3.6: Clustering features to form a codebook

After our last lesson **extracting keypoints and local invariant descriptors (https://gurus.pyimagesearch.com/lessons/extracting-keypoints-and-local-invariant-descriptors/)**, you might be feeling a bit overwhelmed. That single lesson was *jam packed* with a bunch of new concepts including HDF5 databases, index lookups, and feature vector storage — *all coded in a manner to function in the most efficient way possible*.

When I first encountered these concepts, I couldn't help but think to myself *"Wow, how hard is it* really *to build a scalable image search engine? Is this something I can do?"*

What I didn't know at the time, and what I'm telling you now, is ***that it only gets easier from here***. By front-loading our efforts on creating *extremely efficient, highly scalable data structures*, we pave the way for an easier route — a route that leads to an immensely powerful image search engine with a lot less effort.

But don't take my word for it. Today's lesson on *feature clustering* and *codebook construction* will show you how easy it is to build a vocabulary now that our feature vectors have been stored in an efficient manner.

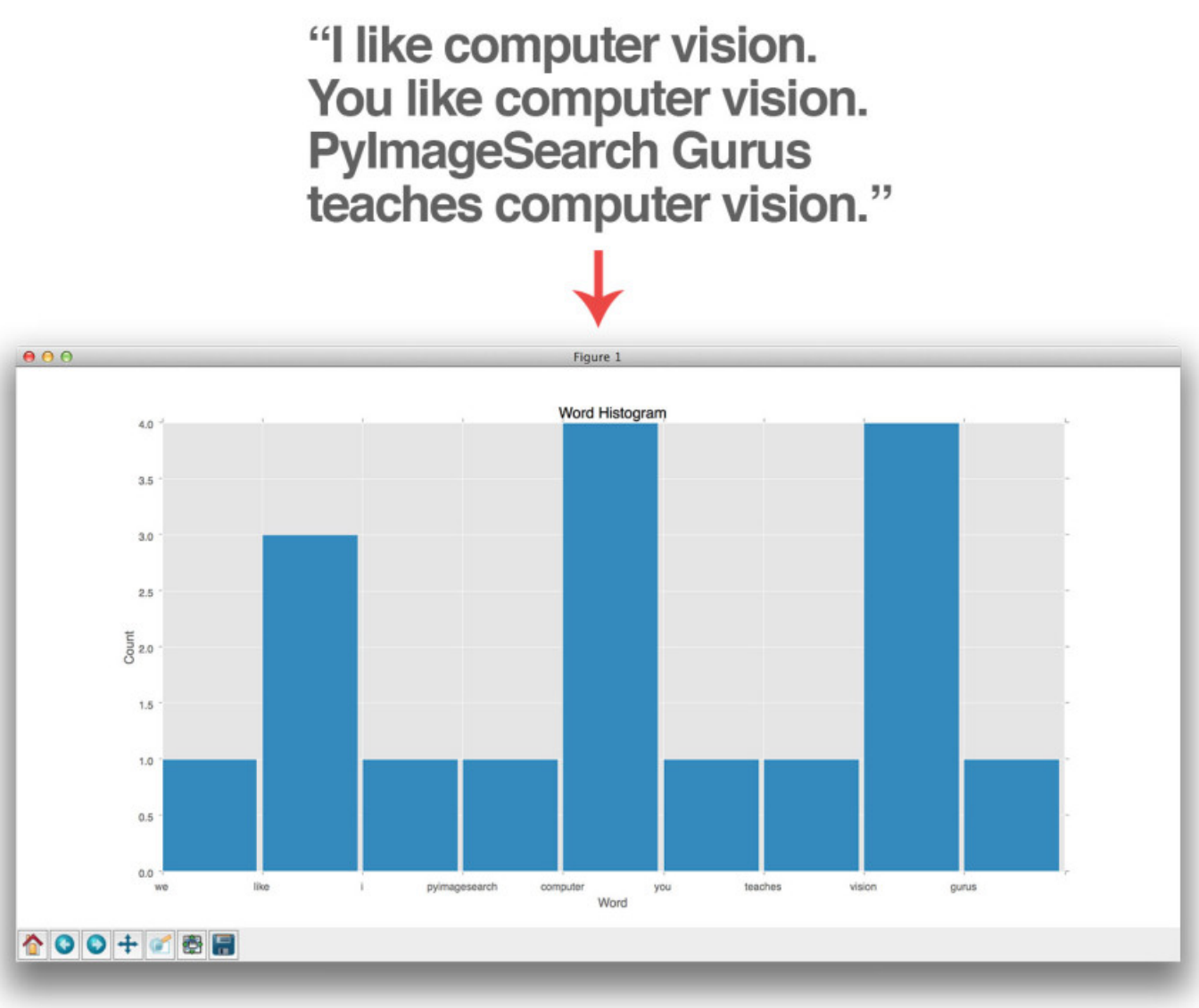
Objectives:

