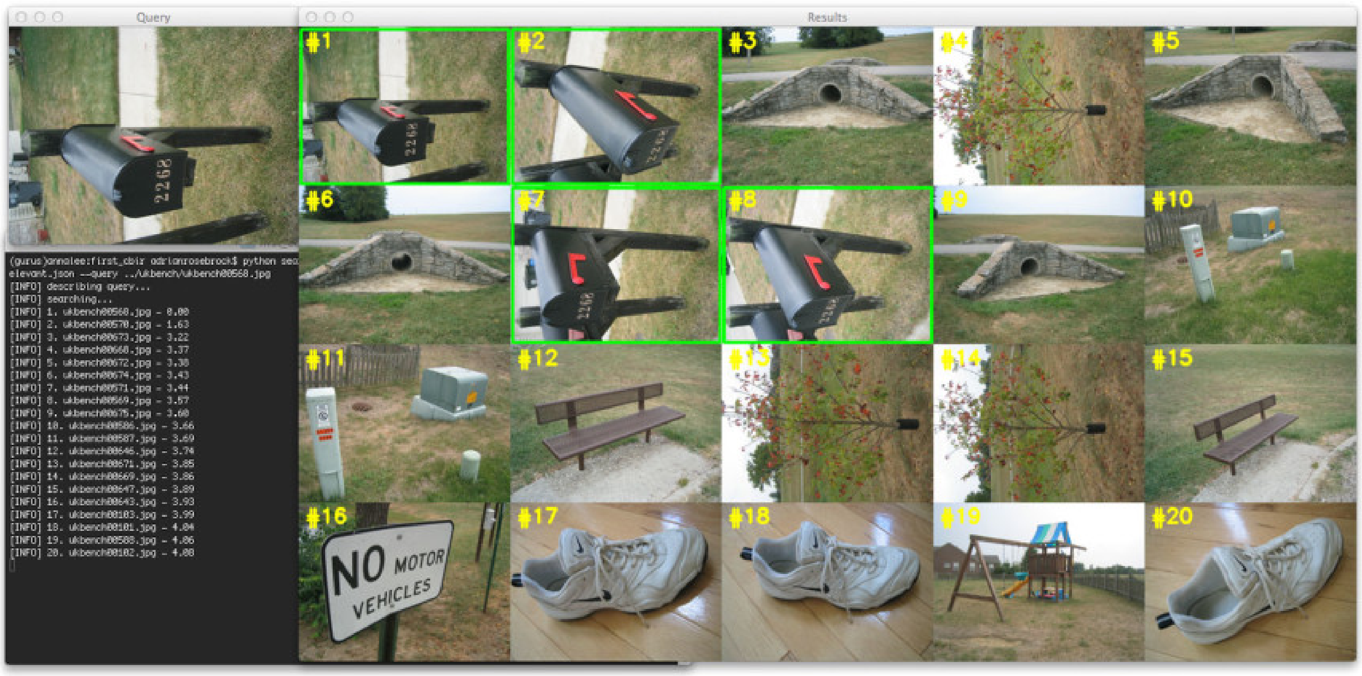
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00696.jpg)

Feedback

FIGURE 16: ONCE AGAIN, OUR BASIC CBIR SYSTEM IS ABLE TO OBTAIN GOOD RESULTS.

