Programming Computer Vision with Python

Jan Erik Solem

Chapter 6

Clustering Images

This chapter introduces several clustering methods and shows how to use them for clustering images for finding groups of similar images. Clustering can be used for recognition, for dividing data sets of images and for organization and navigation. We also look at using clustering for visualizing similarity between images.

6.1 K-means Clustering

K-means is a very simple clustering algorithm that tries to partition the input data in k clusters. K-means works by iteratively refining an initial estimate of *class centroids* as follows:

1. Initialize centroids $\boldsymbol{\mu}_i$, $i = 1 \dots k$, randomly or with some guess.
2. Assign each data point to the class c_i of its nearest centroid.
3. Update the centroids as the average of all data points assigned to that class.
4. Repeat 2 & 3 until convergence.

K-means tries to minimize the *total within-class variance*

$$V = \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in c_i} (\mathbf{x}_j - \boldsymbol{\mu}_i)^2 \ ,$$

where $\mathbf{x}_j$ are the data vectors. The algorithm above is a heuristic refinement algorithm that works fine for most cases but does not guarantee that the best solution is found. To avoid the effects of choosing a bad centroid initialization, the algorithm is often run several times with different initialization centroids. Then the solution with lowest variance V is selected.

The main drawback of this algorithm is that the number of clusters needs to be decided beforehand and an inappropriate choice will give poor clustering results. The benefits are that it is simple to implement, it is parallelizable and works well for a large range of problems without any need for tuning.

The SciPy clustering package

Although simple to implement, there is no need to. The SciPy vector quantization package scipy.cluster.vq comes with a k-means implementation. Here's how to use it.

Let's start with creating some sample 2D data to illustrate.

```
from scipy.cluster.vq import *

class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))
```

This generates two normally distributed classes in two dimensions. To try to cluster the points, run k-means with $k = 2$ like this.

```
centroids,variance = kmeans(features,2)
```

The variance is returned but we don't really need it since the SciPy implementation computes several runs (default is 20) and selects the one with smallest variance for us. Now you can check where each data point is assigned using the vector quantization function in the SciPy package.

```
code,distance = vq(features,centroids)
```

By checking the value of *code* we can see if there are any incorrect assignments. To visualize, we can plot the points and the final centroids.

```
figure()
ndx = where(code==0)[0]
plot(features[ndx,0],features[ndx,1],'*')
ndx = where(code==1)[0]
plot(features[ndx,0],features[ndx,1],'r.')
plot(centroids[:,0],centroids[:,1],'go')
axis('off')
show()
```

Here the function where() gives the indices for each class. This should give a plot like the one in Figure 6.1.

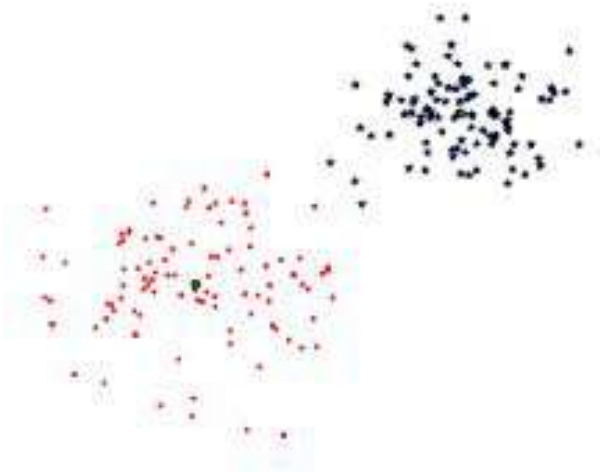


Figure 6.1: An example of k-means clustering of 2D points. Class centroids are marked as green rings and the predicted classes are blue stars and red dots respectively.

Clustering images

Let's try k-means on the font images described on page 28. The file *selectedfontimages.zip* contains 66 images from this font data set (these are selected for easy overview when illustrating the clusters). As descriptor vector for each image we will use the projection coefficients after projecting on the 40 first principal components computed earlier. Loading the model file using `pickle`, projecting the images on the principal components and clustering is then done like this.

```
import imtools
import pickle
from scipy.cluster.vq import *

# get list of images
imlist = imtools.get_imlist('selected_fontimages/')
imnbr = len(imlist)

# load model file
with open('a_pca_modes.pkl','rb') as f:
  immean = pickle.load(f)
  V = pickle.load(f)

# create matrix to store all flattened images
immatrix = array([array(Image.open(im)).flatten()
                        for im in imlist],'f')
```

```
# project on the 40 first PCs
immean = immean.flatten()
projected = array([dot(V[:40],immatrix[i]-immean) for i in range(imnbr)])

# k-means
projected = whiten(projected)
centroids,distortion = kmeans(projected,4)

code,distance = vq(projected,centroids)
```

Same as before, *code* contains the cluster assignment for each image. In this case we tried $k = 4$. We also chose to "whiten" the data using SciPy's `whiten()`, normalizing so that each feature has unit variance. Try to vary parameters like the number of principal components used and the value of k to see how the clustering results change. The clusters can be visualized like this:

```
# plot clusters
for k in range(4):
    ind = where(code==k)[0]
    figure()
    gray()
    for i in range(minimum(len(ind),40)):
        subplot(4,10,i+1)
        imshow(immatrix[ind[i]].reshape((25,25)))
        axis('off')
show()
```

Here we show each cluster in a separate figure window in a grid with maximum 40 images from the cluster shown. We use the PyLab function `subplot()` to define the grid. A sample cluster result can look like the one in Figure 6.2.

For more details on the k-means SciPy implementation and the `scipy.cluster.vq` package see the reference guide `http://docs.scipy.org/doc/scipy/reference/cluster.vq.html`.

Visualizing the images on principal components

To see how the clustering using just a few principal components as above can work, we can visualize the images on their coordinates in a pair of principal component directions. One way is to project on two components by changing the projection to

```
projected = array([dot(V[[0,2]],immatrix[i]-immean) for i in range(imnbr)])
```

to only get the relevant coordinates (in this case $V[[0, 2]]$ gives the first and third). Alternatively, project on all components and afterwards just pick out the columns you need.

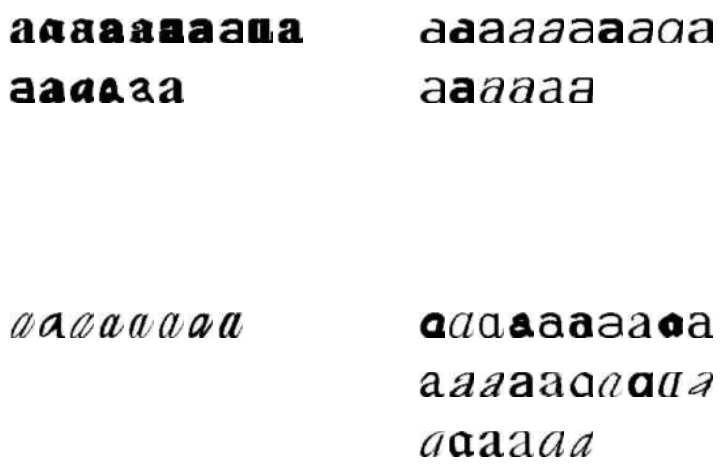


Figure 6.2: An example of k-means clustering with $k = 4$ of the font images using 40 principal components.

For the visualization we will use the `ImageDraw` module in PIL. Assuming that you have the projected images and image list as above, the following short script will generate a plot like the one in Figure 6.3

```
from PIL import Image, ImageDraw

# height and width
h,w = 1200,1200

# create a new image with a white background
img = Image.new('RGB',(w,h),(255,255,255))
draw = ImageDraw.Draw(img)

# draw axis
draw.line((0,h/2,w,h/2),fill=(255,0,0))
draw.line((w/2,0,w/2,h),fill=(255,0,0))

# scale coordinates to fit
scale = abs(projected).max(0)
scaled = floor(array([ (p / scale) * (w/2-20,h/2-20) +
                            (w/2,h/2) for p in projected]))

# paste thumbnail of each image
```

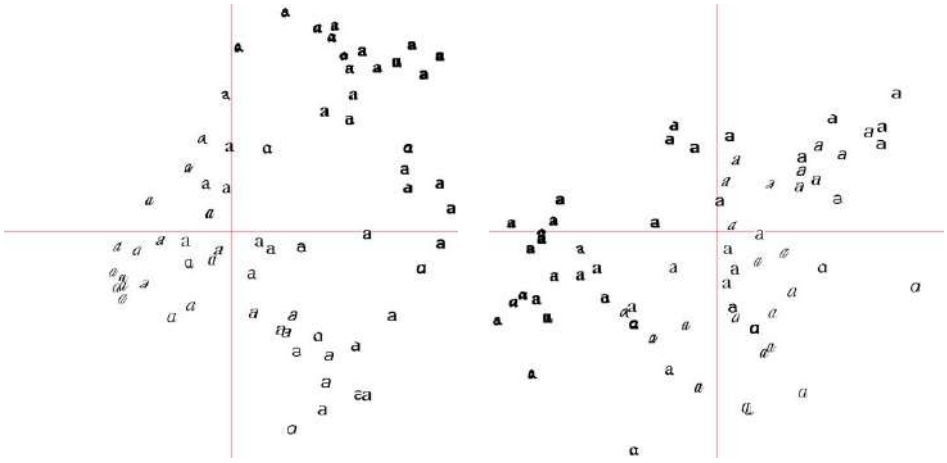


Figure 6.3: The projection of the font images on pairs of principal components. (left) the first and second principal components. (right) the second and third.

```
for i in range(imnbr):
    nodeim = Image.open(imlist[i])
    nodeim.thumbnail((25,25))
    ns = nodeim.size
    img.paste(nodeim,(scaled[i][0]-ns[0]//2,scaled[i][1]-
        ns[1]//2,scaled[i][0]+ns[0]//2+1,scaled[i][1]+ns[1]//2+1))

img.save('pca_font.jpg')
```

Here we used the integer or floor division operator // which returns an integer pixel position by removing any values after the decimal point.

Plots like these illustrate how the images are distributed in the 40 dimensions and can be very useful for choosing a good descriptor. Already in just these two-dimensional projections the closeness of similar font images is clearly visible.

Clustering pixels

Before closing this section we will take a look at an example of clustering individual pixels instead of entire images. Grouping image regions (and pixels) into "meaningful" components is called *image segmentation* and will be the topic of chapter 9. Naively applying k-means on the pixel values will not give anything meaningful except in very simple images. More sophisticated class models (than average pixel color) or spatial consistency is needed to produce useful results. For now, let's just apply k-means to the RGB values and worry about solving segmentation problems later (Section 9.2 has

the details).

The following code sample takes an image, reduces it to a lower resolution version with pixels as mean values of the original image regions (taken over a square grid of size *steps* $\times$ *steps*) and clustering the regions using k-means.

```
from scipy.cluster.vq import *
from scipy.misc import imresize

steps = 50 #image is divided in steps*steps region
im = array(Image.open('empire.jpg'))

dx = im.shape[0] / steps
dy = im.shape[1] / steps

# compute color features for each region
features = []
for x in range(steps):
  for y in range(steps):
    R = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,0])
    G = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,1])
    B = mean(im[x*dx:(x+1)*dx,y*dy:(y+1)*dy,2])
    features.append([R,G,B])
features = array(features,'f') # make into array

# cluster
centroids,variance = kmeans(features,3)
code,distance = vq(features,centroids)

# create image with cluster labels
codeim = code.reshape(steps,steps)
codeim = imresize(codeim,im.shape[:2],interp='nearest')

figure()
imshow(codeim)
show()
```

The input to k-means is an array with *steps***steps* rows, each containing the R, G and B mean values. To visualize the result we use SciPy's `imresize()` function to show the *steps***steps* image at the original image coordinates. The parameter *interp* specifies what type of interpolation to use, here we use nearest neighbor so we don't introduce new pixel values at the transitions between classes.

Figure 6.4 shows results using 50×50 and 100×100 regions for two relatively simple example images. Note the ordering of the k-means labels (in this case the colors in the result images) is arbitrary. As you can see, the result is noisy despite down-sampling to only use a few regions. There is no spatial consistency and it is hard to separate regions, like the boy and the grass in the lower example. Spatial

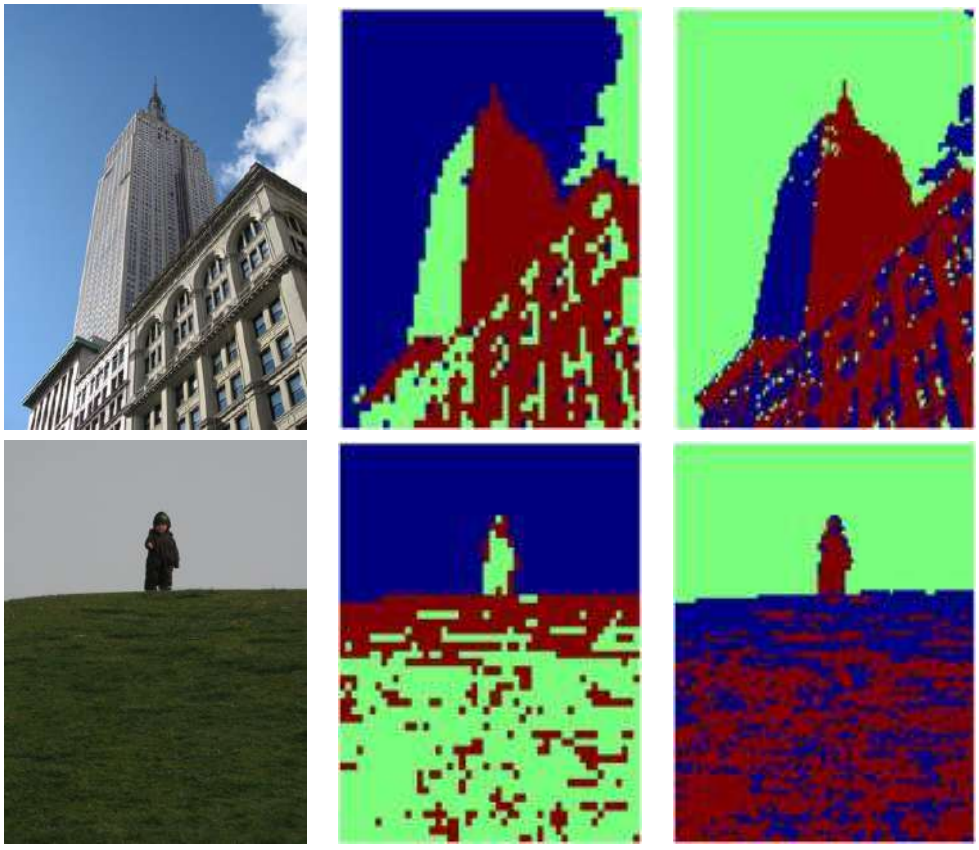


Figure 6.4: Clustering of pixels based on their color value using k-means. (left) original image. (center) cluster result with $k = 3$ and 50×50 resolution. (right) cluster result with $k = 3$ and 100×100 resolution.

consistency and better separation will be dealt with later, together with other image segmentation algorithms. Now let's move on to the next basic clustering algorithm.

6.2 Hierarchical Clustering

Hierarchical clustering (or *agglomerative clustering*) is another simple but powerful clustering algorithm. The idea is to build a similarity tree based on pairwise distances. The algorithm starts with grouping the two closest objects (based on the distance between feature vectors) and creates an "average" node in a tree with the two objects as children. Then the next closest pair is found among the remaining objects but also including any average nodes, and so on. At each node the distance between the two children is also stored. Clusters can then be extracted by traversing this tree and stopping at nodes with distance smaller some threshold that then determines the cluster size.

Hierarchical clustering has several benefits. For example, the tree structure can be used to visualize relationships and show how clusters are related. A good feature vector will give a nice separation in the tree. Another benefit is that the tree can be reused with different cluster thresholds without having to recompute the tree. The drawback is that one needs to choose a threshold if the actual clusters are needed.

Let's see what this looks like in code[1]. Create a file *hcluster.py* and add the following code (inspired by the hierarchical clustering example in [31]).

```
from itertools import combinations

class ClusterNode(object):
  def __init__(self,vec,left,right,distance=0.0,count=1):
    self.left = left
    self.right = right
    self.vec = vec
    self.distance = distance
    self.count = count # only used for weighted average

  def extract_clusters(self,dist):
    """ Extract list of sub-tree clusters from
      hcluster tree with distance<dist. """
    if self.distance < dist:
      return [self]
    return self.left.extract_clusters(dist) + self.right.extract_clusters(dist)
```

[1]There is also a version of hierarchical clustering in the `SciPy` clustering package that you can look at if you like. We will not use that here as parts of the implementation below (creating trees, visualizing dendrograms) is interesting and will be useful later.

```
  def get_cluster_elements(self):
    """ Return ids for elements in a cluster sub-tree. """
    return self.left.get_cluster_elements() + self.right.get_cluster_elements()

  def get_height(self):
    """ Return the height of a node,
      height is sum of each branch. """
    return self.left.get_height() + self.right.get_height()

  def get_depth(self):
    """ Return the depth of a node, depth is
      max of each child plus own distance. """
    return max(self.left.get_depth(), self.right.get_depth()) + self.distance


class ClusterLeafNode(object):
  def __init__(self,vec,id):
    self.vec = vec
    self.id = id

  def extract_clusters(self,dist):
    return [self]

  def get_cluster_elements(self):
    return [self.id]

  def get_height(self):
    return 1

  def get_depth(self):
    return 0


def L2dist(v1,v2):
  return sqrt(sum((v1-v2)**2))


def L1dist(v1,v2):
  return sum(abs(v1-v2))


def hcluster(features,distfcn=L2dist):
  """ Cluster the rows of features using
    hierarchical clustering. """

  # cache of distance calculations
  distances = {}
```

```
# initialize with each row as a cluster
node = [ClusterLeafNode(array(f),id=i) for i,f in enumerate(features)]

while len(node)>1:
  closest = float('Inf')

  # loop through every pair looking for the smallest distance
  for ni,nj in combinations(node,2):
    if (ni,nj) not in distances:
      distances[ni,nj] = distfcn(ni.vec,nj.vec)

    d = distances[ni,nj]
    if d<closest:
      closest = d
      lowestpair = (ni,nj)
  ni,nj = lowestpair

  # average the two clusters
  new_vec = (ni.vec + nj.vec) / 2.0

  # create new node
  new_node = ClusterNode(new_vec,left=ni,right=nj,distance=closest)
  node.remove(ni)
  node.remove(nj)
  node.append(new_node)

return node[0]
```

We created two classes for tree nodes, `ClusterNode` and `ClusterLeafNode`, to be used to create the cluster tree. The function `hcluster()` builds the tree. First a list of leaf nodes is created, then the closest pairs are iteratively grouped together based on the distance measure chosen. Returning the final node will give you the root of the tree. Running `hcluster()` on a matrix with feature vectors as rows will create and return the cluster tree.

The choice of distance measure depends on the actual feature vectors, here we used the Euclidean (L_2) distance (a function for L_1 distance is also provided) but you can create any function and use that as parameter to *hcluster()*. We also used the average feature vector of all nodes in a sub-tree as a new feature vector to represent the sub-tree and treat each sub-tree as objects. There are other choices for deciding which two nodes to merge next, such as using *single linking* (use the minimum distance between objects in two sub-trees) and *complete linking* (use the maximum distance between objects in two sub-trees). The choice of linking will affect the type of clusters produced.

To extract the clusters from the tree you need to traverse the tree from the top until

a node with distance value smaller than some threshold is found. This is easiest done recursively. The `ClusterNode` method `extract_clusters()` handles this by returning a list with the node itself if below the distance threshold, otherwise call the child nodes (leaf nodes always returns themselves). Calling this function will return a list of sub-trees containing the clusters. To get the leaf nodes for each cluster sub-tree that contain the object ids, traverse each sub-tree and return a list of leaves using the method `get_cluster_elements()`.

Let's try this on a simple example to see it all in action. First create some 2D data points (same as for k-means above).

```
class1 = 1.5 * randn(100,2)
class2 = randn(100,2) + array([5,5])
features = vstack((class1,class2))
```

Cluster the points and extract the clusters from the list using some threshold (here we used 5) and print the clusters in the console.

```
import hcluster

tree = hcluster.hcluster(features)

clusters = tree.extract_clusters(5)

print len(clusters)
for c in clusters:
  print c.get_cluster_elements()
```

This should give a printout similar to this:

```
number of clusters 2
[184, 187, 196, 137, 174, 102, 147, 145, 185, 109, 166, 152, 173, 180, 128, 163, 141, 178, 151, 158, 108,
182, 112, 199, 100, 119, 132, 195, 105, 159, 140, 171, 191, 164, 130, 149, 150, 157, 176, 135, 123, 131,
118, 170, 143, 125, 127, 139, 179, 126, 160, 162, 114, 122, 103, 146, 115, 120, 142, 111, 154, 116, 129,
136, 144, 167, 106, 107, 198, 186, 153, 156, 134, 101, 110, 133, 189, 168, 183, 148, 165, 172, 188, 138,
192, 104, 124, 113, 194, 190, 161, 175, 121, 197, 177, 193, 169, 117, 155]

[56, 4, 47, 18, 51, 95, 29, 91, 23, 80, 83, 3, 54, 68, 69, 5, 21, 1, 44, 57, 17, 90, 30, 22, 63, 41, 7, 14, 59,
96, 20, 26, 71, 88, 86, 40, 27, 38, 50, 55, 67, 8, 28, 79, 64, 66, 94, 33, 53, 70, 31, 81, 9, 75, 15, 32, 89, 6,
11, 48, 58, 2, 39, 61, 45, 65, 82, 93, 97, 52, 62, 16, 43, 84, 24, 19, 74, 36, 37, 60, 87, 92, 181, 99, 10, 49,
12, 76, 98, 46, 72, 34, 35, 13, 73, 78, 25, 42, 77, 85]
```

Ideally you should get two clusters but depending on the actual data you might get three or even more. In this simple example of clustering 2D points, one cluster should contain values lower than 100 and the other values 100 and above.

Clustering images

Let's look at an example of clustering images based on their color content. The file *sunsets.zip* contains 100 images downloaded from Flickr using the tag "sunset" or "sunsets". For this example we will use a color histogram of each image as feature vector. This is a bit crude and simple but good enough for illustrating what hierarchical clustering does. Try running the following code in a folder containing the sunset images.

```
import os
import hcluster

# create a list of images
path = 'flickr-sunsets/'
imlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.jpg')]

# extract feature vector (8 bins per color channel)
features = zeros([len(imlist), 512])
for i,f in enumerate(imlist):
  im = array(Image.open(f))

  # multi-dimensional histogram
  h,edges = histogramdd(im.reshape(-1,3),8,normed=True,range=[(0,255),(0,255),(0,255)])
  features[i] = h.flatten()

tree = hcluster.hcluster(features)
```

Here we take the R,G and B color channels as vectors and feed them into NumPy's `histogramdd()` which computes multi-dimensional histograms (in this case three dimensions). We chose 8 bins in each color dimension ($8 * 8 * 8$) which after flattening gives 512 bins in the feature vector. We use the "normed=True" option to normalize the histograms in case the images are of different size and set the range to $0 \ldots 255$ for each color channel. The use of `reshape()` with one dimension set to -1 will automatically determine the correct size and thereby create an input array to the histogram computation consisting of the RGB color values as rows.

To visualize the cluster tree, we can draw a dendrogram. A *dendrogram* is a diagram that shows the tree layout. This often gives useful information on how good a given descriptor vector is and what is considered similar in a particular case. Add the following code to *hcluster.py*.

```
from PIL import Image,ImageDraw

def draw_dendrogram(node,imlist,filename='clusters.jpg'):
  """ Draw a cluster dendrogram and save to a file. """
```

```
  # height and width
  rows = node.get_height()*20
  cols = 1200

  # scale factor for distances to fit image width
  s = float(cols-150)/node.get_depth()

  # create image and draw object
  im = Image.new('RGB',(cols,rows),(255,255,255))
  draw = ImageDraw.Draw(im)

  # initial line for start of tree
  draw.line((0,rows/2,20,rows/2),fill=(0,0,0))

  # draw the nodes recursively
  node.draw(draw,20,(rows/2),s,imlist,im)
  im.save(filename)
  im.show()
```

Here the dendrogram drawing uses a `draw()` method for each node. Add this method to the `ClusterNode` class:

```
def draw(self,draw,x,y,s,imlist,im):
  """ Draw nodes recursively with image
    thumbnails for leaf nodes. """

  h1 = int(self.left.get_height()*20 / 2)
  h2 = int(self.right.get_height()*20 /2)
  top = y-(h1+h2)
  bottom = y+(h1+h2)

  # vertical line to children
  draw.line((x,top+h1,x,bottom-h2),fill=(0,0,0))

  # horizontal lines
  ll = self.distance*s
  draw.line((x,top+h1,x+ll,top+h1),fill=(0,0,0))
  draw.line((x,bottom-h2,x+ll,bottom-h2),fill=(0,0,0))

  # draw left and right child nodes recursively
  self.left.draw(draw,x+ll,top+h1,s,imlist,im)
  self.right.draw(draw,x+ll,bottom-h2,s,imlist,im)
```

The leaf nodes have their own special method to draw thumbnails of the actual images. Add this to the `ClusterLeafNode` class:

```
def draw(self,draw,x,y,s,imlist,im):
  nodeim = Image.open(imlist[self.id])
```

```
  nodeim.thumbnail([20,20])
  ns = nodeim.size
  im.paste(nodeim,[int(x),int(y-ns[1]//2),int(x+ns[0]),int(y+ns[1]-ns[1]//2)])
```

The height of a dendrogram (and the sub parts) is determined by the distance values. These need to be scaled to fit inside the chosen image resolution. The nodes are drawn recursively with the coordinates passed down to the level below. Leaf nodes are drawn with small thumbnail images of 20×20 pixels. Two helper methods are used to get the height and width of the tree, `get_height()` and `get_depth()`

The dendrogram is drawn like this:

```
hcluster.draw_dendrogram(tree,imlist,filename='sunset.pdf')
```

The cluster dendrogram for the sunset images is shown in Figure 6.5. As can be seen, images with similar color are close in the tree. Three example clusters are shown in Figure 6.6. The clusters are in this example extracted as follows.

```
# visualize clusters with some (arbitrary) threshold
clusters = tree.extract_clusters(0.23*tree.distance)

# plot images for clusters with more than 3 elements
for c in clusters:
  elements = c.get_cluster_elements()
  nbr_elements = len(elements)
  if nbr_elements>3:
    figure()
    for p in range(minimum(nbr_elements,20)):
      subplot(4,5,p+1)
      im = array(Image.open(imlist[elements[p]]))
      imshow(im)
      axis('off')
show()
```

As a final example we can create a dendrogram for the font images

```
tree = hcluster.hcluster(projected)
hcluster.draw_dendrogram(tree,imlist,filename='fonts.jpg')
```

where *projected* and *imlist* refer to the variables used in the k-means example in Section 6.1. The resulting font images dendrogram is shown in Figure 6.7.