In this lesson we will:

- Create `Vocabulary` class that can be used to take a HDF5 dataset of features, cluster a subset of the features, and then return the resulting centroids (i.e. our *visual words*).
- Construct a driver script to apply the `Vocabulary` to our HDF5 dataset of UKBench feature vectors.

Clustering features to form a codebook

When working with text data, constructing a bag of visual words is easy: *we already have our dictionary*. It's simply the total set of keywords used in the dataset of documents. All we need to do is loop over each document in the collection and construct a histogram that records each time a given keyword appears:



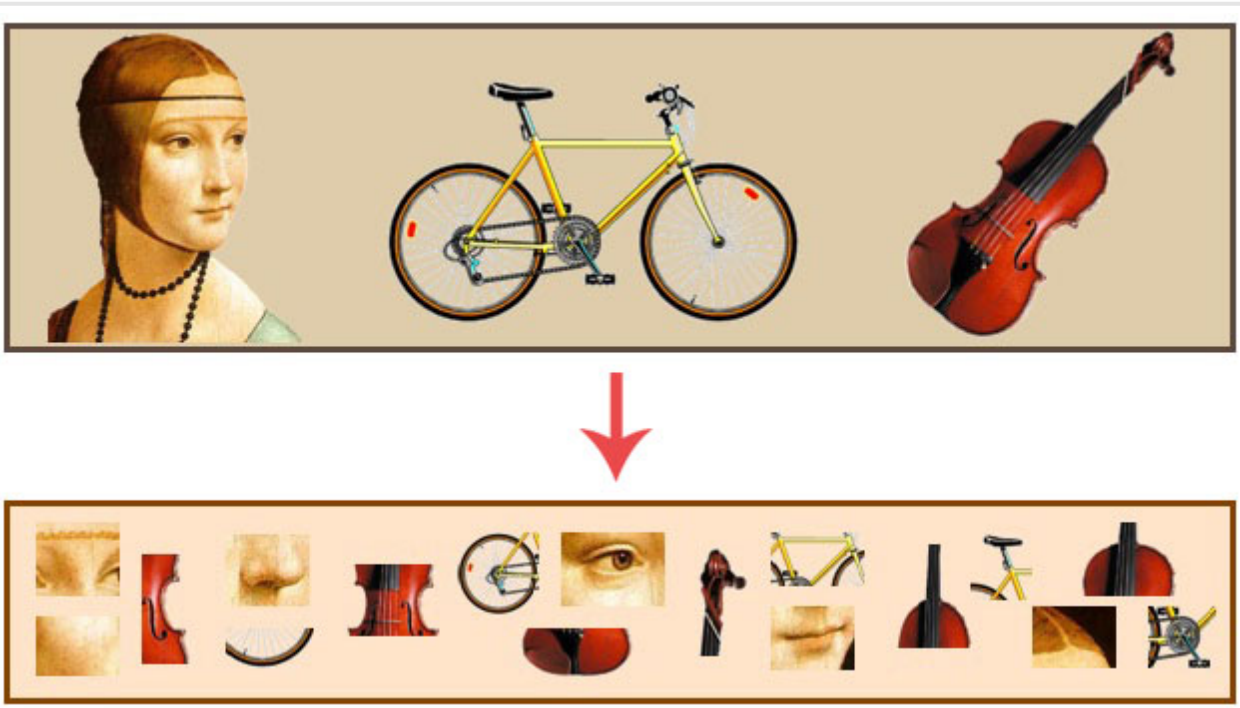
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/09/bovw_word_histogram.jpg)

FIGURE 1: CONSTRUCTING A BAG OF VISUAL WORDS FOR A TEXT DOCUMENT IS EASY — WE SIMPLY COUNT THE NUMBER OF TIMES EACH WORD APPEARS IN THE DOCUMENT.

