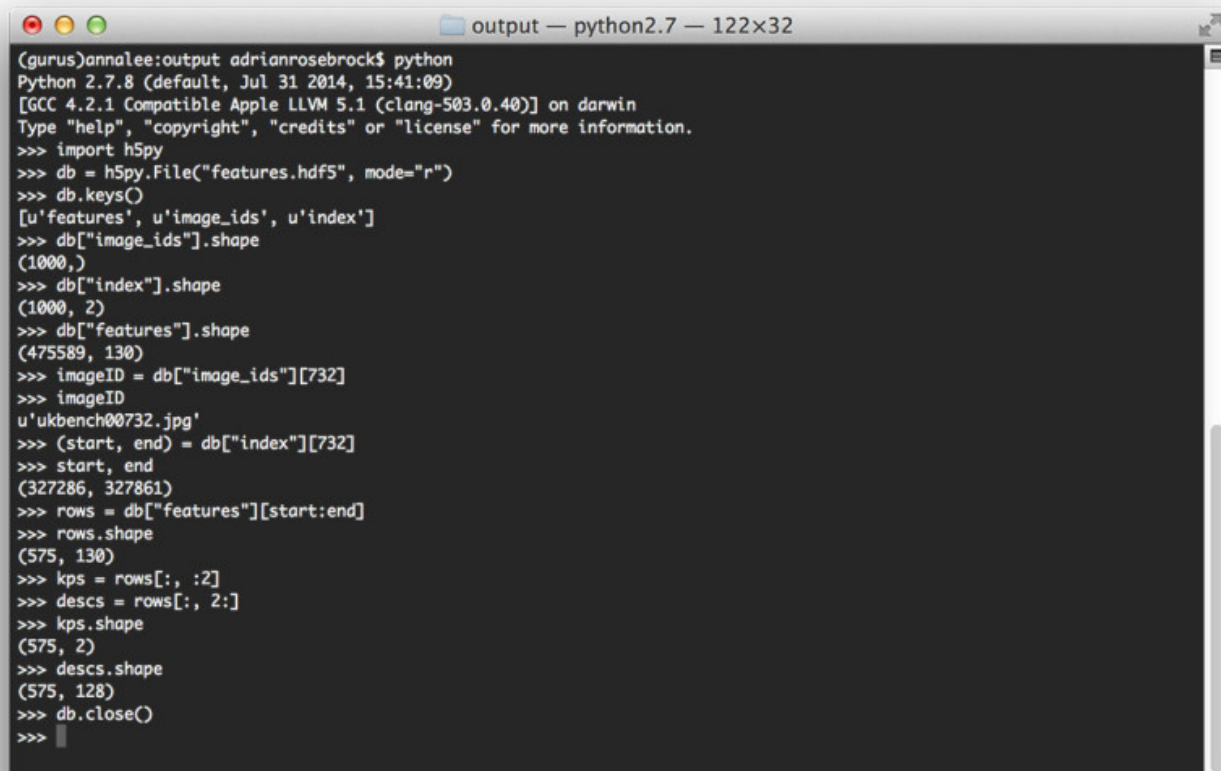


PyImageSearch Gurus Course

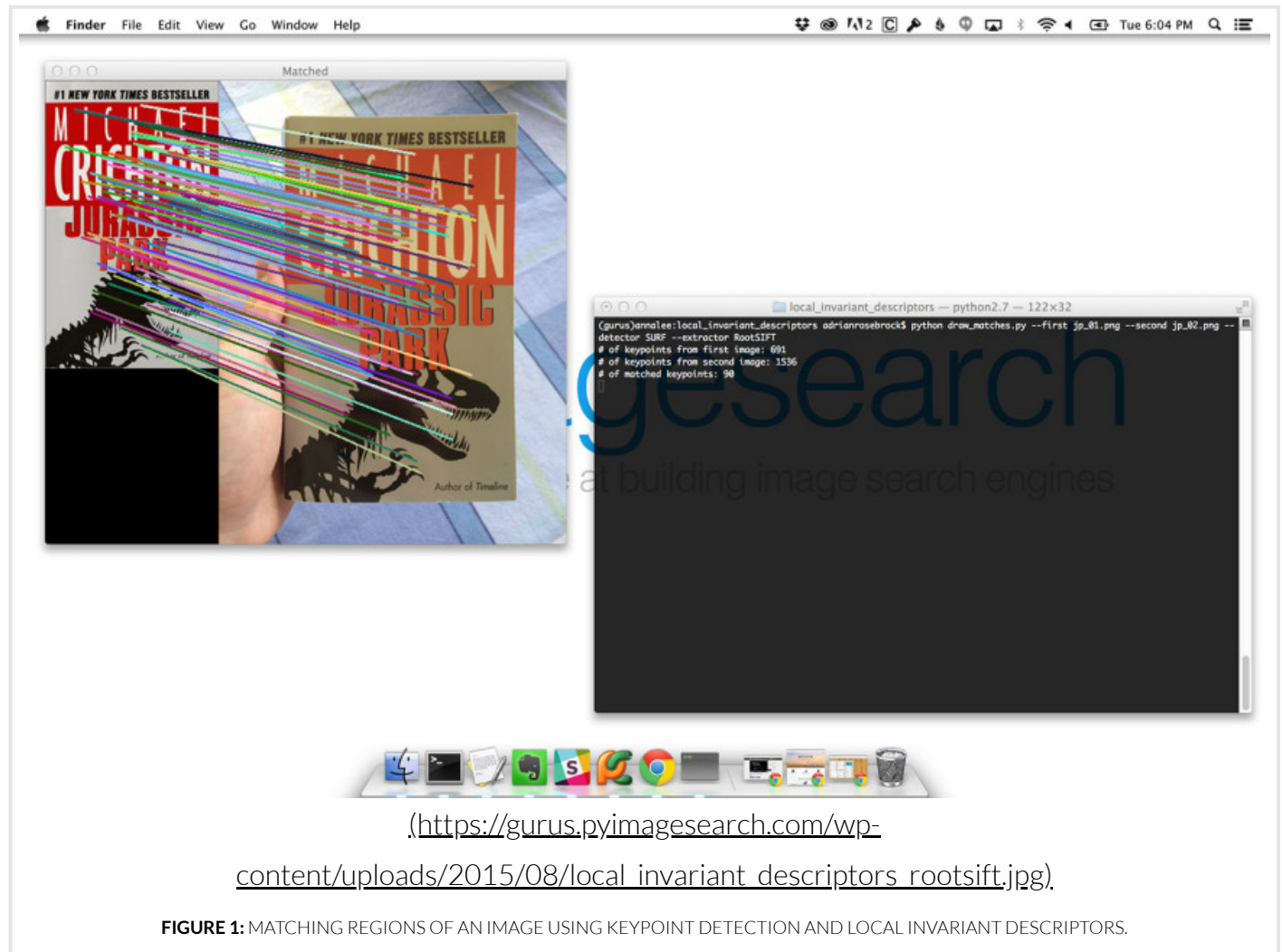
[_ \(HTTPS://GURUS.PYIMAGESEARCH.COM\)](https://gurus.pyimagesearch.com/) >

3.5: Extracting keypoints and local invariant descriptors



```
output — python2.7 — 122x32
(gurus)annalee:output adrianrosebrock$ python
Python 2.7.8 (default, Jul 31 2014, 15:41:09)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import h5py
>>> db = h5py.File("features.hdf5", mode="r")
>>> db.keys()
[u'features', u'image_ids', u'index']
>>> db["image_ids"].shape
(1000,)
>>> db["index"].shape
(1000, 2)
>>> db["features"].shape
(475589, 130)
>>> imageID = db["image_ids"][732]
>>> imageID
u'ukbench00732.jpg'
>>> (start, end) = db["index"][732]
>>> start, end
(327286, 327861)
>>> rows = db["features"][start:end]
>>> rows.shape
(575, 130)
>>> kps = rows[:, :2]
>>> descs = rows[:, 2:]
>>> kps.shape
(575, 2)
>>> descs.shape
(575, 128)
>>> db.close()
>>>
```

We have talked about [keypoint detectors \(https://gurus.pyimagesearch.com/lessons/keypoint-detectors/\)](https://gurus.pyimagesearch.com/lessons/keypoint-detectors/) and [local invariant descriptors \(https://gurus.pyimagesearch.com/lessons/local-invariant-descriptors/\)](https://gurus.pyimagesearch.com/lessons/local-invariant-descriptors/) a fair amount in this course thus far — we have even learned how to take our extracted features and use them to match regions from separate images together:



Feedback

However, we have yet to use keypoint detectors and local invariant descriptors in context of CBIR or image classification — *and that's exactly what this lesson is going to address.*

Current state-of-the-art CBIR systems leverage local invariant descriptors to build powerful, robust image search engines. Building such state-of-the-art image search engines can be broken down into four steps (each with multiple sub-steps, of course):

1. Extracting keypoints and features from the image dataset.
2. Clustering the extracted features using k-means to form a *codebook*.
3. Constructing a bag-of-visual-words (BOVW) representation for each image by *quantizing* the feature vectors associated with *each image* into a histogram using the *codebook* from Step 2.