6.3 Spectral Clustering

An interesting type of clustering algorithms are *spectral clustering* methods which have a different approach compared to k-means and hierarchical clustering.

Figure 6.5: An example of hierarchical clustering of 100 images of sunsets using a 512 bin histogram in RGB coordinates as feature vector. Images close together in the tree have similar color distribution.

Figure 6.6: Example clusters from the 100 images of sunsets obtained with hierarchical clustering using a threshold set to 23% of the maximum node distance in the tree.

A *similarity matrix* (or *affinity matrix*, or sometimes *distance matrix*) for n elements (for example images) is an $n \times n$ matrix with pair-wise similarity scores. Spectral clustering gets its name from the use of the spectrum of a matrix constructed from a similarity matrix. The eigenvectors of this matrix are used for dimensionality reduction and then clustering.

One of the benefits of spectral clustering methods is that the only input needed is this matrix and it can be constructed from any measure of similarity you can think of. Methods like k-means and hierarchical clustering compute mean of feature vectors and this restricts the features (or descriptors) to vectors (in order to be able to compute the mean). With spectral methods, there is no need to have feature vectors of any kind, just a notion of "distance" or "similarity".

Here's how it works. Given a $n \times n$ similarity matrix S with similarity scores s_{ij}, we can create a matrix, called the *Laplacian matrix*[2],

$$L = I - D^{-1/2} S D^{-1/2} \ ,$$

where I is the identity matrix and D is the diagonal matrix containing the row sums

[2]Sometimes $L = D^{-1/2} S D^{-1/2}$ is used as the Laplacian matrix instead but the choice doesn't really matter since it only changes the eigenvalues, not the eigenvectors.

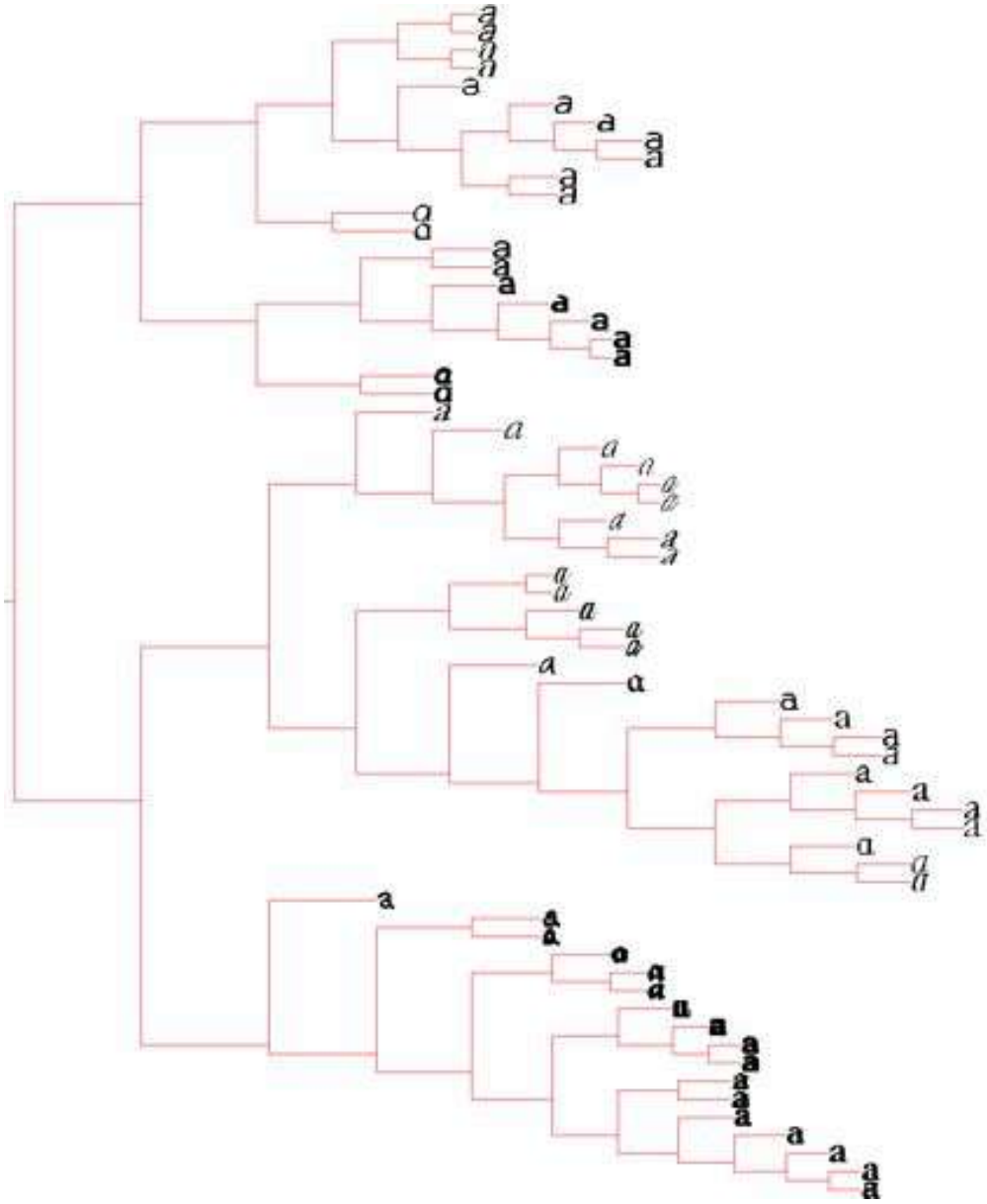


Figure 6.7: An example of hierarchical clustering of 66 selected font images using 40 principal components as feature vector.

of S, $D = \text{diag}(d_i)$, $d_i = \sum_j s_{ij}$. The matrix $D^{-1/2}$ used in the construction of the Laplacian matrix is then

$$D^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{d_1}} & & & \\ & \frac{1}{\sqrt{d_2}} & & \\ & & \ddots & \\ & & & \frac{1}{\sqrt{d_n}} \end{bmatrix}.$$

In order to make the presentation clearer, let's use low values of s_{ij} for similar elements and require $s_{ij} \geq 0$ (the term distance matrix is perhaps more fitting in this case).

The clusters are found by computing the eigenvectors of L and using the k eigenvectors corresponding to the k largest eigenvalues to construct a set of feature vectors (remember that we may not have had any to start with!). Create a matrix with the k eigenvectors as columns, the rows will then be treated as new feature vectors (of length k). These new feature vectors can then be clustered using for example k-means to produce the final clusters. In essence, what the algorithm does is to transform the original data into new feature vectors that can be more easily clustered (and in some cases using cluster algorithms that could not be used in the first place).

Enough about the theory, let's see what it looks like in code when applied to a real example. Again, we take the font images used in the k-means example above (and introduced on page 28).

```
from scipy.cluster.vq import *

n = len(projected)

# compute distance matrix
S = array([[ sqrt(sum((projected[i]-projected[j])**2))
          for i in range(n) ] for j in range(n)], 'f')

# create Laplacian matrix
rowsum = sum(S,axis=0)
D = diag(1 / sqrt(rowsum))
I = identity(n)
L = I - dot(D,dot(S,D))

# compute eigenvectors of L
U,sigma,V = linalg.svd(L)

k = 5
# create feature vector from k first eigenvectors
# by stacking eigenvectors as columns
features = array(V[:k]).T
```

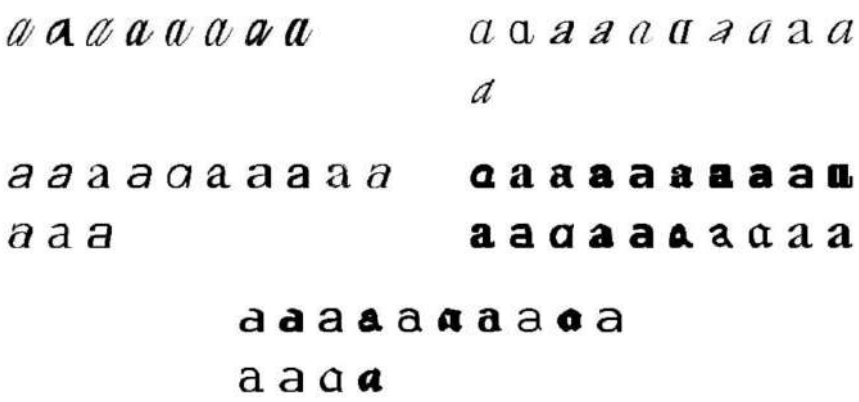


Figure 6.8: Spectral clustering of font images using the eigenvectors of the Laplacian matrix.

```
# k-means
features = whiten(features)
centroids,distortion = kmeans(features,k)
code,distance = vq(features,centroids)

# plot clusters
for c in range(k):
    ind = where(code==c)[0]
    figure()
    for i in range(minimum(len(ind),39)):
        im = Image.open(path+imlist[ind[i]])
        subplot(4,10,i+1)
        imshow(array(im))
        axis('equal')
        axis('off')
show()
```

In this case we just create S using pair-wise Euclidean distances and compute a standard k-means clustering on the k eigenvectors ($k = 5$ in this particular case). Remember that the matrix V contains the eigenvectors sorted with respect to the eigenvalues. Finally, the clusters are plotted. Figure 6.8 shows the clusters for an example run (remember that the k-means step might give different results each run).

We can also try this on an example where we don't have any feature vectors or any strict definition of similarity. The geotagged Panoramio images on page 63 were linked based on how many matching local descriptors were found between them. The matrix

on page 67 is a similarity matrix with scores equal to the number of matching features (without any normalization). With *imlist* containing the filenames of the images and the similarity matrix saved to a file using NumPy's `savetxt()` we only need to modify the first rows of the code above to:

```
n = len(imlist)

# load the similarity matrix and reformat
S = loadtxt('panoramio_matches.txt')
S = 1 / (S + 1e-6)
```

where we invert the scores to have low values for similar images (so we don't have to modify the code above). We add a small number to avoid division with zero. The rest of the code you can leave as is.

Choosing k is a bit tricky in this case. Most people would consider there to be only two classes (the two sides of the White House) and then some junk images. With $k = 2$, you get something like Figure 6.9, with one large cluster of images of one side and the other cluster containing the other side plus all the junk images. Picking a larger value of k like $k = 10$ gives several clusters with only one image (hopefully the junk images) and some real clusters. An example run is shown in Figure 6.10. In this case there were only two actual clusters, each containing images of one side of the White House.

There are many different versions and alternatives to the algorithm presented here. Each of them with its own idea of how to construct the matrix L and what to do with the eigenvectors. For further reading on spectral clustering and the details of some common algorithms see for example the review paper [37].

Exercises

1. *Hierarchical k-means* is a clustering method that applies k-means recursively to the clusters to create a tree of incrementally refined clusters. In this case, each node in the tree will branch to k child nodes. Implement this and try it on the font images.

2. Using the hierarchical k-means from the previous exercise, make a tree visualization (similar to the dendrogram for hierarchical clustering) that shows the average image for each cluster node. Tip: you can take the average PCA coefficients feature vector and use the PCA basis to synthesize an image for each feature vector.

3. By modifying the class used for hierarchical clustering to include the number of images below the node you have a simple and fast way of finding similar (tight)

Figure 6.9: Spectral clustering of geotagged images of the White House with $k = 2$ and the similarity scores as the number of matching local descriptors.

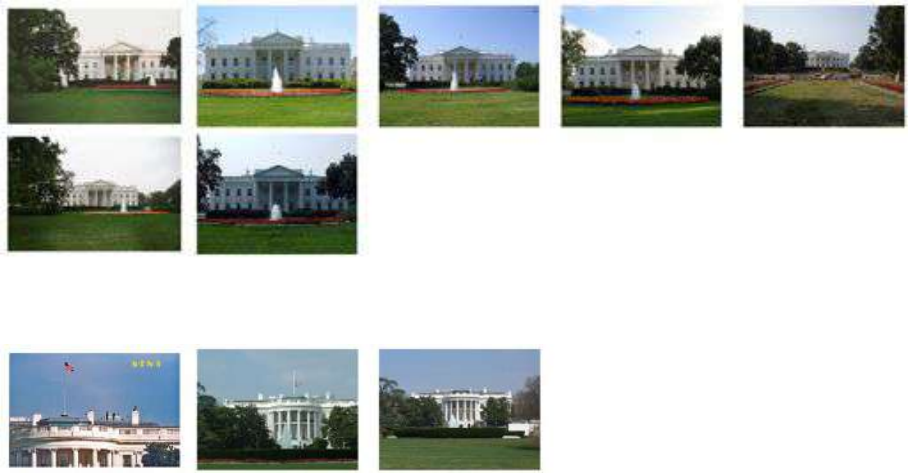


Figure 6.10: Spectral clustering of geotagged images of the White House with $k = 10$ and the similarity scores as the number of matching local descriptors. Only the clusters with more than one image shown.

groups of a given size. Implement this small change and try it out on some real data. How does it perform?

4. Experiment with using single and complete linking for building the hierarchical cluster tree. How do the resulting clusters differ?

5. In some spectral clustering algorithms the matrix $D^{-1}S$ is used instead of L. Try replacing the Laplacian matrix with this and apply this on a few different data sets.

6. Download some image collections from Flickr searches with different tags. Extract the RGB histogram like you did for the sunset images. Cluster the images using one of the methods from this chapter. Can you separate the classes with the clusters?

Chapter 7

Searching Images

This chapter shows how to use text mining techniques to search for images based on their visual content. The basic ideas of using visual words are presented and the details of a complete setup are explained and tested on an example image data set.

7.1 Content-based Image Retrieval

Content-based image retrieval (*CBIR*) deals with the problem of retrieving visually similar images from a (large) database of images. This can be images with similar color, similar textures or similar objects or scenes, basically any information contained in the images themselves.

For high-level queries, like finding similar objects, it is not feasible to do a full comparison (for example using feature matching) between a query image and all images in the database. It would simply take too much time to return any results if the database is large. In the last couple of years, researchers have successfully introduced techniques from the world of text mining for CBIR problems making it possible to search millions of images for similar content.

Inspiration from text mining - the vector space model

The *vector space model* is a model for representing and searching text documents. As we will see, it can be applied to essentially any kind of objects, including images. The name comes from the fact that text documents are represented with vectors that are histograms of the word frequencies in the text[1]. In other words, the vector will contain the number of occurrences of every word (at the position corresponding to that word)

[1] Often you see "term" used instead of "word", the meaning is the same.

and zeros everywhere else. This model is also called a *bag-of-word representation* since order and location of words is ignored.

Documents are indexed by doing a word count to construct the document histogram vector $\mathbf{v}$, usually with common words like "the", "and", "is" etc. ignored. These common words are called *stop words*. To compensate for document length, the vectors can be normalized to unit length by dividing with the total histogram sum. The individual components of the histogram vector are usually weighted according to the importance of each word. Usually, the importance of a word increases proportional to how often it appears in the document but decreases if the word is common in all documents in a data set (or "corpus").

The most common weighting is *tf-idf weighting* (*term frequency - inverse document frequency*) where the *term frequency* of a word w in document d, is

$$\mathrm{tf}_{w,d} = \frac{n_w}{\sum_j n_j} \ ,$$

where n_w is the number of occurrences of w in d. To normalize, this is divided by the total number of occurrences of all words in the document.

The *inverse document frequency* is

$$\mathrm{idf}_{w,d} = \log \frac{|D|}{|\{d \ : \ w \in d\}|} \ ,$$

where $|D|$ is the number of documents in the corpus D and the denominator the number of documents d in D containing w. Multiplying the two gives the tf-idf weight which is then the elements in $\mathbf{v}$. You can read more about tf-idf at `http://en.wikipedia.org/wiki/Tf-idf`.

This is really all we need at the moment. Let's see how to carry this model over to indexing and searching images based on their visual content.

7.2 Visual Words

To apply text mining techniques to images, we first need to create the visual equivalent of words. This is usually done using local descriptors like the SIFT descriptor in Section 2.2. The idea is to quantize the descriptor space into a number of typical examples and assign each descriptor in the image to one of those examples. These typical examples are determined by analyzing a training set of images and can be considered as *visual words* and the set of all words is then a *visual vocabulary* (sometimes called a *visual codebook*). This vocabulary can be created specifically for a given problem or type of images or just try to represent visual content in general.

The visual words are constructed using some clustering algorithm applied to the feature descriptors extracted from a (large) training set of images. The most common choice is k-means[2], which is what we will use here. Visual words are nothing but a collection of vectors in the given feature descriptor space, in the case of k-means they are the cluster centroids. Representing an image with a histogram of visual words is then called a *bag of visual words* model.

Let's introduce an example data set and use that to illustrate the concept. The file *first1000.zip* contains the first 1000 images from the University of Kentucky object recognition benchmark set (also known as "ukbench"). The full set, reported benchmarks and some supporting code can be found at `http://www.vis.uky.edu/~stewe/ukbench/`. The ukbench set contains many sets of four images, each of the same scene or object (stored consecutively so that $0 \ldots 3$, $4 \ldots 7$, etc. belong together). Figure 7.1 shows some examples from the data set. The appendix has the details on the set and how to get the data.

Creating a vocabulary

To create a vocabulary of visual words we first need to extract descriptors. Here we will use the SIFT descriptor. Running the following lines of code, with *imlist*, as usual, containing the filenames of the images,

```
nbr_images = len(imlist)
featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images)]

for i in range(nbr_images):
  sift.process_image(imlist[i],featlist[i])
```

will give you descriptor files for each image. Create a file *vocabulary.py* and add the following code for a vocabulary class and a method for training a vocabulary on some training image data.

```
from scipy.cluster.vq import *
import vlfeat as sift


class Vocabulary(object):

  def __init__(self,name):
    self.name = name
    self.voc = []
    self.idf = []
    self.trainingdata = []
```

[2]Or in the more advanced cases hierarchical k-means.

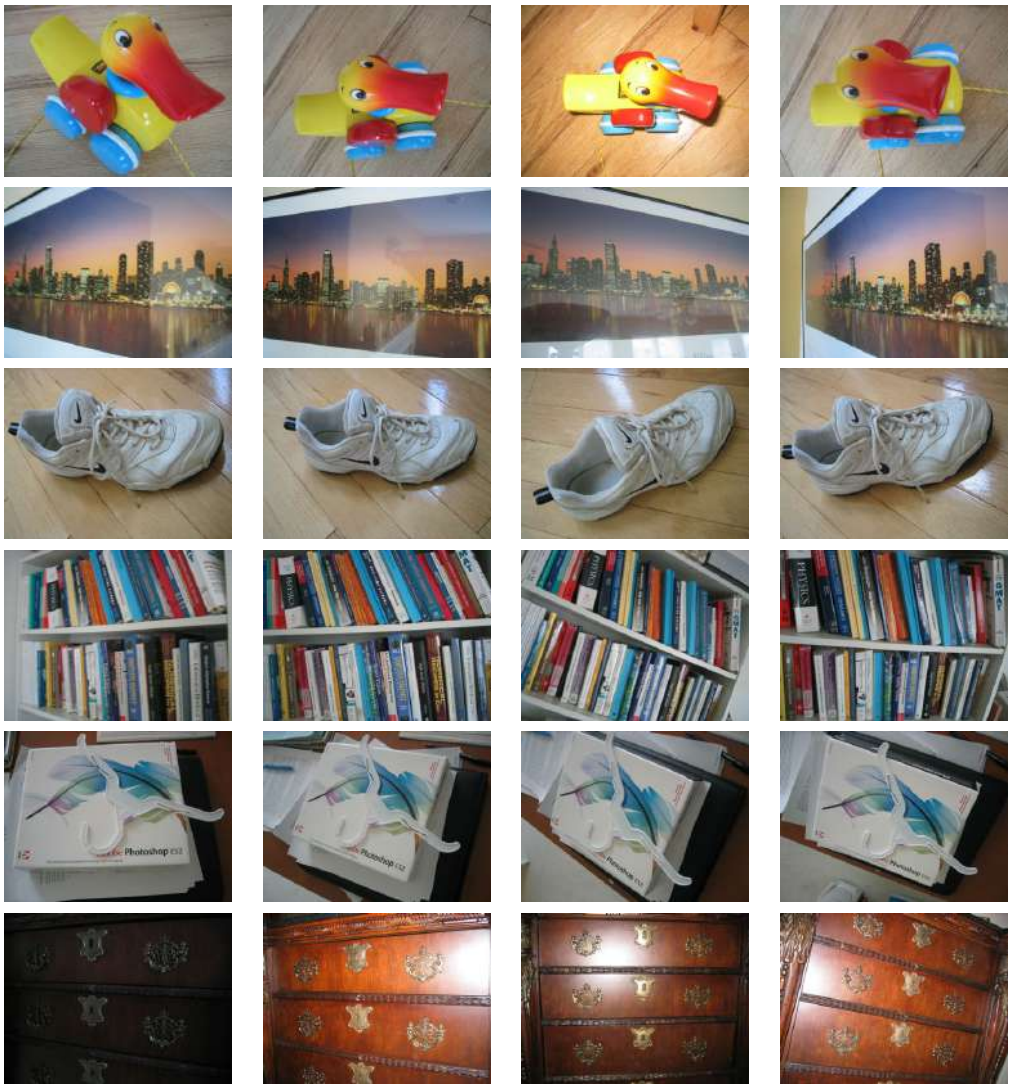


Figure 7.1: Some examples of images from the ukbench (University of Kentucky object recognition benchmark) data set.

```
    self.nbr_words = 0


  def train(self,featurefiles,k=100,subsampling=10):
    """ Train a vocabulary from features in files listed
      in featurefiles using k-means with k number of words.
      Subsampling of training data can be used for speedup. """

    nbr_images = len(featurefiles)
    # read the features from file
    descr = []
    descr.append(sift.read_features_from_file(featurefiles[0])[1])
    descriptors = descr[0] #stack all features for k-means
    for i in arange(1,nbr_images):
      descr.append(sift.read_features_from_file(featurefiles[i])[1])
      descriptors = vstack((descriptors,descr[i]))

    # k-means: last number determines number of runs
    self.voc,distortion = kmeans(descriptors[::subsampling,:],k,1)
    self.nbr_words = self.voc.shape[0]

    # go through all training images and project on vocabulary
    imwords = zeros((nbr_images,self.nbr_words))
    for i in range( nbr_images ):
      imwords[i] = self.project(descr[i])

    nbr_occurences = sum( (imwords > 0)*1 ,axis=0)

    self.idf = log( (1.0*nbr_images) / (1.0*nbr_occurences+1) )
    self.trainingdata = featurefiles


  def project(self,descriptors):
    """ Project descriptors on the vocabulary
      to create a histogram of words. """

    # histogram of image words
    imhist = zeros((self.nbr_words))
    words,distance = vq(descriptors,self.voc)
    for w in words:
      imhist[w] += 1

    return imhist
```

The class `Vocabulary` contains a vector of word cluster centers *voc* together with the idf values for each word. To train the vocabulary on some set of images, the method `train()` takes a list of .sift descriptor files and k, the desired number of words for the

vocabulary. There is also an option of subsampling the training data for the k-means step which (will take a long time if too many features are used).

With the images and the feature files in a folder on your computer, the following code will create a vocabulary of length $k \approx 1000$ (again assuming that *imlist* contains a list of filenames for the images).

```
import pickle
import vocabulary

nbr_images = len(imlist)
featlist = [ imlist[i][:-3]+'sift' for i in range(nbr_images) ]

voc = vocabulary.Vocabulary('ukbenchtest')
voc.train(featlist,1000,10)

# saving vocabulary
with open('vocabulary.pkl', 'wb') as f:
  pickle.dump(voc,f)
print 'vocabulary is:', voc.name, voc.nbr_words
```

The last part also saves the entire vocabulary object for later use using the `pickle` module.

7.3 Indexing Images

Setting up the database

To start indexing images we first need to set up a database. Indexing images in this context means extracting descriptors from the images, converting them to visual words using a vocabulary and storing the visual words and word histograms with information about which image they belong to. This will make it possible to query the database using an image and get the most similar images back as search result.

Here we will use SQLite as database. SQLite is a database which stores everything in a single file and is very easy to set up and use. We are using it here since it is the easiest way to get started without having to go into database and server configurations and other details way outside the scope of this book. The Python version, *pysqlite*, is available from `http://code.google.com/p/pysqlite/` (and also through many package repositories on Mac and Linux). SQLite uses the SQL query language so the transition should be easy if you want to use another database.

To get started we need to create tables and indexes and an indexer class to write image data to the database. First, create a file *imagesearch.py* and add the following code:

imlist
rowid
filename

imwords
imid
wordid
vocname

imhistograms
imid
histogram
vocname

Table 7.1: A simple database schema for storing images and visual words.

```
import pickle
from pysqlite2 import dbapi2 as sqlite

class Indexer(object):

  def __init__(self,db,voc):
    """ Initialize with the name of the database
      and a vocabulary object. """

    self.con = sqlite.connect(db)
    self.voc = voc


  def __del__(self):
    self.con.close()


  def db_commit(self):
    self.con.commit()
```

First of all, we need `pickle` for encoding and decoding these arrays to and from strings. SQLite is imported from the `pysqlite2` module (see appendix for installation details). The `Indexer` class connects to to a database and stores a vocabulary object upon creation (where the `__init__()` method is called). The `__del__()` method makes sure to close the database connection and `db_commit()` writes the changes to the database file.

We only need a very simple database schema of three tables. The table `imlist` contains the filenames of all indexed images, `imwords` contains a *word index* of the words, which vocabulary was used, and which images the words appear in. Finally, `imhistograms` contains the full word histograms for each image. We need those to compare images according to our vector space model. The schema is shown in Table 7.1.

