10
Computer Vision

Image analysis and computer vision have always been important in industrial and scientific applications. With the popularization of cell phones with powerful cameras and Internet connections, images now are increasingly generated by consumers. Therefore, there are opportunities to make use of computer vision to provide a better user experience in new contexts.

In this chapter, we will look at how to apply techniques you have learned in the rest of this book to this specific type of data. In particular, we will learn how to use the mahotas computer vision package to extract features from images. These features can be used as input to the same classification methods we studied in other chapters. We will apply these techniques to publicly available datasets of photographs. We will also see how the same features can be used on another problem, that is, the problem of finding similar looking images.

Finally, at the end of this chapter, we will learn about using local features. These are relatively new methods (the first of these methods to achieve state-of-the-art performance, the **scale-invariant feature transform** (**SIFT**), was introduced in 1999) and achieve very good results in many tasks.

Introducing image processing

From the point of view of the computer, an image is a large rectangular array of pixel values. Our goal is to process this image and to arrive at a decision for our application.

The first step will be to load the image from disk, where it is typically stored in an image-specific format such as PNG or JPEG, the former being a lossless compression format, and the latter a lossy compression one that is optimized for visual assessment of photographs. Then, we may wish to perform preprocessing on the images (for example, normalizing them for illumination variations).

www.it-ebooks.info

We will have a classification problem as a driver for this chapter. We want to be able to learn a support vector machine (or other) classifier that can be trained from images. Therefore, we will use an intermediate representation, extracting numeric features from the images before applying machine learning.

Loading and displaying images

In order to manipulate images, we will use a package called mahotas. You can obtain mahotas from `https://pypi.python.org/pypi/mahotas` and read its manual at `http://mahotas.readthedocs.org`. Mahotas is an open source package (MIT license, so it can be used in any project) that was developed by one of the authors of this book. Fortunately, it is based on NumPy. The NumPy knowledge you have acquired so far can be used for image processing. There are other image packages, such as scikit-image (skimage), the ndimage (n-dimensional image) module in SciPy, and the Python bindings for OpenCV. All of these work natively with NumPy arrays, so you can even mix and match functionality from different packages to build a combined pipeline.

We start by importing mahotas, with the `mh` abbreviation, which we will use throughout this chapter, as follows:

```
>>> import mahotas as mh
```

Now, we can load an image file using `imread` as follows:

```
>>> image = mh.imread('scene00.jpg')
```

The `scene00.jpg` file (this file is contained in the dataset available on this book's companion code repository) is a color image of height `h` and width `w`; the image will be an array of shape `(h, w, 3)`. The first dimension is the height, the second is the width, and the third is red/green/blue. Other systems put the width in the first dimension, but this is the convention that is used by all NumPy-based packages. The type of the array will typically be `np.uint8` (an unsigned 8-bit integer). These are the images that your camera takes or that your monitor can fully display.

Some specialized equipment, used in scientific and technical applications, can take images with higher bit resolution (that is, with more sensitivity to small variations in brightness). Twelve or sixteen bits are common in this type of equipment. Mahotas can deal with all these types, including floating point images. In many computations, even if the original data is composed of unsigned integers, it is advantageous to convert to floating point numbers in order to simplify handling of rounding and overflow issues.

www.it-ebooks.info

Mahotas can use a variety of different input/output backends. Unfortunately, none of them can load all image formats that exist (there are hundreds, with several variations of each). However, loading PNG and JPEG images is supported by all of them. We will focus on these common formats and refer you to the mahotas documentation on how to read uncommon formats.

We can display the image on screen using matplotlib, the plotting library we have already used several times, as follows:

```
>>> from matplotlib import pyplot as plt
>>> plt.imshow(image)
>>> plt.show()
```

As shown in the following, this code shows the image using the convention that the first dimension is the height and the second the width. It correctly handles color images as well. When using Python for numerical computation, we benefit from the whole ecosystem working well together: mahotas works with NumPy arrays, which can be displayed with matplotlib; later we will compute features from images to use with scikit-learn.

www.it-ebooks.info

Thresholding

Thresholding is a very simple operation: we transform all pixel values above a certain threshold to `1` and all those below it to `0` (or by using Booleans, transform it to `True` and `False`). The important question in thresholding is to select a good value to use as the threshold limit. Mahotas implements a few methods for choosing a threshold value from the image. One is called **Otsu**, after its inventor. The first necessary step is to convert the image to grayscale, with `rgb2gray` in the `mahotas.colors` submodule.