Based on *all* the keywords from *all* the documents in our collection we can form our *dictionary*, or the global set of all possible words in our "document universe":

dictionary = {we, like i, pyimagesearch, computer, you, teaches, vision gurus}

But in computer vision, it's not so easy — there is no such thing as "dictionary" of visual words in the same way there is a "dictionary" for text keywords. Instead, we need to build our own dictionary:



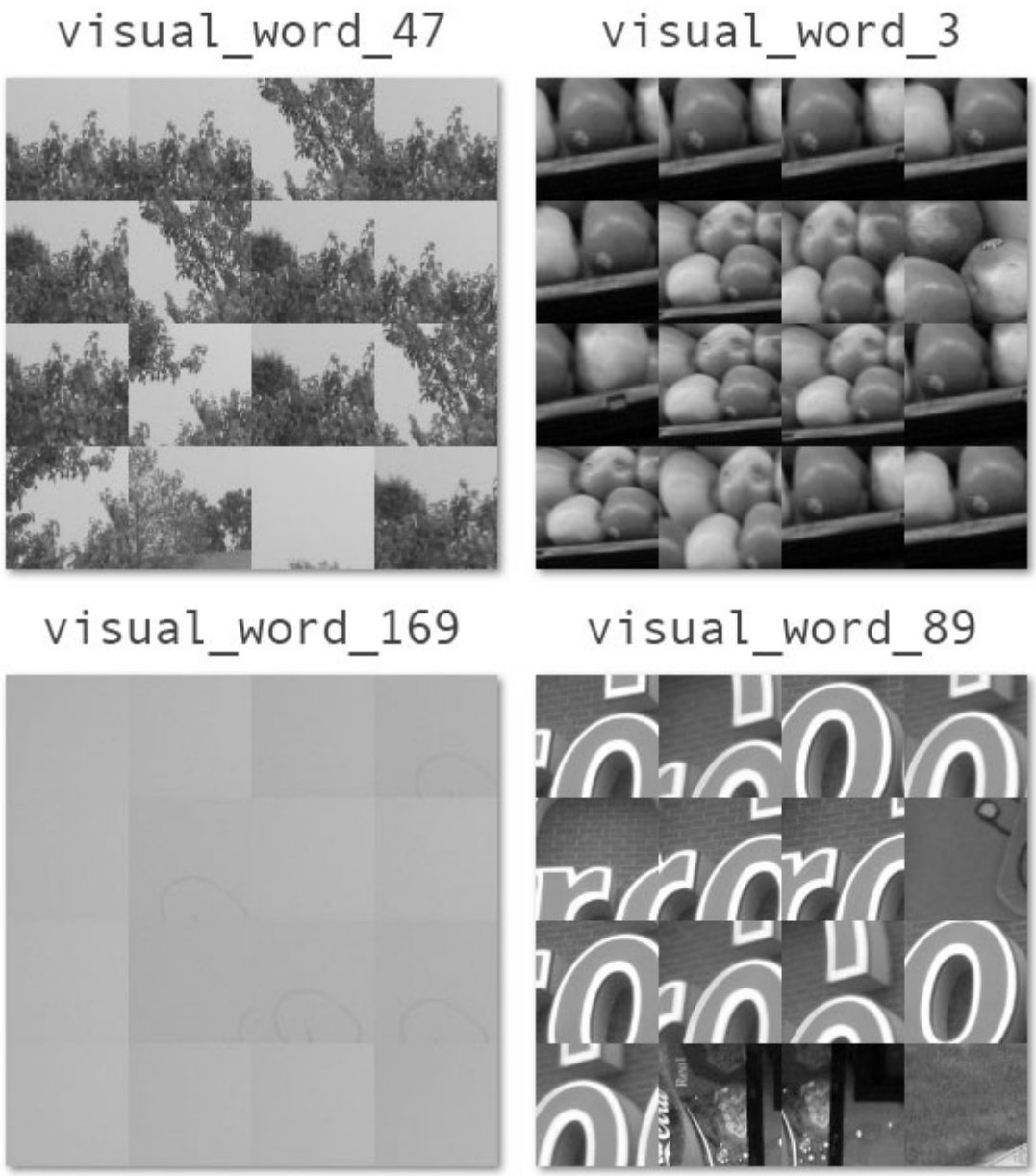
(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/clustering_features_visual_vocab.jpg)

FIGURE 2: TAKING A DATASET OF IMAGES AND EXTRACTING "VISUAL WORDS" (I.E. IMAGE PATCHES AND THEIR CORRESPONDING FEATURE VECTORS) TO FORM OUR DICTIONARY.