The following method for the `Indexer` class creates the tables and some useful indexes to make searching faster.

```
def create_tables(self):
  """ Create the database tables. """

  self.con.execute('create table imlist(filename)')
  self.con.execute('create table imwords(imid,wordid,vocname)')
  self.con.execute('create table imhistograms(imid,histogram,vocname)')
  self.con.execute('create index im_idx on imlist(filename)')
  self.con.execute('create index wordid_idx on imwords(wordid)')
  self.con.execute('create index imid_idx on imwords(imid)')
  self.con.execute('create index imidhist_idx on imhistograms(imid)')
  self.db_commit()
```

Adding images

With the database tables in place, we can start adding images to the index. To do this, we need a method `add_to_index()` for our `Indexer` class. Add this method to *imagesearch.py*.

```
def add_to_index(self,imname,descr):
  """ Take an image with feature descriptors,
    project on vocabulary and add to database. """

  if self.is_indexed(imname): return
  print 'indexing', imname

  # get the imid
  imid = self.get_id(imname)

  # get the words
  imwords = self.voc.project(descr)
  nbr_words = imwords.shape[0]

  # link each word to image
  for i in range(nbr_words):
    word = imwords[i]
    # wordid is the word number itself
    self.con.execute("insert into imwords(imid,wordid,vocname)
          values (?,?,?)", (imid,word,self.voc.name))

  # store word histogram for image
  # use pickle to encode NumPy arrays as strings
  self.con.execute("insert into imhistograms(imid,histogram,vocname)
        values (?,?,?)", (imid,pickle.dumps(imwords),self.voc.name))
```

This method takes the image filename and a `NumPy` array with the descriptors found in the image. The descriptors are projected on the vocabulary and inserted in *imwords*

(word by word) and *imhistograms*. We used two helper functions, `is_indexed()` which checks if the image has been indexed already, and `get_id()` which gives the image id for an image filename. Add these to *imagesearch.py*.

```
def is_indexed(self,imname):
  """ Returns True if imname has been indexed. """

  im = self.con.execute("select rowid from imlist where
        filename='%s'" % imname).fetchone()
  return im != None


def get_id(self,imname):
  """ Get an entry id and add if not present. """

  cur = self.con.execute(
  "select rowid from imlist where filename='%s'" % imname)
  res=cur.fetchone()
  if res==None:
    cur = self.con.execute(
    "insert into imlist(filename) values ('%s')" % imname)
    return cur.lastrowid
  else:
    return res[0]
```

Did you notice that we used `pickle` in `add_to_index()`? Databases like SQLite do not have a standard type for storing objects or arrays. Instead, we can create a string representation using Pickle's `dumps()` function and write the string to the database. Consequently, we need to un-pickle the string when reading from the database. More on that in the next section.

The following code example will go through the ukbench sample images and add them to our index. Here we assume that the lists *imlist* and *featlist* contain the filenames of the images and the descriptor files and that the vocabulary you trained earlier was pickled to a file *vocabulary.pkl*.

```
import pickle
import sift
import imagesearch

nbr_images = len(imlist)

# load vocabulary
with open('vocabulary.pkl', 'rb') as f:
  voc = pickle.load(f)

# create indexer
indx = imagesearch.Indexer('test.db',voc)
```

```
indx.create_tables()

# go through all images, project features on vocabulary and insert
for i in range(nbr_images)[:100]:
  locs,descr = sift.read_features_from_file(featlist[i])
  indx.add_to_index(imlist[i],descr)

# commit to database
indx.db_commit()
```

We can now inspect the contents of our database:

```
from pysqlite2 import dbapi2 as sqlite
con = sqlite.connect('test.db')
print con.execute('select count (filename) from imlist').fetchone()
(1000,)
print con.execute('select * from imlist').fetchone()
(u'ukbench00000.jpg',)
```

If you try `fetchall()` instead of `fetchone()` in the last line you will get a long list of all the filenames.

7.4 Searching the Database for Images

With a set of images indexed we can search the database for similar images. Here we have used a bag-of-word representation for the whole image but the procedure explained here is generic and can be used to find similar objects, similar faces, similar colors etc. It all depends on the images and descriptors used.

To handle searches we introduce a `Searcher` class to *imagesearch.py*.

```
class Searcher(object):

  def __init__(self,db,voc):
    """ Initialize with the name of the database. """
    self.con = sqlite.connect(db)
    self.voc = voc


  def __del__(self):
    self.con.close()
```

A new `Searcher` object connects to the database and closes the connection upon deletion, same as for the `Indexer` class before.

If the number of images is large, it is not feasible to do a full histogram comparison across all images in the database. We need a way to find a reasonably sized set of

candidates (where "reasonable" can be determined by search response time, memory requirements etc.). This is where the word index comes into play. Using the index we can get a set of candidates and then do the full comparison against that set.

Using the index to get candidates

We can use our index to find all images that contain a particular word. This is just a simple query to the database. Add `candidates_from_word()` as a method for the `Searcher` class.

```
def candidates_from_word(self,imword):
  """ Get list of images containing imword. """

  im_ids = self.con.execute(
      "select distinct imid from imwords where wordid=%d" % imword).fetchall()
  return [i[0] for i in im_ids]
```

This gives the image ids for all images containing the word. To get candidates for more than one word, for example all the nonzero entries in a word histogram, we can loop over each word, get images with that word and aggregate the lists[3]. Here we should also keep track of how many times each image id appears in the aggregate list since this shows how many words that matches the ones in the word histogram. This can be done with the following `Searcher` method:

```
def candidates_from_histogram(self,imwords):
  """ Get list of images with similar words. """

  # get the word ids
  words = imwords.nonzero()[0]

  # find candidates
  candidates = []
  for word in words:
    c = self.candidates_from_word(word)
    candidates+=c

  # take all unique words and reverse sort on occurrence
  tmp = [(w,candidates.count(w)) for w in set(candidates)]
  tmp.sort(cmp=lambda x,y:cmp(x[1],y[1]))
  tmp.reverse()

  # return sorted list, best matches first
  return [w[0] for w in tmp]
```

[3] If you don't want to use all words, try ranking them according to their idf weight and use the ones with highest weights.

This method creates a list of word ids from the nonzero entries in a word histogram of an image. Candidates for each word are retrieved and aggregated in the list *candidates*. Then we create a list of tuples (word id, count) with the number of occurrences of each word in the candidate list and sort this list (in place for efficiency) using `sort()` with a custom comparison function that compares the second element in the tuple. The comparison function is declared inline using `lambda` functions, convenient one-line function declarations. The result is returned as a list of image ids with the best matching image first.

Consider the example

```
src = imagesearch.Searcher('test.db')
locs,descr = sift.read_features_from_file(featlist[0])
iw = voc.project(descr)

print 'ask using a histogram...'
print src.candidates_from_histogram(iw)[:10]
```

which prints the first 10 lookups from the index and gives the output (this will vary depending on your vocabulary):

```
ask using a histogram...
[655, 656, 654, 44, 9, 653, 42, 43, 41, 12]
```

None of the top 10 candidates are correct. Don't worry, we can now take any number of elements from this list and compare histograms. As you will see, this improves things considerably.

Querying with an image

There is not much more needed to do a full search using an image as query. To do word histogram comparisons a `Searcher` object needs to be able to read the image word histograms from the database. Add this method to the `Searcher` class.

```
def get_imhistogram(self,imname):
  """ Return the word histogram for an image. """

  im_id = self.con.execute(
      "select rowid from imlist where filename='%s'" % imname).fetchone()
  s = self.con.execute(
      "select histogram from imhistograms where rowid='%d'" % im_id).fetchone()

  # use pickle to decode NumPy arrays from string
  return pickle.loads(str(s[0]))
```

Again we use pickle to convert between string and `NumPy` arrays, this time with `loads()`.

Now we can combine everything into a query method:

```
def query(self,imname):
    """ Find a list of matching images for imname"""

    h = self.get_imhistogram(imname)
    candidates = self.candidates_from_histogram(h)

    matchscores = []
    for imid in candidates:
        # get the name
        cand_name = self.con.execute(
            "select filename from imlist where rowid=%d" % imid).fetchone()
        cand_h = self.get_imhistogram(cand_name)
        cand_dist = sqrt( sum( (h-cand_h)**2 ) ) #use L2 distance
        matchscores.append( (cand_dist,imid) )

    # return a sorted list of distances and database ids
    matchscores.sort()
    return matchscores
```

This `Searcher` method takes the filename of an image, retrieves the word histogram and a list of candidates (which should be limited to some maximum number if you have a large data set). For each candidate, we compare histograms using standard Euclidean distance and return a sorted list of tuples containing distance and image id.

Let's try a query for the same image as in the previous section:

```
src = imagesearch.Searcher('test.db')
print 'try a query...'
print src.query(imlist[0])[:10]
```

This will again print the top 10 results, including the distance, and should look something like this:

```
try a query...
[(0.0, 1), (100.03999200319841, 2), (105.45141061171255, 3), (129.47200469599596, 708),
(129.73819792181484, 707), (132.68006632497588, 4), (139.89639023220005, 10),
(142.31654858097141, 706), (148.1924424523734, 716), (148.22955170950223, 663)]
```

Much better. The image has distance zero to itself and two out of the three images of the same scene are on the first two positions. The third image coming in on position five.

Benchmarking and plotting the results

To get a feel for how good the search results are, we can compute the number of correct images on the top four positions. This is the measure used to report performance for the ukbench image set. Here's a function that computes this score. Add it to *imagesearch.py* and you can start optimizing your queries.

```
def compute_ukbench_score(src,imlist):
    """ Returns the average number of correct
        images on the top four results of queries."""

    nbr_images = len(imlist)
    pos = zeros((nbr_images,4))
    # get first four results for each image
    for i in range(nbr_images):
        pos[i] = [w[1]-1 for w in src.query(imlist[i])[:4]]

    # compute score and return average
    score = array([ (pos[i]//4)==(i//4) for i in range(nbr_images)])*1.0
    return sum(score) / (nbr_images)
```

This function gets the top four results and subtracts one from the index returned by `query()` since the database index starts at one and the list of images at zero. Then we compute the score using integer division, using the fact that the correct images are consecutive in groups of four. A perfect result gives a score of 4, nothing right gives a score of 0 and only retrieving the identical images gives a score of 1. Finding the identical image together with two of the three other images gives a score of 3.

Try it out like this:

```
imagesearch.compute_ukbench_score(src,imlist)
```

or if you don't want to wait (it will take some time to do 1000 queries), just use a subset of the images

```
imagesearch.compute_ukbench_score(src,imlist[:100])
```

We can consider a score close to 3 as pretty good in this case. The state-of-the-art results reported on the ukbench website are just over 3 (note that they are using more images and your score will decrease with a larger set).

Finally, a function for showing the actual search results will be useful. Add this function,

```
def plot_results(src,res):
    """ Show images in result list 'res'."""

    figure()
    nbr_results = len(res)
    for i in range(nbr_results):
        imname = src.get_filename(res[i])
        subplot(1,nbr_results,i+1)
        imshow(array(Image.open(imname)))
        axis('off')
    show()
```

which can be called with any number of search results in the list *res*. For example like this:

```
nbr_results = 6
res = [w[1] for w in src.query(imlist[0])[:nbr_results]]
imagesearch.plot_results(src,res)
```

The helper function

```
def get_filename(self,imid):
    """ Return the filename for an image id"""

    s = self.con.execute(
        "select filename from imlist where rowid='%d'" % imid).fetchone()
    return s[0]
```

translates image id to filenames which we need for loading the images when plotting. Some example queries on our data set are shown using `plot_results()` in Figure 7.2.

7.5 Ranking Results using Geometry

Let's briefly look at a common way of improving results obtained using a bag of visual words model. One of the drawbacks if the model is that the visual words representation of an image does not contain the positions of the image features. This was the price paid to get speed and scalability.

One way to have the feature points improve results is to re-rank the top results using some criteria that takes the features geometric relationships into account. The most common approach is to fit homographies between the feature locations in the query image and the top result images.

To make this efficient the feature locations can be stored in the database and correspondences determined by the word id of the features (this only works if the vocabulary is large enough so that the word id matches contain mostly correct matches). This would require a major rewrite of our database and code above and complicate the presentation. To illustrate we will just reload the features for the top images and match them.

Here is what a complete example of loading all the model files and re-ranking the top results using homographies looks like.

```
import pickle
import sift
import imagesearch
import homography

# load image list and vocabulary
```

Figure 7.2: Some example search results on the ukbench data set. The query image is shown on the far left followed by the top five retrieved images.

```
with open('ukbench_imlist.pkl','rb') as f:
  imlist = pickle.load(f)
  featlist = pickle.load(f)

nbr_images = len(imlist)

with open('vocabulary.pkl', 'rb') as f:
  voc = pickle.load(f)


src = imagesearch.Searcher('test.db',voc)

# index of query image and number of results to return
q_ind = 50
nbr_results = 20

# regular query
res_reg = [w[1] for w in src.query(imlist[q_ind])[:nbr_results]]
print 'top matches (regular):', res_reg


# load image features for query image
q_locs,q_descr = sift.read_features_from_file(featlist[q_ind])
fp = homography.make_homog(q_locs[:,:2].T)

# RANSAC model for homography fitting
model = homography.RansacModel()

rank = {}
# load image features for result
for ndx in res_reg[1:]:
  locs,descr = sift.read_features_from_file(featlist[ndx])

  # get matches
  matches = sift.match(q_descr,descr)
  ind = matches.nonzero()[0]
  ind2 = matches[ind]
  tp = homography.make_homog(locs[:,:2].T)

  # compute homography, count inliers. if not enough matches return empty list
  try:
    H,inliers = homography.H_from_ransac(fp[:,ind],tp[:,ind2],model,match_theshold=4)
  except:
    inliers = []

  # store inlier count
  rank[ndx] = len(inliers)
```

```
# sort dictionary to get the most inliers first
sorted_rank = sorted(rank.items(), key=lambda t: t[1], reverse=True)
res_geom = [res_reg[0]]+[s[0] for s in sorted_rank]
print 'top matches (homography):', res_geom


# plot the top results
imagesearch.plot_results(src,res_reg[:8])
imagesearch.plot_results(src,res_geom[:8])
```

First the image list, feature list (containing the filenames of the images and SIFT feature files respectively) and the vocabulary is loaded. Then a `Searcher` object is created and a regular query is performed and stored in the list *res_reg*. The features for the query image are loaded. Then for each image in the result list, the features are loaded and matched against the query image. Homographies are computed from the matches and the number of inliers counted. If the homography fitting fails we set the inlier list to an empty list. Finally we sort the dictionary *rank* that contains image index and inlier count according to decreasing number of inliers. The result lists are printed to the console and the top images visualized.

The output looks like this:

```
top matches (regular): [39, 22, 74, 82, 50, 37, 38, 17, 29, 68, 52, 91, 15, 90, 31, ... ]
top matches (homography): [39, 38, 37, 45, 67, 68, 74, 82, 15, 17, 50, 52, 85, 22, 87, ... ]
```

Figure 7.3 shows some sample results with the regular and re-ranked top images.

7.6 Building Demos and Web Applications

In this last section on searching we'll take a look at a simple way of building demos and web applications with Python. By making demos as web pages, you automatically get cross platform support and an easy way to show and share your project with minimal requirements. In the sections below we will go through an example of a simple image search engine.

Creating web applications with CherryPy

To build these demos we will use the `CherryPy` package, available at `http://www.cherrypy.org/`. `CherryPy` is a pure Python lightweight web server that uses an object oriented model. See the appendix for more details on how to install and configure `CherryPy`. Assuming that you have studied the tutorial examples enough to have an initial idea of how `CherryPy` works, let's build an image search web demo on top of the image searcher you created earlier in this chapter.

Figure 7.3: Some example search results with re-ranking based on geometric consistency using homographies. For each example, the top row is the regular result and the bottom row the re-ranked result.

Image search demo

First we need to initialize with a few html tags and load the data using Pickle. We need the vocabulary for the Searcher object that interfaces with the database. Create a file *searchdemo.py* and add the following class with two methods.

```
import cherrypy, os, urllib, pickle
import imagesearch

class SearchDemo(object):

  def __init__(self):
    # load list of images
    with open('webimlist.txt') as f:
      self.imlist = f.readlines()

    self.nbr_images = len(self.imlist)
    self.ndx = range(self.nbr_images)

    # load vocabulary
    with open('vocabulary.pkl', 'rb') as f:
      self.voc = pickle.load(f)

    # set max number of results to show
    self.maxres = 15

    # header and footer html
    self.header = """
      <!doctype html>
      <head>
      <title>Image search example</title>
      </head>
      <body>
      """
    self.footer = """
      </body>
      </html>
      """

  def index(self,query=None):
    self.src = imagesearch.Searcher('web.db',self.voc)

    html = self.header
    html += """
      <br />
      Click an image to search. <a href='?query='>Random selection</a> of images.
      <br /><br />
      """
```

```
    if query:
      # query the database and get top images
      res = self.src.query(query)[:self.maxres]
      for dist,ndx in res:
        imname = self.src.get_filename(ndx)
        html += "<a href='?query="+imname+"'>"
        html += "<img src='"+imname+"' width='100' />"
        html += "</a>"
    else:
      # show random selection if no query
      random.shuffle(self.ndx)
      for i in self.ndx[:self.maxres]:
        imname = self.imlist[i]
        html += "<a href='?query="+imname+"'>"
        html += "<img src='"+imname+"' width='100' />"
        html += "</a>"

    html += self.footer
    return html

  index.exposed = True

cherrypy.quickstart(SearchDemo(), '/',
          config=os.path.join(os.path.dirname(__file__), 'service.conf'))
```

As you can see, this simple demo consists of a single class with one method for initialization and one for the "index" page (the only page in this case). Methods are automatically mapped to URLs and arguments to the methods can be passed directly in the URL. The `index` method has a query parameter which in this case is the query image to sort the others agains. If it is empty, a random selection of images is shown instead. The line

```
index.exposed = True
```

makes the index URL accessible and the last line starts the `CherryPy` web server with configurations read from *service.conf*. Our configuration file for this example has the following lines

```
[global]
server.socket_host = "127.0.0.1"
server.socket_port = 8080
server.thread_pool = 50
tools.sessions.on = True

[/]
tools.staticdir.root = "tmp/"
tools.staticdir.on = True
tools.staticdir.dir = ""
```

The first part specifies which IP address and port to use. The second part enables a local folder for reading (in this case "tmp/"). This should be set to the folder containing your images.

Note: *Don't put anything secret in that folder if you plan to show this to people. The content of the folder will be accessible through CherryPy.*

Start you web server with

```
$ python searchdemo.py
```

from the command line. Opening your browser and pointing it at the right URL (in this case `http://127.0.0.1:8080/`) should show the initial screen with a random selection of images. This should look like Figure 7.4a. Clicking an image starts a query and shows the top results. Clicking an image in the results starts a new query with that image, and so on. There is a link to get back to the starting state of a random selection (by passing an empty query). Some examples are shown in Figure 7.4.

This example shows a full integration from webpage to database queries and presentation of results. Naturally, we kept the styling and options to a minimum and there are many possibilities for improvement. For example, adding a stylesheet to make it prettier or upload files to use as queries.

Exercises

1. Try to speed up queries by only using part of the words in the query image to construct the list of candidates. Use the idf weight as a criteria for which words to use.

2. Implement a visual stop word list of the most common visual words in your vocabulary (say the top 10%) and ignore these words. How does this change the search quality?

3. Visualize a visual word by saving all image features that are mapped to a given word id. Crop image patches around the feature locations (at the given scale) and plot them in a figure. Do the patches for a given word look the same?

4. Experiment with using different distance measures and weighting in the `query()` method. Use the score from `compute_ukbench_score()` to measure your progress.

5. Throughout this chapter we only used SIFT features in our vocabulary. This completely disregards the color information as you can see in the example results

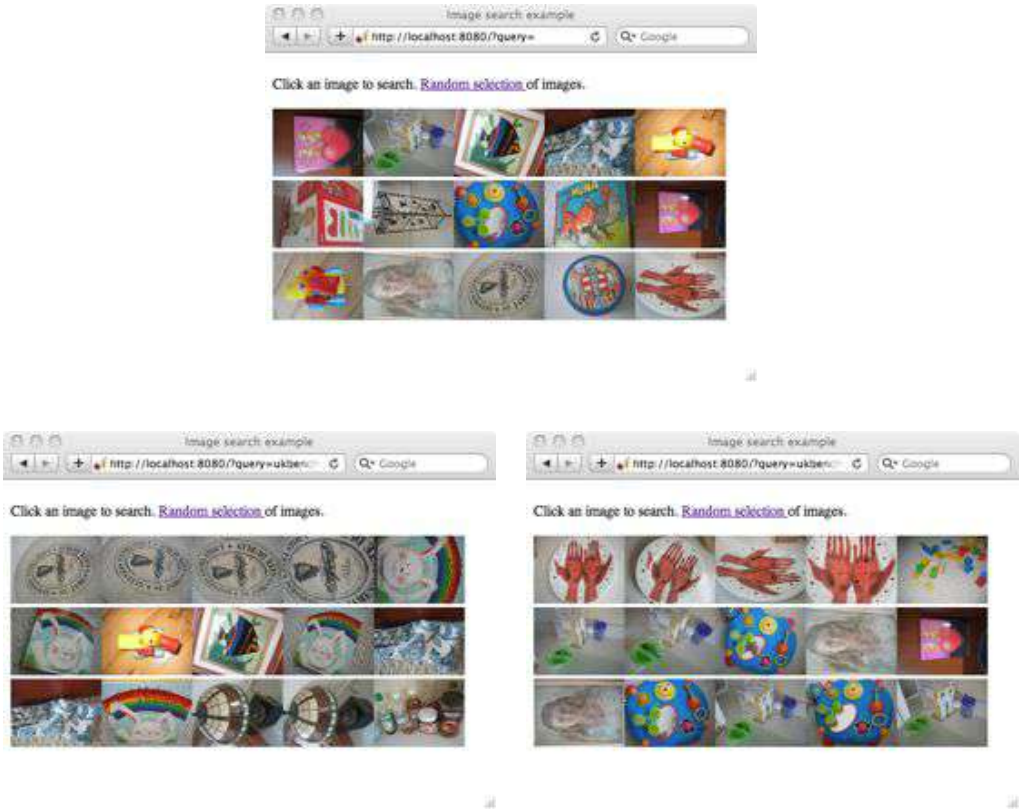


Figure 7.4: Some example search results on the ukbench data set. (top) The starting page which shows a random selection of the images. (bottom) Sample queries. The query image is shown on the top left corner followed by the top image results shown row-wise.

in Figure 7.2. Try to add color descriptors and see if you can improve the search results.

6. For large vocabularies using arrays to represent the visual word frequencies is inefficient since most of the entries will be zero (think of the case with a few hundred thousand words and images with typically a thousand features). One way to overcome this inefficiency is to use dictionaries as sparse array representations. Replace the arrays with a sparse class of your own and add the necessary methods. Alternatively, try to use the `scipy.sparse` module.

7. As you try to increase the size of the vocabulary the clustering time will take too long and the projection of features to words also becomes slower. Implement a hierarchical vocabulary using hierarchical k-mean clustering and see how this improves scaling. See the paper [23] for details and inspiration.

Chapter 8

Classifying Image Content

This chapter introduces algorithms for classifying images and image content. We look at some simple but effective methods as well as state of the art classifiers and apply them to two-class and multi-class problems. We show examples with applications in gesture recognition and object recognition.

8.1 K-Nearest Neighbors

One of the simplest and most used methods for classification is the *k-nearest neighbor classifier* (*kNN*). The algorithm simply compares an object (for example a feature vector) to be classified with all objects in a training set with known class labels and lets the k nearest vote for which class to assign. This method often performs well but has a number of drawbacks. Same as with the k-means clustering algorithm, the number k needs to be chosen and the value will affect performance. Furthermore, the method requires the entire training set to be stored and if this set is large it will be slow to search. For large training sets some form of binning is usually used to reduce the number of comparisons needed[1]. On the positive side, there are no restrictions on what distance measure to use, practically anything you can think of will work (which is not the same as saying that it will perform well). The algorithm is also trivially parallelizable.

Implementing kNN in a basic form is pretty straightforward. Given a set of training examples and a list of associated labels, the code below does the job. The training examples and labels can be rows in an array or just in a list. They can be numbers, strings, whatever you like. Add this class to a file called *knn.py*.

```
class KnnClassifier(object):
```

[1] Another option is to only keep a selected subset of the training set. This can however impact accuracy.

```
  def __init__(self,labels,samples):
    """ Initialize classifier with training data. """

    self.labels = labels
    self.samples = samples


  def classify(self,point,k=3):
    """ Classify a point against k nearest
      in the training data, return label. """

    # compute distance to all training points
    dist = array([L2dist(point,s) for s in self.samples])

    # sort them
    ndx = dist.argsort()

    # use dictionary to store the k nearest
    votes = {}
    for i in range(k):
      label = self.labels[ndx[i]]
      votes.setdefault(label,0)
      votes[label] += 1

    return max(votes)

def L2dist(p1,p2):
  return sqrt( sum( (p1-p2)**2) )
```

It is easiest to define a class and initialize with the training data. That way we don't have to store and pass the training data as arguments every time we want to classify something. Using a dictionary for storing the k nearest labels makes it possible to have labels as text strings or numbers. In this example we used the Euclidean (L_2) distance measure, if you have other measures, just add them as functions.

A simple 2D example

Let's first create some simple 2D example data sets to illustrate and visualize how this classifier works. The following script will create two different 2D point sets, each with two classes, and save the data using Pickle.

```
from numpy.random import randn
import pickle

# create sample data of 2D points
```

```
n = 200

# two normal distributions
class_1 = 0.6 * randn(n,2)
class_2 = 1.2 * randn(n,2) + array([5,1])
labels = hstack((ones(n),-ones(n)))

# save with Pickle
with open('points_normal.pkl', 'w') as f:
  pickle.dump(class_1,f)
  pickle.dump(class_2,f)
  pickle.dump(labels,f)


# normal distribution and ring around it
class_1 = 0.6 * randn(n,2)
r = 0.8 * randn(n,1) + 5
angle = 2*pi * randn(n,1)
class_2 = hstack((r*cos(angle),r*sin(angle)))
labels = hstack((ones(n),-ones(n)))

# save with Pickle
with open('points_ring.pkl', 'w') as f:
  pickle.dump(class_1,f)
  pickle.dump(class_2,f)
  pickle.dump(labels,f)
```

Run the script twice with different filenames, for example *points_normal_test.pkl* and *points_ring_test.pkl* the second time. You will now have four files with 2D data sets, two files for each of the distributions. We can use one for training and the other for testing.

Let's see how to do that with the kNN classifier. Create a script with the following commands.

```
import pickle
import knn
import imtools

# load 2D points using Pickle
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

model = knn.KnnClassifier(labels,vstack((class_1,class_2)))
```

This will create a kNN classifier *model* using the data in the Pickle file. Now add the following:

```
# load test data using Pickle
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# test on the first point
print model.classify(class_1[0])
```

This loads the other data set (the test set) and prints the estimated class label of the first point to your console.