4. Accepting a query image from the user, constructing the BOVW representation for the query, and performing the actual search.

In this lesson, we'll be focusing on **Step 1** of building a state-of-the-art CBIR system — *feature extraction*. We'll extract keypoints and local invariant descriptors from our UKBench dataset and store them in a highly efficient, easily accessible manner. Instead of simply writing the feature vectors to disk using a `.csv` file (which can be *extremely inefficient*) or a serializing using a `cPickle` representation, we'll use a format that you perhaps have never used or heard of before – **HDF5** (<https://www.hdfgroup.org/HDF5/>).

Brace yourselves, because we'll be covering **a lot** of content in this lesson, including some advanced, production-level code. While this may seem like a daunting or even overwhelming task, don't worry — I'll be breaking down these higher-level concepts into small, easy to digest pieces.

With that said, let's go ahead and get started extracting keypoints and local invariant descriptors from our dataset of images.

Objectives:

In this lesson, we will:

- Learn about HDF5 and how we can efficiently store *billions* of feature vectors in a single, hierarchically structured file boasting fast read access.
- Create advanced “indexer” classes to take the keypoints and descriptors extracted from our images and store them in HDF5.
- Write a driver Python script that can be used to easily extract features from a dataset of images, specifically our UKBench dataset.

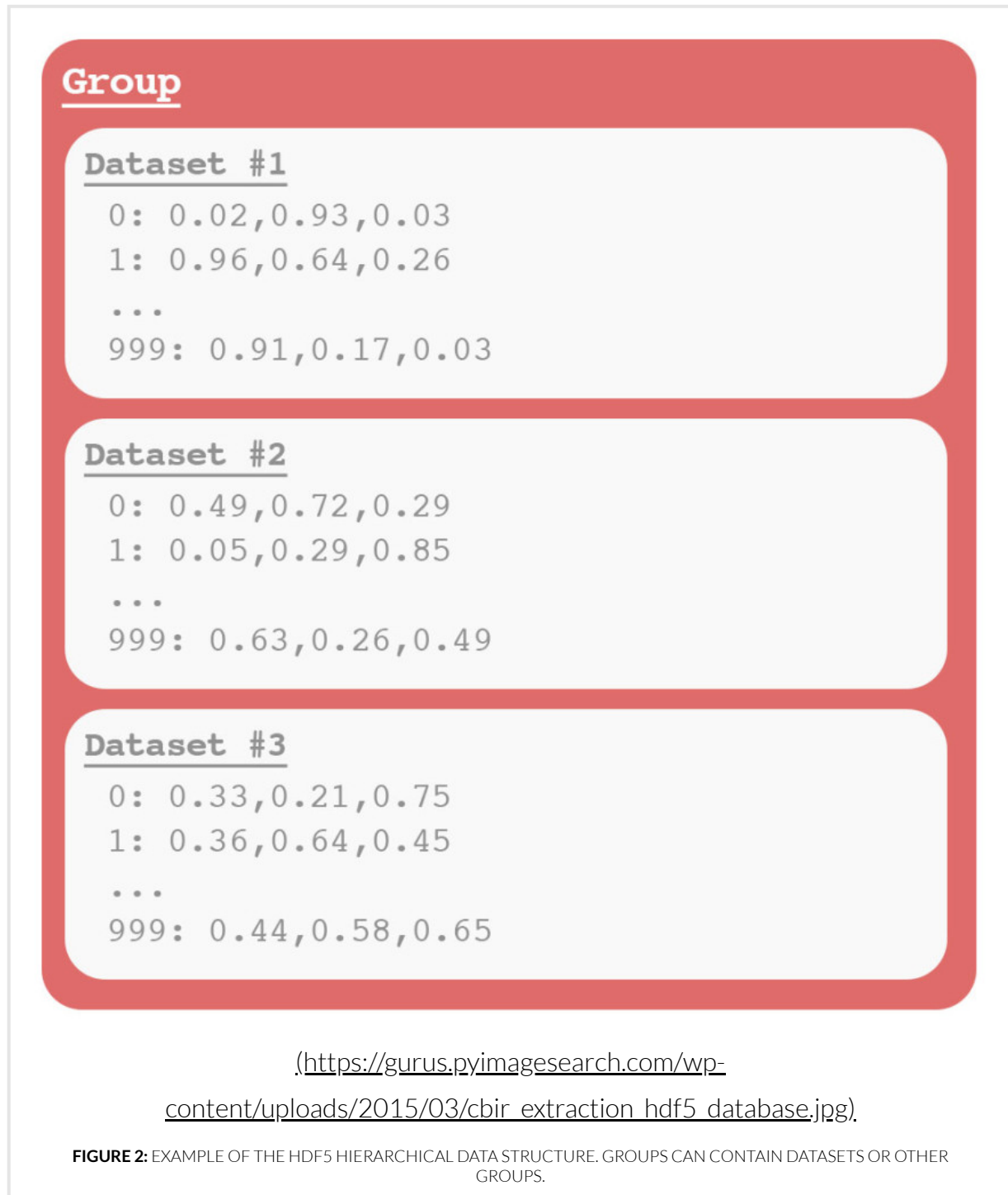
Extracting keypoints and local invariant descriptors

Before we dive too deep into extracting features from images, let's briefly review what HDF5 is and why it's so important when working with large amounts of feature vectors.

What is HDF5?

HDF5 is a binary data format created by the **HDF group** (<https://www.hdfgroup.org/>) to store gigantic numerical datasets on disk, while facilitating easy access and computation on the rows in the datasets.

Data in HDF5 is stored *hierarchically*, similar to how a filesystem stores data. Data is first defined in *groups*, where a group is a container-like structure which can hold *datasets* and other *groups*. Once a group has been defined, a *dataset* can be created within the group. A *dataset* can be thought of as a multi-dimensional array of a homogeneous data type.



HDF5 is written in C; however, by using the **h5py module** (<http://www.h5py.org/>), we can gain access to the underlying C API using the Python programming language.

What makes `h5py` so awesome is the ease of interaction with the data. We can store *huge* amounts of data in our HDF5 dataset and manipulate the data using NumPy.

For example, we can use standard Python syntax to access and slice rows from *multi-terabyte datasets* stored on disk as if they were simple NumPy arrays loaded into memory. Thanks to specialized data structures, these slices and row accesses are *lightning quick*.

When using HDF5 with `h5py`, you can think of your data as a gigantic NumPy array that is too large to fit into main memory, but can still be accessed and manipulated just the same.

Perhaps best of all, the HDF5 format is *standardized*, meaning that datasets stored in HDF5 format are inherently portable and can be accessed by other developers using different programming languages, such as C, MATLAB, and Java.

In the rest of this module, we'll be using HDF5 to store our keypoints and descriptors extracted from the UKBench dataset — but you'll be able to *extract code* to index datasets of your choice. Like I said, this code is indeed production ready.

The cost of feature vector storage

As we already know from our keypoint detector and local invariant descriptor lessons, we end up extracting *multiple* feature vectors per image — these feature vectors can quickly add up in size, sometimes consuming more memory than the original image!

The work of Philbin et al. in their 2007 paper, *Object retrieval with large vocabularies and fast spatial matching* (https://gurus.pyimagesearch.com/wp-content/uploads/2015/08/philbin_2007.pdf), reported on the size of descriptors extracted from the **Oxford5k dataset** (<http://www.robots.ox.ac.uk/~vgg/data/oxbuildings/>). For **5,000 images**, **16 million feature vectors** were extracted totaling **1.9GB**.

When they increased the dataset to **100,000 images**, the number of feature vectors extracted jumped to **277 million**, weighing in at **33.1GB**.

Finally, when **1 million** images were included, **over 1.1 billion** features were extracted, requiring a total of **141GB** of space.

It quickly becomes clear that we need an efficient, disk-based method to store our extracted feature vectors — and that's exactly what HDF5 gives us, all while being incredibly simplistic and easy to use.