Instead of `rgb2gray`, we could also have just the mean value of the red, green, and blue channels, by callings `image.mean(2)`. The result, however, would not be the same, as `rgb2gray` uses different weights for the different colors to give a subjectively more pleasing result. Our eyes are not equally sensitive to the three basic colors.

```
>>> image = mh.colors.rgb2grey(image, dtype=np.uint8)
>>> plt.imshow(image) # Display the image
```

By default, matplotlib will display this single-channel image as a false color image, using red for high values and blue for low values. For natural images, a grayscale is more appropriate. You can select it with:

```
>>> plt.gray()
```

Now the image is shown in gray scale. Note that only the way in which the pixel values are interpreted and shown has changed and the image data is untouched. We can continue our processing by computing the threshold value:

```
>>> thresh = mh.thresholding.otsu(image)
>>> print('Otsu threshold is {}.'.format(thresh))
Otsu threshold is 138.
>>> plt.imshow(image > thresh)
```


www.it-ebooks.info

When applied to the previous screenshot, this method finds the threshold to be 138, which separates the ground from the sky above, as shown in the following image:

Gaussian blurring

Blurring your image may seem odd, but it often serves to reduce noise, which helps with further processing. With mahotas, it is just a function call:

```
>>> im16 = mh.gaussian_filter(image, 16)
```


www.it-ebooks.info

Notice that we did not convert the grayscale image to unsigned integers: we just made use of the floating point result as it is. The second argument to the `gaussian_filter` function is the size of the filter (the standard deviation of the filter). Larger values result in more blurring, as shown in the following screenshot:

www.it-ebooks.info

We can use the screenshot on the left and threshold with Otsu (using the same previous code). Now, the boundaries are smoother, without the jagged edges, as shown in the following screenshot:

Putting the center in focus

The final example shows how to mix NumPy operators with a tiny bit of filtering to get an interesting result. We start with the Lena image and split it into the color channels:

```
>>> im = mh.demos.load('lena')
```


www.it-ebooks.info

This is an image of a young woman that has been often for image processing demos. It is shown in the following screenshot:

To split the red, green, and blue channels, we use the following code:

```
>>> r,g,b = im.transpose(2,0,1)
```

Now, we filter the three channels separately and build a composite image out of it with `mh.as_rgb`. This function takes three two-dimensional arrays, performs contrast stretching to make each be an 8-bit integer array, and then stacks them, returning a color RGB image:

```
>>> r12 = mh.gaussian_filter(r, 12.)
>>> g12 = mh.gaussian_filter(g, 12.)
>>> b12 = mh.gaussian_filter(b, 12.)
>>> im12 = mh.as_rgb(r12, g12, b12)
```

Now, we blend the two images from the center away to the edges. First, we need to build a weights array `W`, which will contain at each pixel a normalized value, which is its distance to the center:

```
>>> h, w = r.shape # height and width
>>> Y, X = np.mgrid[:h,:w]
```


www.it-ebooks.info

We used the `np.mgrid` object, which returns arrays of size `(h, w)`, with values corresponding to the *y* and *x* coordinates, respectively. The next steps are as follows:

```
>>> Y = Y - h/2. # center at h/2
>>> Y = Y / Y.max() # normalize to -1 .. +1

>>> X = X - w/2.
>>> X = X / X.max()
```

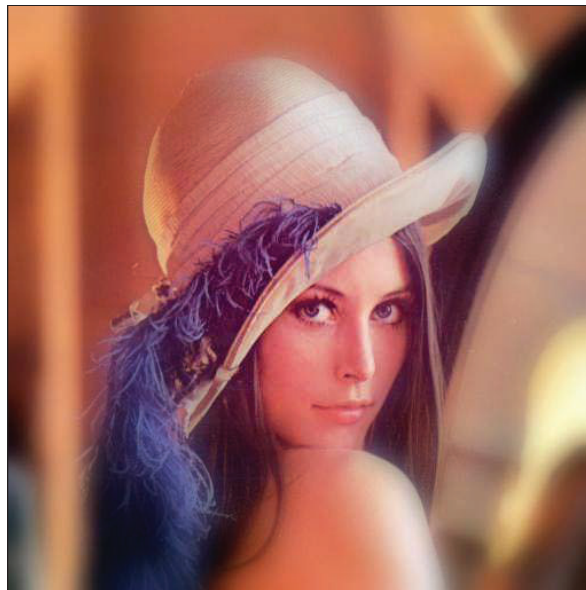
We now use a Gaussian function to give the center region a high value:

```
>>> C = np.exp(-2.*(X**2+ Y**2))

>>> # Normalize again to 0..1
>>> C = C - C.min()
>>> C = C / C.ptp()
>>> C = C[:,:,None] # This adds a dummy third dimension to C
```