To visualize the classification of all the test points and show how well the classifier separates the two classes we can add these lines.

```
# define function for plotting
def classify(x,y,model=model):
  return array([model.classify([xx,yy]) for (xx,yy) in zip(x,y)])

# plot the classification boundary
imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
show()
```

Here we created a small helper function that takes arrays of 2D coordinates x and y and the classifier and returns an array of estimated class labels. Now we can pass this function as an argument to the actual plotting function. Add the following function to your file *imtools*.

```
def plot_2D_boundary(plot_range,points,decisionfcn,labels,values=[0]):
  """ Plot_range is (xmin,xmax,ymin,ymax), points is a list
    of class points, decisionfcn is a funtion to evaluate,
    labels is a list of labels that decisionfcn returns for each class,
    values is a list of decision contours to show. """

  clist = ['b','r','g','k','m','y'] # colors for the classes

  # evaluate on a grid and plot contour of decision function
  x = arange(plot_range[0],plot_range[1],.1)
  y = arange(plot_range[2],plot_range[3],.1)
  xx,yy = meshgrid(x,y)
  xxx,yyy = xx.flatten(),yy.flatten() # lists of x,y in grid
  zz = array(decisionfcn(xxx,yyy))
  zz = zz.reshape(xx.shape)
  # plot contour(s) at values
  contour(xx,yy,zz,values)
```

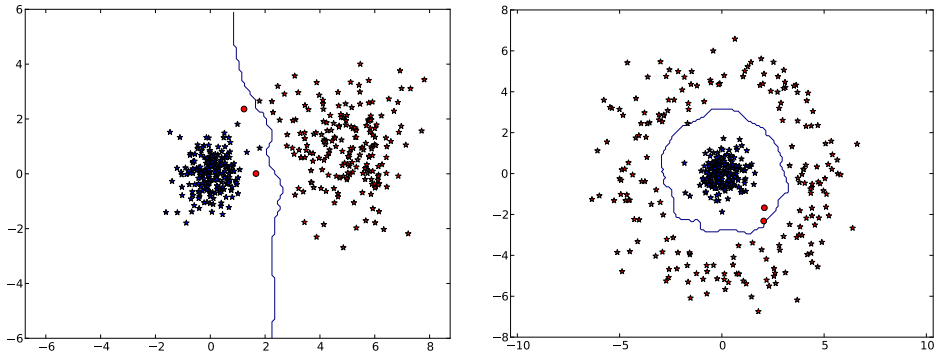


Figure 8.1: Classifying 2D data using a k nearest neighbor classifier. For each example the color shows the class label (blue and red). Correctly classified points are shown with stars and misclassified points with circles. The curve is the classifier decision boundary.

```
# for each class, plot the points with '*' for correct, 'o' for incorrect
for i in range(len(points)):
  d = decisionfcn(points[i][:,0],points[i][:,1])
  correct_ndx = labels[i]==d
  incorrect_ndx = labels[i]!=d
  plot(points[i][correct_ndx,0],points[i][correct_ndx,1],'*',color=clist[i])
  plot(points[i][incorrect_ndx,0],points[i][incorrect_ndx,1],'o',color=clist[i])

axis('equal')
```

This function takes a decision function (the classifier) and evaluates it on a grid using `meshgrid()`. The contours of the decision function can be plotted to show where the boundaries are. The default is the zero contour. The resulting plots look like the ones in Figure 8.1. As you can see the kNN decision boundary can adapt to the distribution of the classes without any explicit modeling.

Dense SIFT as image feature

Let's look at classifying some images. To do so we need a feature vector for the image. We saw feature vectors with average RGB and PCA coefficients as examples in the clustering chapter. Here we will introduce another representation, the *dense SIFT* feature vector.

A dense SIFT representation is created by applying the descriptor part of SIFT

to a regular grid across the whole image[2]. We can use the same executables as in Section 2.2 and get dense SIFT features by adding some extra parameters. Create a file *dsift.py* as a place-holder for the dense SIFT computation and add the following function:

```
import sift

def process_image_dsift(imagename,resultname,size=20,steps=10,
                force_orientation=False,resize=None):
  """ Process an image with densely sampled SIFT descriptors
    and save the results in a file. Optional input: size of features,
    steps between locations, forcing computation of descriptor orientation
    (False means all are oriented upwards), tuple for resizing the image."""

  im = Image.open(imagename).convert('L')
  if resize!=None:
    im = im.resize(resize)
  m,n = im.size

  if imagename[-3:] != 'pgm':
    # create a pgm file
    im.save('tmp.pgm')
    imagename = 'tmp.pgm'

  # create frames and save to temporary file
  scale = size/3.0
  x,y = meshgrid(range(steps,m,steps),range(steps,n,steps))
  xx,yy = x.flatten(),y.flatten()
  frame = array([xx,yy,scale*ones(xx.shape[0]),zeros(xx.shape[0])])
  savetxt('tmp.frame',frame.T,fmt='%03.3f')

  if force_orientation:
    cmmd = str("sift "+imagename+" --output="+resultname+
          " --read-frames=tmp.frame --orientations")
  else:
    cmmd = str("sift "+imagename+" --output="+resultname+
          " --read-frames=tmp.frame")
  os.system(cmmd)
  print 'processed', imagename, 'to', resultname
```

Compare this to the function `process_image()` in Section 2.2. We use the function `savetxt()` to store the *frame* array in a text file for command line processing. The last parameter of this function can be used to resize the image before extracting the descriptors. For example, passing *imsize*=(100,100) will resize to square images 100×100 pixels. Lastly, if *force_orientation* is true the descriptors will be normalized based

[2]Another common name is *Histogram of Oriented Gradients* (*HOG*).

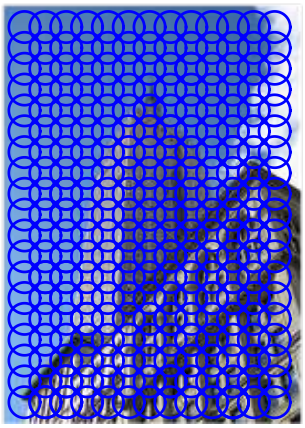


Figure 8.2: An example of applying dense SIFT descriptors over an image.

on the local dominant gradient direction, if it is false all descriptors are simply oriented upwards.

Use it like this to compute the dense SIFT descriptors and visualize the locations:

```
import dsift,sift

dsift.process_image_dsift('empire.jpg','empire.sift',90,40,True)
l,d = sift.read_features_from_file('empire.sift')

im = array(Image.open('empire.jpg'))
sift.plot_features(im,l,True)
show()
```

This will compute SIFT features densely across the image with the local gradient orientation used to orient the descriptors (by setting *force_orientation* to true). The locations are shown in Figure 8.2.

Classifying images - hand gesture recognition

In this application, we will look at applying the dense SIFT descriptor to images of hand gestures to build a simple hand gesture recognition system. We will use some images from the Static Hand Posture Database (available at `http://www.idiap.ch/resource/gestures/`) to illustrate. Download the smaller test set ("test set 16.3Mb" on the webpage) and take all the images in the "uniform" folders and split each class evenly into two folders called "train" and "test".

Process the images with the dense SIFT function above to get feature vectors for

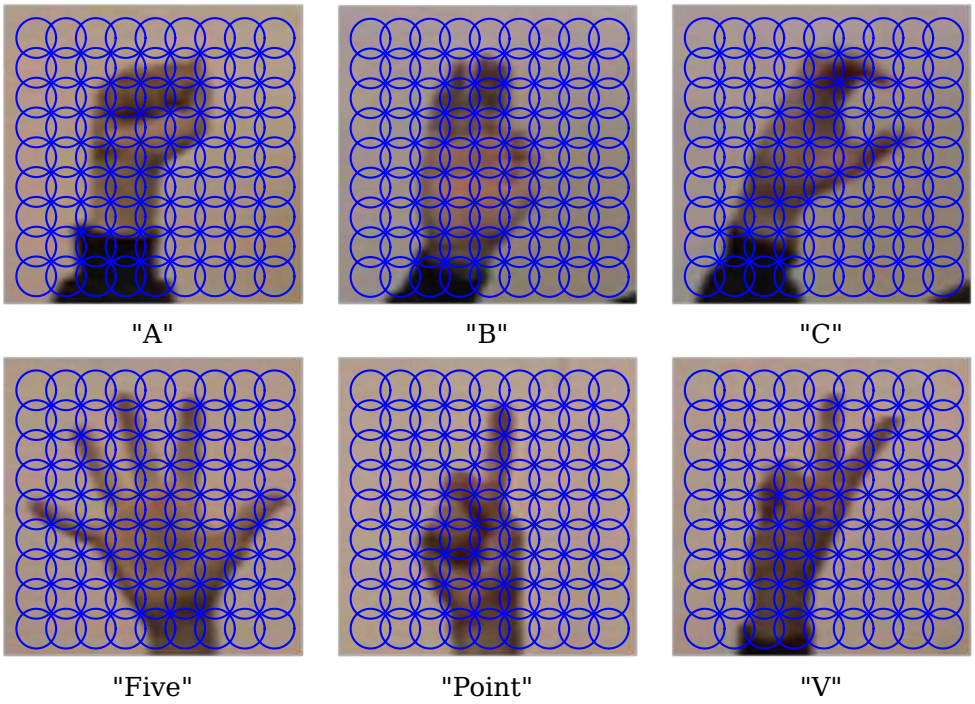


Figure 8.3: Dense SIFT descriptors on sample images from the six categories of hand gesture images. (images from the Static Hand Posture Database)

all images. Again, assuming the filenames are in a list *imlist*, this is done like this.

```
import dsift

# process images at fixed size (50,50)
for filename in imlist:
  featfile = filename[:-3]+'dsift'
  dsift.process_image_dsift(filename,featfile,10,5,resize=(50,50))
```

This creates feature files for each image with the extension ".dsift". *Note the resizing of the images to some common fixed size*. This is very important, otherwise your images will have varying number of descriptors and therefore varying length of the feature vectors. This will cause errors when comparing them later. Plotting the images with the descriptors looks like in Figure 8.3.

Define a helper function for reading the dense SIFT descriptor files as this

```
import os, sift

def read_gesture_features_labels(path):
```

```
  # create list of all files ending in .dsift
  featlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.dsift')]

  # read the features
  features = []
  for featfile in featlist:
    l,d = sift.read_features_from_file(featfile)
    features.append(d.flatten())
  features = array(features)

  # create labels
  labels = [featfile.split('/')[-1][0] for featfile in featlist]

  return features,array(labels)
```

Then we can read the features and labels for our test and training sets using the following commands.

```
features,labels = read_gesture_features_labels('train/')

test_features,test_labels = read_gesture_features_labels('test/')

classnames = unique(labels)
```

Here we used the first letter in the filename to create class labels. Using the `NumPy` function `unique()` we get a sorted list of unique class names.

Now we can try our nearest neighbor code on this data:

```
# test kNN
k = 1
knn_classifier = knn.KnnClassifier(labels,features)
res = array([knn_classifier.classify(test_features[i],k) for i in
          range(len(test_labels))])

# accuracy
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Accuracy:', acc
```

First the classifier object is created with the training data and labels as input. Then we iterate over the test set and classify each image using the `classify()` method. The accuracy is computed by multiplying the boolean array with one and summing. In this case the true values are 1 so it is a simple thing to count the correct classifications. This gives a printout like this

```
Accuracy: 0.811518324607
```

which means that 81% were correctly classified in this case. The value will vary with the choice of k and the parameters for the dense SIFT image descriptor.

The accuracy above shows how many correct classifications there are for a given test set but does not tell us which signs are hard to classify or which mistakes are typically made. A *confusion matrix* is a matrix that shows how many samples of each class is classified as each of the classes. It shows how the errors are distributed and what classes are often "confused" for each other.

The following function will print the labels and the corresponding confusion matrix.

```
def print_confusion(res,labels,classnames):

  n = len(classnames)

  # confusion matrix
  class_ind = dict([(classnames[i],i) for i in range(n)])

  confuse = zeros((n,n))
  for i in range(len(test_labels)):
    confuse[class_ind[res[i]],class_ind[test_labels[i]]] += 1

  print 'Confusion matrix for'
  print classnames
  print confuse
```

The printout of running

```
print_confusion(res,test_labels,classnames)
```

looks like this:

```
Confusion matrix for
['A' 'B' 'C' 'F' 'P' 'V']
[[ 26.   0.   2.   0.   1.   1.]
 [  0.  26.   0.   1.   1.   1.]
 [  0.   0.  25.   0.   0.   1.]
 [  0.   3.   0.  37.   0.   0.]
 [  0.   1.   2.   0.  17.   1.]
 [  3.   1.   3.   0.  14.  24.]]
```

This shows that, for example, in this case "P" ("Point") is often misclassified as "V".

8.2 Bayes Classifier

Another simple but powerful classifier is the *Bayes classifier*[3] (or *naive Bayes classifier*). A Bayes classifier is a probabilistic classifier based on applying Bayes' theorem for conditional probabilities. The assumption is that all features are independent and unrelated to each other (this is the "naive" part). Bayes classifiers can be trained very

[3] After Thomas Bayes, an 18th century English mathematician and minister.

efficiently since the model chosen is applied to each feature independently. Despite their simplistic assumptions, Bayes classifiers have been very successful in practical applications, in particular for email spam filtering. Another benefit of this classifier is that once the model is learned, no training data needs to be stored. Only the model parameters are needed.

The classifier is constructed by multiplying the individual conditional probabilities from each feature to get the total probability of a class. Then the class with highest probability is selected.

Let's look at a basic implementation of a Bayes classifier using Gaussian probability distribution models. This means that each feature is individually modeled using the feature mean and variance, computed from a set of training data. Add the following classifier class to a file called *bayes.py*.

```
class BayesClassifier(object):

  def __init__(self):
    """ Initialize classifier with training data. """

    self.labels = []  # class labels
    self.mean = []    # class mean
    self.var = []   # class variances
    self.n = 0      # nbr of classes

  def train(self,data,labels=None):
    """ Train on data (list of arrays n*dim).
      Labels are optional, default is 0...n-1. """

    if labels==None:
      labels = range(len(data))
    self.labels = labels
    self.n = len(labels)

    for c in data:
      self.mean.append(mean(c,axis=0))
      self.var.append(var(c,axis=0))

  def classify(self,points):
    """ Classify the points by computing probabilities
      for each class and return most probable label. """

    # compute probabilities for each class
    est_prob = array([gauss(m,v,points) for m,v in zip(self.mean,self.var)])

    # get index of highest probability, this gives class label
    ndx = est_prob.argmax(axis=0)
    est_labels = array([self.labels[n] for n in ndx])
```

```
    return est_labels, est_prob
```

The model has two variables per class, the class mean and covariance. The `train()` method takes a lists of feature arrays (one per class) and computes mean and covariance for each. The method `classify()` computes the class probabilities for an array of data points and selects the class with highest probability. The estimated class labels and probabilities are returned. The helper function for the actual Gaussian function is also needed:

```
def gauss(m,v,x):
  """ Evaluate Gaussian in d-dimensions with independent
    mean m and variance v at the points in (the rows of) x. """

  if len(x.shape)==1:
    n,d = 1,x.shape[0]
  else:
    n,d = x.shape

  # covariance matrix, subtract mean
  S = diag(1/v)
  x = x-m
  # product of probabilities
  y = exp(-0.5*diag(dot(x,dot(S,x.T))))

  # normalize and return
  return y * (2*pi)**(-d/2.0) / ( sqrt(prod(v)) + 1e-6)
```

This function computes the product of the individual Gaussian distributions and returns the probability for a given pair of model parameters m,v. For more details on this function see for example `http://en.wikipedia.org/wiki/Multivariate_normal_distribution`.

Try this Bayes classifier on the 2D data from the previous section. This script will load the exact same point sets and train a classifier.

```
import pickle
import bayes
import imtools

# load 2D example points using Pickle
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# train Bayes classifier
bc = bayes.BayesClassifier()
bc.train([class_1,class_2],[1,-1])
```

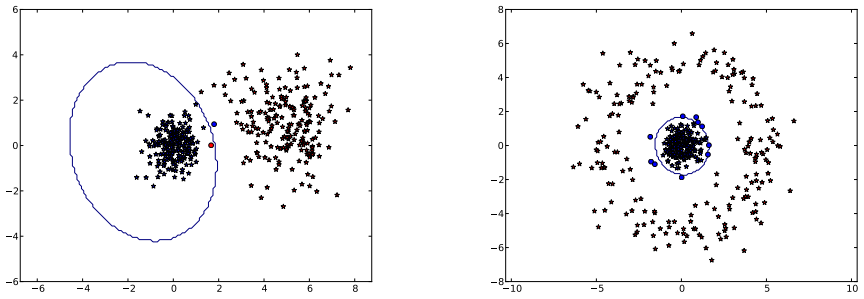


Figure 8.4: Classifying 2D data using a Bayes classifier. For each example the color shows the class label (blue and red). Correctly classified points are shown with stars and misclassified points with circles. The curve is the classifier decision boundary.

Now, we can load the other one and test the classifier.

```
# load test data using Pickle
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# test on some points
print bc.classify(class_1[:10])[0]

# plot points and decision boundary
def classify(x,y,bc=bc):
  points = vstack((x,y))
  return bc.classify(points.T)[0]

imtools.plot_2D_boundary([-6,6,-6,6],[class_1,class_2],classify,[1,-1])
show()
```

This prints the classification result for the first 10 points to the console. It might look like this:

```
[1 1 1 1 1 1 1 1 1 1]
```

Again, we used a helper function `classify()` to pass to the plotting function for visualizing the classification results by evaluating the function on a grid. The plots for the two sets look like Figure 8.4. The decision boundary in this case will be the ellipse-like level curves of a 2D Gaussian function.

Using PCA to reduce dimensions

Now, let's try the gesture recognition problem. Since the feature vectors are very large for the dense SIFT descriptor (more than 10000 for the parameter choices in the example above) it is a good idea to do dimensionality reduction before fitting models to the data. Principal Component Analysis, PCA, (see Section 1.3) usually does a good job. Try the following script that uses PCA from the file *pca.py* (page 28):

```
import pca

V,S,m = pca.pca(features)

# keep most important dimensions
V = V[:50]
features = array([dot(V,f-m) for f in features])
test_features = array([dot(V,f-m) for f in test_features])
```

Here *features* and *test_features* are the same arrays that we loaded for the kNN example. In this case we apply PCA on the training data and keep the 50 dimensions with most variance. This is done by subtracting the mean m (computed on the training data) and multiplying with the basis vectors V. The same transformation is applied to the test data.

Train and test the Bayes classifier like this:

```
# test Bayes
bc = bayes.BayesClassifier()
blist = [features[where(labels==c)[0]] for c in classnames]

bc.train(blist,classnames)
res = bc.classify(test_features)[0]
```

Since `BayesClassifier` takes a list of arrays (one array for each class) we transform the data before passing it to the `train()` method. Since we don't need the probabilities for now we chose to return only the labels of the classification.

Checking the accuracy

```
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Accuracy:', acc
```

gives something like this

```
Accuracy: 0.717277486911
```

and checking the confusion matrix

```
print_confusion(res,test_labels,classnames)
```

gives a print out like this:

```
Confusion matrix for
['A' 'B' 'C' 'F' 'P' 'V']
[[ 20.   0.   0.   4.   0.   0.]
 [  0.  26.   1.   7.   2.   2.]
 [  1.   0.  27.   5.   1.   0.]
 [  0.   2.   0.  17.   0.   0.]
 [  0.   1.   0.   4.  22.   1.]
 [  8.   2.   4.   1.   8.  25.]]
```

Not as good as the kNN classifier but with the Bayes classifier we don't need to keep any training data, just the model parameters for each of the classes. The result will vary greatly with the choice of dimensions after PCA.

8.3 Support Vector Machines

Support Vector Machines (*SVM*) are a powerful type of classifiers that often give state-of-the-art results for many classification problems. In its simplest form an SVM finds a linear separating hyperplane (a plane in higher dimensional spaces) with the best possible separation between two classes. The decision function for a feature vector $\mathbf{x}$ is

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b \ ,$$

where $\mathbf{w}$ is the hyperplane normal and b an offset constant. The zero level of this function then ideally separates the two classes so that one class has positive values and the other negative. The parameters $\mathbf{w}$ and b are found by solving an optimization problem on a training set of labelled feature vectors $\mathbf{x}_i$ with labels $y_i \in \{-1, 1\}$ so that the hyperplane has maximal separation between the two classes. The normal is a linear combination of some of the training feature vectors

$$\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i \ ,$$

so that the decision function can be written

$$f(\mathbf{x}) = \sum_i \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} - b \ .$$

Here i runs over a selection of the training vectors, the selected vectors $\mathbf{x}_i$ are called *support vectors* since they help define the classification boundary.

One of the strengths of SVM is that by using *kernel functions*, that is functions that map the feature vectors to a different (higher) dimensional space, non-linear or very difficult classification problems can be effectively solved while still keeping some control over the decision function. Kernel functions replace the inner product of the classification function, $\mathbf{x}_i \cdot \mathbf{x}$, with a function $K(\mathbf{x}_i, \mathbf{x})$.

Some of the most common kernel functions are:

- *linear*, a hyperplane in feature space, the simplest case, $K(\mathbf{x}_i, \mathbf{x}) = \mathbf{x}_i \cdot \mathbf{x}$.
- *polynomial*, features are mapped with polynomials of a defined degree d, $K(\mathbf{x}_i, \mathbf{x}) = (\gamma \mathbf{x}_i \cdot \mathbf{x} + r)^d$, $\gamma > 0$.
- *radial basis functions*, exponential functions, usually a very effective choice, $K(\mathbf{x}_i, \mathbf{x}) = e^{(-\gamma ||\mathbf{x}_i - \mathbf{x}||^2)}$, $\gamma > 0$.
- *sigmoid*, a smoother alternative to hyperplane, $K(\mathbf{x}_i, \mathbf{x}) = \tanh(\gamma \mathbf{x}_i \cdot \mathbf{x} + r)$.

The parameters of each kernel are also determined during training.

For multi-class problems, the usual procedure is to train multiple SVMs that each separates one class from the rest (also known as "one-versus-all" classifiers). For more details on SVMs see for example the book [9] and the online references at `http://www.support-vector.net/references.html`.

Using LibSVM

We will use one of the best and most commonly used implementation available, LibSVM [7]. LibSVM comes with a nice Python interface (there are also interfaces for many other programming languages). For installation instructions, see Appendix A.4.

Let's use LibSVM on the sample 2D point data to see how it works. This script will load the same points and train a SVM classifier using radial basis functions.

```
import pickle
from svmutil import *
import imtools

# load 2D example points using Pickle
with open('points_normal.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# convert to lists for libsvm
class_1 = map(list,class_1)
class_2 = map(list,class_2)
labels = list(labels)
samples = class_1+class_2 # concatenate the two lists

# create SVM
prob = svm_problem(labels,samples)
param = svm_parameter('-t 2')
# train SVM on data
m = svm_train(prob,param)
```

```
# how did the training do?
res = svm_predict(labels,samples,m)
```

Loading the data set is the same as before but this time we have to convert the arrays to lists since LibSVM does not support array objects as input. Here we used Python's built in `map()` function that applies the conversion function `list()` to each element. The next lines create a SVM problem object and sets some parameters. The `svm_train()` call solves the optimization problem for determining the model parameters. The model can then be used in a predictions. The last call to `svm_predict()` will classify the training data with the model *m* and shows how successful the training was. The print out looks something like this:

```
Accuracy = 100% (400/400) (classification)
```

This means that the classifier completely separates the training data and correctly classifies all 400 data points.

Note that we added a string of parameter choices in the call to train the classifier. These parameters are used to control the kernel type, degree and other choices for the classifier. Most of them are outside the scope of this book but the important ones to know are "t" and "k". The parameter "t" determines the type of kernel used. The options are:

"-t"	**kernel**
0	linear
1	polynomial
2	radial basis function (default)
3	sigmoid

The parameter "k" determines the degree of the polynomial (default is 3).

Now, load the other point set and test the classifier:

```
# load test data using Pickle
with open('points_normal_test.pkl', 'r') as f:
  class_1 = pickle.load(f)
  class_2 = pickle.load(f)
  labels = pickle.load(f)

# convert to lists for libsvm
class_1 = map(list,class_1)
class_2 = map(list,class_2)

# define function for plotting
def predict(x,y,model=m):
  return array(svm_predict([0]*len(x),zip(x,y),model)[0])
```

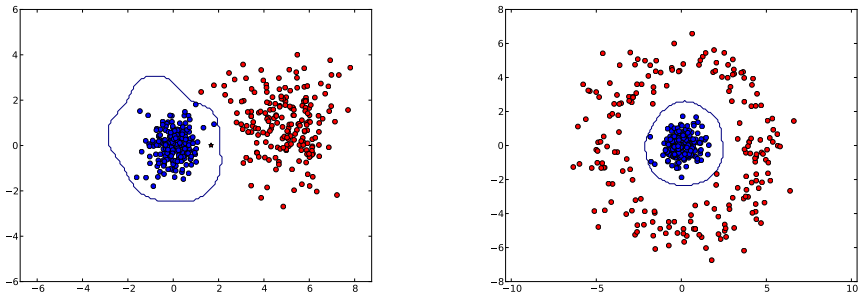


Figure 8.5: Classifying 2D data using a Support Vector Machine classifier. For each example the color shows the class label (blue and red). Correctly classified points are shown with stars and misclassified points with circles. The curve is the classifier decision boundary.

```
# plot the classification boundary
imtools.plot_2D_boundary([-6,6,-6,6],[array(class_1),array(class_2)],predict,[-1,1])
show()
```

Again we have to convert the data to lists for LibSVM. As before, we also define a helper function `predict()` for plotting the classification boundary. Note the use of a list of zeros `[0]*len(x)` as a replacement for the label list if true labels are unavailable. You can use any list as long as it has the correct length. The 2D plots for the two different point data sets are shown in Figure 8.5.

Hand gesture recognition again

Using LibSVM on our multi-class hand gesture recognition problem is fairly straight forward. Multiple classes are automatically handled, we only need to format the data so that the input and output matches the requirements of LibSVM.

With training and testing data *features* and *test_features* as in the previous examples, the following will load the data and train a linear SVM classifier.

```
features = map(list,features)
test_features = map(list,test_features)

# create conversion function for the labels
transl = {}
for i,c in enumerate(classnames):
  transl[c],transl[i] = i,c
```

```
# create SVM
prob = svm_problem(convert_labels(labels,transl),features)
param = svm_parameter('-t 0')

# train SVM on data
m = svm_train(prob,param)

# how did the training do?
res = svm_predict(convert_labels(labels,transl),features,m)

# test the SVM
res = svm_predict(convert_labels(test_labels,transl),test_features,m)[0]
res = convert_labels(res,transl)
```

Same as before, we convert the data to lists using a `map()` call. Then the labels need to be converted since LibSVM does not handle string labels. The dictionary *transl* will contain a conversion between string and integer labels. Try to print it to your console to see what happens. The parameter "-t 0" makes it a linear classifier and the decision boundary will be a hyperplane in the original feature space of some 10000 dimensions.

Now compare the labels, just like before

```
acc = sum(1.0*(res==test_labels)) / len(test_labels)
print 'Accuracy:', acc

print_confusion(res,test_labels,classnames)
```

The output using this linear kernel should look like this:

```
Accuracy: 0.916230366492
Confusion matrix for
['A' 'B' 'C' 'F' 'P' 'V']
[[ 26.   0.   1.   0.   2.   0.]
 [  0.  28.   0.   0.   1.   0.]
 [  0.   0.  29.   0.   0.   0.]
 [  0.   2.   0.  38.   0.   0.]
 [  0.   1.   0.   0.  27.   1.]
 [  3.   0.   2.   0.   3.  27.]]
```

Now if we apply PCA to reduce the dimensions to 50, as we did in Section 8.2, this changes the accuracy to

```
Accuracy: 0.890052356021
```

Not bad seeing that the feature vectors are about 200 times smaller than the original data (and the space to store the support vectors then also 200 times less).