Installing HDF5 and h5py

Unlike other libraries in this course, HDF5 is actually very easy to install. On **OSX**, just let `brew` install it for you:

Installing HDF5 on OSX	Shell
1 \$ <code>brew install hdf5</code>	

On **Ubuntu**, you can use `apt-get` :

Installing HDF5 on Ubuntu	Shell
1 \$ <code>sudo apt-get install libhdf5-serial-dev</code>	

Next (regardless of operating system), be sure to access your `gurus` environment and use `pip` to install `h5py` :

Installing h5py	Shell
1 \$ <code>workon gurus</code>	
2 \$ <code>pip install h5py</code>	

Note: If you are using > v1.1 of the [PyImageSearch Gurus Virtual Machine](https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/) (<https://gurus.pyimagesearch.com/pyimagesearch-virtual-machine/>), then both HDF5 and `h5py` are already pre-configured and pre-installed.

After that, you're good to go!

Directory structure of the feature extraction pipeline

Before we start coding, let's briefly review the directory structure of our project:

Directory structure of our feature extraction pipeline	Python
<pre>1 --- pyimagesearch 2 --- __init__.py 3 --- descriptors 4 --- __init__.py 5 --- detectanddescribe.py 6 --- indexer 7 --- __init__.py 8 --- baseindexer.py 9 --- featureindexer.py 10 --- index_features.py</pre>	

The `descriptors` sub-module contains implementations to extract keypoints and local invariant descriptors from our dataset of images. We'll be using [RootSIFT](https://gurus.pyimagesearch.com/topic/rootsift/) (<https://gurus.pyimagesearch.com/topic/rootsift/>) as our local invariant descriptor.

The `detectanddescribe.py` file will contain a class to easily detect keypoints and extract features using arbitrary detectors and descriptors with only a *single method call*.

We then have the `indexer` sub-module that will contain our object-oriented interfaces to the HDF5 dataset to store features.

Finally, `index_features.py` is our driver script used to glue all of the pieces together.

Now that we have our project structure defined, let's write some actual code to extract features from a dataset of images and efficiently store the resulting feature vectors on disk.

Feature extraction and indexing

In previous lessons in the PyImageSearch Gurus course, I normally save the driver implementation until **last**. However, now that we are getting into more advanced computer vision techniques and algorithms, I think it's better to look at the driver **first** — that way you can get a good idea of *what we're building* and *how the pieces fit together* (without becoming overwhelmed in the details all at once).

That said, let's take a peek at our `index_features.py` script:

index_features.py	Python
<pre>1 # import the necessary packages 2 from __future__ import print_function 3 from pyimagesearch.descriptors import DetectAndDescribe 4 from pyimagesearch.indexer import FeatureIndexer 5 from imutils.feature import FeatureDetector_create, DescriptorExtractor_create 6 from imutils import paths 7 import argparse 8 import imutils 9 import cv2 10 11 # construct the argument parser and parse the arguments 12 ap = argparse.ArgumentParser() 13 ap.add_argument("-d", "--dataset", required=True, 14 help="Path to the directory that contains the images to be indexed") 15 ap.add_argument("-f", "--features-db", required=True, 16 help="Path to where the features database will be stored") 17 ap.add_argument("-a", "--approx-images", type=int, default=500, 18 help="Approximate # of images in the dataset") 19 ap.add_argument("-b", "--max-buffer-size", type=int, default=50000, 20 help="Maximum buffer size for # of features to be stored in memory") 21 args = vars(ap.parse_args())</pre>	

Feedback

We'll start off by importing our required packages on **Lines 2-9**. Notice how we'll be using the

`DetectAndDescribe` class to reduce keypoint detection and feature extraction into a *single line of code*.

We'll import our `FeatureIndexer`, so we can store the extract RootSIFT features in our HDF5 dataset.

Finally, we'll import `FeatureDetector_create` and `DescriptorExtractor_create` from `imutils` so that we can work with RootSIFT across OpenCV versions.

Let's move on to defining each of the command line arguments:

- `--dataset` : This switch could be pretty self-explanatory — it's simply the path to the directory containing our UKBench images.
- `--features-db` : Here, we specify the path to where our HDF5 database will be stored on disk.
- `--approx-images` : This(optional) switch allows us to specify the approximate number of images in our dataset. In the UKbench dataset, we know there are a total of 1,000 images. However, if you only had an estimate of the number of images, you could use this switch to specify the approximation. This is an important value because it allows us to *estimate the size of our HDF5 dataset* when the `FeatureIndexer` is initialized. Having a good estimation of the size of the dataset can increase efficiency as we write feature vectors to disk.
- `--max-buffer-size` : Writing feature vectors to HDF5 one at a time is *highly inefficient*. Instead, it's much more effective to collect feature vectors into a large array in memory and then dump them to HDF5 when the buffer is full. The value of `--max-buffer-size` specifies how many *feature vectors* can be stored in memory until the buffer is flushed to HDF5. We specify a default value of 50,000 feature vectors, but for machines with a fair amount of RAM, this value can easily be an order of magnitude (or two orders) larger — it all depends on the available memory on your machine.

Feedback

Now that our command line arguments have been parsed, let's continue working through the script:

index_features.py	Python
<pre>23 # initialize the keypoint detector, local invariant descriptor, and the descriptor 24 # pipeline 25 detector = FeatureDetector_create("SURF") 26 descriptor = DescriptorExtractor_create("RootSIFT") 27 dad = DetectAndDescribe(detector, descriptor) 28 29 # initialize the feature indexer 30 fi = FeatureIndexer(args["features_db"], estNumImages=args["approx_images"], 31 maxBufferSize=args["max_buffer_size"], verbose=True)</pre>	

On **Line 25**, we initialize our keypoint `detector` to use the **Fast Hessian method** (<https://gurus.pyimagesearch.com/topic/fast-hessian/>). We'll then initialize our `RootSIFT` descriptor and pass both of the instantiations into the `DetectAndDescribe` class (**Line 27**). The `DetectAndDescribe` class can accept arbitrary keypoint detectors and descriptor methods, and in turn allows us to detect and describe regions of an image in a single line of code.

Last, we need to construct our `FeatureIndexer` on **Lines 30 and 31**. The `FeatureIndexer` requires we pass in the path to our output HDF5 dataset, the approximate number of images in our dataset (so we can roughly approximate the number of total feature vectors we'll be extracting), the maximum buffer size of feature vectors to be stored in memory, and a verbosity level for debugging.

Now that all of our classes are set up and initialized, we can perform the actual extraction:

index_features.py	Python
<pre>33 # loop over the images in the dataset 34 for (i, imagePath) in enumerate(sorted(paths.list_images(args["dataset"]))): 35 # check to see if progress should be displayed 36 if i > 0 and i % 10 == 0: 37 fi._debug("processed {} images".format(i), msgType="[PROGRESS]") 38 39 # extract the image filename (i.e. the unique image ID) from the image 40 # path, then load the image itself 41 filename = imagePath[imagePath.rfind("/") + 1:] 42 image = cv2.imread(imagePath) 43 image = imutils.resize(image, width=320) 44 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) 45 46 # describe the image 47 (kps, descs) = dad.describe(image) 48 49 # if either the keypoints or descriptors are None, then ignore the image 50 if kps is None or descs is None: 51 continue 52 53 # index the features 54 fi.add(filename, kps, descs) 55 56 # finish the indexing process 57 fi.finish()</pre>	

Feedback

We start looping over the images in our dataset on **Line 34**. Each image is then loaded, resized, and converted to grayscale (**Lines 41-44**) in preparation for keypoint detection and feature extraction.

The actual extraction of keypoints and features takes place on **Line 47**, where we make a call to the `describe` method of our `DetectAndDescribe` class.

If no keypoints are detected (and therefore no features extracted), we skip the image and continue on to the next one (**Lines 50 and 51**). Otherwise, we add the image filename, keypoints, and descriptors to the HDF5 dataset (**Line 54**) and finish the feature indexing process (**Line 57**).

Keypoints and descriptors

We saw from the driver script above that we are using the Fast Hessian keypoint detector and the RootSIFT method to extract feature vectors, a common choice when building CBIR or image classification systems.