Notice how all of these manipulations are performed using NumPy arrays and not some mahotas-specific methodology. Finally, we can combine the two images to have the center be in sharp focus and the edges softer:

```
>>> ringed = mh.stretch(im*C + (1-C)*im12)
```



www.it-ebooks.info

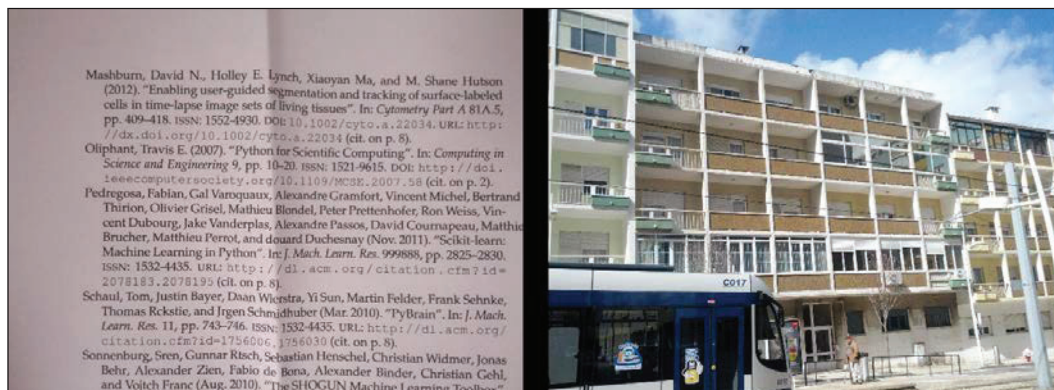
Basic image classification

We will start with a small dataset that was collected especially for this book. It has three classes: buildings, natural scenes (landscapes), and pictures of texts. There are 30 images in each category, and they were all taken using a cell phone camera with minimal composition. The images are similar to those that would be uploaded to a modern website by users with no photography training. This dataset is available from this book's website or the GitHub code repository. Later in this chapter, we will look at a harder dataset with more images and more categories.

When classifying images, we start with a large rectangular array of numbers (pixel values). Nowadays, millions of pixels are common. We could try to feed all these numbers as features into the learning algorithm. This is not a very good idea. This is because the relationship of each pixel (or even each small group of pixels) to the final result is very indirect. Also, having millions of pixels, but only as a small number of example images, results in a very hard statistical learning problem. This is an extreme form of the P greater than N type of problem we discussed in *Chapter 7, Regression*. Instead, a good approach is to compute features from the image and use those features for classification.

Having said that, I will point out that, in fact, there are a few methods that do work directly from the pixel values. They have feature computation submodules inside them. They may even attempt to learn good features automatically. These methods are the topic of current research. They typically work best with very large datasets (millions of images).

We previously used an example of the scene class. The following are examples of the text and building classes:



Mashburn, David N., Holley E. Lynch, Xiaoyan Ma, and M. Shane Hutson (2012). "Enabling user-guided segmentation and tracking of surface-labeled cells in time-lapse image sets of living tissues". In: *Cytometry Part A* 81A.5, pp. 409–418. ISSN: 1552-4930. DOI: 10.1002/cyto.a.22034. URL: http://dx.doi.org/10.1002/cyto.a.22034 (cit. on p. 8).

Oliphant, Travis E. (2007). "Python for Scientific Computing". In: *Computing in Science and Engineering* 9, pp. 10–20. ISSN: 1521-9615. DOI: http://doi.ieeecomputersociety.org/10.1109/MCSE.2007.58 (cit. on p. 2).

Pedregosa, Fabian, Gal Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthie Brucher, Matthieu Perrot, and douard Duchesnay (Nov. 2011). "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 999888, pp. 2825–2830. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2078183.2078195 (cit. on p. 8).