With this in mind, we are now moving on to **Step 2** in building a *bag of visual words model*: clustering our extracted feature vectors to form our "vocabulary", or simply the resulting cluster centers generated by the **k-means (https://gurus.pyimagesearch.com/lessons/k-means-clustering/)** clustering algorithm.

As detailed in our **bag of (visual) words model (https://gurus.pyimagesearch.com/lessons/the-bag-of-visual-words-model/)** lesson, our goal is to take an image that is represented using *multiple* feature vectors (such as SIFT, SURF, etc.) and then construct a histogram for each image of image patch occurrences that tabulate the frequency of each of these prototype vectors.

A "prototype" vector is simply a "visual word" — it's an abstract quantification of a region in an image. Some visual words may encode for corner regions. Others visual words may represent edges. Even other visual words symbolize areas of low texture:

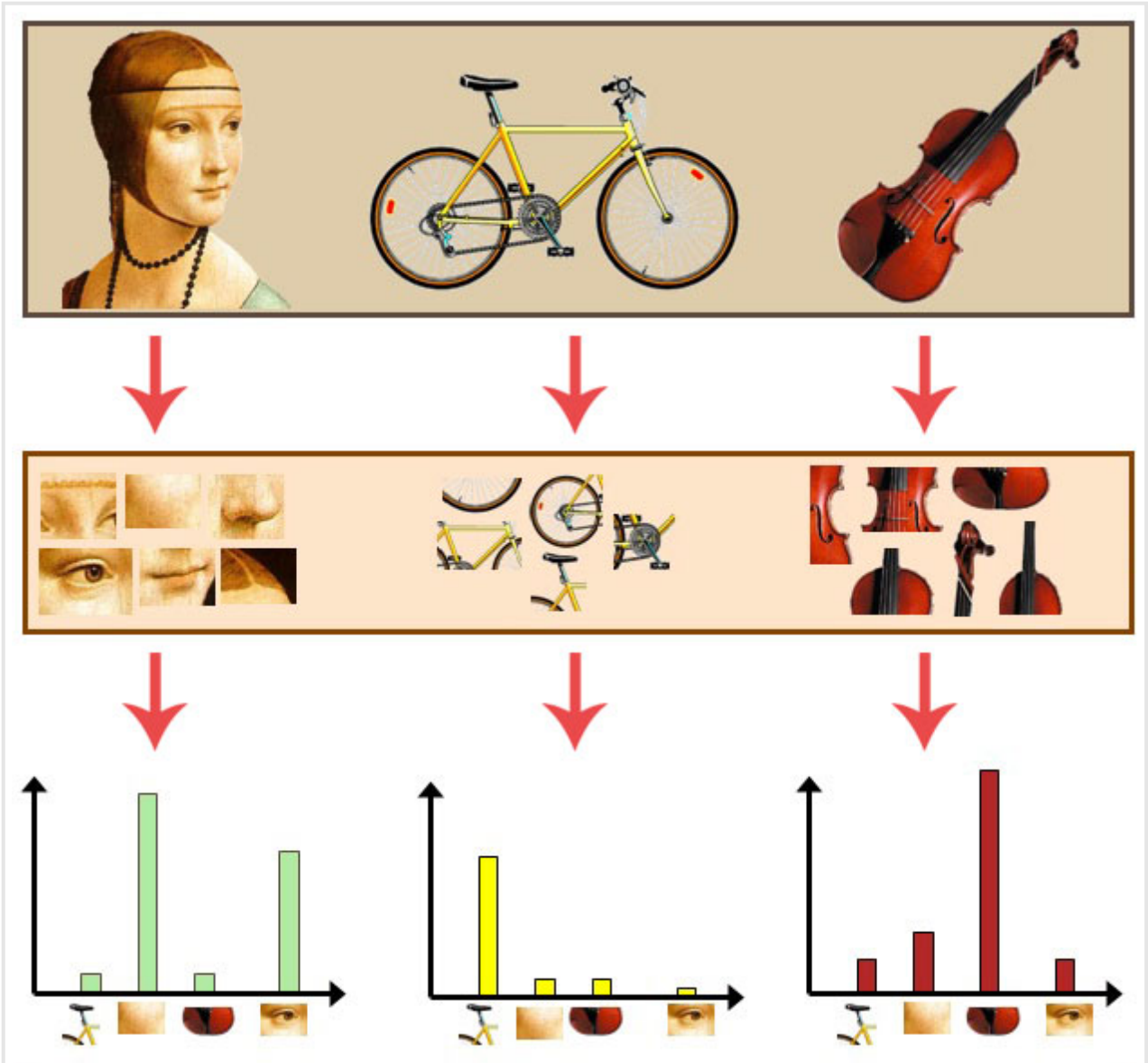


(https://gurus.pyimagesearch.com/wp-content/uploads/2015/03/clustering_features_vocab_example.jpg)