Let's now define our `DetectAndDescribe` class to encapsulate keypoint detection and feature extraction into a single method call:

detectanddescribe.py	Python
<pre>1 # import the necessary packages 2 import numpy as np 3 4 class DetectAndDescribe: 5 def __init__(self, detector, descriptor): 6 # store the keypoint detector and local invariant descriptor 7 self.detector = detector 8 self.descriptor = descriptor</pre>	

Line 5 defines the constructor of our `DetectAndDescribe` class, which requires a `detector` and a `descriptor` method to be passed in. The `detector` should have a `detect` method that can be called to detect keypoints. Similarly, the `descriptor` needs a `compute` method that accepts an image and a list of keypoints so that it may extract a feature vector for each keypoint region.

detectanddescribe.py	Python
<pre>10 def describe(self, image, useKpList=True): 11 # detect keypoints in the image and extract local invariant descriptors 12 kps = self.detector.detect(image) 13 (kps, descs) = self.descriptor.compute(image, kps) 14 15 # if there are no keypoints or descriptors, return None 16 if len(kps) == 0: 17 return (None, None) 18 19 # check to see if the keypoints should be converted to a NumPy array 20 if useKpList: 21 kps = np.int0([kp.pt for kp in kps]) 22 23 # return a tuple of the keypoints and descriptors 24 return (kps, descs)</pre>	

Feedback

Given all our previous work in keypoint detectors and local invariant descriptors, the `describe` method should look fairly straightforward. This function requires only a single parameter, the `image` that we want to extract features from.

Lines 12 and 13 handle detecting keypoints and extracting feature vectors. If no keypoints are detected, we return an empty tuple on **Lines 16 and 17**. Otherwise, we make a check to see if the `KeyPoint` OpenCV object should be converted to a NumPy array (**Lines 20 and 21**).

Last, we return the tuple of keypoints and descriptors to the calling function on **Line 24**.

So far, we haven't covered anything too advanced, but that's going to change when we jump into the next section on feature storage and indexing as we build classes to buffer features in memory and flush the features to HDF5 when the buffer is full.

Feature storage and indexing

What makes CBIR and building image search engines challenging is not always the computer vision algorithms — it's also the intimate knowledge of data structures being used. Being an expert in image search engines not only takes a strong understanding of image descriptors and features, but also a deep knowledge of data structures (and their associated implementations) to make storage and search as efficient (and fast, without sacrificing accuracy) as possible.

In the rest of this section, I'll detail two Python classes *from my own personal CBIR implementations* that can be used to store features in HDF5 — and automatically expand and grow as the number of feature vectors does.

I've implemented these classes in an object oriented manner, so let's start with the root class — the

`BaseIndexer` :

baseindexer.py	Python
<pre>1 # import the necessary packages 2 from __future__ import print_function 3 import numpy as np 4 import datetime 5 6 class BaseIndexer(object): 7 def __init__(self, dbPath, estNumImages=500, maxBufferSize=50000, dbResizeFactor=2, 8 verbose=True): 9 # store the database path, estimated number of images in the dataset, max 10 # buffer size, the resize factor of the database and the verbosity setting 11 self.dbPath = dbPath 12 self.estNumImages = estNumImages 13 self.maxBufferSize = maxBufferSize 14 self.dbResizeFactor = dbResizeFactor 15 self.verbose = verbose 16 17 # initialize the indexes dictionary 18 self.idx = {}</pre>	

After looking at the `index_features.py` driver script, the constructor for the `BaseIndexer` should look somewhat familiar. The `BaseIndexer` requires a path to store our HDF5 dataset, the estimated number of images in the dataset, the maximum number of feature vectors to be stored in memory prior to

flushing to disk, a resize factor (which I'll explain later in this lesson), and a verbosity level. All of these options are stored on **Lines 11-15**.

We also initialize a *very special* empty dictionary called `idxs` on **Line 18**.

Remember how I mentioned that HDF5 datasets can be viewed as gigantic NumPy arrays? Well, in order to access a given row or insert data into the HDF5 dataset, we need to know the *index* of the row we are accessing, just like when working with NumPy arrays. This `idxs` dictionary will store the current *indexes* into the respective datasets — *this is an incredibly important variable*, so pay attention to how we use it through the remainder of this lesson.

baseindexer.py	Python
20	<code>def _writeBuffers(self):</code>
21	<code> pass</code>
22	
23	<code>def _writeBuffer(self, dataset, datasetName, buf, idxName, sparse=False):</code>
24	<code> # if the buffer is a list, then compute the ending index based on</code>
25	<code> # the lists length</code>
26	<code> if type(buf) is list:</code>
27	<code> end = self.idxs[idxName] + len(buf)</code>
28	
29	<code> # otherwise, assume that the buffer is a NumPy/SciPy array, so</code>
30	<code> # compute the ending index based on the array shape</code>
31	<code> else:</code>
32	<code> end = self.idxs[idxName] + buf.shape[0]</code>
33	
34	<code> # check to see if the dataset needs to be resized</code>
35	<code> if end > dataset.shape[0]:</code>
36	<code> self._debug("triggering `{}` db resize".format(datasetName))</code>
37	<code> self._resizeDataset(dataset, datasetName, baseSize=end)</code>
38	
39	<code> # if this is a sparse matrix, then convert the sparse matrix to a</code>
40	<code> # dense one so it can be written to file</code>
41	<code> if sparse:</code>
42	<code> buf = buf.toarray()</code>
43	
44	<code> # dump the buffer to file</code>
45	<code> self._debug("writing `{}` buffer".format(datasetName))</code>
46	<code> dataset[self.idxs[idxName]:end] = buf</code>

When our buffer of feature vectors reaches the `maxBufferSize`, we need to flush the buffer to our HDF5 dataset and reset the buffer. A call to `_writeBuffers` on **Line 20** should *take all available buffers and flush them*; however, since this is the root class, we don't have any concept on the number of buffers that need to be flushed.

However, we can still define the `_writeBuffer` (singular) method on **Line 23** that accepts a single buffer and writes it to disk. This method requires a handful of arguments, including:

- `dataset` : The HDF5 dataset object that we are writing the buffer to.
- `datasetName` : The name of the HDF5 dataset (e.g. internal, hierarchical path).
- `buf` : The buffer that will be flushed to disk.
- `idxName` : The name of the **key** into the `idx` dictionary. This variable will allow us to access the current pointer into the HDF5 dataset and ensure our features are written in sequential order, without overlapping each other.
- `sparse` : A flag indicating whether or not the buffer is of a sparse matrix (<http://docs.scipy.org/doc/scipy/reference/sparse.html>). data type.

Lines 26 and 27 handle the case where our `buf` is a list data type. In this situation, the `buf` has no `shape` (since it's not a NumPy array), so we use the `len` of the list to compute the **ending index** into the dataset.

However, if our `buf` is indeed a NumPy matrix, then we do have a `shape` attribute, so we can use it to compute the **ending index** (**Lines 31 and 32**).

In either case, the `end` index is simply the current index value plus the number of entries in the buffer.

Since HDF5 datasets are actually NumPy arrays, this means that they have a pre-defined dimension, both in terms of number of rows *and* number of columns. **Line 35** makes a check to see if there is enough room in the dataset to append the current number of entries in the buffer. If not, we need to trigger a resizing of the dataset and increase the number of rows it can store (**Line 37**). Resizing the dataset is handled by the `_resizeDataset`, which we'll review in the next codeblock.

HDF5 can only store dense arrays, so we need to make a check on **Line 41** to see if our `buf` is a sparse SciPy matrix. If it is, we'll need to convert it to a *dense matrix* first.

Finally, **Line 46** writes the `buf` to the HDF5 dataset using standard Python array slicing. The start of the slice is specified by `self.idx[s[idxName]]`, which is simply an integer pointing to the next open row in the dataset. The end of the slice is determined by our calculation of `end` earlier in the `_writeBuffer` dataset.

We'll leave resetting the buffers and incrementing the `idxs` value to the function that called `_writeBuffer`.

Now that we can write a buffer to file, let's see how we can resize the dataset to accommodate for more (or less) feature vectors:

```

48     def _resizeDataset(self, dataset, dbName, baseSize=0, finished=0):
49         # grab the original size of the dataset
50         origSize = dataset.shape[0]
51
52         # check to see if we are finished writing rows to the dataset, and if
53         # so, make the new size the current index
54         if finished > 0:
55             newSize = finished
56
57         # otherwise, we are enlarging the dataset so calculate the new size
58         # of the dataset
59         else:
60             newSize = baseSize * self.dbResizeFactor
61
62         # determine the shape of (to be) the resized dataset
63         shape = list(dataset.shape)
64         shape[0] = newSize
65
66         # show the old versus new size of the dataset
67         dataset.resize(tuple(shape))
68         self._debug("old size of `{}`: {:,}; new size: {:,}".format(dbName, origSize,
69             newSize))

```

When resizing our HDF5 dataset, we need to be able to both *increase* the size of the dataset (such as when we're indexing images and adding more features to the dataset) and *compact* the size of the dataset (such as when the feature indexing process has completed, and we need to reclaim unused space).

Luckily, increasing and decreasing the size of a given dataset is handled by the same `h5py` function — all we need to do is determine the new size (i.e. NumPy shape) of the dataset.