Schaul, Tom, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rckstie, and Jrgen Schmidhuber (Mar. 2010). "PyBrain". In: *J. Mach. Learn. Res.* 11, pp. 743–746. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1756006.1756030 (cit. on p. 8).

Sonnenburg, Sren, Gunnar Rtsch, Sebastian Henschel, Christian Widmer, Jonas Behr, Alexander Zien, Fabio de Bona, Alexander Binder, Christian Gehl, and Vojtch Franc (Aug. 2010). "The SHOGUN Machine Learning Toolbox".

www.it-ebooks.info

Computing features from images

With mahotas, it is very easy to compute features from images. There is a submodule named `mahotas.features`, where feature computation functions are available.

A commonly used set of texture features is the Haralick. As with many methods in image processing, the name is due to its inventor. These features are texture-based: they distinguish between images that are smooth from those that are patterned, and between different patterns. With mahotas, it is very easy to compute them as follows:

```
>>> haralick_features = mh.features.haralick(image)
>>> haralick_features_mean = np.mean(haralick_features, axis=0)
>>> haralick_features_all = np.ravel(haralick_features)
```

The `mh.features.haralick` function returns a 4x13 array. The first dimension refers to four possible directions in which to compute the features (vertical, horizontal, diagonal, and the anti-diagonal). If we are not interested in the direction specifically, we can use the average over all the directions (shown in the earlier code as `haralick_features_mean`). Otherwise, we can use all the features separately (using `haralick_features_all`). This decision should be informed by the properties of the dataset. In our case, we reason that the horizontal and vertical directions should be kept separately. Therefore, we will use `haralick_features_all`.

There are a few other feature sets implemented in mahotas. Linear binary patterns are another texture-based feature set, which is very robust against illumination changes. There are other types of features, including local features, which we will discuss later in this chapter.

With these features, we use a standard classification method such as logistic regression as follows:

```
>>> from glob import glob
>>> images = glob('SimpleImageDataset/*.jpg')
>>> features = []
>>> labels = []
>>> for im in images:
...   labels.append(im[:-len('00.jpg')])
...   im = mh.imread(im)
...   im = mh.colors.rgb2gray(im, dtype=np.uint8)
...   features.append(mh.features.haralick(im).ravel())


>>> features = np.array(features)
>>> labels = np.array(labels)
```


www.it-ebooks.info

The three classes have very different textures. Buildings have sharp edges and big blocks where the color is similar (the pixel values are rarely exactly the same, but the variation is slight). Text is made of many sharp dark-light transitions, with small black areas in a sea of white. Natural scenes have smoother variations with fractal-like transitions. Therefore, a classifier based on texture is expected to do well.

As a classifier, we are going to use a logistic regression classifier with preprocessing of the features as follows:

```
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.linear_model import LogisticRegression
>>> clf = Pipeline([('preproc', StandardScaler()),
                    ('classifier', LogisticRegression())])
```

Since our dataset is small, we can use leave-one-out regression as follows:

```
>>> from sklearn import cross_validation
>>> cv = cross_validation.LeaveOneOut(len(images))
>>> scores = cross_validation.cross_val_score(
...     clf, features, labels, cv=cv)
>>> print('Accuracy: {:.1%}'.format(scores.mean()))
Accuracy: 81.1%
```

Eighty-one percent is not bad for the three classes (random guessing would correspond to 33 percent). We can do better, however, by writing our own features.

Writing your own features

A feature is nothing magical. It is simply a number that we computed from an image. There are several feature sets already defined in the literature. These often have the added advantage that they have been designed and studied to be invariant to many unimportant factors. For example, linear binary patterns are completely invariant to multiplying all pixel values by a number or adding a constant to all these values. This makes this feature set robust against illumination changes of images.

However, it is also possible that your particular use case would benefit from a few specially designed features.

A simple type of feature that is not shipped with mahotas is a color histogram. Fortunately, this feature is easy to implement. A color histogram partitions the color space into a set of bins, and then counts how many pixels fall into each of the bins.

www.it-ebooks.info

The images are in RGB format, that is, each pixel has three values: R for red, G for green, and B for blue. Since each of these components is an 8-bit value, the total is 17 million different colors. We are going to reduce this number to only 64 colors by grouping colors into bins. We will write a function to encapsulate this algorithm as follows:

```
def chist(im):
```

To bin the colors, we first divide the image by 64, rounding down the pixel values as follows:

```
    im = im // 64
```

This makes the pixel values range from 0 to 3, which gives a total of 64 different colors.