FIGURE 3: AN EXAMPLE OF FOUR VISUAL WORDS. THE *TOP-LEFT* VISUAL WORD CAPTURES TREE, BRANCH, AND SKY-LIKE PATCHES. THE *TOP-RIGHT* DESCRIBES FRUIT AND FRUIT-STAND PATCHES. THE *BOTTOM-LEFT* PATCH HAS RELATIVELY ZERO TEXTURE. FINALLY, THE *BOTTOM-RIGHT* PATCH SHOWS TEXT-LIKE STRUCTURES.

Note: We'll learn how to construct visual word representations like the one above in our next lesson on **visualizing words in a codebook (https://gurus.pyimagesearch.com/lessons/visualizing-words-in-a-codebook/)**.

And when we tabulate how many each of these "visual words" occur, we obtain a *single histogram* that neatly and compactly quantifies the contents of an image.



(https://gurus.pyimagesearch.com/wp-content/uploads/2015/09/bovw_visual_word_counts.jpg)

FIGURE 4: TAKING A *(1)* A DICTIONARY OF VISUAL WORDS, *(2)* EXTRACTED IMAGE PATCHES, AND *(3)* CONSTRUCTING A HISTOGRAM THAT COUNTS THE FREQUENCY OF EACH VISUAL WORD IN THE DICTIONARY.

This process all starts with forming our dictionary (again, which I'll refer to as a codebook or vocabulary interchangeably) .

Let's revisit the structure of our CBIR project and add in two new files: `vocabulary.py` and `cluster_features.py` :

```
Directory structure of our codebook extraction                    Shell
```

```
|--- pyimagesearch
|    |--- __init__.py
|    |--- descriptors
|    |    |---- __init__.py
|    |    |--- detectanddescribe.py
|    |--- indexer
|    |    |---- __init__.py
|    |    |--- baseindexer.py
|    |    |--- featureindexer.py
|    |--- ir
|    |    |---- __init__.py
|    |    |--- vocabulary.py
|--- cluster_features.py
|--- index_features.py
```

We'll define a `Vocabulary` class inside the `vocabulary.py` file of the `ir` sub-module (where "ir" stands for "Information Retrieval" since we are borrowing concepts from the IR field) used to ingest a HDF5 dataset of features and then return a dictionary (i.e., cluster centers) of visual words. These visual words will serve as our vector prototypes when we quantize the feature vectors into a single histogram of visual word occurrences in our **vector quantization (https://gurus.pyimagesearch.com/lessons/vector-quantization-and-constructing-a-bovw/)** lesson.

We'll also create a `cluster_features.py` script. As the name suggests, this driver script will kick-off the `Vocabulary` formulation process.

If you're starting to feel overwhelmed, don't worry — I promise that this is all a lot easier than it sounds. With that in mind, let's open up `vocabulary.py` and start working on our `Vocabulary` class:

vocabulary.py Python

```
# import the necessary packages
from __future__ import print_function
from sklearn.cluster import MiniBatchKMeans
import numpy as np
import datetime
import h5py

class Vocabulary:
    def __init__(self, dbPath, verbose=True):
        # store the database path and the verbosity setting
        self.dbPath = dbPath
        self.verbose = verbose
```

We'll start off by importing our necessary packages on **Lines 2-6**. These are all imports you have seen before, with the exception of `MiniBatchKMeans` which is just a more efficient and scalable version (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html) of the original k-means algorithm [Sculley 2010 (https://gurus.pyimagesearch.com/wp-content/uploads/2015/09/sculley_2010.pdf)].

While I won't go into the details of the mini-batch k-means algorithm, it essentially works by breaking the dataset into small segments, clustering each of the segments individually, then merging the clusters resulting from each of these segments together to form the final solution. This is in stark contrast to the standard k-means algorithm which clusters *all* of the data in a single segment. While the clusters obtained from mini-batch k-means aren't necessarily as "accurate" as the ones using the standard k-means algorithm, the primary benefit is that mini-batch k-means is that it's often ***an order of magnitude*** (or more) faster than standard k-means. In reality, we don't need the perfect cluster centers from k-means, reasonable approximations will often do, hence why it's a good idea to opt for mini-batch k-means over standard k-means whenever possible.