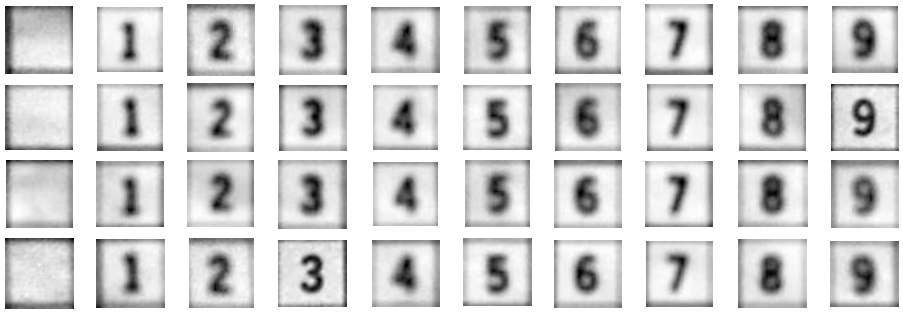


Figure 8.6: Sample training images for the 10 classes of the sudoku OCR classifier.

8.4 Optical Character Recognition

As an example of a multi-class problem, let's look at interpreting images of sudokus. *Optical character recognition* (*OCR*) is the process of interpreting images of hand- or machine written text. A common example is text extraction from scanned documents such as zip codes on letters or book pages as the library volumes in Google Books (`http://books.google.com/`). Here we will look at a simple OCR problem of recognizing numbers in images of printed sudokus. Sudokus are a form of logic puzzles where the goal is to fill a 9×9 grid with the numbers $1 \ldots 9$ so that each column, each row, and each 3×3 sub-grid contains all nine digits[4]. In this example we are just interested in reading the puzzle and interpreting it correctly, actually solving the puzzle we leave to you.

Training a classifier

For this classification problem we have ten classes, the numbers $1 \ldots 9$ and the empty cells. Let's give the empty cells the label 0 so that our class labels are $0 \ldots 9$. To train this ten-class classifier, we will use a dataset of images of cropped sudoku cells[5]. In the file *sudoku_images.zip* are two folders, "ocr_data" and "sudokus". The latter contains images of sudokus under varying conditions. We will save those for later. For now, take a look at the folder "ocr_data". It contains two sub-folders with images one for training and one for testing. The images are named with the first character equal to the class ($0 \ldots 9$). Figure 8.6 shows some samples from the training set. The images are grayscale and roughly 80×80 pixels (with some variation).

[4] See `http://en.wikipedia.org/wiki/Sudoku` for more details if you are unfamiliar with the concept.

[5] Images courtesy of Martin Byröd [4], `http://www.maths.lth.se/matematiklth/personal/byrod/`, collected and cropped from photos of actual sudokus.

Selecting features

We need to decide on what feature vector to use for representing each cell image. There are many good choices, here we'll try something simple but still effective. The following function takes an image and returns a feature vector of the flattened grayscale values.

```
def compute_feature(im):
  """ Returns a feature vector for an
    ocr image patch. """

  # resize and remove border
  norm_im = imresize(im,(30,30))
  norm_im = norm_im[3:-3,3:-3]

  return norm_im.flatten()
```

This function uses the resizing function `imresize()` from *imtools* to reduce the length of the feature vector. We also crop away about 10% border pixels since the crops often get parts of the grid lines on the edges as you can see in Figure 8.6.

Now we can read the training data using a function like this.

```
def load_ocr_data(path):
  """ Return labels and ocr features for all images
    in path. """

  # create list of all files ending in .jpg
  imlist = [os.path.join(path,f) for f in os.listdir(path) if f.endswith('.jpg')]
  # create labels
  labels = [int(imfile.split('/')[-1][0]) for imfile in imlist]

  # create features from the images
  features = []
  for imname in imlist:
    im = array(Image.open(imname).convert('L'))
    features.append(compute_feature(im))
  return array(features),labels
```

The labels are extracted as the first character of the filename of each of the JPEG files and stored in the *labels* list as integers. The feature vectors are computed using the function above and stored in an array.

Multi-class SVM

With the training data in place, we are ready to learn a classifier. Here we'll use a multi-class support vector machine. The code looks just like in the previous section.

```
from svmutil import *

# TRAINING DATA
features,labels = load_ocr_data('training/')

# TESTING DATA
test_features,test_labels = load_ocr_data('testing/')

# train a linear SVM classifier
features = map(list,features)
test_features = map(list,test_features)

prob = svm_problem(labels,features)
param = svm_parameter('-t 0')

m = svm_train(prob,param)

# how did the training do?
res = svm_predict(labels,features,m)

# how does it perform on the test set?
res = svm_predict(test_labels,test_features,m)
```

This trains a linear SVM classifier and tests the performance on the unseen images in the test set. You should get the following printout from the last two `svm_predict()` calls.

```
Accuracy = 100% (1409/1409) (classification)
Accuracy = 99.2979% (990/997) (classification)
```

Great news. The 1409 images of the training set are perfectly separated in the ten classes and the recognition performance on the test set is around 99%. We can now use this classifier on crops from new sudoku images.

Extracting cells and recognizing characters

With a classifier that recognizes cell contents, the next step is to automatically find the cells. Once we solve that, we can crop them and pass the crops to the classifier. Let's for now assume that the image of the sudoku is aligned so that the horizontal and vertical lines of the grid are parallel to the image sides (like the left image of Figure 8.8). Under these conditions, we can threshold the image and sum up the pixel values horizontally and vertically. Since the edges will have values of one and the other parts values of zeros, this should give strong response at the edges and tells us where to crop.

The following function takes a grayscale image and a direction and returns the ten edges for that direction.

```
from scipy.ndimage import measurements

def find_sudoku_edges(im,axis=0):
  """ Finds the cell edges for an aligned sudoku image. """

  # threshold and sum rows and columns
  trim = 1*(im<128)
  s = trim.sum(axis=axis)

  # find center of strongest lines
  s_labels,s_nbr = measurements.label(s>(0.5*max(s)))
  m = measurements.center_of_mass(s,s_labels,range(1,s_nbr+1))
  x = [int(x[0]) for x in m]

  # if only the strong lines are detected add lines in between
  if len(x)==4:
    dx = diff(x)
    x = [x[0],x[0]+dx[0]/3,x[0]+2*dx[0]/3,
        x[1],x[1]+dx[1]/3,x[1]+2*dx[1]/3,
        x[2],x[2]+dx[2]/3,x[2]+2*dx[2]/3,x[3]]

  if len(x)==10:
    return x
  else:
    raise RuntimeError('Edges not detected.')
```

First the image is thresholded at the midpoint to give ones on the dark areas. Then these are added up in the specified direction (axis=0 or 1). The `scipy.ndimage` package contains a module, `measurements`, that is very useful for counting and measuring regions in binary or label arrays. First `labels()` finds the connected components of a binary array computed by thresholding the sum at the midpoint. Then the `center_of_mass()` function computes the center point of each independent component. Depending on the graphic design of the sudoku (all lines equally strong or the sub-grid lines stronger than the other) you might get four or ten points. In the case of four, the intermediary lines are interpolated at even intervals. If the end result does not have ten lines, an exception is raised.

In the "sudokus" folder are a collection of sudoku images of varying difficulty. There is also a file for each image containing the true values of the sudoku so that we can check our results. Some of the images are aligned with the image sides. Picking one of them, you can check the performance of the cropping and classification like this.

```
imname = 'sudokus/sudoku18.jpg'
vername = 'sudokus/sudoku18.sud'
```

```
im = array(Image.open(imname).convert('L'))

# find the cell edges
x = find_sudoku_edges(im,axis=0)
y = find_sudoku_edges(im,axis=1)

# crop cells and classify
crops = []
for col in range(9):
  for row in range(9):
    crop = im[y[col]:y[col+1],x[row]:x[row+1]]
    crops.append(compute_feature(crop))

res = svm_predict(loadtxt(vername),map(list,crops),m)[0]
res_im = array(res).reshape(9,9)

print 'Result:'
print res_im
```

The edges are found and then crops are extracted for each cell. The crops are passed to the same feature extraction function used for the training and stored in an array. These feature vectors are classified using `svm_predict()` with the true labels read using `loadtxt()`. The result in your console should be:

```
Accuracy = 100% (81/81) (classification)
Result:
[[ 0.  0.  0.  0.  1.  7.  0.  5.  0.]
 [ 9.  0.  3.  0.  0.  5.  2.  0.  7.]
 [ 0.  0.  0.  0.  0.  0.  4.  0.  0.]
 [ 0.  1.  6.  0.  0.  4.  0.  0.  2.]
 [ 0.  0.  0.  8.  0.  1.  0.  0.  0.]
 [ 8.  0.  0.  5.  0.  0.  6.  4.  0.]
 [ 0.  0.  9.  0.  0.  0.  0.  0.  0.]
 [ 7.  0.  2.  1.  0.  0.  8.  0.  9.]
 [ 0.  5.  0.  2.  3.  0.  0.  0.  0.]]
```

Now, this was one of the easier images. Try some of the others and see what the errors look like and where the classifier makes mistakes.

If you plot the crops using a 9×9 subplot, they should look like the right image of Figure 8.7.

Rectifying images

If you are happy with the performance of your classifier, the next challenge is to apply it to non-aligned images. We will end our sudoku example with a simple way of rectifying an image given that the four outer corner points of the grid have been detected or

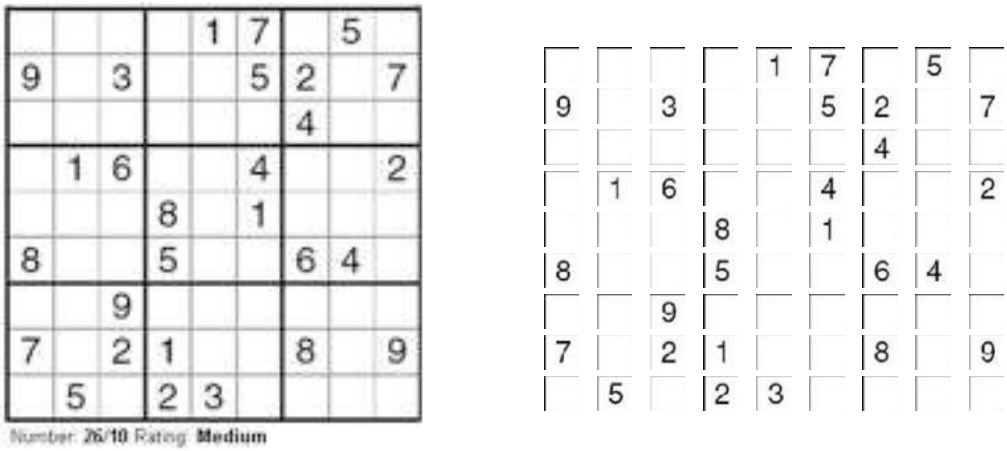


Figure 8.7: An example of detecting and cropping the fields of a sudoku grid. (left) image of a sudoku grid. (right) the 9×9 cropped images of the individual cells to be sent to the OCR classifier.

marked manually. The left image in Figure 8.8 shows an example of a sudoku image with strong perspective effects.

A homography can map the grid to align the edges as in the examples above, all we need to do is estimate the transform. The example below shows the case of manually marking the four corner points and then warping the image to a square target image of 1000×1000 pixels.

```
from scipy import ndimage
import homography

imname = 'sudoku8.jpg'
im = array(Image.open(imname).convert('L'))

# mark corners
figure()
imshow(im)
gray()
x = ginput(4)

# top left, top right, bottom right, bottom left
fp = array([array([p[1],p[0],1]) for p in x]).T
tp = array([[0,0,1],[0,1000,1],[1000,1000,1],[1000,0,1]]).T

# estimate the homography
H = homography.H_from_points(tp,fp)
```

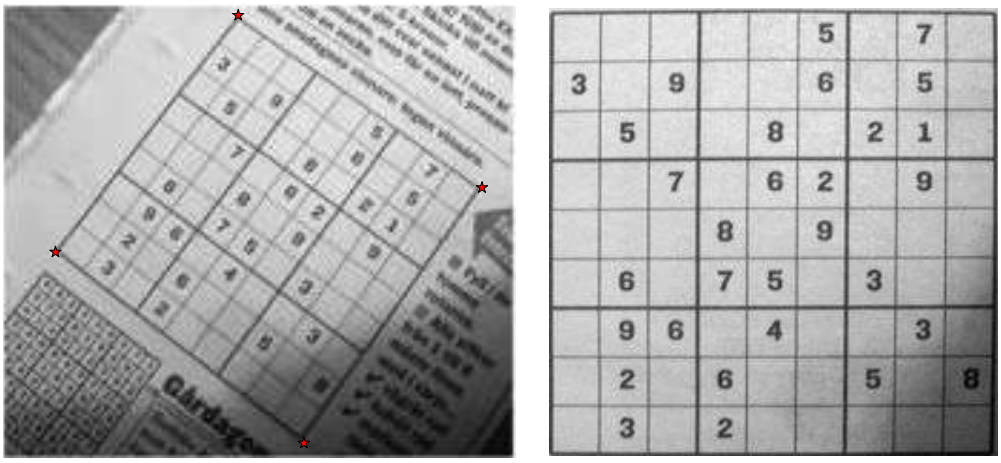


Figure 8.8: An example of rectifying an image using a full perspective transform. (left) the original image with the four corners of the sudoku marked. (right) rectified image warped to a square image of 1000×1000 pixels.

```
# helper function for geometric_transform
def warpfcn(x):
  x = array([x[0],x[1],1])
  xt = dot(H,x)
  xt = xt/xt[2]
  return xt[0],xt[1]

# warp image with full perspective transform
im_g = ndimage.geometric_transform(im,warpfcn,(1000,1000))
```

In most of these sample images an affine transform, as we used in Chapter 3, is not enough. Here we instead used the more general transform function `geometric_transform()` from `scipy.ndimage`. This function takes a 2D to 2D mapping instead of a transform matrix so we need to use a helper function (using a piece-wise affine warp on triangles will introduce artifacts in this case). The warped image is shown to the right in Figure 8.8.

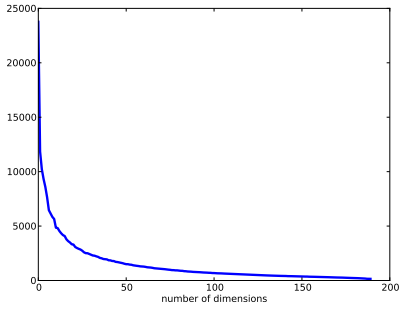
This concludes our sudoku OCR example. There are many improvements to be made and alternatives to investigate. Some are mentioned in the following exercises, the rest we leave to you.

Exercises

1. The performance of the kNN classifier depends on the value of k. Try to vary this number and see how the accuracy changes. Plot the decision boundaries of the 2D point sets to see how they change.

2. The hand gesture data set in Figure 8.3 also contains images with more complex background (in the "complex/" folders). Try to train and test a classifier on these images. What is the difference in performance? Can you suggest improvements to the image descriptor?

3. Try to vary the number of dimensions after PCA projection of the gesture recognition features for the Bayes classifier. What is a good choice? Plot the singular values S, they should give a typical "knee" shaped curve as the one below. A good compromise between ability to generate the variability of the training data and keeping the number of dimensions low is usually found at a number before the curve flattens out.



4. Modify the Bayes classifier to use a different probability model than Gaussian distributions. For example, try using the frequency counts of each feature in the training set. Compare the results to using a Gaussian distribution for some different datasets.

5. Experiment with using non-linear SVMs for the gesture recognition problem. Try polynomial kernels and increase the degree (using the "-d" parameter) incrementally. What happens to the classification performance on the training set and the test set. With a non-linear classifier there is a risk of training and optimizing it for a specific set so that performance is close to perfect on the training set but the classifier has poor performance on other test sets. This phenomenon of breaking the generalization capabilities of a classifier is called *overfitting* and should be avoided.

6. Try some more advanced feature vectors for the sudoku character recognition problem. If you need inspiration, look at [4].

7. Implement a method for automatically aligning the sudoku grid. Try for example feature detection with RANSAC, line detection or detecting the cells using morphological and measurement operations from `scipy.ndimage` (`http://docs.scipy.org/doc/scipy/reference/ndimage.html`). Bonus task, solve the rotation ambiguity of finding the "up" direction. For example, you could try rotating the rectified grid and let the OCR classifier's accuracy vote for the best orientation.

8. For a more challenging classification problem than the sudoku digits, take a look at the MNIST database of *handwritten digits* `http://yann.lecun.com/exdb/mnist/`. Try to extract some features and apply SVM to that set. Check where your performance ends up on the ranking of best methods (some are insanely good).

9. If you want to dive deeper in classifiers and machine learning algorithms, take a look at the `scikit.learn` package (`http://scikit-learn.org/`) and try some of the algorithms on the data in this chapter.

Chapter 9

Image Segmentation

Image segmentation is the process of partitioning an image into meaningful regions. Regions can be foreground versus background or individual objects in the image. The regions are constructed using some feature such as color, edges or neighbor similarity. In this chapter we will look at some different techniques for segmentation.

9.1 Graph Cuts

A *graph* is a set of nodes (sometimes called vertices) with edges between them. See Figure 9.1 for an example[1]. The edges can be directed (as illustrated with arrows in Figure 9.1) or undirected and may have weights associated with them.

A *graph cut* is the partitioning of a directed graph into two disjoint sets. Graph cuts can be used for solving many different computer vision problems like stereo depth reconstruction, image stitching and image segmentation. By creating a graph from image pixels and their neighbors and introducing an energy or a "cost" it is possible to use a graph cut process to segment an image in two or more regions. The basic idea is that similar pixels that are also close to each other should belong to the same partition.

The cost of a graph cut C (where C is a set of edges) is defined as the sum of the edge weights of the cuts

$$E_{cut} = \sum_{(i,j) \in C} w_{ij} \ , \tag{9.1}$$

where w_{ij} is the weight of the edge (i, j) from node i to node j in the graph and the sum is taken over all edges in the cut C.

[1] You also saw graphs in action in Section 2.3, this time we are going to use them to partition images.

The idea behind graph cut segmentation is to partition a graph representation of the image such that the cut cost E_{cut} is minimized. In this graph representation, two additional nodes, a source and a sink node, are added to the graph and only cuts that separate the source and sink are considered.

Finding the *minimum cut* (or *min cut*) is equivalent to finding the *maximum flow* (or *max flow*) between the source and the sink (see [2] for details). There are efficient algorithms for solving these max flow / min cut problems.

For our graph cut examples we will use the `python-graph` package. This package contains many useful graph algorithms. The website with downloads and documentation is `http://code.google.com/p/python-graph/`. We will need the function `maximum_flow()` which computes the max flow / min cut using the Edmonds-Karp algorithm `http://en.wikipedia.org/wiki/Edmonds-Karp_algorithm`. The good thing about using a package written fully in Python is ease of installation and compatibility, the downside is speed. Performance is adequate for our purposes but for anything but small images, a faster implementation is needed.

Here's a simple example of using `python-graph` to compute the max flow / min cut of a small graph[2].

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

gr = digraph()
gr.add_nodes([0,1,2,3])

gr.add_edge((0,1), wt=4)
gr.add_edge((1,2), wt=3)
gr.add_edge((2,3), wt=5)
gr.add_edge((0,2), wt=3)
gr.add_edge((1,3), wt=4)

flows,cuts = maximum_flow(gr,0,3)
print 'flow is:', flows
print 'cut is:', cuts
```

First a directed graph is created with four nodes with index $0 \ldots 3$. Then the edges are added using `add_edge()` with an edge weight specified. This will be used as the maximum flow capacity of the edge. The maximum flow is computed with node 0 as source and node 3 as sink. The flow and the cuts are printed and should look like this:

```
flow is: {(0, 1): 4, (1, 2): 0, (1, 3): 4, (2, 3): 3, (0, 2): 3}
cut is: {0: 0, 1: 1, 2: 1, 3: 1}
```

[2] Same graph as the example at `http://en.wikipedia.org/wiki/Max-flow_min-cut_theorem`.

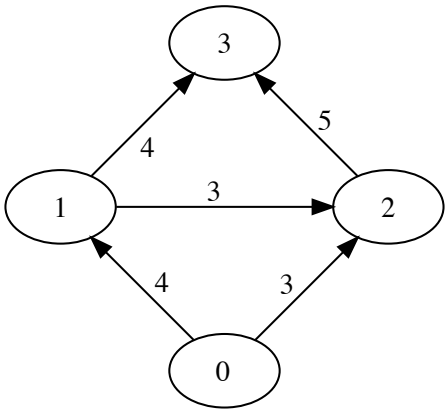


Figure 9.1: A simple directed graph created using `python − graph`.

These two python dictionaries contain the flow through each edge and the label for each node, 0 for the part of the graph containing the source, 1 for the nodes connected to the sink. You can verify manually that the cut is indeed the minimum. The graph is shown in Figure 9.1.

Graphs from images

Given a neighborhood structure, we can define a graph using the image pixels as nodes. Here we will focus on the simplest case of 4-neighborhood of pixels and two image regions (which we can call foreground and background). A *4-neighborhood* is where a pixel is connected to the pixels directly above, below, left, and right[3].

In addition to the pixel nodes, we will also need two special nodes a "source" node and a "sink" node, representing the foreground and background respectively. We will use a simple model where all pixels are connected to the source and the sink.

Here's how to build the graph:

- Every pixel node has an incoming edge from the source node.
- Every pixel node has an outgoing edge to the sink node.

[3]Another common option is 8-neighborhood where the diagonal pixels are also connected

- Every pixel node has one incoming and one outgoing edge to each of its neighbors.

To determine the weights on these edges, you need a segmentation model that determines the edge weights (representing the maximum flow allowed for that edge) between pixels and between pixels and the source and sink. As before we call the edge weight between pixel i and pixel j, w_{ij}. Let's call the weight from the source to pixel i, w_{si}, and from pixel i to the sink, w_{it}.

Let's look at using a naive Bayesian classifier from Section 8.2 on the color values of the pixels. Given that we have trained a Bayes classifier on foreground and background pixels (from the same image or from other images), we can compute the probabilities $p_F(I_i)$ and $p_B(I_i)$ for the foreground and background. Here I_i is the color vector of pixel i.

We can now create a model for the edge weights as follows:

$$w_{si} = \frac{p_F(I_i)}{p_F(I_i) + p_B(I_i)}$$

$$w_{it} = \frac{p_B(I_i)}{p_F(I_i) + p_B(I_i)}$$

$$w_{ij} = \kappa \, e^{-|I_i - I_j|^2/\sigma} \quad .$$

With this model, each pixel is connected to the foreground and background (source and sink) with weights equal to a normalized probability of belonging to that class. The w_{ij} describe the pixel similarity between neighbors, similar pixels have weight close to κ, dissimilar close to 0. The parameter σ determines how fast the values decay towards zero with increasing dissimilarity.

Create a file *graphcut.py* and add the following function that creates this graph from an image.

```
from pygraph.classes.digraph import digraph
from pygraph.algorithms.minmax import maximum_flow

import bayes

def build_bayes_graph(im,labels,sigma=1e2,kappa=2):
  """ Build a graph from 4-neighborhood of pixels.
    Foreground and background is determined from
    labels (1 for foreground, -1 for background, 0 otherwise)
    and is modeled with naive Bayes classifiers."""

  m,n = im.shape[:2]
```

```
# RGB vector version (one pixel per row)
vim = im.reshape((-1,3))

# RGB for foreground and background
foreground = im[labels==1].reshape((-1,3))
background = im[labels==-1].reshape((-1,3))
train_data = [foreground,background]

# train naive Bayes classifier
bc = bayes.BayesClassifier()
bc.train(train_data)

# get probabilities for all pixels
bc_lables,prob = bc.classify(vim)
prob_fg = prob[0]
prob_bg = prob[1]

# create graph with m*n+2 nodes
gr = digraph()
gr.add_nodes(range(m*n+2))

source = m*n # second to last is source
sink = m*n+1 # last node is sink

# normalize
for i in range(vim.shape[0]):
  vim[i] = vim[i] / linalg.norm(vim[i])

# go through all nodes and add edges
for i in range(m*n):
  # add edge from source
  gr.add_edge((source,i), wt=(prob_fg[i]/(prob_fg[i]+prob_bg[i])))

  # add edge to sink
  gr.add_edge((i,sink), wt=(prob_bg[i]/(prob_fg[i]+prob_bg[i])))

  # add edges to neighbors
  if i%n != 0: # left exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-1])**2)/sigma)
    gr.add_edge((i,i-1), wt=edge_wt)
  if (i+1)%n != 0: # right exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+1])**2)/sigma)
    gr.add_edge((i,i+1), wt=edge_wt)
  if i//n != 0: # up exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i-n])**2)/sigma)
    gr.add_edge((i,i-n), wt=edge_wt)
  if i//n != m-1: # down exists
    edge_wt = kappa*exp(-1.0*sum((vim[i]-vim[i+n])**2)/sigma)
```

```
      gr.add_edge((i,i+n), wt=edge_wt)

  return gr
```

Here we used a label image with values 1 for foreground training data and -1 for background training data. Based on this labeling, a Bayes classifier is trained on the RGB values. Then classification probabilities are computed for each pixel. These are then used as edge weights for the edges going from the source and to the sink. A graph with $n * m + 2$ nodes is created. Note the index of the source and sink, we choose them as the last two to simplify the indexing of the pixels.

To visualize the labeling overlaid on the image we can use the function `contourf()` which fills the regions between contour levels of an image (in this case the label image). The *alpha* variable sets the transparency. Add the following function to *graphcut.py*.

```
def show_labeling(im,labels):
  """ Show image with foreground and background areas.
    labels = 1 for foreground, -1 for background, 0 otherwise."""

  imshow(im)
  contour(labels,[-0.5,0.5])
  contourf(labels,[-1,-0.5],colors='b',alpha=0.25)
  contourf(labels,[0.5,1],colors='r',alpha=0.25)
  axis('off')
```

Once the graph is built it needs to be cut at the optimal location. The following function computes the min cut and reformats the output to a binary image of pixel labels.

```
def cut_graph(gr,imsize):
  """ Solve max flow of graph gr and return binary
    labels of the resulting segmentation."""

  m,n = imsize
  source = m*n # second to last is source
  sink = m*n+1 # last is sink

  # cut the graph
  flows,cuts = maximum_flow(gr,source,sink)

  # convert graph to image with labels
  res = zeros(m*n)
  for pos,label in cuts.items()[:-2]: #don't add source/sink
    res[pos] = label

  return res.reshape((m,n))
```

Again, note the indices for the source and sink. We need to take the size of the image as input to compute these indices and to reshape the output before returning the segmentation. The cut is returned as a dictionary which needs to be copied to an image of segmentation labels. This is done using the *.items()* method that returns a list of (key, value) pairs. Again we skip the last two elements of that list.

Let's see how to use these functions for segmenting an image. The following is a complete example of reading an image and creating a graph with class probabilities estimated from two rectangular image regions.