Separate the red, green, and blue channels as follows:

```
    r,g,b = im.transpose((2,0,1))
    pixels = 1 * r + 4 * b + 16 * g
    hist = np.bincount(pixels.ravel(), minlength=64)
    hist = hist.astype(float)
```

Convert to log scale, as seen in the following code snippet. This is not strictly necessary, but makes for better features. We use `np.log1p`, which computes *log(h+1)*. This ensures that zero values are kept as zero values (mathematically, the logarithm of zero is not defined, and NumPy prints a warning if you attempt to compute it).

```
    hist = np.log1p(hist)
    return hist
```

We can adapt the previous processing code to use the function we wrote very easily:

```
>>> features = []
>>> for im in images:
...   image = mh.imread(im)
...   features.append(chist(im))
```

Using the same cross-validation code we used earlier, we obtain 90 percent accuracy. The best results, however, come from combining all the features, which we can implement as follows:

```
>>> features = []
>>> for im in images:
...   imcolor = mh.imread(im)
...   im = mh.colors.rgb2gray(imcolor, dtype=np.uint8)
```


www.it-ebooks.info

```
...     features.append(np.concatenate([
...             mh.features.haralick(im).ravel(),
...             chist(imcolor),
...         ]))
```

By using all of these features, we get 95.6 percent accuracy, as shown in the following code snippet:

```
>>> scores = cross_validation.cross_val_score(
...     clf, features, labels, cv=cv)
>>> print('Accuracy: {:.1%}'.format(scores.mean()))
Accuracy: 95.6%
```

This is a perfect illustration of the principle that good algorithms are the easy part. You can always use an implementation of state-of-the-art classification from scikit-learn. The real secret and added value often comes in feature design and engineering. This is where knowledge of your dataset is valuable.

Using features to find similar images

The basic concept of representing an image by a relatively small number of features can be used for more than just classification. For example, we can also use it to find similar images to a given query image (as we did before with text documents).

We will compute the same features as before, with one important difference: we will ignore the bordering area of the picture. The reason is that due to the amateur nature of the compositions, the edges of the picture often contain irrelevant elements. When the features are computed over the whole image, these elements are taken into account. By simply ignoring them, we get slightly better features. In the supervised example, it is not as important, as the learning algorithm will then learn which features are more informative and weigh them accordingly. When working in an unsupervised fashion, we need to be more careful to ensure that our features are capturing important elements of the data. This is implemented in the loop as follows:

```
>>> features = []
>>> for im in images:
...     imcolor = mh.imread(im)
...     # ignore everything in the 200 pixels closest to the borders
...     imcolor = imcolor[200:-200, 200:-200]
...     im = mh.colors.rgb2gray(imcolor, dtype=np.uint8)
...     features.append(np.concatenate([
```


www.it-ebooks.info

```
...             mh.features.haralick(im).ravel(),
...             chist(imcolor),
...         ]))
```

We now normalize the features and compute the distance matrix as follows:

```
>>> sc = StandardScaler()
>>> features = sc.fit_transform(features)
>>> from scipy.spatial import distance
>>> dists = distance.squareform(distance.pdist(features))
```

We will plot just a subset of the data (every 10th element) so that the query will be on top and the returned "nearest neighbor" at the bottom, as shown in the following:

```
>>> fig, axes = plt.subplots(2, 9)
>>> for ci,i in enumerate(range(0,90,10)):
...     left = images[i]
...     dists_left = dists[i]
...     right = dists_left.argsort()
...     # right[0] is same as left[i], so pick next closest
...     right = right[1]
...     right = images[right]
...     left = mh.imread(left)
...     right = mh.imread(right)
...     axes[0, ci].imshow(left)
...     axes[1, ci].imshow(right)
```

The result is shown in the following screenshot:

www.it-ebooks.info

It is clear that the system is not perfect, but can find images that are at least visually similar to the queries. In all but one case, the image found comes from the same class as the query.

Classifying a harder dataset

The previous dataset was an easy dataset for classification using texture features. In fact, many of the problems that are interesting from a business point of view are relatively easy. However, sometimes we may be faced with a tougher problem and need better and more modern techniques to get good results.

We will now test a public dataset, which has the same structure: several photographs split into a small number of classes. The classes are animals, cars, transportation, and natural scenes.