Line 9 defines the constructor of our `Vocabulary` class. This class only requires a single parameter — the `dbPath` to our HDF5 database file.

Let's move on to some code that is a bit more exciting:

vocabulary.py — Python

```
14      def fit(self, numClusters, samplePercent, randomState=None):
15          # open the database and grab the total number of features
16          db = h5py.File(self.dbPath)
17          totalFeatures = db["features"].shape[0]
18
19          # determine the number of features to sample, generate the indexes of the
20          # sample, sorting them in ascending order to speedup access time from the
21          # HDF5 database
22          sampleSize = int(np.ceil(samplePercent * totalFeatures))
23          idxs = np.random.choice(np.arange(0, totalFeatures), (sampleSize), replace=False)
24          idxs.sort()
25          data = []
26          self._debug("starting sampling...")
27
28          # loop over the randomly sampled indexes and accumulate the features to
29          # cluster
30          for i in idxs:
31              data.append(db["features"][i][2:])
```

Line 14 defines our `fit` method in spirit of the scikit-learn library. Our `fit` method requires two arguments: the number of clusters to generate and the percentage of the dataset to sample prior to running mini-batch k-means. Since k-means can be a slow algorithm to run on large datasets (and feature vectors extracted from images can already number at 500,000+ for only 1,000 images), we commonly only use 10-25% of the feature vectors when clustering.

Next up, we open our HDF5 dataset for reading and then grab the total number of feature vectors from the `features` dataset (**Lines 16 and 17**).

Using the `totalFeatures` and the `samplePercent`, we can compute the `sampleSize` for the number of feature vectors to select for clustering (**Line 22**).

We'll use the NumPy's `choice` method to sample indexes ***without replacement*** in the range *[0, totalFeatures]*. Since HDF5 expects index accesses to be supplied in sorted order (or otherwise read accesses will be very slow and inefficient), we'll then sort the `idxs` array.

At this point, all that is left to do is loop over each of the `idxs` (**Line 30**) and accumulate the `data` matrix of sampled feature vectors.

Given our `data` matrix, we are now ready to perform k-means clustering:

vocabulary.py Python

```
33          # cluster the data
34          self._debug("sampled {:,} features from a population of {:,}".format(
35              len(idxs), totalFeatures))
36          self._debug("clustering with k={:,}".format(numClusters))
37          clt = MiniBatchKMeans(n_clusters=numClusters, random_state=randomState)
38          clt.fit(data)
39          self._debug("cluster shape: {}".format(clt.cluster_centers_.shape))
40
41          # close the database
42          db.close()
43
44          # return the cluster centroids
45          return clt.cluster_centers_
46
47      def _debug(self, msg, msgType="[INFO]"):
48          # check to see the message should be printed
49          if self.verbose:
50              print("{} {} - {}".format(msgType, msg, datetime.datetime.now()))
```

On **Line 37** we instantiate our `MiniBatchKMeans` class, implemented in scikit-learn (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html). Here we specify the `numClusters` to generate along with an (optional) random state, allowing us to repeat our clustering results if need be. Finally, the clustering process is started on **Line 38** with a call to `fit`.

After the clustering process finishes, we close the database and then return the resulting cluster centers (i.e. our visual words) to the calling method.

Hopefully by now you can see that I wasn't joking about building a CBIR system getting easier. Most of the code in our `Vocabulary` class is either (1) imports, (2) debug statements, or (3) code used to grab the randomly sampled feature vectors from our HDF5 dataset — only *two* lines of code are required to perform the actual clustering.

Now that we have our `Vocabulary` class defined, let's create the `cluster_features.py` driver script:

cluster_features.py Python

```python
# import the necessary packages
from __future__ import print_function
from pyimagesearch.ir import Vocabulary
import argparse
import pickle

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-f", "--features-db", required=True,
    help="Path to where the features database will be stored")
ap.add_argument("-c", "--codebook", required=True,
    help="Path to the output codebook")
ap.add_argument("-k", "--clusters", type=int, default=64,
    help="# of clusters to generate")
ap.add_argument("-p", "--percentage", type=float, default=0.25,
    help="Percentage of total features to use when clustering")
args = vars(ap.parse_args())

# create the visual words vocabulary
voc = Vocabulary(args["features_db"])
vocab = voc.fit(args["clusters"], args["percentage"])

# dump the clusters to file
print("[INFO] storing cluster centers...")
f = open(args["codebook"], "wb")
f.write(pickle.dumps(vocab))
f.close()
```