We'll start by defining our `_resizeDataset` method on **Line 48** to accept four parameters:

- `dataset` : The HDF5 dataset object that we are resizing.
- `dbName` : The name of the HDF5 dataset.
- `baseSize` : The base size of the dataset is assumed to be the *total number of rows in the dataset plus the total number of entries in the buffer*.
- `finished` : An integer indicating whether we are finished indexing feature vectors, and thus the dataset should be *compacted* rather than *expanded*. This value will be 0 if we are expanding the dataset and > 0 if the dataset is to be compacted.

In order to resize our dataset, we need to know the total number of rows currently in the matrix (both empty and filled rows), which we grab on **Line 50**.

If we are compacting the dataset, then the `newSize` should be the value of `finished` , or the total number of filled rows in the dataset (**Lines 54 and 55**).

Otherwise, we must be expanding the dataset (**Lines 59 and 60**), so we'll multiply the `baseSize` by the `resizeFactor` (which defaults to `resizeFactor=2`) to essentially double the size of our dataset. You might wonder: isn't doubling the size of our dataset wasteful? Aren't we essentially allocating a ton of extra space that we might not need?

You would be correct in saying that we are allocating a lot of extra rows in the dataset; however, *this operation is not wasteful* since we will be compacting the dataset after the feature indexing process has finished. Furthermore, it's likely that we'll be performing *expanding* operations multiple times, whereas a *compaction* operation is guaranteed to only happen once. Since resizing operations tend to be expensive, it's much more beneficial to over-allocate space during the feature vector insertion process.

Note: This is the same argument that is used to justify doubling the size of hash tables once they reach a certain percentage of “full-ness”. For more information on hash table expansion, take a look at the [following lesson from MIT \(http://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf\)](http://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf).

Last, **Lines 63 and 64** grab the current dimensions of the `dataset` and insert the value of `newSize` as the new number of rows. A call to the `resize` method of the `dataset` object handles resizing the dataset to the calculated new size.

Let's wrap up our `BaseIndexer` by looking at two utility methods:

baseindexer.py	Python
<pre>71 def _debug(self, msg, msgType="[INFO]"): 72 # check to see the message should be printed 73 if self.verbose: 74 print("{} {} - {}".format(msgType, msg, datetime.datetime.now())) 75 76 @staticmethod 77 def featureStack(array, accum=None, stackMethod=np.vstack): 78 # if the accumulated array is None, initialize it 79 if accum is None: 80 accum = array 81 82 # otherwise, stack the arrays 83 else: 84 accum = stackMethod([accum, array]) 85 86 # return the accumulated array 87 return accum</pre>	

Line 71 defines the `_debug` method, which can be used to (optionally) write debugging messages to our terminal.

We also define a static method, `featureStack`, on **Line 77** that is heavily used by buffer operations. Since buffers can be NumPy arrays, we should use NumPy/SciPy methods to stack and append them together. This method simply accepts two arrays: `array`, which will be appended to `accum`, the accumulated array (i.e. the buffer). Given these two arrays, the rest of the `featureStack` method handles stacking and appending these arrays based on the supplied `stackMethod`.

If you are unfamiliar with the concept of array stacking, I advise reading the **joining arrays** (<http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html#joining-arrays>) section of the NumPy documentation.

Structuring our HDF5 dataset

Before we extend our `BaseIndexer` class, let's talk about the structure of our HDF5 dataset and how it will be leveraged to efficiently store image identifiers, keypoints, and feature vectors.

As I mentioned, HDF5 files are meant to store multiple *groups* and *datasets*. A *group* is a container-like structure which can hold datasets (and other groups). A *dataset*, however, is meant to store a multi-dimensional array of a homogenous data type — essentially a NumPy matrix.

However, unlike NumPy arrays, HDF5 datasets support additional features such as compression, error detection, and chunked I/O. All that said, from our point of view, we'll be treating HDF5 datasets as large NumPy arrays with `shape`, `size`, and `dtype` attributes.

In order to build our CBIR system, we'll need three *datasets* (again, which are essentially NumPy arrays). Below follows the structure of our HDF5 group to index and store our feature vectors:

features.hdf5

image_ids

```
0: ukbench000001.jpg
1: ukbench000002.jpg
...
999: ukbench00999.jpg
```

index

```
0: 0,651
1: 651,1386
...
999: 475280,475589
```

features

```
0: 175,45,0.07
1: 123,57,0.07
...
999: 212,99,0.02
```

([https://gurus.pyimagesearch.com/wp-](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/cbir_extraction_hdf5_database_layout.jpg)

[content/uploads/2015/03/cbir_extraction_hdf5_database_layout.jpg](https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/cbir_extraction_hdf5_database_layout.jpg)).

FIGURE 3: THE STRUCTURE OF OUR HDF5 DATABASE. THE IMAGE_IDS DATASET WILL STORE THE UNIQUE IMAGE FILENAMES, INDEX WILL STORE THE MAPPINGS OF THE IMAGE_IDS TO THE FEATURE VECTORS, AND FEATURES WILL STORE THE RAW FEATURE VECTORS THEMSELVES.

The `image_ids` dataset has shape $(N,)$ where N is the total number of images in the dataset — the `image_ids` could be a filename, a UUID (https://en.wikipedia.org/wiki/Universally_unique_identifier), or any value used to *uniquely identify the image*.

The `index` dataset has shape $(N, 2)$ and stores two integers — **these integers are the indexes into the `features` dataset for image i .**

Finally, the `features` dataset has shape $(M, 130)$, where M is the total number of feature vectors extracted from the N images in the dataset. The `features` dataset stores the raw feature vectors themselves. We derived 130 columns for the `features` dataset since the RootSIFT feature vectors are 128-dim – and then we need an extra two columns for the (x, y) -coordinates of the keypoint associated with the feature vector.

For example, take a look at the HDF5 group structure in **Figure 3** above: `ukbench00001.jpg` has an index of 0. Looking at index 0 in the `index` dataset, we receive two integers: 0 and 651.

This implies that image `ukbench00001.jpg` has $651 - 0 = 651$ total feature vectors associated with it. These feature vectors can be accessed in the `features` dataset in the range $[0, 651)$.

Let's look at another example: `ukbench00002.jpg` has index 1 in `image_ids`. Looking at index 1 in the `index` dataset, we get the integers 651 and 1386. This implies that `ukbench00002.jpg` has $1386 - 651 = 735$ feature vectors which can be sliced from the `features` dataset in the range $[651, 1386)$.

The FeatureIndexer

Now that we understand the structure of our HDF5 datasets, let's look at the actual `FeatureIndexer` class:

featureindexer.py	Python
<pre>1 # import the necessary packages 2 from .baseindexer import BaseIndexer 3 import numpy as np 4 import h5py 5 import sys 6 7 class FeatureIndexer(BaseIndexer): 8 def __init__(self, dbPath, estNumImages=500, maxBufferSize=50000, dbResizeFactor=2, 9 verbose=True): 10 # call the parent constructor 11 super(FeatureIndexer, self).__init__(dbPath, estNumImages=estNumImages, 12 maxBufferSize=maxBufferSize, dbResizeFactor=dbResizeFactor, 13 verbose=verbose)</pre>	

To start, we'll import our necessary packages. First, we'll import our `BaseIndexer`, so we can extend it and create our `FeatureIndexer`. We'll also import `h5py`, so we can work with HDF5.

Overall, the constructor for the `FeatureIndexer` class is essentially identical to the `BaseIndexer`, which calls the super constructor on **Line 11**.

Now that the `BaseIndexer` has been initialized as well, let's look at the rest of the `FeatureIndexer` constructor and see what type of variables we'll be using:

featureindexer.py	Python
15	# open the HDF5 database for writing and initialize the datasets within
16	# the group
17	self.db = h5py.File(self.dbPath, mode="w")
18	self.imageIDDB = None
19	self.indexDB = None
20	self.featuresDB = None
21	
22	# initialize the image IDs buffer, index buffer, and the keypoints +
23	# features buffer
24	self.imageIDBuffer = []
25	self.indexBuffer = []
26	self.featuresBuffer = None
27	
28	# initialize the total number of features in the buffer along with the
29	# indexes dictionary
30	self.totalFeatures = 0
31	self.idx = {"index": 0, "features": 0}

We start off by opening our HDF5 database for writing on **Line 17**. Then, we'll initialize the `image_id`, `index`, and `features` datasets on **Lines 18-20**. Each of these datasets will start off as `None` until our buffers are initially full, then we'll explicitly create the datasets.

Speaking of buffers, let's initialize the image IDs, index, and features buffers, respectively on **Lines 24-26**. The `imageIDBuffer` and `indexBuffer` will be lists, whereas the `featuresBuffer` will be a NumPy array (which is initialized as `None` so the buffer can be accumulated using our `featuresStack` utility function.