When compared to the three class problem we discussed previously, these classes are harder to tell apart. Natural scenes, buildings, and texts have very different textures. In this dataset, however, texture and color are not as clear marker, of the image class. The following is one example from the animal class:

And here is another example from the car class:

www.it-ebooks.info

Both objects are against natural backgrounds, and with large smooth areas inside the objects. This is a harder problem than the simple dataset, so we will need to use more advanced methods. The first improvement will be to use a slightly more powerful classifier. The logistic regression that scikit-learn provides is a penalized form of logistic regression, which contains an adjustable parameter, `C`. By default, `C = 1.0`, but this may not be optimal. We can use grid search to find a good value for this parameter as follows:

```
>>> from sklearn.grid_search import GridSearchCV
>>> C_range = 10.0 ** np.arange(-4, 3)
>>> grid = GridSearchCV(LogisticRegression(), param_grid={'C' : C_range})
>>> clf = Pipeline([('preproc', StandardScaler()),
...                 ('classifier', grid)])
```

The data is not organized in a random order inside the dataset: similar images are close together. Thus, we use a cross-validation schedule that considers the data shuffled so that each fold has a more representative training set, as shown in the following:

```
>>> cv = cross_validation.KFold(len(features), 5,
...                      shuffle=True, random_state=123)
>>> scores = cross_validation.cross_val_score(
...    clf, features, labels, cv=cv)
>>> print('Accuracy: {:.1%}'.format(scores.mean()))
Accuracy: 72.1%
```

This is not so bad for four classes, but we will now see if we can do better by using a different set of features. In fact, we will see that we need to combine these features with other methods to get the best possible results.

Local feature representations

A relatively recent development in the computer vision world has been the development of local-feature based methods. Local features are computed on a small region of the image, unlike the previous features we considered, which had been computed on the whole image. Mahotas supports computing a type of these features, **Speeded Up Robust Features** (**SURF**). There are several others, the most well-known being the original proposal of SIFT. These features are designed to be robust against rotational or illumination changes (that is, they only change their value slightly when illumination changes).

www.it-ebooks.info

When using these features, we have to decide where to compute them. There are three possibilities that are commonly used:

- Randomly
- In a grid
- Detecting interesting areas of the image (a technique known as keypoint detection or interest point detection)

All of these are valid and will, under the right circumstances, give good results. Mahotas supports all three. Using interest point detection works best if you have a reason to expect that your interest point will correspond to areas of importance in the image.

We will be using the interest point method. Computing the features with mahotas is easy: import the right submodule and call the `surf.surf` function as follows:

```
>>> from mahotas.features import surf
>>> image = mh.demos.load('lena')
>>> image = mh.colors.rgb2gray(im, dtype=np.uint8)
>>> descriptors = surf.surf(image, descriptor_only=True)
```

The `descriptors_only=True` flag means that we are only interested in the descriptors themselves, and not in their pixel location, size, or orientation. Alternatively, we could have used the dense sampling method, using the `surf.dense` function as follows:

```
>>> from mahotas.features import surf
>>> descriptors = surf.dense(image, spacing=16)
```

This returns the value of the descriptors computed on points that are at a distance of 16 pixels from each other. Since the position of the points is fixed, the metainformation on the interest points is not very interesting and is not returned by default. In either case, the result (descriptors) is an n-times-64 array, where *n* is the number of points sampled. The number of points depends on the size of your images, their content, and the parameters you pass to the functions. In this example, we are using the default settings, and we obtain a few hundred descriptors per image.

We cannot directly feed these descriptors to a support vector machine, logistic regressor, or similar classification system. In order to use the descriptors from the images, there are several solutions. We could just average them, but the results of doing so are not very good as they throw away all location specific information. In that case, we would have just another global feature set based on edge measurements.

www.it-ebooks.info

The solution we will use here is the **bag of words** model, which is a very recent idea. It was published in this form first in 2004. This is one of those obvious-in-hindsight ideas: it is very simple to implement and achieves very good results.

It may seem strange to speak of *words* when dealing with images. It may be easier to understand if you think that you have not written words, which are easy to distinguish from each other, but orally spoken audio. Now, each time a word is spoken, it will sound slightly different, and different speakers will have their own pronunciation. Thus, a word's waveform will not be identical every time it is spoken. However, by using clustering on these waveforms, we can hope to recover most of the structure so that all the instances of a given word are in the same cluster. Even if the process is not perfect (and it will not be), we can still talk of grouping the waveforms into words.