```
from scipy.misc import imresize
import graphcut

im = array(Image.open('empire.jpg'))
im = imresize(im,0.07,interp='bilinear')
size = im.shape[:2]

# add two rectangular training regions
labels = zeros(size)
labels[3:18,3:18] = -1
labels[-18:-3,-18:-3] = 1

# create graph
g = graphcut.build_bayes_graph(im,labels,kappa=1)

# cut the graph
res = graphcut.cut_graph(g,size)


figure()
graphcut.show_labeling(im,labels)

figure()
imshow(res)
gray()
axis('off')

show()
```

We use the `imresize()` function to make the image small enough for our Python graph library, in this case uniform scaling to 7% of the original size. The graph is cut and the result plotted together with an image showing the training regions. Figure 9.2 shows the training regions overlaid on the image and the final segmentation result.

The variable *kappa* (κ in the equations) determines the relative weight of the edges between neighboring pixels. The effect of changing *kappa* can be seen in Figure 9.3. With increasing value, the segmentation boundary will be smoother and details will

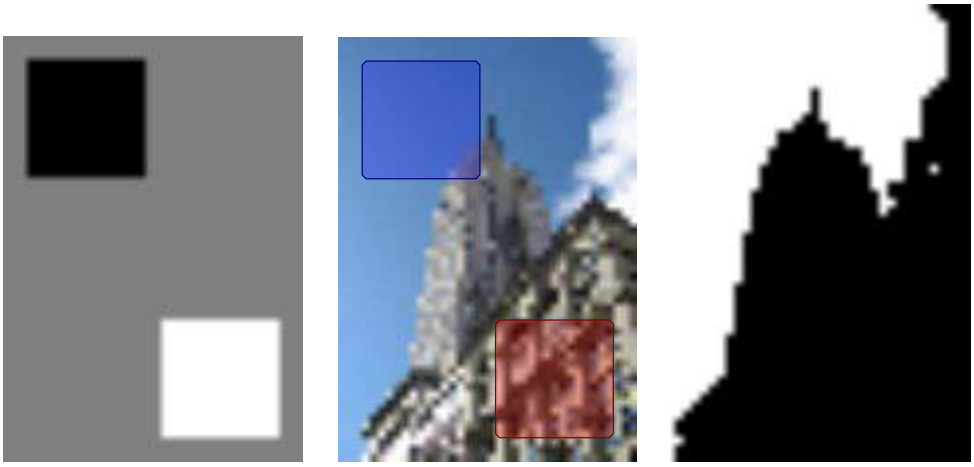


Figure 9.2: An example of graph cut segmentation using a Bayesian probability model. Image is downsampled to size 54×38. (left) label image for model training. (center) training regions shown on the image. (right) segmentation.

be lost. Choosing the right value is up to you, the right value will depend on your application and the type of result you desire.

Segmentation with user input

Graph cut segmentation can be combined with user input in a number of ways. For example, a user can supply markers for foreground and background by drawing on an image. Another way is to select a region that contains the foreground with a bounding box or using a "lasso" tool.

Let's look at this last example using some images from the Grab Cut dataset from Microsoft Research Cambridge, see [27] and Appendix B.5 for details.

These images come with ground truth labels for measuring segmentation performance. They also come with annotations simulating a user selecting a rectangular image region or drawing on the image with a "lasso" type tool to mark foreground and background. We can use these user inputs to get training data and apply graph cuts to segment the image guided by the user input.

The user input is encoded in bitmap images with the following meaning.

(a) (b) (c) (d)

Figure 9.3: The effect of changing the relative weighting between pixel similarity and class probability. The same segmentation as in Figure 9.2 with: (a) $\kappa = 1$, (b) $\kappa = 2$, (c) $\kappa = 5$ and (d) $\kappa = 10$.

pixel value	meaning
0, 64	background
128	unknown
255	foreground

Here's a complete code example of loading an image and annotations and passing that to our graph cut segmentation routine.

```
from scipy.misc import imresize
import graphcut

def create_msr_labels(m,lasso=False):
  """ Create label matrix for training from
    user annotations. """

  labels = zeros(im.shape[:2])

  # background
  labels[m==0] = -1
  labels[m==64] = -1

  # foreground
  if lasso:
    labels[m==255] = 1
  else:
    labels[m==128] = 1

  return labels
```

```
# load image and annotation map
im = array(Image.open('376043.jpg'))
m = array(Image.open('376043.bmp'))

# resize
scale = 0.1
im = imresize(im,scale,interp='bilinear')
m = imresize(m,scale,interp='nearest')

# create training labels
labels = create_msr_labels(m,False)

# build graph using annotations
g = graphcut.build_bayes_graph(im,labels,kappa=2)

# cut graph
res = graphcut.cut_graph(g,im.shape[:2])

# remove parts in background
res[m==0] = 1
res[m==64] = 1


# plot the result
figure()
imshow(res)
gray()
xticks([])
yticks([])
savefig('labelplot.pdf')
```

First we define a helper function to read the annotation images and format them so we can pass them to our function for training background and foreground models. The bounding rectangles contain only background labels. In this case we set the foreground training region to the whole "unknown" region (the inside of the rectangle). Next we build the graph and cut it. Since we have user input we remove results that have any foreground in the marked background area. Last, we plot the resulting segmentation and remove the tick markers by setting them to an empty list. That way we get a nice bounding box (otherwise the boundaries of the image will be hard to see in this black and white plot).

Figure 9.4 shows some results using RGB vector as feature with the original image, a downsampled mask and downsampled resulting segmentation. The image on the right is the plot generated by the script above.

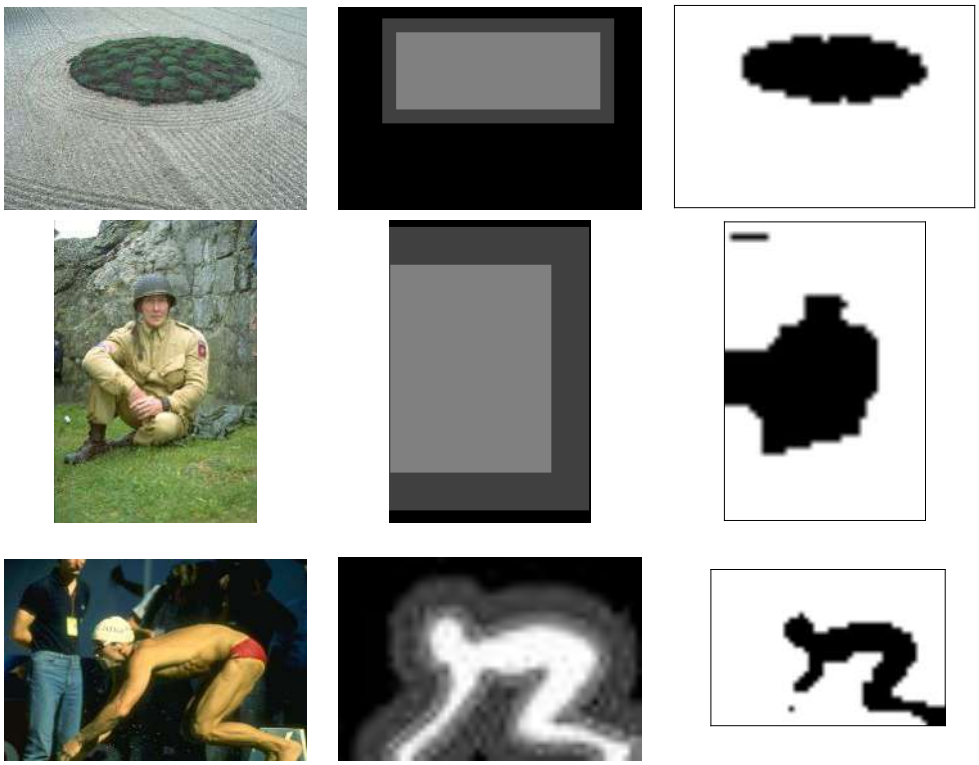


Figure 9.4: Sample graph cut segmentation results using images from the Grab Cut data set. (left) original image, downsampled. (middle) mask used for training. (right) resulting segmentation using RGB values as feature vectors.

9.2 Segmentation using Clustering

The graph cut formulation in the previous section solves the segmentation problem by finding a discrete solution using max flow / min cut over an image graph. In this section we will look at an alternative way to cut the image graph. The *normalized cut* algorithm, based on spectral graph theory, combines pixel similarities with spatial proximity to segment the image.

The idea comes from defining a cut cost that takes into account the size of the groups and "normalizes" the cost with the size of the partitions. The normalized cut formulation modifies the cut cost of equation (9.1) to

$$E_{ncut} = \frac{E_{cut}}{\sum_{i \in A} w_{ix}} + \frac{E_{cut}}{\sum_{j \in B} w_{jx}} \ ,$$

where A and B indicate the two sets of the cut and the sums add the weights from A and B respectively to all other nodes in the graph (which is pixels in the image in this case). This sum is called the *association* and for images where pixels have the same number of connections to other pixels it is a rough measure of the size of the partitions. In the paper [32] the cost function above was introduced together with an algorithm for finding a minimizer. The algorithm is derived for two-class segmentation and will be described next.

Define W as the edge weight matrix with elements w_{ij} containing the weight of the edge connecting pixel i with pixel j. Let D be the diagonal matrix of the row sums of S, $D = \mathrm{diag}(d_i)$, $d_i = \sum_j w_{ij}$ (same as in Section 6.3). The normalized cut segmentation is obtained as the minimum of the following optimization problem

$$\min_{\mathbf{y}} \frac{\mathbf{y}^T(D - W)\mathbf{y}}{\mathbf{y}^T D \mathbf{y}} \ ,$$

where the vector $\mathbf{y}$ contains the discrete labels that satisfy the constraints $y_i \in \{1, -b\}$ for some constant b (meaning that $\mathbf{y}$ only takes two discrete values) and $\mathbf{y}^T D$ sum to zero. Because of these constraints, this is not easily solvable[4].

However, by relaxing the constraints and letting $\mathbf{y}$ take any real value, the problem becomes an eigenvalue problem that is easily solved. The drawback is that you need to threshold or cluster the output to make it a discrete segmentation again.

Relaxing the problem results in solving for eigenvectors of a Laplacian matrix

$$L = D^{-1/2} W D^{-1/2} \ ,$$

just like the spectral clustering case. The only remaining difficulty is now to define the between-pixel edge weights w_{ij}. Normalized cuts have many similarities to spectral

[4] In fact this problem is NP-hard.

clustering and the underlying theory overlaps somewhat, see [32] for an explanation and the details.

Let's use the edge weights from the original normalized cuts paper [32]. The edge weight connecting two pixels i and j is given by

$$w_{ij} = e^{-|I_i - I_j|^2/\sigma_g} \; e^{-|\mathbf{x}_i - \mathbf{x}_j|^2/\sigma_d} \; .$$

The first part measures the pixel similarity between the pixels with I_i and I_j denoting either the RGB vectors or the grayscale values. The second part measures the proximity between the pixels in the image with $\mathbf{x}_i$ and $\mathbf{x}_j$ denoting the coordinate vector of each pixel. The scaling factors σ_g and σ_d determine the relative scales and how fast each component approaches zero.

Let's see what this looks like in code. Add the following function to a file *ncut.py*.

```
def ncut_graph_matrix(im,sigma_d=1e2,sigma_g=1e-2):
  """ Create matrix for normalized cut. The parameters are
    the weights for pixel distance and pixel similarity. """

  m,n = im.shape[:2]
  N = m*n

  # normalize and create feature vector of RGB or grayscale
  if len(im.shape)==3:
    for i in range(3):
      im[:,:,i] = im[:,:,i] / im[:,:,i].max()
    vim = im.reshape((-1,3))
  else:
    im = im / im.max()
    vim = im.flatten()

  # x,y coordinates for distance computation
  xx,yy = meshgrid(range(n),range(m))
  x,y = xx.flatten(),yy.flatten()

  # create matrix with edge weights
  W = zeros((N,N),'f')
  for i in range(N):
    for j in range(i,N):
      d = (x[i]-x[j])**2 + (y[i]-y[j])**2
      W[i,j] = W[j,i] = exp(-1.0*sum((vim[i]-vim[j])**2)/sigma_g) * exp(-d/sigma_d)

  return W
```

This function takes an image array and creates a feature vector using either RGB values or grayscale values depending on the input image. Since the edge weights contain a distance component we use `meshgrid()` to get the x and y values for each

pixel feature vector. Then the function loops over all N pixels and fills out the values in the $N \times N$ normalized cut matrix W.

We can compute the segmentation either by sequentially cutting each eigenvector or by taking a number of eigenvectors and apply clustering. We chose the second approach which also works without modification for any number of segments. We take the top *ndim* eigenvectors of the Laplacian matrix corresponding to W and cluster the pixels. The following function implements the clustering, as you can see it is almost the same as the spectral clustering example in Section 6.3.

```
from scipy.cluster.vq import *

def cluster(S,k,ndim):
  """ Spectral clustering from a similarity matrix."""

  # check for symmetry
  if sum(abs(S-S.T)) > 1e-10:
    print 'not symmetric'

  # create Laplacian matrix
  rowsum = sum(abs(S),axis=0)
  D = diag(1 / sqrt(rowsum + 1e-6))
  L = dot(D,dot(S,D))

  # compute eigenvectors of L
  U,sigma,V = linalg.svd(L)

  # create feature vector from ndim first eigenvectors
  # by stacking eigenvectors as columns
  features = array(V[:ndim]).T

  # k-means
  features = whiten(features)
  centroids,distortion = kmeans(features,k)
  code,distance = vq(features,centroids)

  return code,V
```

Here we used the k-means clustering algorithm (see Section 6.1 for details) to group the pixels based on the values in the eigenvector images. You could try any clustering algorithm or grouping criteria if you feel like experimenting with the results.

Now we are ready to try this on some sample images. The following script shows a complete example:

```
import ncut
from scipy.misc import imresize
```

```
im = array(Image.open('C-uniform03.ppm'))
m,n = im.shape[:2]

# resize image to (wid,wid)
wid = 50
rim = imresize(im,(wid,wid),interp='bilinear')
rim = array(rim,'f')

# create normalized cut matrix
A = ncut.ncut_graph_matrix(rim,sigma_d=1,sigma_g=1e-2)

# cluster
code,V = ncut.cluster(A,k=3,ndim=3)

# reshape to original image size
codeim = imresize(code.reshape(wid,wid),(m,n),interp='nearest')

# plot result
figure()
imshow(codeim)
gray()
show()
```

Here we resize the image to a fixed size (50×50 in this example) in order to make the eigenvector computation fast enough. The NumPy `linalg.svd()` function is not fast enough to handle large matrices (and sometimes gives inaccurate results for too large matrices). We use bilinear interpolation when resizing the image but nearest neighbor interpolation when resizing the resulting segmentation label image since we don't want to interpolate the class labels. Note the use of first reshaping the one-dimensional array to (*wid,wid*) followed by resizing to the original image size.

In the example we used one of the hand gesture images from the Static Hand Posture Database (see Section 8.1 for more details) with $k = 3$. The resulting segmentation is shown in Figure 9.5 together with the first four eigenvectors.

The eigenvectors are returned as the array V in the example and can be visualized as images like this:

```
imshow(imresize(V[i].reshape(wid,wid),(m,n),interp='bilinear'))
```

This will show eigenvector i as an image at the original image size.

Figure 9.6 shows some more examples using the same script above. The airplane image is from the "airplane" category in the Caltech 101 dataset. For these examples we kept the parameters σ_d and σ_g to the same values as above. Changing them can give you smoother more regularized results and quite different eigenvector images. We leave the experimentation to you.

Figure 9.5: Image segmentation using the normalized cuts algorithm. (top) the original image and the resulting three-class segmentation. (bottom) the first four eigenvectors of the graph similarity matrix.

It is worth noting that even for these fairly simple examples a thresholding of the image would not have given the same result, neither would clustering the RGB or graylevel values. This is because neither of these take the pixel neighborhoods into account.

9.3 Variational Methods

In this book you have seen a number of examples of minimizing a cost or energy to solve computer vision problems. In the previous sections it was minimizing the cut in a graph but we also saw examples like the ROF de-noising, k-means and support vector machines. These are examples of optimization problems.

When the optimization is taken over functions, the problems are called *variational problems* and algorithms for solving such problems are called *variational methods*. Let's look at a simple and effective variational model.

The *Chan-Vese segmentation* model [6] assumes a piece-wise constant image model for the image regions to be segmented. Here we will focus on the case of two regions, for example foreground and background, but the model extends to multiple regions as well, see for example [38]. The model can be described as follows.

If we let a collection of curves Γ separate the image into two regions Ω_1 and Ω_2 as

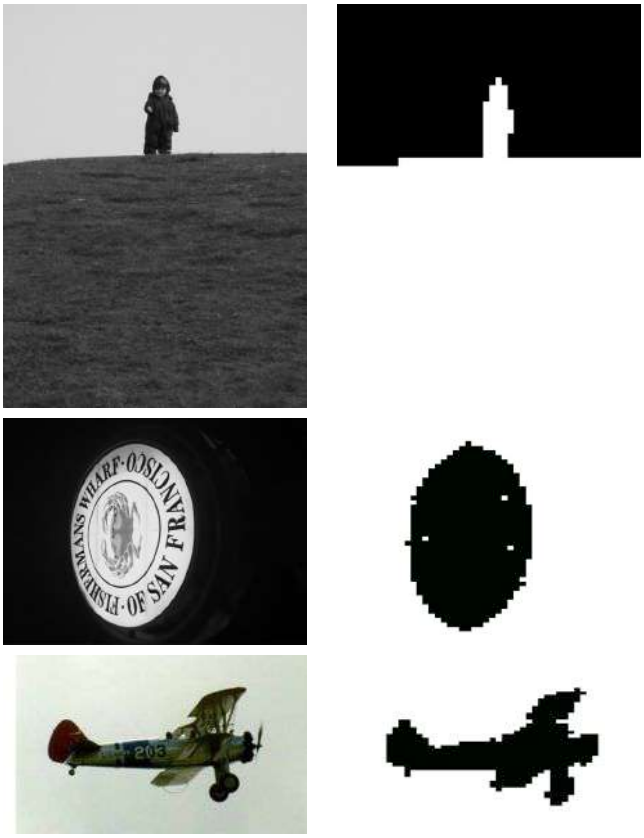


Figure 9.6: Examples of two-class image segmentation using the normalized cuts algorithm. (left) original image. (right) segmentation result.

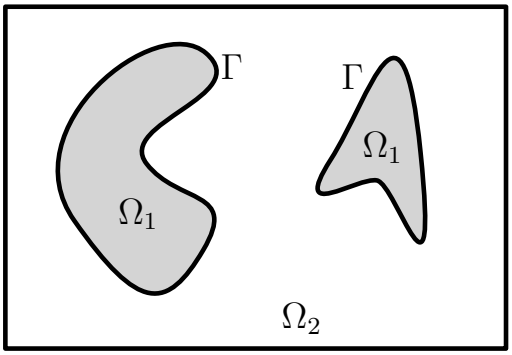


Figure 9.7: The piece-wise constant Chan-Vese segmentation model.

in Figure 9.7 the segmentation is given by minima of the Chan-Vese model energy

$$E(\Gamma) = \lambda \text{ length}(\Gamma) + \int_{\Omega_1} (I - c_1)^2 d\mathbf{x} + \int_{\Omega_2} (I - c_2)^2 d\mathbf{x} \ ,$$

which measures the deviation from the constant graylevels in each region, c_1 and c_2. Here the integrals are taken over each region and the length of the separating curves are there to prefer smoother solutions.

With a piece-wise constant image $U = \chi_1 c_1 + \chi_2 c_2$ this can we re-written as

$$E(\Gamma) = \lambda \frac{|c_1 - c_2|}{2} \int |\nabla U| d\mathbf{x} + ||I - U||^2 \ ,$$

where χ_1 and χ_2 are characteristic (indicator) functions for the two regions[5]. This transformation is non-trivial and requires some heavy mathematics that are not needed for understanding and are well outside the scope of this book.

The point is that this equation is now the same as the ROF equation (1.1) with λ replaced by $\lambda|c_1 - c_2|$. The only difference is that in the Chan-Vese case we are looking for an image U which is piece-wise constant. It can be shown that thresholding the ROF solution will give a good minimizer. The interested reader can check [8] for the details.

Minimizing the Chan-Vese model now becomes a ROF de-noising followed by thresholding.

[5] *Characteristic functions* are 1 in the region and 0 outside.

(a) (b) (c)

Figure 9.8: Examples image segmentation by minimizing the Chan-Vese model using ROF de-noising. (a) original image, (b) image after ROF de-noising. (c) final segmentation.

```
import rof

im = array(Image.open('ceramic-houses_t0.png').convert("L"))
U,T = rof.denoise(im,im,tolerance=0.001)
t = 0.4 #threshold

import scipy.misc
scipy.misc.imsave('result.pdf',U < t*U.max())
```

In this case we turn down the tolerance threshold for stopping the ROF iterations to make sure we get enough iterations. Figure 9.8 shows the result on two rather difficult images.

Exercises

1. It is possible to speed up computation for the graph cut optimization by reducing the number of edges. This graph construction is described in Section 4.2 of [16].

Try this out and measure the difference graph size and in segmentation time compared to the simpler construction we used.

2. Create a user interface or simulate a user selecting regions for graph cut segmentation. Then try "hard coding" background and foreground by setting weights to some large value.

3. Change the feature vector in the graph cut segmentation from a RGB vector to some other descriptor. Can you improve on the segmentation results?

4. Implement an iterative segmentation approach using graph cut where a current segmentation is used to train new foreground and background models for the next. Does it improve segmentation quality?

5. The Microsoft Research Grab Cut dataset contains ground truth segmentation maps. Implement a function that measures the segmentation error and evaluate different settings and some of the ideas in the exercises above.

6. Try to vary the parameters of the normalized cuts edge weight and see how they affect the eigenvector images and the segmentation result.

7. Compute image gradients on the first normalized cuts eigenvectors. Combine these gradient images to detect image contours of objects.

8. Implement a linear search over the threshold value for the de-noised image in Chan-Vese segmentation. For each threshold, store the energy $E(\Gamma)$ and pick the segmentation with the lowest value.

Chapter 10

OpenCV

This chapter gives a brief overview of how to use the popular computer vision library OpenCV through the Python interface. OpenCV is a C++ library for real time computer vision initially developed by Intel, now maintained by Willow Garage. OpenCV is open source and released under a BSD license, meaning it is free for both academic and commercial use. As of version 2.0, Python support has been greatly improved. We will go through some basic examples and look deeper into tracking and video.

10.1 The OpenCV Python Interface

OpenCV is a C++ library with modules that cover many areas of computer vision. Besides C++ (and C) there is growing support for Python as a simpler scripting language through a Python interface on top of the C++ code base. The Python interface is still under development and not all parts of OpenCV are exposed and many functions are undocumented. This is likely to change as there is an active community behind this interface. The Python interface is documented at `http://opencv.willowgarage.com/documentation/python/index.html`. See the appendix for installation instructions.

The current OpenCV version (2.3.1) actually comes with two Python interfaces. The old `cv` module uses internal OpenCV datatypes and can be a little tricky to use from `NumPy`. The new `cv2` module uses `NumPy` arrays and is much more intuitive to use[1]. The module is available as

```
import cv2
```

and the old module can be accessed as

[1]The names and location of these two modules are likely to change over time, check the online documentation for changes.

```
import cv2.cv
```

We will focus on the cv2 module in this chapter. Look out for future name changes, as well as changes in function names and definitions in future versions. OpenCV and the Python interface is under rapid development.

10.2 OpenCV Basics

OpenCV comes with functions for reading and writing images as well as matrix operations and math libraries. For the details on OpenCV, there is an excellent book [3] (C++ only). Let's look at some of the basic components and how to use them.

Reading and writing images

This short example will load an image, print the size and convert and save the image in .png format.

```
import cv2

# read image
im = cv2.imread('empire.jpg')
h,w = im.shape[:2]
print h,w

# save image
cv2.imwrite('result.png',im)
```

The function `imread()` returns the image as a standard `NumPy` array and can handle a wide range of image formats. You can use this function as an alternative to the PIL image reading if you like. The function `imwrite()` automatically takes care of any conversion based on the file ending.

Color spaces

In OpenCV images are not stored using the conventional RGB color channels, they are stored in BGR order (the reverse order). When reading an image the default is BGR, however there are several conversions available. Color space conversion are done using the function `cvtColor()`. For example, converting to grayscale is done like this.

```
im = cv2.imread('empire.jpg')
# create a grayscale version
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
```

After the source image there is an OpenCV color conversion code. Some of the most useful conversion codes are:

- cv2.COLOR_BGR2GRAY
- cv2.COLOR_BGR2RGB
- cv2.COLOR_GRAY2BGR

In each of these, the number of color channels for resulting images will match the conversion code (single channel for gray and three channels for RGB and BGR). The last version converts grayscale images to BGR and is useful if you want to plot or overlay colored objects on the images. We will use this in the examples.

Displaying images and results

Let's look at some examples of using OpenCV for image processing and how to show results with OpenCV plotting and window management.

The first example reads an image from file and creates an integral image representation.

```
import cv2

# read image
im = cv2.imread('fisherman.jpg')
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

# compute integral image
intim = cv2.integral(gray)

# normalize and save
intim = (255.0*intim) / intim.max()
cv2.imwrite('result.jpg',intim)
```

After reading the image and converting to grayscale the function `integral()` creates an image where the value at each pixel is the sum of the intensities above and to the left. This is a very useful trick for quickly evaluating features. Integral images are used in OpenCV's `CascadeClassifier` which is based on a framework introduced by Viola and Jones [39]. Before saving the resulting image, we normalize the values to $0 \ldots 255$ by dividing with the largest value. Figure 10.1 shows the result for an example image.

The second example applies flood filling starting from a seed pixel.

```
import cv2
```

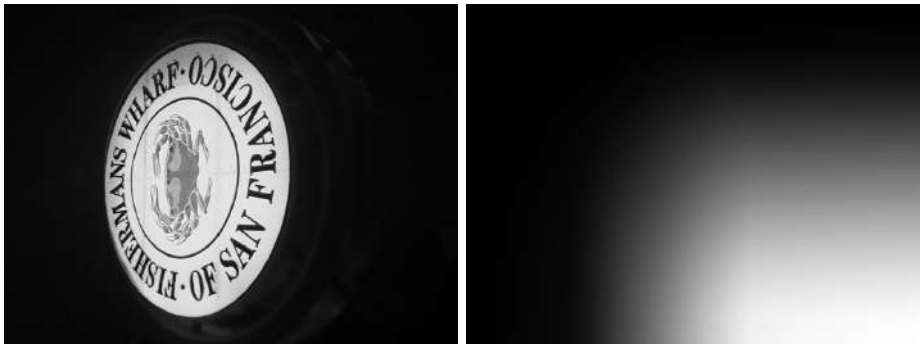


Figure 10.1: Example of computing an integral image using OpenCV's `integral()` function.