Feedback

The vast majority of the `cluster_features.py` script is spent parsing command line arguments, each of which I detail below:

- `--features-db` : This is the path to our HDF5 database that contains our `image_ids` , `index` , and most importantly for this script, `features` datasets.
- `--codebook` : After we run mini-batch k-means on our sampled feature vectors we need a place to store the resulting cluster centers on disk — this is the path to where our visual words dictionary will be stored.
- `--clusters` : The number of clusters (i.e., visual words) that mini-batch k-means will generate. The number of visual words to use is always a point of contention and not an easy parameter to tune. I'll discuss this caveat in more detail at the end of this lesson.
- `--percentage` : As I mentioned earlier, we often don't use the *entire* set of feature vectors when clustering, it would simply take too long. Instead, we sample *without replacement* from the dataset and then cluster only on the sampled features. As the name suggests, the `--percentage` controls the size of the feature vector sample size.

After parsing our command line arguments, we instantiate our `Vocabulary` on **Line 20** and obtain the resulting visual words on **Line 21** — the entire sampling and clustering process is abstracted by the `Vocabulary` class.

Finally, all that's left is to take our NumPy array of cluster centroids and dump them to file in `cPickle` format on **Lines 25-27**.

As I said, clustering feature vectors and obtaining the resulting codebook becomes a very easy task when you have the proper dataset infrastructure setup — which is *exactly* why we spent so much time detailing how to construct an efficient dataset using HDF5 in our **previous lesson (https://gurus.pyimagesearch.com/lessons/extracting-keypoints-and-local-invariant-descriptors/)**.

Generating the codebook

Now that our driver script is done, let's generate our codebook:

```
cluster_features.py                                                    Shell
1 python cluster_features.py --features-db output/features.hdf5 --codebook output/vocab.cpickle \
2     --clusters 1536 --percentage 0.25
```

On my machine, this script will take approximately **3m 1s** to generate *k=1,536* cluster centers, where much of the time is spent accessing the HDF5 dataset. While HDF5 is *extremely fast* for NumPy-like slice operations (which is exactly what we did when we grabbed entire rows of feature vectors from the dataset in the previous lesson), it's not very efficient for *random access* which is required to build the sample of feature vectors to cluster.

That said, trading slower random access for near infinite NumPy matrix storage in a manner that can quickly and efficiently be accessed via slicing is well worth it — this point will become even more clear when we get to the topic of *spatial verification* later in this module.

After the `cluster_features.py` script executes on your system, you should have an `vocab.cpickle` in your `output` directory. Go ahead and change to the `output` directory and explore the visual vocabulary:

```
Exploring the visual vocabulary                                        Shell
```

```
$ python
>>> import pickle
>>> vocab = pickle.loads(open("vocab.cpickle", "rb").read())
>>> vocab.shape
(1536, 128)
>>> vocab[0]
array([ 0.11123683,  0.10837882,  0.08192626,  0.05929523,  0.05589489,
        0.08150352,  0.07448516,  0.07174279,  0.11738609,  0.16164516,
        0.07359745,  0.03630495,  0.06993915,  0.1109158 ,  0.0650181 ,
        0.05576601,  0.05814129,  0.0762072 ,  0.01825257,  0.01422932,
        0.06747166,  0.0654792 ,  0.02392728,  0.01855116,  0.00830103,
        0.01224354,  0.0051013 ,  0.01074217,  0.05364205,  0.04901215,
        0.01407343,  0.00195085,  0.13296704,  0.12198274,  0.0743954 ,
        0.05419618,  0.06686475,  0.08821686,  0.08003182,  0.09280612,
        0.16785861,  0.16664424,  0.08651175,  0.06746129,  0.15242398,
        0.15400148,  0.0809649 ,  0.07800419,  0.15796522,  0.11693972,
        0.03031797,  0.03874443,  0.1174408 ,  0.08106413,  0.03301498,
        0.04894181,  0.00278801,  0.00550713,  0.00350455,  0.01324967,
        0.05598286,  0.03992321,  0.013384  ,  0.00450074,  0.10976932,
        0.14684016,  0.09832302,  0.05836446,  0.06744226,  0.09754715,
        0.0834092 ,  0.08068604,  0.17025123,  0.15566562,  0.1113125 ,
        0.0972802 ,  0.15191865,  0.1241293 ,  0.08671884,  0.11270912,
        0.1697216 ,  0.08386806,  0.04417334,  0.06336761,  0.1202775 ,
        0.06074962,  0.03859587,  0.08716761,  0.00506535,  0.00901106,
        0.01175004,  0.02415728,  0.05119791,  0.02458242,  0.00522437,
        0.00392357,  0.09368528,  0.11348678,  0.07761503,  0.05527597,
        0.05956403,  0.11254   ,  0.11691682,  0.09052995,  0.1345629 ,
        0.12094013,  0.08693528,  0.08216955,  0.08940452,  0.09487003,
        0.12011178,  0.13208795,  0.11120937,  0.04996527,  0.03905898,
        0.0638725 ,  0.06894389,  0.0304223 ,  0.03478306,  0.08201623,
        0.00428349,  0.00794239,  0.01368838,  0.02736828,  0.03475052,
        0.00951142,  0.00257433,  0.0029725 ])
```