Finally, **Line 30** initializes `totalFeatures`, the total number of feature vectors in the `featuresBuffer`, whereas **Line 31** constructs the `idx` dictionary. The `idx` dictionary contains two entries:

- `index`: An integer representing the current row (i.e. the next empty row) in both the `image_ids` and `index` datasets.
- `features`: An integer representing the next empty row in the `features` dataset.

Again, when you think of these two values as indexes into a NumPy array, they make a lot more sense. Since all three `image_ids`, `index`, and `features` datasets are empty, the next empty rows in the respective arrays are 0. It's important to note that both of these values will be incremented as the buffers are filled and flushed to disk.

Now, let's define the `add` method, which will be used to add an image and its associated keypoints and features to the HDF5 database:

featureindexer.py	Python
<pre>33 def add(self, imageID, kps, features): 34 # compute the starting and ending index for the features lookup 35 start = self.idx["features"] + self.totalFeatures 36 end = start + len(features) 37 38 # update the image IDs buffer, features buffer, and index buffer, 39 # followed by incrementing the feature count 40 self.imageIDBuffer.append(imageID) 41 self.featuresBuffer = BaseIndexer.featureStack(np.hstack([kps, features]), 42 self.featuresBuffer) 43 self.indexBuffer.append((start, end)) 44 self.totalFeatures += len(features) 45 46 # check to see if we have reached the maximum buffer size 47 if self.totalFeatures >= self.maxBufferSize: 48 # if the databases have not been created yet, create them 49 if None in (self.imageIDDB, self.indexDB, self.featuresDB): 50 self._debug("initial buffer full") 51 self._createDatasets() 52 53 # write the buffers to file 54 self._writeBuffers()</pre>	

The `add` method requires three arguments: the unique identifier of the image to be added, followed by the keypoints and descriptors associated with the image.

Lines 35 and 36 compute the `start` and `end` index of keypoints and descriptors into the `features` dataset. It's important that we make note of these values because they will also be stored in the `index` dataset, allowing us to quickly look up the starting and ending slice values for a particular image — which then implies that we slice the keypoints and feature vectors associated from an image very efficiently.

Lines 40-43 updates our buffers. The `imageID` is appended to the `imageIDBuffer`. The set of keypoints and feature vectors are stacked and accumulated into the `featuresBuffer`. We also update the `indexBuffer`, inserting the `start` and `end` row indexes into the `features` dataset for the current set of features. **Line 44** then increments the total number of feature vectors we have examined so far.

Of course, our buffers will eventually fill up — which is exactly what **Line 47** checks for. In the case that the total number of features in `featuresBuffer` exceeds the maximum number of features to be kept in memory, we need to stop adding features to the buffer and flush it to HDF5.

However, the `imageIDDB`, `indexDB`, and `featuresDB` have all been initialized to `None` — therefore, we must create them before flushing the buffers (**Lines 49-51**).

Finally, **Line 54** calls the `_writeBuffers` method which takes all three buffers, dumps them to file, and empties the buffers so that they can be filled up again.

Speaking of creating datasets, let's see how that is done with HDF5:

featureindexer.py	Python
<pre>56 def _createDatasets(self): 57 # compute the average number of features extracted from the initial buffer 58 # and use this number to determine the approximate number of features for 59 # the entire dataset 60 avgFeatures = self.totalFeatures / float(len(self.imageIDBuffer)) 61 approxFeatures = int(avgFeatures * self.estNumImages) 62 63 # grab the feature vector size 64 fvectorSize = self.featuresBuffer.shape[1] 65 66 # handle h5py datatype for Python 2.7 67 if sys.version_info[0] < 3: 68 dt = h5py.special_dtype(vlen=unicode) 69 70 # otherwise use a datatype compatible with Python 3+ 71 else: 72 dt = h5py.special_dtype(vlen=str) 73 74 # initialize the datasets 75 self._debug("creating datasets...") 76 self.imageIDDB = self.db.create_dataset("image_ids", (self.estNumImages,), 77 maxshape=(None,), dtype=dt) 78 self.indexDB = self.db.create_dataset("index", (self.estNumImages, 2), 79 maxshape=(None, 2), dtype="int") 80 self.featuresDB = self.db.create_dataset("features", 81 (approxFeatures, fvectorSize), maxshape=(None, fvectorSize), 82 dtype="float")</pre>	

Feedback

The `_createDatasets` method starts off by estimating the total number of feature vectors that will be stored in our `features` dataset (**Lines 60 and 61**). To perform this estimation, the constructor to our `FeatureIndexer` class accepted a special argument called `estNumImages`, which is the *approximate* number of images in our dataset. By taking this approximate number of images and the *average number of features associated with each image*, we can in turn construct a *rough estimate* to the total number of feature vectors (i.e. rows) that the `features` dataset will need to hold.

Line 64 defines a variable named `fvectorSize`, which — as the name suggests — is the size of the feature vector (i.e. in this case, the number of columns) that is being written to the HDF5 dataset. Given a 128-dim RootSIFT feature vector and the (x, y)-coordinates of the keypoint associated with the feature vector, our `fvectorSize` is thus 130.

(2018-02-07) Update for Python 3+: On **Lines 67-72** we handle creating the `h5py` datatype for either Python 2.7 or Python 3+.

Lines 75-82 handle constructing the actual HDF5 datasets for the image IDs, index, and features. At a bare-minimum, the `create_dataset` method requires two arguments: the *name* of the dataset, followed by the initial *shape* of the dataset. The `shape` is used in the exact same manner as defining a NumPy array.

For the `image_ids` dataset, we'll make the shape `(self.estNumImages,)`, implying that we'll be storing approximate `self.estNumImages` in our dataset.

However, a **very important attribute** to pay attention to is the `maxshape` parameter. As the name suggests, this value controls the **maximum size** of the HDF5 dataset. If no `maxshape` is defined, the maximum number of rows and columns is assumed to be the `shape` of the matrix. More importantly, if `maxshape` is not defined, **you will not be able to resize your dataset later!** Instead, you can specify a value of `None` along any dimension that you want to make resizable.

Since we do not know the number of images we'll be storing in `image_ids`, we'll make the total number of rows `None`, indicating that we'll be able to resize our `image_ids` dataset and store as many image IDs as we want.

Finally, we specify the `dtype` of the `imageIDDB` to be `unicode` since we'll be storing image filenames as strings.

The `indexDB` dataset is defined in a similar way, only using integers as the `dtype` and a shape of `(self.estNumImages, 2)`, where the two integers will be our lookups into the `features` dataset. We'll also specify `None` for the `maxshape` rows, indicating we can add as many rows as we want (but the number of columns will stay fixed at 2).

Finally, we define the `featuresDB`, naming it `features` and giving it an initial shape of `(approxFeatures, fvectorSize)`. This dataset will be a floating point data type since RootSIFT features are floats. Finally, we'll allow this dataset to have an unlimited number of rows as well.

Now that we have our datasets initialized, we need a way to write the buffers to them once the buffers are filled:

featureindexer.py

Python

```

84     def _writeBuffers(self):
85         # write the buffers to disk
86         self._writeBuffer(self.imageIDDB, "image_ids", self.imageIDBuffer,
87                           "index")
88         self._writeBuffer(self.indexDB, "index", self.indexBuffer, "index")
89         self._writeBuffer(self.featuresDB, "features", self.featuresBuffer,
90                           "features")
91
92         # increment the indexes
93         self.idxs["index"] += len(self.imageIDBuffer)
94         self.idxs["features"] += self.totalFeatures
95
96         # reset the buffers and feature counts
97         self.imageIDBuffer = []
98         self.indexBuffer = []
99         self.featuresBuffer = None
100        self.totalFeatures = 0

```

Lines 86-90 are relatively straightforward. For each of the `imageIDDB`, `indexDB`, and `featuresDB` datasets, we make a call to the `_writeBuffer` method (defined in the `BaseIndexer` class) to write each of the respective buffers to HDF5. For each call of `_writeBuffer`, we pass in the *dataset object*, the *name of the dataset*, the *buffer list/array*, and finally the *key into the `self.idxs` array*.

Once the calls to `_writeBuffer` have finished, we can then increment the `idxs` values on **Lines 93 and 94**. The `index` value will be incremented by the *total number of images* in the `imageIDBuffer`, whereas the `features` value will be incremented by the *total number of feature vectors* in the `featuresBuffer`.

Finally, the buffers and feature counts are reset on **Lines 97-100**.