We perform the same operation with image data: we cluster together similar looking regions from all images and call these **visual words**.

> The number of words used does not usually have a big impact on the final performance of the algorithm. Naturally, if the number is extremely small (10 or 20, when you have a few thousand images), then the overall system will not perform well. Similarly, if you have too many words (many more than the number of images, for example), the system will also not perform well. However, in between these two extremes, there is often a very large plateau, where you can choose the number of words without a big impact on the result. As a rule of thumb, using a value such as 256, 512, or 1,024 if you have very many images should give you a good result.

We are going to start by computing the features as follows:

```
>>> alldescriptors = []
>>> for im in images:
...   im = mh.imread(im, as_grey=True)
...   im = im.astype(np.uint8)
...   alldescriptors.append(surf.dense(image, spacing=16))
>>> # get all descriptors into a single array
>>> concatenated = np.concatenate(alldescriptors)
>>> print('Number of descriptors: {}'.format(
...        len(concatenated)))
Number of descriptors: 2489031
```


www.it-ebooks.info

This results in over 2 million local descriptors. Now, we use k-means clustering to obtain the centroids. We could use all the descriptors, but we are going to use a smaller sample for extra speed, as shown in the following:

```
>>> # use only every 64th vector
>>> concatenated = concatenated[::64]
>>> from sklearn.cluster import KMeans
>>> k = 256
>>> km = KMeans(k)
>>> km.fit(concatenated)
```

After this is done (which will take a while), the `km` object contains information about the centroids. We now go back to the descriptors and build feature vectors as follows:

```
>>> sfeatures = []
>>> for d in alldescriptors:
...   c = km.predict(d)
...   sfeatures.append(
...       np.array([np.sum(c == ci) for ci in range(k)])
...   )
>>> # build single array and convert to float
>>> sfeatures = np.array(sfeatures, dtype=float)
```

The end result of this loop is that `sfeatures[fi, fj]` is the number of times that the image `fi` contains the element `fj`. The same could have been computed faster with the `np.histogram` function, but getting the arguments just right is a little tricky. We convert the result to floating point as we do not want integer arithmetic (with its rounding semantics).

The result is that each image is now represented by a single array of features, of the same size (the number of clusters, in our case 256). Therefore, we can use our standard classification methods as follows:

```
>>> scores = cross_validation.cross_val_score(
...    clf, sfeatures, labels, cv=cv)
>>> print('Accuracy: {:.1%}'.format(scores.mean()))
Accuracy: 62.6%
```

This is worse than before! Have we gained nothing?

www.it-ebooks.info

In fact, we have, as we can combine all features together to obtain 76.1 percent accuracy, as follows:

```
>>> combined = np.hstack([features, features])
>>> scores = cross_validation.cross_val_score(
...     clf, combined, labels, cv=cv)
>>> print('Accuracy: {:.1%}'.format(scores.mean()))
Accuracy: 76.1%
```

This is the best result we have, better than any single feature set. This is due to the fact that the local SURF features are different enough to add new information to the global image features we had before and improve the combined result.

Summary

We learned the classical feature-based approach to handling images in a machine learning context: by converting from a million pixels to a few numeric features, we are able to directly use a logistic regression classifier. All of the technologies that we learned in the other chapters suddenly become directly applicable to image problems. We saw one example in the use of image features to find similar images in a dataset.

We also learned how to use local features, in a bag of words model, for classification. This is a very modern approach to computer vision and achieves good results while being robust to many irrelevant aspects of the image, such as illumination, and even uneven illumination in the same image. We also used clustering as a useful intermediate step in classification rather than as an end in itself.

We focused on mahotas, which is one of the major computer vision libraries in Python. There are others that are equally well maintained. Skimage (scikit-image) is similar in spirit, but has a different set of features. OpenCV is a very good C++ library with a Python interface. All of these can work with NumPy arrays and you can mix and match functions from different libraries to build complex computer vision pipelines.

In the next chapter, you will learn a different form of machine learning: dimensionality reduction. As we saw in several earlier chapters, including when using images in this chapter, it is very easy to computationally generate many features. However, often we want to have a reduced number of features for speed and visualization, or to improve our results. In the next chapter, we will see how to achieve this.

www.it-ebooks.info

www.it-ebooks.info