Creating features using visual codebook and vector quantization

In order to build an object recognition system, we need to extract feature vectors from each image. Each image needs to have a signature that can be used for matching. We use a concept called **visual codebook** to build image signatures. This codebook is basically the dictionary that we will use to come up with a representation for the images in our training dataset. We use vector quantization to cluster many feature points and come up with centroids. These centroids will serve as the elements of our visual codebook. You can learn more about this at **http://mi.eng.cam.ac.uk/~cipolla/lectures/PartIB/old/IB-visualcodebook.pdf**.

Before you start, make sure that you have some training images. You were provided with a sample training dataset that contains three classes, where each class has 20 images. These images were downloaded from **http://www.vision.caltech.edu/html-files/archive.html**.

To build a robust object recognition system, you need tens of thousands of images. There is a dataset called `Caltech256` that's very popular in this field! It contains 256 classes of images, where each class contains thousands of samples. You can download this dataset at **http://www.vision.caltech.edu/Image_Datasets/Caltech256**.

How to do it...

1. This is a lengthy recipe, so we will only look at the important functions. The full code is given in the `build_features.py` file that is already provided to you. Let's look at the class defined to extract features:

```
class FeatureBuilder(object):
```

2. Define a method to extract features from the input image. We will use the Star detector to get the keypoints and then use SIFT to extract descriptors from these locations:

```
def extract_ features(self, img):
    keypoints = StarFeatureDetector().detect(img)
    keypoints, feature_vectors = compute_sift_features(img, keypoints)
    return feature_vectors
```

3. We need to extract centroids from all the descriptors:

```
def get_codewords(self, input_map, scaling_size, max_samples=12):
    keypoints_all = []

    count = 0
    cur_label = ''
```

4. Each image will give rise to a large number of descriptors. We will just use a small number of images because the centroids won't change

much after this:

```
            for item in input_map:
                if count >= max_samples:
                    if cur_class != item['object_class']:
                        count = 0
                else:
                    continue

            count += 1
```

5. The print progress is as follows:

```
            if count == max_samples:
                print "Built centroids for", item['object_class']
```

6. Extract the current label:

```
            cur_class = item['object_class']
```

7. Read the image and resize it:

```
            img = cv2.imread(item['image_path'])
            img = resize_image(img, scaling_size)
```

8. Set the number of dimensions to 128 and extract the features:

```
            num_dims = 128
            feature_vectors = self.extract_image_features(img)
            keypoints_all.extend(feature_vectors)
```

9. Use vector quantization to quantize the feature points. **Vector quantization** is the *N*-dimensional version of "rounding off". You can learn more about it at **http://www.data-compression.com/vq.shtml**.

```
            kmeans, centroids = BagOfWords().cluster(keypoints_all)
            return kmeans, centroids
```

10. Define the class to handle bag of words model and vector quantization:

```
    class BagOfWords(object):
        def __init__(self, num_clusters=32):
            self.num_dims = 128
            self.num_clusters = num_clusters
            self.num_retries = 10
```

11. Define a method to quantize the datapoints. We will use **k-means clustering** to achieve this:

```
    def cluster(self, datapoints):
        kmeans = KMeans(self.num_clusters,
            n_init=max(self.num_retries, 1),
            max_iter=10, tol=1.0)
```

12. Extract the centroids, as follows:

```
        res = kmeans.fit(datapoints)
        centroids = res.cluster_centers_
        return kmeans, centroids
```

13. Define a method to normalize the data:

```
def normalize(self, input_data):
    sum_input = np.sum(input_data)

    if sum_input > 0:
        return input_data / sum_input
    else:
        return input_data
```

14. Define a method to get the feature vector:

```
def construct_feature(self, img, kmeans, centroids):
    keypoints = StarFeatureDetector().detect(img)
    keypoints, feature_vectors = compute_sift_features(img, keypoints)
    labels = kmeans.predict(feature_vectors)
    feature_vector = np.zeros(self.num_clusters)
```

15. Build a histogram and normalize it:

```
        for i, item in enumerate(feature_vectors):
            feature_vector[labels[i]] += 1

            feature_vector_img = np.reshape(feature_vector,
((1, feature_vector.shape[0])))
            return self.normalize(feature_vector_img)
```

16. Define a method the extract the SIFT features:

```
# Extract SIFT features
def compute_sift_features(img, keypoints):
    if img is None:
        raise TypeError('Invalid input image')

    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    keypoints, descriptors = cv2.xfeatures2d.SIFT_create().compute(img_gray, keypoints)
    return keypoints, descriptors
```

As mentioned earlier, please refer to build_features.py for the complete code. You should run the code in the following way:

```
$ python build_features.py --data-folder /path/to/training_images/ --codebook-file codebook
```

This will generate two files called `codebook.pkl` and `feature_map.pkl`. We will use these files in the next recipe.