Almost done! All we need to do now is create a `finish` method that should be called when all images in the UKBench dataset (or any other arbitrary image dataset) have been processed:

featureindexer.py

Python

```

102     def finish(self):
103         # if the databases have not been initialized, then the original
104         # buffers were never filled up
105         if None in (self.imageIDDB, self.indexDB, self.featuresDB):
106             self._debug("minimum init buffer not reached", msgType="[WARN]")
107             self._createDatasets()
108
109         # write any unempty buffers to file
110         self._debug("writing un-empty buffers...")
111         self._writeBuffers()
112
113         # compact datasets
114         self._debug("compacting datasets...")
115         self._resizeDataset(self.imageIDDB, "image_ids", finished=self.idxs["index"])
116         self._resizeDataset(self.indexDB, "index", finished=self.idxs["index"])
117         self._resizeDataset(self.featuresDB, "features", finished=self.idxs["features"])
118
119         # close the database
120         self.db.close()

```

Again, the `finish` method is only meant to be called after *all* images in the dataset have been processed.

We'll start off the `finish` function by checking to see if the HDF5 datasets have been initialized or not – if not, we'll want to create them (**Lines 105-107**). Ideally, the buffers should be dumped to disk a handful of times during the indexing process; however, for very large values of `maxBufferSize` and small values of `estNumImages`, it's possible that the buffers never filled up and have thus never been written to disk. In this case, we'll need to create the HDF5 datasets.

Line 111 makes a call to `_writeBuffers` which empties any leftover entries in the buffers to disk. We then compact each of the `image_ids`, `index`, and `features` datasets on **Lines 115-117**, claiming back any empty rows.

Our long journey ends on **Line 120** by closing the HDF5 database itself.

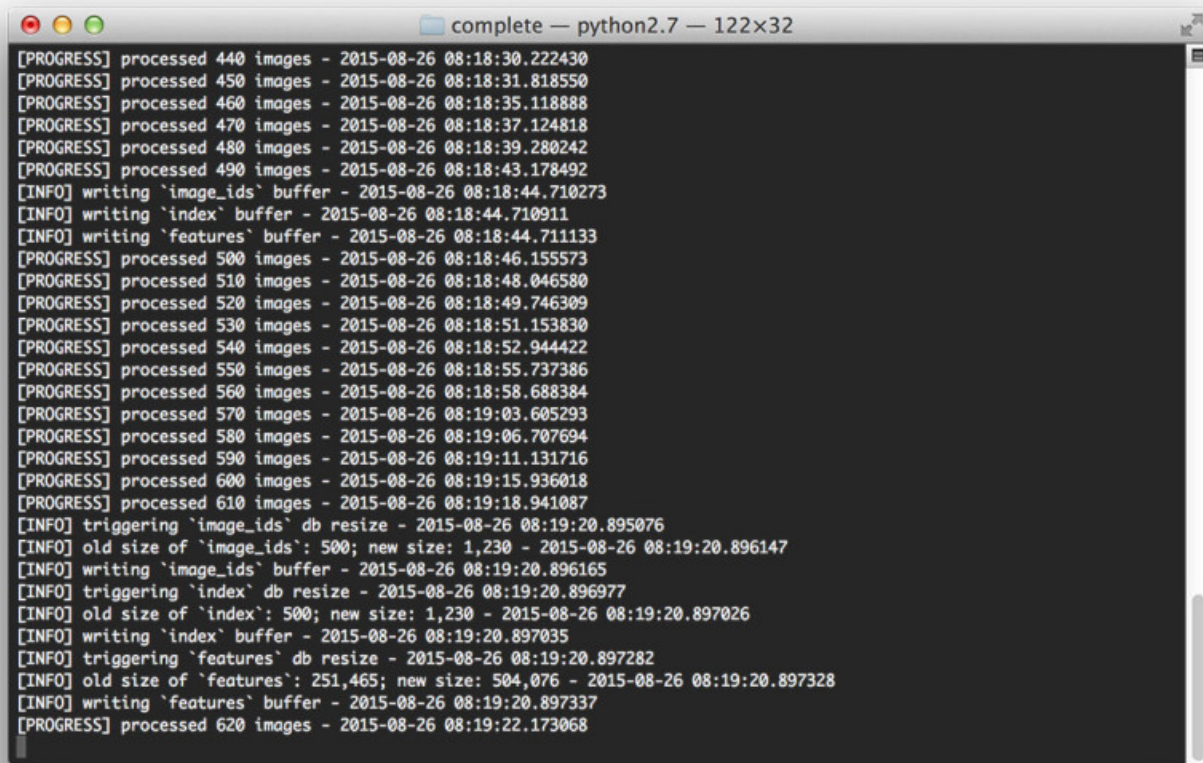
Running our feature extractor

Finally, after all this work, we can now execute our `index_features.py` script and extract features from our UKBench dataset.

Open up a terminal and execute the following commands:

index_features.py	Shell
1 \$ mkdir output	
2 \$ python index_features.py --dataset ../ukbench_sample --features-db output/features.hdf5	

This will kick off the indexing process:



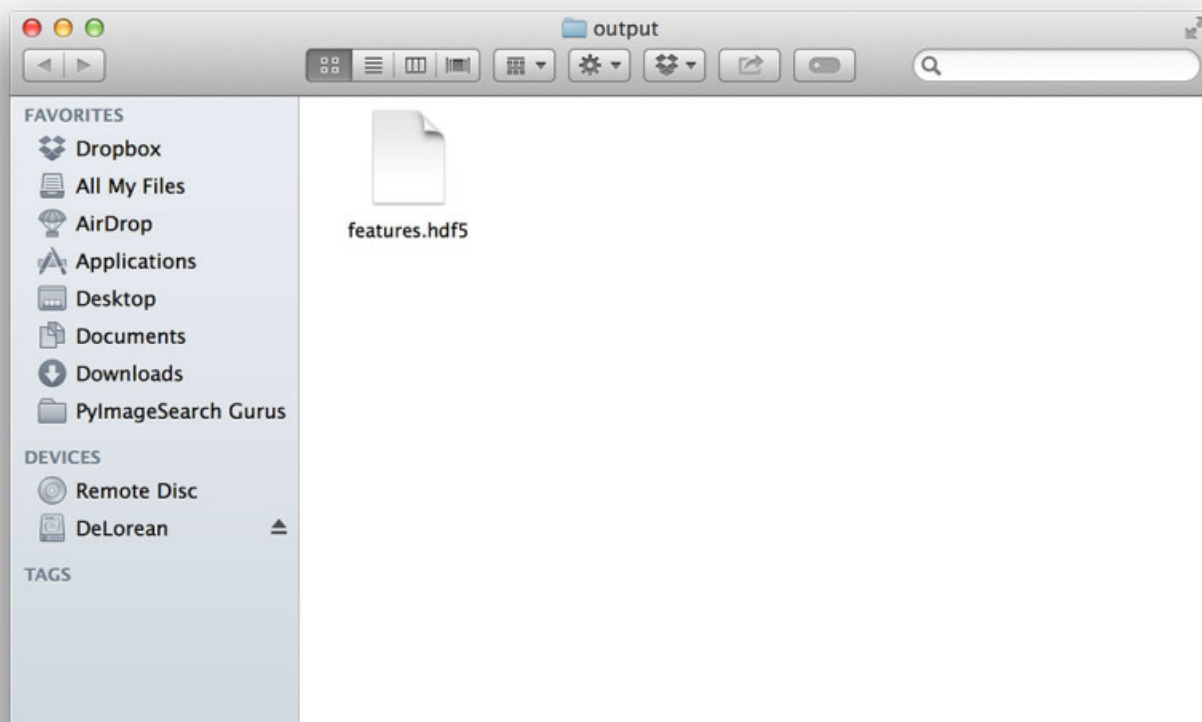
```
[PROGRESS] processed 440 images - 2015-08-26 08:18:30.222430
[PROGRESS] processed 450 images - 2015-08-26 08:18:31.818550
[PROGRESS] processed 460 images - 2015-08-26 08:18:35.118888
[PROGRESS] processed 470 images - 2015-08-26 08:18:37.124818
[PROGRESS] processed 480 images - 2015-08-26 08:18:39.280242
[PROGRESS] processed 490 images - 2015-08-26 08:18:43.178492
[INFO] writing 'image_ids' buffer - 2015-08-26 08:18:44.710273
[INFO] writing 'index' buffer - 2015-08-26 08:18:44.710911
[INFO] writing 'features' buffer - 2015-08-26 08:18:44.711133
[PROGRESS] processed 500 images - 2015-08-26 08:18:46.155573
[PROGRESS] processed 510 images - 2015-08-26 08:18:48.046580
[PROGRESS] processed 520 images - 2015-08-26 08:18:49.746309
[PROGRESS] processed 530 images - 2015-08-26 08:18:51.153830
[PROGRESS] processed 540 images - 2015-08-26 08:18:52.944422
[PROGRESS] processed 550 images - 2015-08-26 08:18:55.737386
[PROGRESS] processed 560 images - 2015-08-26 08:18:58.688384
[PROGRESS] processed 570 images - 2015-08-26 08:19:03.605293
[PROGRESS] processed 580 images - 2015-08-26 08:19:06.707694
[PROGRESS] processed 590 images - 2015-08-26 08:19:11.131716
[PROGRESS] processed 600 images - 2015-08-26 08:19:15.936018
[PROGRESS] processed 610 images - 2015-08-26 08:19:18.941087
[INFO] triggering 'image_ids' db resize - 2015-08-26 08:19:20.895076
[INFO] old size of 'image_ids': 500; new size: 1,230 - 2015-08-26 08:19:20.896147
[INFO] writing 'image_ids' buffer - 2015-08-26 08:19:20.896165
[INFO] triggering 'index' db resize - 2015-08-26 08:19:20.896977
[INFO] old size of 'index': 500; new size: 1,230 - 2015-08-26 08:19:20.897026
[INFO] writing 'index' buffer - 2015-08-26 08:19:20.897035
[INFO] triggering 'features' db resize - 2015-08-26 08:19:20.897282
[INFO] old size of 'features': 251,465; new size: 504,076 - 2015-08-26 08:19:20.897328
[INFO] writing 'features' buffer - 2015-08-26 08:19:20.897337
[PROGRESS] processed 620 images - 2015-08-26 08:19:22.173068
```