Feedback

You'll notice that the `vocab` variable is simply a raw NumPy array with *1,536* rows and *128* columns. The *rows* are our "visual words" — and each row has 128 columns since our **RootSIFT (https://gurus.pyimagesearch.com/topic/rootsift/)** feature vectors are of 128-dim. Thus, a single visual word is just the cluster center of a set of RootSIFT feature vectors!

In the next lesson, I'll demonstrate how to use our visual vocabulary to quantize the set of the RootSIFT features associated with an image into a *single histogram* that concisely and compactly represents the visual contents of an image.

How do you choose the correct codebook size?

Choosing the optimal codebook/vocabulary size is one of the hardest aspects of using the bag of visual words. If the codebook is too small, then you won't be able to discriminate between different visual contents of your image. It would be like me telling you to describe both a *mountain* and an *ocean* using ***only*** the words "the", "because, and "balloon" — you would simply lack the vocabulary to adequately describe a mountain and an ocean.

Similarly, if your codebook is too large, then your bag of visual words will have *too many* components and images with very similar contents may be regarded as "not similar" when, in reality, they are. A good example of this problem would be me telling you to describe the difference between an *ocean* and a *lake* — but only by using words you find in the thesaurus for "water", "lagoon", and "waves". A simple search in a thesaurus for these keywords could easily return *1,000's* of possible keywords — and your descriptions of both *lake* and *pond* would look very different, even though they are both bodies of water.

All that said, determining the optimal size of a codebook in the real world is done via experimentation, starting by:

1. Deciding on a set of cluster sizes *K* to evaluate.
2. Clustering the feature vectors using each *k* in *K*.
3. Evaluating the performance of the codebook according to some metric (ex. raw accuracy, precision, recall, f-measure, etc.)

While we haven't written any code to actually *evaluate* a CBIR system, we will very soon in the upcoming lessons in this module.

In general, you'll want to use *smaller*, *dense* codebook sizes when performing machine learning since you're trying to "learn" what a given class looks like, especially since there will (likely) be a large amount of visual variation within each class.

On the other hand, *larger*, *sparse* codebook sizes are better for actual *instance recognition* — for example when performing CBIR and wanting to find a *single*, *particular motorcycle* out of a dataset of 10,000 motorcycles. In that particular case, a large codebook would be extremely helpful.

Summary

The bag of visual words model was one of the hardest computer vision concepts for me to wrap my head around as an undergraduate in college — I simply couldn't understand what a "codebook" or "dictionary" was.

It wasn't until I sat down and wrote out the code myself that it became evidently clear — a codebook/dictionary/vocabulary is simply the ***output*** of running k-means on a dataset of feature vectors. That's it!

From that point on, I understood the logic behind a bag of visual words. Given the codebook, all I needed to do was take the feature vectors associated with a given image, find their closest neighbors in the codebook, and then tabulate the number of nearest neighbors into a histogram — which is exactly what we are going to cover in our next lesson.

Downloads:

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

	Quizzes	Status
1	Clustering Features to Form a Codebook Quiz (https://gurus.pyimagesearch.com/quizzes/clustering-features-to-form-a-codebook-quiz/)	

← Previous Lesson (https://gurus.pyimagesearch.com/lessons/extracting-keypoints-and-local-invariant-descriptors/) Next Lesson → (https://gurus.pyimagesearch.com/lessons/visualizing-words-in-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/)

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

/