However, not all of our results are perfect. Take a look at the results when we submit a mailbox query image:

```
search.py                                                                    Shell
$ python search.py --index index.csv --dataset ../ukbench --relevant ../ukbench/relevant.json \
    --query ../ukbench/ukbench00568.jpg
```



```
(gurus)annalee:first_cbir adrianrosebrock$ python sea
elevant.json --query ../ukbench/ukbench00568.jpg
[INFO] describing query...
[INFO] searching...
[INFO] 1. ukbench00568.jpg - 0.00
[INFO] 2. ukbench00570.jpg - 1.63
[INFO] 3. ukbench00673.jpg - 3.22
[INFO] 4. ukbench00668.jpg - 3.37
[INFO] 5. ukbench00672.jpg - 3.38
[INFO] 6. ukbench00674.jpg - 3.43
[INFO] 7. ukbench00571.jpg - 3.44
[INFO] 8. ukbench00569.jpg - 3.57
[INFO] 9. ukbench00675.jpg - 3.60
[INFO] 10. ukbench00586.jpg - 3.66
[INFO] 11. ukbench00587.jpg - 3.69
[INFO] 12. ukbench00646.jpg - 3.74
[INFO] 13. ukbench00671.jpg - 3.85
[INFO] 14. ukbench00669.jpg - 3.86
[INFO] 15. ukbench00647.jpg - 3.89
[INFO] 16. ukbench00643.jpg - 3.93
[INFO] 17. ukbench00103.jpg - 3.99
[INFO] 18. ukbench00101.jpg - 4.04
[INFO] 19. ukbench00588.jpg - 4.06
[INFO] 20. ukbench00102.jpg - 4.08
```

(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/first_cbir_results_ukbench00568.jpg)

FIGURE 17: WHILE WE ARE ABLE TO FIND THE FOUR MAILBOX IMAGES IN OUR DATASET, THEY ARE NOT IN THE TOP-4 RESULTS.

While we were able to find the four corresponding mailbox images in our dataset, they are not in the top-4 results. This example demonstrates a limitation with our color histogram-based approach — sometimes images with *different visual contents* can have *similar color distributions*, thus affecting the search results.

In order to place all four of these mailbox images in the top-4 results, we'll need to use more powerful image/feature descriptors and apply more advanced search methods later in this module.

Summary

In this lesson, we applied the **four steps required to build any CBIR system (https://gurus.pyimagesearch.com/lessons/what-is-content-based-image-retrieval/)** to make the collection of UKBench images visually searchable.

We utilized a color histogram to characterize the color distribution of five spatial regions in an image. Then, we indexed our dataset using our color descriptor, extracting color histograms from each of the images in the dataset.

To compare images, we utilized the chi-squared distance, a popular choice when comparing discrete probability distributions.

From there, we implemented the necessary logic to accept a query image and then return relevant results.

While this is a very basic CBIR system, I hope that it piqued your curiosity. In the remainder of this module, we'll be taking a deeper look at the four steps required to build a CBIR system, followed by exploring various algorithms to make our image search engine more accurate.

See you in the next lesson!

Downloads:

Download the Code (https://gurus.pyimagesearch.com/protected/code/cbir/first_cbir.zip)

Download the UKBench sample dataset (https://gurus.pyimagesearch.com/protected/code/cbir/ukbench.zip)

Quizzes		Status
1	Your First Image Search Engine Quiz (https://gurus.pyimagesearch.com/quizzes/your-first-image-search-engine-quiz/)	

Feedback

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/what-is-content-based-image-retrieval/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/the-4-steps-of-building-any-image-search-engine/)

Course Progress

Ready to continue the course?

Click the button below to **continue your journey to computer vision guru**.

I'm ready, let's go! (/pyimagesearch-gurus-course/)

Resources & Links

- PyImageSearch Gurus Community (https://community.pyimagesearch.com/)
- PyImageSearch Virtual Machine (https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/)
- Setting up your own Python + OpenCV environment (https://gurus.pyimagesearch.com/setting-up-your-python-opencv-development-environment/)
- Course Syllabus & Content Release Schedule (https://gurus.pyimagesearch.com/course-syllabus-content-release-schedule/)
- Member Perks & Discounts (https://gurus.pyimagesearch.com/pyimagesearch-gurus-discounts-perks/)
- Your Achievements (https://gurus.pyimagesearch.com/achievements/)
- Official OpenCV documentation (http://docs.opencv.org/index.html)

Your Account

- Account Info (https://gurus.pyimagesearch.com/account/)
- Support (https://gurus.pyimagesearch.com/contact/)
- Logout (https://gurus.pyimagesearch.com/wp-login.php?action=logout&redirect_to=https%3A%2F%2Fgurus.pyimagesearch.com%2F&_wpnonce=5736b21cae)

Search

Feedback