```
# read image
filename = 'fisherman.jpg'
im = cv2.imread(filename)
h,w = im.shape[:2]

# flood fill example
diff = (6,6,6)
mask = zeros((h+2,w+2),uint8)
cv2.floodFill(im,mask,(10,10), (255,255,0),diff,diff)

# show the result in an OpenCV window
cv2.imshow('flood fill',im)
cv2.waitKey()

# save the result
cv2.imwrite('result.jpg',im)
```

This example applies flood fill to the image and shows the result in an OpenCV window. The function `waitKey()` pauses until a key is pressed and the window is automatically closed. Here the function `floodFill()` takes the image (grayscale or color), a mask with non-zero pixels indicating areas not to be filled, a seed pixel, the new color value to replace the flooded pixels together with lower and upper difference thresholds to accept new pixels. The flood fill starts at the seed pixel and keeps expanding as long as new pixels can be added within the difference thresholds. The difference thresholds are given as tuples (R,G,B). The result looks like Figure 10.2.

As a third and final example, we look at extracting SURF features, a faster version of SIFT introduced by [1]. Here we also show how to use some basic OpenCV plotting commands.

Figure 10.2: Flood fill of a color image. The cyan area marks all pixels filled using a single seed in the upper left corner.

```
import cv2

# read image
im = cv2.imread('empire.jpg')

# down sample
im_lowres = cv2.pyrDown(im)

# convert to grayscale
gray = cv2.cvtColor(im_lowres,cv2.COLOR_RGB2GRAY)

# detect feature points
s = cv2.SURF()
mask = uint8(ones(gray.shape))
keypoints = s.detect(gray,mask)

# show image and points
vis = cv2.cvtColor(gray,cv2.COLOR_GRAY2BGR)

for k in keypoints[::10]:
  cv2.circle(vis,(int(k.pt[0]),int(k.pt[1])),2,(0,255,0),-1)
  cv2.circle(vis,(int(k.pt[0]),int(k.pt[1])),int(k.size),(0,255,0),2)

cv2.imshow('local descriptors',vis)
cv2.waitKey()
```

After reading the image it is down sampled using the function `pyrDown()` which if no new size is given, creates a new image half the size of the original. Then the image is converted to grayscale and passed to a `SURF` keypoint detection object. The *mask* determines what areas to apply the keypoint detector. When it comes to plotting we

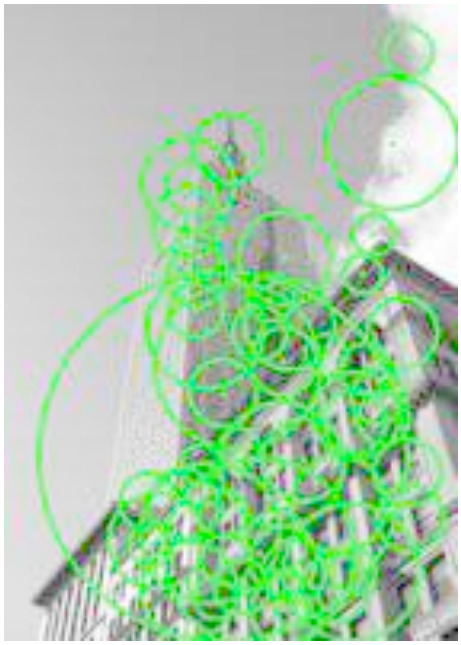


Figure 10.3: Sample SURF features extracted and plotted using OpenCV.

convert the grayscale image to a color image and use the green channel for plotting. We loop over every tenth keypoint and plot a circle at the center and one circle showing the scale (size) of the keypoint. The plotting function `circle()` takes an image, a tuple with image coordinates (integer only), a radius, a tuple with plot color and finally the line thickness (-1 gives a solid circle). Figure 10.3 shows the result.

10.3 Processing Video

Video with pure Python is hard. There is speed, codecs, cameras, operating systems and file formats to consider. There is currently no video library for Python. OpenCV with its Python interface is the only good option. In this section we'll look at some basic examples using video.

Video input

Reading video from a camera is very well supported in OpenCV. A basic complete example that captures frames and shows them in an OpenCV window looks like this.

```
import cv2
```

```
# setup video capture
cap = cv2.VideoCapture(0)

while True:
  ret,im = cap.read()
  cv2.imshow('video test',im)
  key = cv2.waitKey(10)
  if key == 27:
    break
  if key == ord(' '):
    cv2.imwrite('vid_result.jpg',im)
```

The capture object `VideoCapture` captures video from cameras or files. Here we pass an integer at initialization. This is the id of the video device, with a single camera connected this is 0. The method `read()` decodes and returns the next video frame. The first value is a success flag and the second the actual image array. The `waitKey()` function waits for a key to be pressed and quit the application if the 'esc' key (Ascii number 27) is pressed or saves the frame if the 'space' key is pressed.

Let's extend this example with some simple processing by taking the camera input and show a blurred (color) version of the input in an OpenCV window. This is only a slight modification to the base example above:

```
import cv2

# setup video capture
cap = cv2.VideoCapture(0)

# get frame, apply Gaussian smoothing, show result
while True:
  ret,im = cap.read()
  blur = cv2.GaussianBlur(im,(0,0),5)
  cv2.imshow('camera blur',blur)
  if cv2.waitKey(10) == 27:
    break
```

Each frame is passed to the function `GaussianBlur()` which applies a Gaussian filter to the image. In this case we are passing a color image so each color channel is blurred separately. The function takes a tuple for filter size and the standard deviation for the Gaussian function (in this case 5). If the filter size is set to zero, it will automatically be determined from the standard deviation. The result looks like Figure 10.4.

Reading video from files works the same way but with the call to `VideoCapture()` taking the video filename as input.

```
capture = cv2.VideoCapture('filename')
```

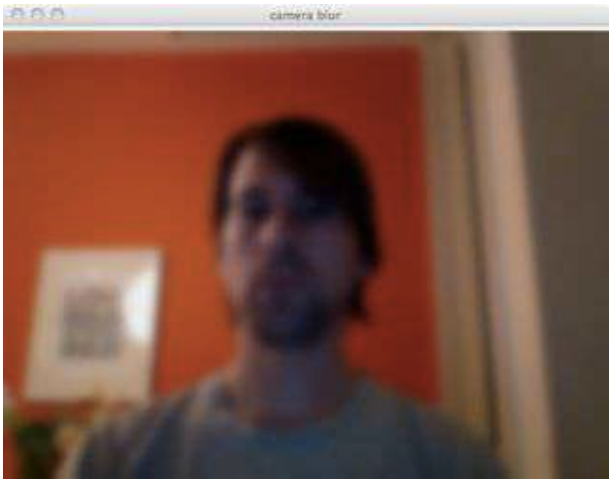


Figure 10.4: Screenshot of a blurred video of the author as he's writing this chapter.

Reading video to NumPy arrays

Using OpenCV it is possible to read video frames from a file and convert them to NumPy arrays. Here is an example of capturing video from a camera and storing the frames in a NumPy array.

```
import cv2

# setup video capture
cap = cv2.VideoCapture(0)

frames = []
# get frame, store in array
while True:
  ret,im = cap.read()
  cv2.imshow('video',im)
  frames.append(im)
  if cv2.waitKey(10) == 27:
    break
frames = array(frames)

# check the sizes
print im.shape
print frames.shape
```

Each frame array is added to the end of a list until the capturing is stopped. The resulting array will have size (number of frames,height,width,3). The printout confirms

this:

```
(480, 640, 3)
(40, 480, 640, 3)
```

In this case there were 40 frames recorded. Arrays with video data like this are useful for video processing such as computing frame differences and tracking.

10.4 Tracking

Optical flow

Optical flow (sometimes called *optic flow*) is the image motion of objects as the objects, scene or camera moves between two consecutive images. It is a 2D vector field of within-image translation. Is is a classic and well studied field in computer vision with many successful applications in for example video compression, motion estimation, object tracking and image segmentation.

Optical flow relies on three major assumptions.

1. *Brightness constancy:* The pixel intensities of an object in an image does not change between consecutive images.

2. *Temporal regularity:* The between-frame time is short enough to consider the motion change between images using differentials (used to derive the central equation below).

3. *Spatial consistency:* Neighboring pixels have similar motion.

In many cases these assumptions break down, but for small motions and short time steps between images it is a good model. Assuming that an object pixel $I(x, y, t)$ at time t has the same intensity at time $t + \delta t$ after motion $[\delta x, \delta y])$ means that $I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t)$. Differentiating this constraint gives the *optical flow equation*:

$$\nabla I^T \mathbf{v} = -I_t \ ,$$

where $\mathbf{v} = [u, v]$ is the motion vector and I_t the time derivative. For individual points in the image, this equation is under-determined and cannot be solved (one equation with two unknowns in $\mathbf{v}$). By enforcing some spatial consistency, it is possible to obtain solutions though. In the Lucas-Kanade algorithm below we will see how that assumption is used.

OpenCV contains several optical flow implementations, `CalcOpticalFlowBM()` which uses block matching, `CalcOpticalFlowHS()` which uses [15] (both of these currently only in the old `cv` module), the pyramidal Lucas-Kanade algorithm [19] `calcOpticalFlowPyrLK()`

and finally `calcOpticalFlowFarneback()` based on [10]. The last one is considered one of the best methods for obtaining dense flow fields. Let's look at an example of using this to find motion vectors in video (the Lucas-Kanade version is the subject of the next section).

Try running the following script.

```
import cv2

def draw_flow(im,flow,step=16):
  """ Plot optical flow at sample points
    spaced step pixels apart. """

  h,w = im.shape[:2]
  y,x = mgrid[step/2:h:step,step/2:w:step].reshape(2,-1)
  fx,fy = flow[y,x].T

  # create line endpoints
  lines = vstack([x,y,x+fx,y+fy]).T.reshape(-1,2,2)
  lines = int32(lines)

  # create image and draw
  vis = cv2.cvtColor(im,cv2.COLOR_GRAY2BGR)
  for (x1,y1),(x2,y2) in lines:
    cv2.line(vis,(x1,y1),(x2,y2),(0,255,0),1)
    cv2.circle(vis,(x1,y1),1,(0,255,0), -1)
  return vis


# setup video capture
cap = cv2.VideoCapture(0)

ret,im = cap.read()
prev_gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

while True:
  # get grayscale image
  ret,im = cap.read()
  gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)

  # compute flow
  flow = cv2.calcOpticalFlowFarneback(prev_gray,gray,None,0.5,3,15,3,5,1.2,0)
  prev_gray = gray

  # plot the flow vectors
  cv2.imshow('Optical flow',draw_flow(gray,flow))
  if cv2.waitKey(10) == 27:
    break
```

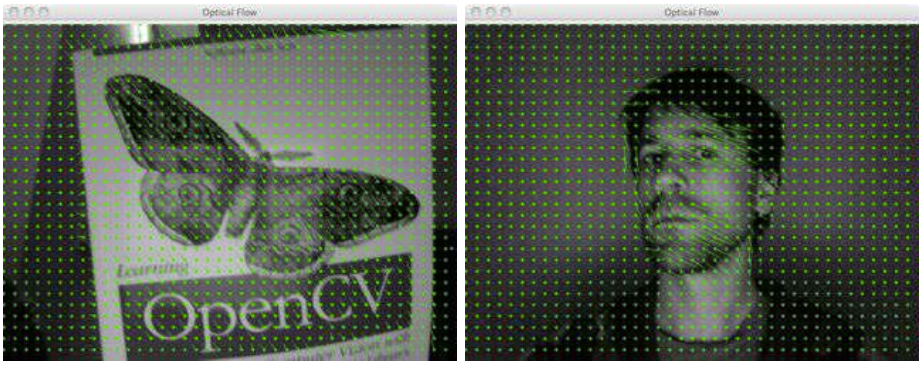


Figure 10.5: Optical flow vectors (sampled at every 16th pixel) shown on video of a translating book and a turning head.

This example will capture images from a webcam and call the optical flow estimation on every consecutive pair of images. The motion flow vectors are stored in the two-channel image *flow* returned by `calcOpticalFlowFarneback()`. Besides the previous frame and the current frame, this function takes a sequence of parameters. Look them up in the documentation if you are interested. The helper function `draw_flow()` plots the motion vectors at regularly sample points in the image. It uses the OpenCV drawing functions `line()` and `circle()` and the variable *step* controls the spacing of the flow samples. The result can look like the screenshots in Figure 10.5. Here the positions of the flow samples are shown as a grid of green circles and the flow vectors with lines show how each sample point moves.

The Lucas-Kanade algorithm

Tracking is the process of following objects through a sequence of images or video. The most basic form of tracking is to follow interest points such as corners. A popular algorithm for this is the *Lucas-Kanade tracking algorithm* which uses a sparse optical flow algorithm.

Lucas-Kanade tracking can be applied to any type of features but usually makes use of corner points similar to the Harris corner points in Section 2.1. The function `goodFeaturesToTrack()` detects corners according to an algorithm by Shi and Tomasi [33] where corners are points with two large eigenvalues of the structure tensor (Harris matrix) equation (2.2) and where the smaller eigenvalue is above a threshold.

The optical flow equation is under-determined (meaning that there are too many unknowns per equation) if considered on a per-pixel basis. Using the assumption that neighboring pixels have the same motion it is possible to stack many of these equations

into one system of equations like this

$$\begin{bmatrix} \nabla I^T(\mathbf{x}_1) \\ \nabla I^T(\mathbf{x}_2) \\ \dots \\ \nabla I^T(\mathbf{x}_n) \end{bmatrix} \mathbf{v} = \begin{bmatrix} I_x(\mathbf{x}_1) & I_y(\mathbf{x}_1) \\ I_x(\mathbf{x}_2) & I_y(\mathbf{x}_2) \\ \dots & \\ I_x(\mathbf{x}_n) & I_y(\mathbf{x}_n) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(\mathbf{x}_1) \\ I_t(\mathbf{x}_2) \\ \dots \\ I_t(\mathbf{x}_n) \end{bmatrix}$$

for some neighborhood of n pixels. This has the advantage that the system now has more equations than unknowns and can be solved with least square methods. Typically, the contribution from the surrounding pixels is weighted so that pixels farther away have less influence. A Gaussian weighting is the most common choice. This turns the matrix above into the structure tensor in equation (2.2) and we have the relation

$$\overline{\mathbf{M}}_I \mathbf{v} = - \begin{bmatrix} I_t(\mathbf{x}_1) \\ I_t(\mathbf{x}_2) \\ \dots \\ I_t(\mathbf{x}_n) \end{bmatrix} \quad \text{or simpler} \quad A\mathbf{v} = \mathbf{b} \ .$$

This over-determined equation system can be solved in a least square sense and the motion vector is given by

$$\mathbf{v} = (A^T A)^{-1} A^T \mathbf{b} \ .$$

This is solvable only when $A^T A$ is invertible, which it is by construction if applied at Harris corner points or the "good features to track" of Shi-Tomasi. This is how the motion vectors are computed in the Lucas-Kanade tracking algorithms.

Standard Lucas-Kanade tracking works for small displacements. To handle larger displacements a hierarchical approach is used. In this case the optical flow is computed at coarse to fine versions of the image. This is what the OpenCV function `calcOpticalFlowPyrLK()` does.

The Lucas-Kanade functions are included in OpenCV. Let's look at how to use those to build a Python tracker class. Create a file *lktrack.py* and add the following class and constructor.

```
import cv2

# some constants and default parameters
lk_params = dict(winSize=(15,15),maxLevel=2,
        criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,10,0.03))

subpix_params = dict(zeroZone=(-1,-1),winSize=(10,10),
           criteria = (cv2.TERM_CRITERIA_COUNT | cv2.TERM_CRITERIA_EPS,20,0.03))

feature_params = dict(maxCorners=500,qualityLevel=0.01,minDistance=10)
```

```
class LKTracker(object):
  """ Class for Lucas-Kanade tracking with
    pyramidal optical flow."""

  def __init__(self,imnames):
    """ Initialize with a list of image names. """

    self.imnames = imnames
    self.features = []
    self.tracks = []
    self.current_frame = 0
```

The tracker object is initialized with a list of filenames. The variables *features* and *tracks* will hold the corner points and their tracked positions. We also use a variable to keep track of the current frame. We define three dictionaries with parameters for the feature extraction, the tracking, and the subpixel feature point refinement.

Now, to start detecting points, we need to load the actual image, create grayscale version and extract the "good features to track" points. The OpenCV function doing the main work is `goodFeaturesToTrack()`. Add this `detect_points()` method to the class.

```
def detect_points(self):
  """ Detect 'good features to track' (corners) in the current frame
    using sub-pixel accuracy. """

  # load the image and create grayscale
  self.image = cv2.imread(self.imnames[self.current_frame])
  self.gray = cv2.cvtColor(self.image,cv2.COLOR_BGR2GRAY)

  # search for good points
  features = cv2.goodFeaturesToTrack(self.gray, **feature_params)

  # refine the corner locations
  cv2.cornerSubPix(self.gray,features, **subpix_params)

  self.features = features
  self.tracks = [[p] for p in features.reshape((-1,2))]

  self.prev_gray = self.gray
```

The point locations are refined using `cornerSubPix()` and stored in the member variables *features* and *tracks*. Note that running this function clears the track history.

Now that we can detect the points, we also need to track them. First we need to get the next frame, then apply the OpenCV function `calcOpticalFlowPyrLK()` that finds out where the points moved and remove and clean the lists of tracked points.

The method `track_points()` below does this.

```
def track_points(self):
  """ Track the detected features. """

  if self.features != []:
    self.step() # move to the next frame

    # load the image and create grayscale
    self.image = cv2.imread(self.imnames[self.current_frame])
    self.gray = cv2.cvtColor(self.image,cv2.COLOR_BGR2GRAY)

    # reshape to fit input format
    tmp = float32(self.features).reshape(-1, 1, 2)

    # calculate optical flow
    features,status,track_error = cv2.calcOpticalFlowPyrLK(self.prev_gray,
                                self.gray,tmp,None,**lk_params)

    # remove points lost
    self.features = [p for (st,p) in zip(status,features) if st]

    # clean tracks from lost points
    features = array(features).reshape((-1,2))
    for i,f in enumerate(features):
      self.tracks[i].append(f)
    ndx = [i for (i,st) in enumerate(status) if not st]
    ndx.reverse() #remove from back
    for i in ndx:
      self.tracks.pop(i)

    self.prev_gray = self.gray
```

This makes use of a simple helper method `step()` that moves to the next available frame.

```
def step(self,framenbr=None):
  """ Step to another frame. If no argument is
    given, step to the next frame. """

  if framenbr is None:
    self.current_frame = (self.current_frame + 1) % len(self.imnames)
  else:
    self.current_frame = framenbr % len(self.imnames)
```

This method jumps to a given frame or just to the next if no argument is given.

Finally, we also want to be able to draw the result using OpenCV windows and drawing functions. Add this `draw()` method to the `LKTracker` class.

```
def draw(self):
  """ Draw the current image with points using
    OpenCV's own drawing functions.
    Press ant key to close window."""

  # draw points as green circles
  for point in self.features:
    cv2.circle(self.image,(int(point[0][0]),int(point[0][1])),3,(0,255,0),-1)

  cv2.imshow('LKtrack',self.image)
  cv2.waitKey()
```

Now we have a complete self-contained tracking system using OpenCV functions.

Using the tracker Let's tie it all together by using this tracker class on a real tracking scenario. The following script will initialize a tracker object, detect and track points through the sequence and draw the result.

```
import lktrack

imnames = ['bt.003.pgm', 'bt.002.pgm', 'bt.001.pgm', 'bt.000.pgm']

# create tracker object
lkt = lktrack.LKTracker(imnames)

# detect in first frame, track in the remaining
lkt.detect_points()
lkt.draw()
for i in range(len(imnames)-1):
  lkt.track_points()
  lkt.draw()
```

The drawing is one frame at a time and show the points currently tracked. Pressing any key will move to the next image in the sequence. The resulting figure windows for the first four images of the Oxford corridor sequence (one of the Oxford multi-view datasets available at `http://www.robots.ox.ac.uk/~vgg/data/data-mview.html`) looks like Figure 10.6.

Using generators Add the following method to the `LKTracker` class.

```
def track(self):
  """ Generator for stepping through a sequence."""

  for i in range(len(self.imnames)):
    if self.features == []:
```

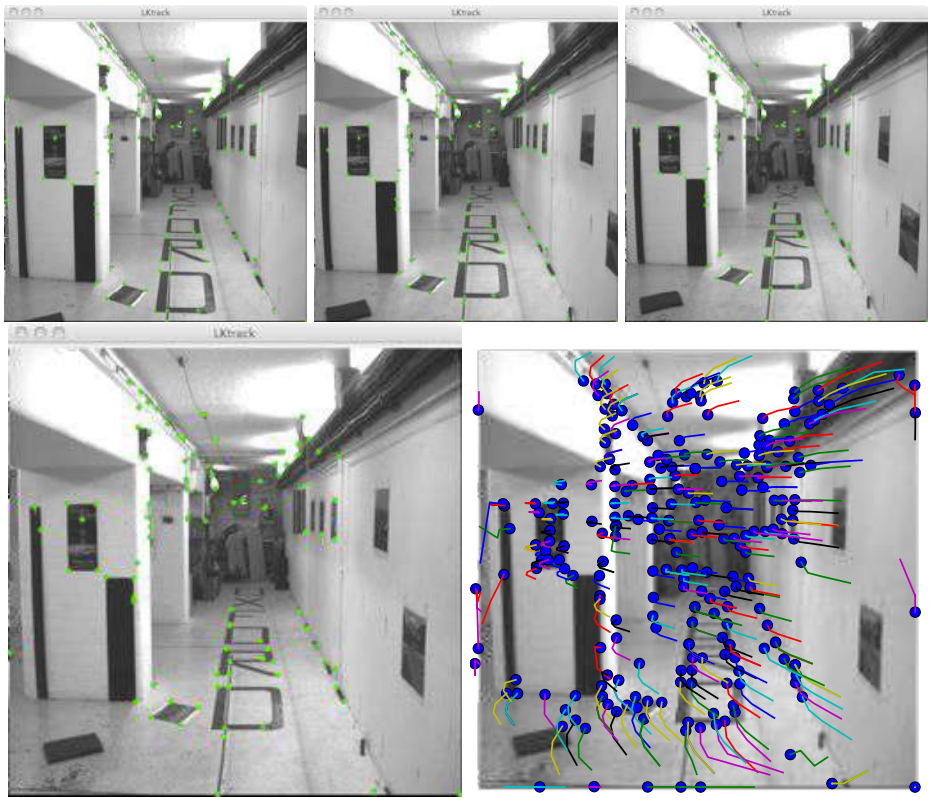


Figure 10.6: Tracking using the Lucas-Kanade algorithm through the LKTrack class.