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

FIGURE 4: STARTING THE INDEXING PROCESS.

On my MacBook Pro, the feature extraction process took roughly 5m 38s.

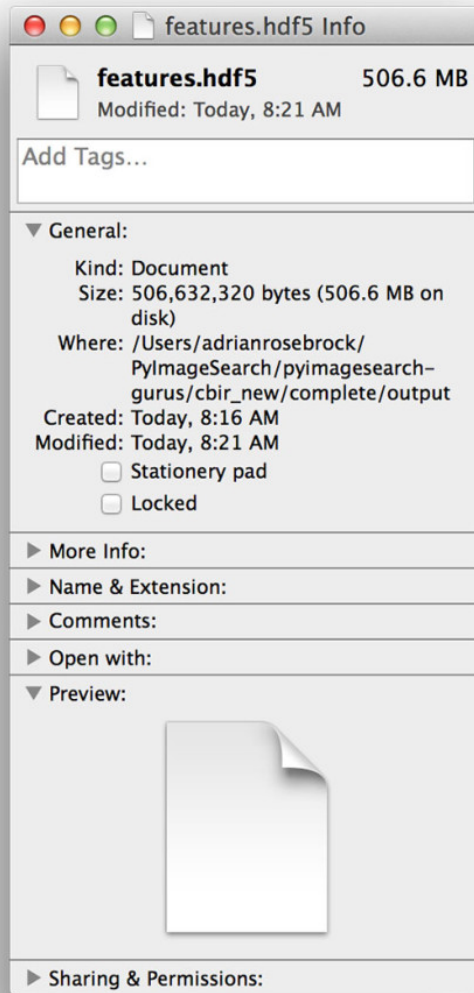
After the `index_features.py` script finishes running, you'll see a `features.hdf5` file in the `output` directory:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/cbir_extracting_hdf5_output.jpg).

FIGURE 5: THE OUTPUT FEATURES.HDF5 FILE.

This file is our HDF5 database and contains the `image_ids` , `index` , and `features` datasets. This file is also quite large, weighing in at 506mb:



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/cbir_extracting_hdf5_file_size.jpg).

FIGURE 6: STORING RAW LOCAL INVARIANT FEATURE VECTORS OFTEN TAKES MORE DISK SPACE THAN THE IMAGES THEMSELVES.

As you can see, storing the feature vectors associated with our 1,000 images takes more space than the image files themselves! However, this was to be expected — see **The cost of feature vector storage** section earlier in this lesson for more information on the space requirements for storing feature vectors on disk.

Given our HDF5 database, let's fire up Python IDLE and poke around with our newly extracted feature vectors. First, let's open database and examine the size of each of the datasets:

Extracting keypoints and local invariant descriptors	Python

```

1 >>> import h5py
2 >>> db = h5py.File("features.hdf5", mode="r")
3 >>> list(db.keys())
4 [u'features', u'image_ids', u'index']
5 >>> db["image_ids"].shape
6 (1000,)
7 >>> db["index"].shape
8 (1000, 2)
9 >>> db["features"].shape
10 (523498, 130)

```

Here, we can see that our `db` has three keys, just like a dictionary: `features` , `image_ids` , and `index` .

The `image_ids` dataset has 1,000 rows, one for each unique image filename. The `index` dataset also has 1,000 rows, but stores two integers: the starting and ending slice ranges into the `features` dataset for the image at index i . Finally, the `features` dataset has 523,498 rows, nearly half a million feature vectors extracted from our UKBench dataset. You'll also notice that the `features` dataset has 130 columns, two for the (x, y)-coordinates of the keypoints, and 128 for the 128-dim RootSIFT feature vector.

Now, let's see how we can use all three datasets in conjunction with each other to fill out all keypoints and feature vectors associated with a given image:

Extracting keypoints and local invariant descriptors

Python

```

1 >>> imageID = db["image_ids"][732]
2 >>> imageID
3 u'ukbench00894.jpg'

```

Here, I randomly choose an `imageID` from the `image_ids` dataset — I supply a value of 732 as the index into the dataset. This corresponds to the image filename `ukbench00894.jpg` .

I can use this same value of 732 to grab the starting and ending slices into the `features` dataset by first checking my `index` dataset:

Extracting keypoints and local invariant descriptors

Python

```

1 >>> (start, end) = db["index"][732]
2 >>> start, end
3 (382710, 384063)
4 >>> end - start
5 1353

```

The values `382710` and `384063` represent the starting and ending slice ranges into `features` , respectively. Subtracting the two from each other, I get a value of 1353, indicating that image `ukbench00894.jpg` has 1353 keypoints and feature vectors associated with it.

Pulling out these 1353 feature vectors is also equally simple:

Extracting keypoints and local invariant descriptors	Python
<pre> 1 >>> rows = db["features"][start:end] 2 >>> rows.shape 3 (1353, 130) 4 >>> kps = rows[:, :2] 5 >>> desc = rows[:, 2:] 6 >>> kps.shape 7 (1353, 2) 8 >>> desc.shape 9 (1353, 128) </pre>	

All I need to do is supply `start` and `end` as the slice ranges to the `features` dataset, and I receive my 1353 feature vectors in return. From there, it's just simple NumPy slice operations to split the keypoints and feature vectors from each other.

Note: *Your results may differ in your environment.*

Summary

In this lesson, we learned how to extract keypoints and local invariant descriptors from our UKBench dataset. We then used HDF5 to efficiently store these feature vectors for later use.

Future lessons in this module will detail how to take these extracted feature vectors, cluster them to form a codebook, construct a bag-of-visual-words model for each image in the dataset, perform the initial search, and finally utilize spatial verification to improve search results.

There was *a lot* content covered in this lesson and many new techniques introduced, especially related to the HDF5 format and hierarchical structure. I *highly suggest* that you read through this lesson *multiple times*, not to mention download the code and play around with it until you feel comfortable you know what each block of code is doing. Future lessons in the rest of this CBIR module will build upon this content and utilize HDF5, so make sure you are familiar with the concept of storing and accessing large NumPy arrays on disk before you move forward.

Downloads:

Download the Code

(https://gurus.pyimagesearch.com/protected/code/cbir/extracting_keypoints_and_descs.py)

Quizzes	Status
1	

Extracting Keypoints and Local Invariant Descriptors Quiz (<https://gurus.pyimagesearch.com/quizzes/extracting-keypoints-and-local-invariant-descriptors-quiz/>)

[← Previous Lesson \(https://gurus.pyimagesearch.com/lessons/the-bag-of-visual-words-model/\)](https://gurus.pyimagesearch.com/lessons/the-bag-of-visual-words-model/) [Next Lesson → \(https://gurus.pyimagesearch.com/lessons/clustering-features-to-form-a-codebook/\)](https://gurus.pyimagesearch.com/lessons/clustering-features-to-form-a-codebook/)

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! \(/pyimagesearch-gurus-course/\)](/pyimagesearch-gurus-course/)

Resources & Links

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

Your Account

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

Q Search