```
      self.detect_points()
    else:
      self.track_points()

    # create a copy in RGB
    f = array(self.features).reshape(-1,2)
    im = cv2.cvtColor(self.image,cv2.COLOR_BGR2RGB)
    yield im,f
```

This creates a generator which makes it easy to step through a sequence and get tracks and the images as RGB arrays so that it is easy to plot the result. To use it on the classic Oxford "dinosaur" sequence (from the same multi-view dataset page as the corridor above) and plot the points and their tracks, the code looks like this:

```
import lktrack

imnames = ['viff.000.ppm', 'viff.001.ppm',
    'viff.002.ppm', 'viff.003.ppm', 'viff.004.ppm']

# track using the LKTracker generator
lkt = lktrack.LKTracker(imnames)
for im,ft in lkt.track():
  print 'tracking %d features' % len(ft)

# plot the tracks
figure()
imshow(im)
for p in ft:
  plot(p[0],p[1],'bo')
for t in lkt.tracks:
  plot([p[0] for p in t],[p[1] for p in t])
axis('off')
show()
```

This generator makes it really easy to use the tracker class and completely hides the OpenCV functions from the user. The example generates a plot like the one shown in Figure 10.7 and the bottom right of Figure 10.6.

10.5 More Examples

With OpenCV comes a number of useful sample examples of how to use the python interface. These are in the sub-directory samples/python2/ and are a good way to get familiar with OpenCV. Here are a few selected examples to illustrate some other capabilities of OpenCV.

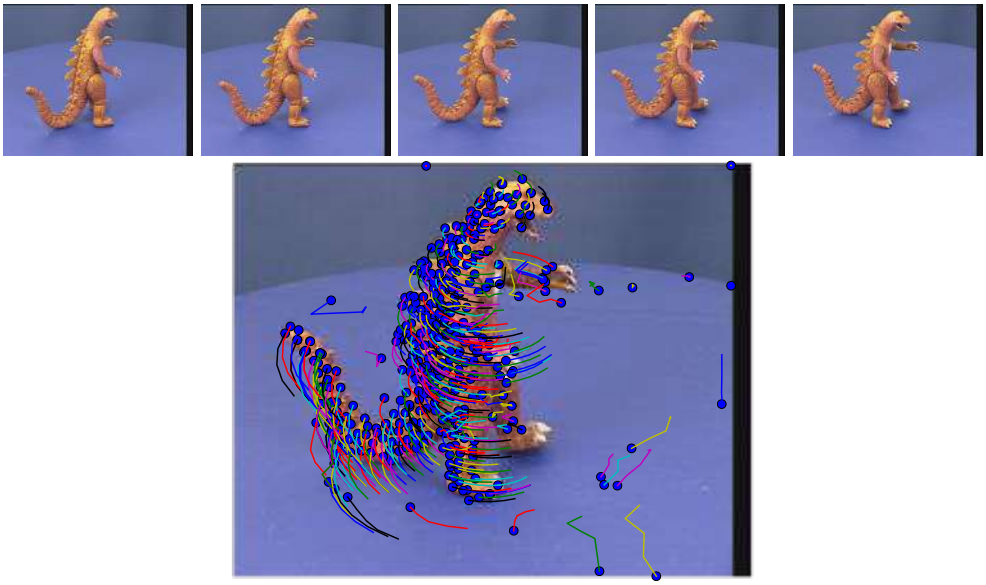


Figure 10.7: An example of using Lucas-Kanade tracking on a turntable sequence and plotting the tracks of points.

Inpainting

The reconstruction of lost or deteriorated parts of images is called *inpainting*. This covers both algorithms to recover lost or corrupted parts of image data for restoration purposes as well as removing red-eyes or objects in photo editing applications. Typically a region of the image is marked as "corrupt" and needs to be filled using the data from the rest of the image.

Try the following command:

```
$ python inpaint.py empire.jpg
```

This will open an interactive window where you can draw regions to be inpainted. The results are shown in a separate window. An example is shown in Figure 10.8.

Segmentation with the watershed transform

Watershed is an image processing technique that can be used for segmentation. A (graylevel) image is treated as a topological landscape that is "flooded" from a number of seed regions. Usually a gradient magnitude image is used since this has ridges at strong edges and will make the segmentation stop at image edges.

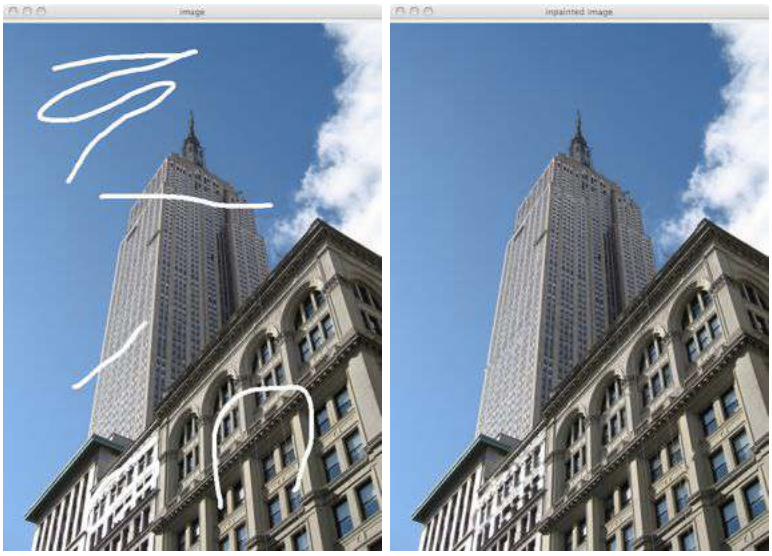


Figure 10.8: An example of inpainting with OpenCV. The left image shows areas marked by a user as "corrupt". The right image shows the result after inpainting.

The implementation in OpenCV uses an algorithm by Meyer [22]. Try it using the command:

```
$ python watershed.py empire.jpg
```

This will open an interactive window where you can draw the seed regions you want the algorithm to use as input. The results are shown in a second window with colors representing regions overlaid on a grayscale version of the input image.

Line detection with a Hough transform

The *Hough transform* (`http://en.wikipedia.org/wiki/Hough_transform`) is a method for finding shapes in images. It works by using a voting procedure in the parameter space of the shapes. The most common use is to find line structures in images. In that case edges and line segments can be grouped together by them voting for the same line parameters in the 2D parameter space of lines.

The OpenCV sample detects lines using this approach[2]. Try the following command:

```
$ python houghlines.py empire.jpg
```

[2]This sample is currently in the /samples/python folder.

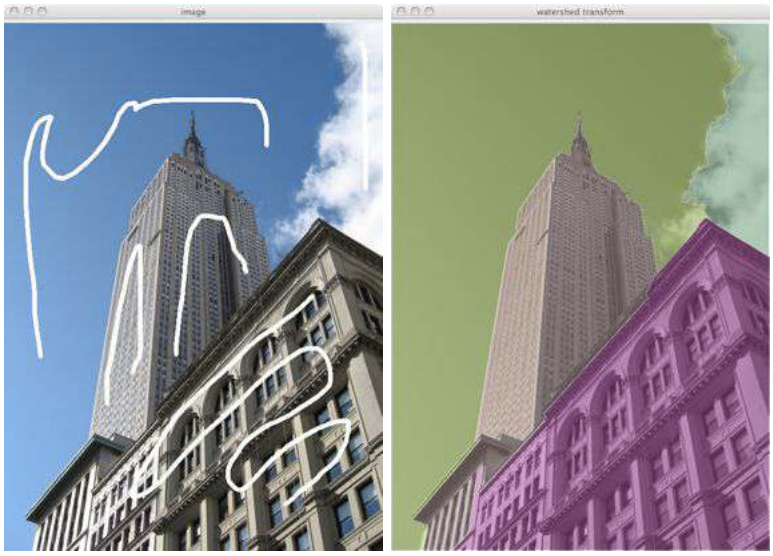


Figure 10.9: An example of segmenting an image using a watershed transform. The left image is the input image with seed regions drawn. The right image shows the resulting segmentation starting.

This gives two windows like the ones shown in Figure 10.10. One window shows the source image in grayscale, the other shows the edge map used together with lines detected as those with most votes in parameter space. Note that the lines are always infinite, if you want to find the endpoints of line segments in the image you can use the edge map to try to find them.

Exercises

1. Use optical flow to build a simple gesture recognition system. For example, you could sample the flow as in the plotting function and use these sample vectors as input.

2. There are two warp functions available in OpenCV, `cv2.warpAffine()` and `cv2.warpPerspective()` Try to use them on some of the examples from Chapter 3.

3. Use the flood fill function to do background subtraction on the Oxford "dinosaur" images used in Figure 10.7. Create new images with the dinosaur placed on a different color background or on a different image.

Figure 10.10: An example of detecting lines using a Hough transform. The left image is the source in grayscale. The right image shows an edge map with detected lines in red.

4. OpenCV has a function `cv2.findChessboardCorners()` which automatically finds the corners of a chessboard pattern. Use this function to get correspondences for calibrating a camera with the function `cv2.calibrateCamera()`.

5. If you have two cameras, mount them in a stereo rig setting and capture stereo image pairs using `cv2.VideoCapture()` with different video device ids. Try 0 and 1 for starters. Compute depth maps for some varying scenes.

6. Use Hu moments with `cv2.HuMoments()` as features for the sudoku OCR classification problem in Section 8.4 and check the performance.

7. OpenCV has an implementation of the Grab Cut segmentation algorithm. Use the function `cv2.grabCut()` on the Microsoft Research Grab Cut dataset (see Section 9.1). Hopefully you will get better results that the low resolution segmentation in our examples.

8. Modify the Lucas-Kanade tracker class to take a video file as input and write a script that tracks points between frames and detects new points every k frames.

Appendix A

Installing Packages

Here are short installation instructions for the packages used in the book. They are written based on the latest versions as of writing of this book. Things change (urls change!), so if the instructions become outdated, check the individual project websites for help.

In addition to the specific instructions, an option that often works on most platforms is Python's easy_install. If you run into problems with the installation instructions given here, easy_install is worth a try. Find out more on the package website `http://packages.python.org/distribute/easy_install.html`.

A.1 `NumPy` and `SciPy`

Installing `NumPy` and `SciPy` is a little different depending on your operating system. Follow the applicable instructions below. The current versions are 2.0 (`NumPy`) and 0.11 (`SciPy`) on most platforms. A package that currently works on all major platforms is the Enthought EPD Free bundle, a free light version of the commercial Enthought distribution, available for free at `http://enthought.com/products/epd_free.php`.

Windows

The easiest way to install `NumPy` and `SciPy` is to download and install the binary distributions from `http://www.scipy.org/Download`.

Mac OS X

Later versions of Mac OS X (10.7.0 (Lion) and up) comes with `NumPy` pre-installed.

An easy way to install NumPy and SciPy for Mac OS X is with the "superpack" from https://github.com/fonnesbeck/ScipySuperpack. This also gives you Matplotlib.

Another alternative is to use the package system MacPorts (http://www.macports.org/). This also works for Matplotlib instead of the instructions below.

If none of those work, the project webpage has other alternatives listed (http://scipy.org/).

Linux

Installation requires that you have administrator rights on your computer. On some distributions NumPy comes pre-installed, on others not. Both NumPy and SciPy is easiest installed with the built in package handler (for example Synaptic on Ubuntu). You can also use the package handler for Matplotlib instead of the instructions below.

A.2 Matplotlib

Here are instructions for installing Matplotlib in case your NumPy/SciPy installation did not also install Matplotlib. Matplotlib is freely available at http://matplotlib.sourceforge.net/. Click the "download" link and download the installer for the latest version for your system and Python version. Currently the latest version is 1.1.0.

Alternatively, just download the source and unpack. Run

```
$ python setup.py install
```

from the command line and everything should work. General tips on installing for different systems can be found at http://matplotlib.sourceforge.net/users/installing.html but the process above should work for most platforms and Python versions.

A.3 PIL

PIL, the Python Imaging Library is available at http://www.pythonware.com/products/pil/. The latest free version is 1.1.7. Download the source kit and unpack the folder. In the downloaded folder run

```
$ python setup.py install
```

from the command line.

You need to have JPEG (libjpeg) and PNG (zlib) supported if you want to save images using PIL. See the README file or the PIL website if you encounter any problems.

A.4 LibSVM

The current release is version 3.1 (released April 2011). Download the zip file from the LibSVM website `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`. Unzip the file (a directory "libsvm-3.1" will be created). In a terminal window go to this directory and type "make".

```
$ cd libsvm-3.0
$ make
```

Then go to the "python" directory and do the same:

```
$ cd python/
$ make
```

This should be all you need to do. To test your installation, start python from the command line and try

```
import svm
```

The authors wrote a practical guide for using LivSVM [7]. This is a good starting point.

A.5 OpenCV

Installing OpenCV is a bit different depending on your operating system. Follow the applicable instructions below.

To check your installation, start python and try the cookbook examples `http://opencv.willowgarage.com/documentation/python/cookbook.html`. The online OpenCV Python reference guide gives more examples and details `http://opencv.willowgarage.com/documentation/python/index.html` on how to use OpenCV with Python.

Windows and Unix

There are installers for Windows and Unix available at the SourceForge repository `http://sourceforge.net/projects/opencvlibrary/`.

Mac OS X

Mac OS X support has been lacking but is on the rise. There are several ways to install from source as described on the OpenCV wiki `http://opencv.willowgarage.com/wiki/InstallGuide`. MacPorts is one option that works well if you are using Python/Numpy/Scipy/Matplotlib also from MacPorts. Building OpenCV from source can be done like this:

```
$ svn co https://code.ros.org/svn/opencv/trunk/opencv
$ cd opencv/
$ sudo cmake -G "Unix Makefiles" .
$ sudo make -j8
$ sudo make install
```

If you have all the dependencies in place, everything should build and install properly. If you get an error like

```
import cv2
Traceback (most recent call last):
  File "", line 1, in
ImportError: No module named cv2
```

then you need to add the directory containing cv2.so to PYTHONPATH. For example:

```
$ export PYTHONPATH=/usr/local/lib/python2.7/site-packages/
```

Linux

Linux users could try the package installer for the distribution (the package is usually called "opencv") or install from source as described in the Mac OS X section.

A.6 VLFeat

To install VLFeat, download and unpack the latest binary package from `http://vlfeat.org/download.html` (currently the latest version is 0.9.14). Add the paths to your environment or copy the binaries to a directory in your path. The binaries are in the bin/ directory, just pick the sub-directory for your platform.

The use of the VLFeat command line binaries is described in the src/ sub-directory. Alternatively you can find the documentation online at `http://vlfeat.org/man/man.html`.

A.7 PyGame

PyGame can be downloaded from `http://www.pygame.org/download.shtml`. The latest version is 1.9.1. The easiest way is to get the binary install package for your system and Python version.

Alternatively, you can download the source and in the downloaded folder do:

```
$ python setup.py install
```

from the command line.

A.8 PyOpenGL

Installing PyOpenGL is easiest done by downloading the package from `http://pypi.python.org/pypi/PyOpenGL` as suggested on the PyOpenGL webpage `http://pyopengl.sourceforge.net/`. Get the latest version, currently 3.0.1.

In the downloaded folder do the usual:

```
$ python setup.py install
```

from the command line. If you get stuck or need information on dependencies etc, more documentation can be found at `http://pyopengl.sourceforge.net/documentation/installation.html`. Some good demo scripts for getting started are available at `http://pypi.python.org/pypi/PyOpenGL-Demo`.

A.9 Pydot

Begin by installing the dependencies, GraphViz and Pyparsing. Go to `http://www.graphviz.org/` and download the latest GraphViz binary for your platform. The install files should install GraphViz automatically.

Then, go to the Pyparsing project page `http://pyparsing.wikispaces.com/`. The download page is at `http://sourceforge.net/projects/pyparsing/`. Get the latest version (currently 1.5.5) and unzip the file to a directory. Type

```
$ python setup.py install
```

from the command line.

Finally, go to the project page `http://code.google.com/p/pydot/` and click "download". From the download page, download the latest version (currently 1.0.4). Unzip and again type

```
$ python setup.py install
```

from the command line. Now you should be able to import pydot in your python sessions.

A.10 Python-graph

Python-graph is a python module for working with graphs and contains lots of useful algorithms like traversals, shortest path, pagerank and maximum flow. The latest version is 1.8.1 and can be found on the project website `http://code.google.com/p/python-graph/`. If you have easy_install on your system, the simplest way to get python-graph is:

```
$ easy_install python-graph-core
```

Alternatively, download the source code from `http://code.google.com/p/python-graph/downloads/list` and run

```
$ python setup.py install
```

To write and visualize the graphs (using the DOT language) you need python-graph-dot which comes with the download or through easy_install:

```
$ easy_install python-graph-dot
```

Python-graph-dot depends on pydot, see above. The documentation (in html) is in the "docs/" folder.

A.11 Simplejson

Simplejson is the independently maintained version of the JSON module that comes with later versions of python (2.6 or later). The syntax is the same for both modules but simplejson is more optimized and will give better performance.

To install, go to the project page `https://github.com/simplejson/simplejson` and click the Download button. Then select the latest version from the "Download Packages" section (currently this is 2.1.3). Unzip the folder and type

```
$ python setup.py install
```

from the command line. This should be all you need.

A.12 PySQLite

PySQLite is an SQLite binding for python. SQLite is a lightweight disk-based database that can be queried with SQL and is easy to install and use. The latest version is 2.6.3, see the project website `http://code.google.com/p/pysqlite/` for more details.

To install, download from `http://code.google.com/p/pysqlite/downloads/list` and unzip to a folder. Run

```
$ python setup.py install
```

from the command line.

A.13 CherryPy

CherryPy (`http://www.cherrypy.org/`) is a fast, stable and lightweight web server built on python using an object oriented model. CherryPy is easy to install, just download the latest version from `http://www.cherrypy.org/wiki/CherryPyInstall`. The latest stable release is 3.2.0. Unpack and run

```
$ python setup.py install
```

from the command line. After installing, look at the ten tiny tutorial examples that come with CherryPy in the *cherrypy/tutorial/* folder. These examples show you how to pass GET/POST variables, inheritance of page properties, file upload and download etc.

Appendix B

Image Datasets

B.1 Flickr

The immensely popular photo sharing site Flickr (`http://flickr.com/`) is a gold mine for computer vision researchers and hobbyists. With hundreds of millions of images, many of them tagged by users, it is a great resource to get training data or for doing experiments on real data. Flickr has an API for interfacing with the service that makes it possible to upload, download and annotate images (and much more). A full description of the API is available here `http://flickr.com/services/api/` and there are kits for many programming languages, including Python.

Let's look at using a library called *flickrpy* available freely at `http://code.google.com/p/flickrpy/`. Download the file *flickr.py*. You will need an API Key from Flickr to get this to work. Keys are free for non-commercial use and can be requested for commercial use. Just click the link "Apply for a new API Key" on the Flickr API page and follow the instructions. Once you have an API key, open *flickr.py* and replace the empty string on the line

```
API_KEY = ''
```

with your key. It should look something like this:

```
API_KEY = '123fbbb81441231123cgg5b123d92123'
```

Let's create a simple command line tool that downloads images tagged with a particular tag. Add the following code to a new file called *tagdownload.py*.

```
import flickr
import urllib, urlparse
import os
import sys
```

```
if len(sys.argv)>1:
    tag = sys.argv[1]
else:
    print 'no tag specified'

# downloading image data
f = flickr.photos_search(tags=tag)
urllist = [] #store a list of what was downloaded

# downloading images
for k in f:
    url = k.getURL(size='Medium', urlType='source')
    urllist.append(url)
    image = urllib.URLopener()
    image.retrieve(url, os.path.basename(urlparse.urlparse(url).path))
    print 'downloading:', url
```

If you also want to write the list of urls to a text file, add the following lines at the end.

```
# write the list of urls to file
fl = open('urllist.txt', 'w')
for url in urllist:
    fl.write(url+'\n')
fl.close()
```

From the command line, just type

```
$ python tagdownload.py goldengatebridge
```

and you will get the 100 latest images tagged with "goldengatebridge". As you can see, we chose to take the "Medium" size. If you want thumbnails or full size originals or something else, there are many sizes available, check the documentation on the Flickr website `http://flickr.com/api/`.

Here we were just interested in downloading images, for API calls that require authentication the process is slightly more complicated. See the API documentation for more information on how to set up authenticated sessions.

B.2 Panoramio

A good source of geotagged images is Google's photo-sharing service Panoramio (`http://www.panoramio.com/`). This web service has an API to access content programmatically. The API is described at `http://www.panoramio.com/api/`. You can get website widgets and access the data using JavaScript objects. To download images, the simplest way is to use a GET call. For example:

```
http://www.panoramio.com/map/get_panoramas.php?order=popularity&set=public&
from=0&to=20&minx=-180&miny=-90&maxx=180&maxy=90&size=medium
```

where *minx*, *miny*, *maxx*, *maxy* define the geographic area to select photos from (minimum longitude, latitude, maximum longitude and latitude, respectively). The response will be in JSON and look like this:

```
{"count": 3152, "photos":
[{"upload_date": "02 February 2006", "owner_name": "***", "photo_id": 9439, "longitude":
-151.75, "height": 375, "width": 500, "photo_title": "***", "latitude": -16.5, "owner_url":
"http://www.panoramio.com/user/1600", "owner_id": 1600, "photo_file_url":
 "http://mw2.google.com/mw-panoramio/photos/medium/9439.jpg", "photo_url": "http://www.panoramio.com/photo/9439"},
{"upload_date": "18 January 2011", "owner_name": "***", "photo_id": 46752123, "longitude":
120.52718600000003, "height": 370, "width": 500, "photo_title": "***", "latitude": 23.327833999999999, "owner_url":
 "http://www.panoramio.com/user/2780232", "owner_id": 2780232, "photo_file_url":
 "http://mw2.google.com/mw-panoramio/photos/medium/46752123.jpg", "photo_url": "http://www.panoramio.com/photo/46752123"},
{"upload_date": "20 January 2011", "owner_name": "***", "photo_id": 46817885, "longitude":
-178.13709299999999, "height": 330, "width": 500, "photo_title": "***", "latitude": -14.310613, "owner_url":
 "http://www.panoramio.com/user/919358", "owner_id": 919358, "photo_file_url":
 "http://mw2.google.com/mw-panoramio/photos/medium/46817885.jpg", "photo_url": "http://www.panoramio.com/photo/46817885"},
...
...
], "has_more": true}
```

Using a JSON package you can get the "photo_file_url" field of the result, see Section 2.3 for an example.

B.3 Oxford Visual Geometry Group

The Visual Geometry research group at Oxford University has many datasets available at `http://www.robots.ox.ac.uk/~vgg/data/`. We used some of the multi-view datasets in this book, for example the "Merton1", "Model House", "dinosaur" and "corridor" sequences. The data is available for download (some with camera matrices and point tracks) at `http://www.robots.ox.ac.uk/~vgg/data/data-mview.html`.

B.4 University of Kentucky Recognition Benchmark Images

The UK Benchmark image set, also called the "ukbench" set, is a set with 2550 groups of images. Each group has four images of an object or scene from varying viewpoints. This is a good set to test object recognition and image retrieval algorithms. The data set is available for download (the full set is around 1.5GB) at `http://www.vis.uky.edu/~stewe/ukbench/`. It is described in detail in the paper [23].

In this book we used a smaller subset using only the first 1000 images.

B.5 Other

Prague Texture Segmentation Datagenerator and Benchmark

This set used in the segmentation chapter can generate many different types of texture segmentation images. Available at http://mosaic.utia.cas.cz/index.php.

MSR Cambridge Grab Cut Dataset

Originaly used in the Grab Cut paper [27], this set provides segmentation images with user annotations. The data set and some papers are available from http://research.microsoft.com/en-us/um/cambridge/projects/visionimagevideoediting/segmentation/grabcut.htm. The original images in the data set are from a data set that now is part of the Berkeley Segmentation Dataset http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/.

Caltech 101

This is a classic dataset that contains pictures of objects from 101 different categories and can be used to test object recognition algorithms. The data set is available at http://www.vision.caltech.edu/Image_Datasets/Caltech101/.

Static Hand Posture Database

This dataset from Sebastien Marcel is available at http://www.idiap.ch/resource/gestures/ together with a few other sets with hands and gestures.

Middlebury Stereo Datasets

These are datasets used to benchmark stereo algorithms. They are available for download at http://vision.middlebury.edu/stereo/data/. Every stereo pair comes with ground truth depth images to compare results against.

Appendix C

Image Credits

Throughout this book we have made use of publicly available datasets and images available from web services, these were listed in Appendix B. The contributions of the researchers behind these datasets are greatly appreciated.

Some of the reoccurring example images are the author's own. You are free to use these images under a Creative Commons Attribution 3.0 (CC BY 3.0) license `http://creativecommons.org/licenses/by/3.0/`, for example by citing this book.

These images are:

- The Empire State building image used in almost every example throughout the book.
- The low contrast image in Figure 1.7.
- The feature matching examples used in Figures 2.2, 2.5, 2.6, and 2.7.
- The Fisherman's Wharf sign used in Figures 9.6, 10.1, and 10.2.
- The little boy on top of a hill used in Figures 6.4, 9.6.
- The book image for calibration used in Figures 4.3.
- The two images of the O'Reilly open source book used in Figures 4.4, 4.5, and 4.6.

Images from Flickr

We used some images from Flickr available with a Creative Commons Attribution 2.0 Generic (CC BY 2.0) license `http://creativecommons.org/licenses/by/2.0/deed.en`. The contributions from these photographers is greatly appreciated.

The images used from Flickr are (names are the ones used in the examples, not the original filenames):

- *billboard_for_rent.jpg* by @striatic `http://flickr.com/photos/striatic/21671910/` used in Figures 3.2.
- *blank_billboard.jpg* by @mediaboytodd `http://flickr.com/photos/23883605@N06/2317982570/` used in Figures 3.3.
- *beatles.jpg* by @oddsock `http://flickr.com/photos/oddsock/82535061/` used in Figures 3.2, 3.3.
- *turningtorso1.jpg* by @rutgerblom `http://www.flickr.com/photos/rutgerblom/2873185336/` used in Figure 3.5.
- *sunset_tree.jpg* by @jpck `http://www.flickr.com/photos/jpck/3344929385/` used in Figure 3.5.

Other images

- The face images used in Figures 3.6, 3.7, and 3.8 are courtesy of JK Keller. The eye and mouth annotations are the author's.
- The Lund University building images used in Figures 3.9, 3.11, and 3.12 are from a dataset used at the Mathematical Imaging Group, Lund University. Photographer was probably Magnus Oskarsson.
- The toy plane 3D model used in Figure 4.6 is from Gilles Tran (Creative Commons License By Attribution).
- The Alcatraz images in Figures 5.7 and 5.8 are courtesy of Carl Olsson.
- The font data set used in Figures 1.8, 6.2, 6.3 6.7, and 6.8 is courtesy of Martin Solli.
- The sudoku images in Figures 8.6, 8.7, and 8.8 are courtesy of Martin Byröd.

Illustrations

The epipolar geometry illustration in Figure 5.1 is based on an illustration by Klas Josephson and adapted for this book.

Bibliography

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. SURF: Speeded up robust features. In *European Conference on Computer Vision*, 2006.

[2] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:2001, 2001.

[3] Gary Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media Inc., 2008.

[4] Martin Byröd. An optical sudoku solver. In *Swedish Symposium on Image Analysis, SSBA*. `http://www.maths.lth.se/matematiklth/personal/byrod/papers/sudokuocr.pdf`, 2007.

[5] Antonin Chambolle. Total variation minimization and a class of binary mrf models. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, Lecture Notes in Computer Science, pages 136–152. Springer Berlin / Heidelberg, 2005.

[6] T. Chan and L. Vese. Active contours without edges. *IEEE Trans. Image Processing*, 10(2):266–277, 2001.

[7] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[8] D. Cremers, T. Pock, K. Kolev, and A. Chambolle. Convex relaxation techniques for segmentation, stereo and multiview reconstruction. In *Advances in Markov Random Fields for Vision and Image Processing*. MIT Press, 2011.

[9] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000.

[10] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, pages 363–370, 2003.

[11] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications-of-the-ACM*, 24(6):381–95, 1981.

[12] C. Harris and M. Stephens. A combined corner and edge detector. In *Proc. Alvey Conf.*, pages 189–192, 1988.

[13] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[14] Richard Hartley. In defense of the eight-point algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:580–593, 1997.

[15] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artifical Intelligence*, 17:185–203, 1981.

[16] Vladimir Kolmogorov and Ramin Zabih. What energy functions can be minimized via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:65–81, 2004.

[17] David G. Lowe. Object recognition from local scale-invariant features. In *International Conference on Computer Vision*, pages 1150–1157, 1999.

[18] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[19] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. pages 674–679, 1981.

[20] Mark Lutz. *Learning Python*. O'Reilly Media Inc., 2009.

[21] Will McGugan. *Beginning Game Development with Python and Pygame*. Apress, 2007.

[22] F. Meyer. Color image segmentation. In *Proc. of the 4th Conference on Image Processing and its Applications*, pages 302–306, 1992.

[23] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 2161–2168, 2006.

[24] Travis E. Oliphant. *Guide to NumPy*. http://www.tramy.us/numpybook.pdf, 2006.

[25] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch. Visual modeling with a hand-held camera. *International Journal of Computer Vision*, 59(3):207–232, 2004.

[26] Marc Pollefeys. Visual 3d modeling from images – tutorial notes. Technical report, University of North Carolina âĂŞ- Chapel Hill.

[27] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23:309–314, 2004.

[28] L. I. Rudin, S. J. Osher, and E. Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D*, 60:259–268, 1992.

[29] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 2001.

[30] Daniel Scharstein and Richard Szeliski. High-accuracy stereo depth maps using structured light. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.

[31] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, 2007.

[32] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22:888–905, August 2000.

[33] Jianbo Shi and Carlo Tomasi. Good features to track. In *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 593 – 600, 1994.

[34] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846. ACM Press, 2006.

[35] Bill Triggs, Philip F. McLauchlan, Richard I. Hartley, and Andrew W. Fitzgibbon. Bundle adjustment - a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, ICCV '99, pages 298–372. Springer-Verlag, 2000.

[36] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/, 2008.

[37] Deepak Verma and Marina Meila. A comparison of spectral clustering algorithms. Technical report, 2003.

[38] Luminita A. Vese and Tony F. Chan. A multiphase level set framework for image segmentation using the mumford and shah model. *Int. J. Comput. Vision*, 50:271–293, December 2002.

[39] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[40] Marco Zuliani. Ransac for dummies. Technical report, Vision Research Lab, UCSB, 2011.

Index