Первый шаг – загрузить изображение с диска, где оно обычно хранится в одном из графических форматов, например PNG или JPEG. Отметим, что в формате PNG изображение сжимается без потери информации, а в формате JPEG – с потерей, но он оптимизирован для визуального восприятия фотографий. Затем мы можем выполнить предварительную обработку изображения (например, нормировать с учетом разных условий освещения).

Вся эта глава построена вокруг задачи классификации. Мы хотим обучить классификатор на основе метода опорных векторов (или какого-нибудь другого алгоритма) на изображениях. Но перед тем как приступать к машинному обучению, нужно придумать промежуточное представление и выделить из изображений числовые признаки.

Загрузка и показ изображения

Для манипулирования изображениями мы будем использовать пакет mahotas. Скачать его можно со страницы `https://pypi.python.org/pypi/`, а прочитать руководство – на странице – `http://mahotas.readthedocs.org`. Mahotas – пакет с открытым исходным кодом (распространяемый по лицензии MIT, то есть его можно использовать в любом проекте), разработанный одним из авторов этой книги. По счастью, он основан на NumPy. Знания о NumPy, полученные к этому моменту, можно применить к обработке изображений. Существуют и другие пакеты на эту тему, например scikit-image (skimage), модуль ndimage (*n*-мерные изображения) в SciPy и интерфейс из Python к библиотеке OpenCV. Все они работают с массивами NumPy, поэтому можно без опаски комбинировать функции, реализованные в разных пакетах, для построения конвейера.

Для начала импортируем mahotas, назначив ему сокращенное имя `mh`, которым и будем пользоваться в этой главе:

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

Теперь можно загрузить файл изображения методом `imread`:

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

Файл `scene00.jpg` (имеется в составе набора данных в репозитории на сопроводительном сайте книги) содержит цветное изображение высотой `h` и шириной `w`; это изображение будет храниться в массиве

формы (h, w, 3). Первое измерение – высота, второе – ширина, а третье – цвет в виде трех компонент: красной, зеленой и синей. В других системах первое измерение соответствует ширине, но во всех пакетах на основе NumPy принято именно такое соглашение. Тип данных в массиве обычно np.uint8 (8-разрядное целое без знака). Именно так представлены изображения, которые вы снимаете камерой или отображаете на экране монитора.

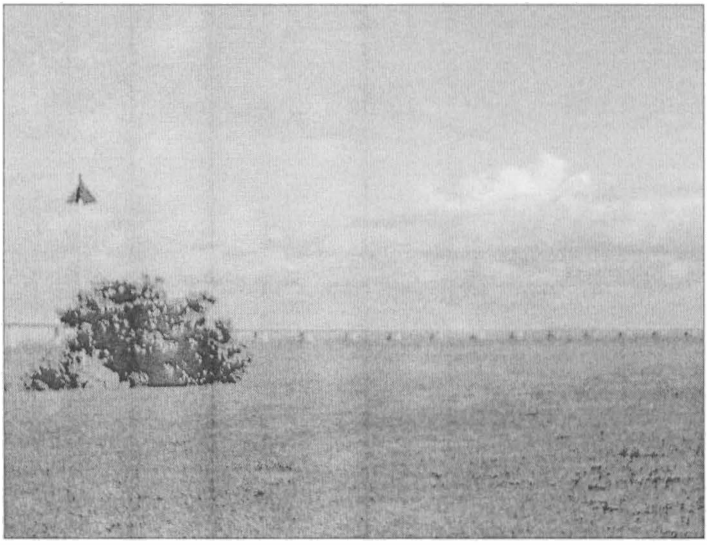
Специальное оборудование, применяемое в научно-технических приложениях, умеет работать с изображениями более высокой разрядности (то есть с большей чувствительностью к малым изменениям яркости), обычно 12- или 16-разрядными. Mahotas умеет работать и с такими типами, в том числе с числами с плавающей точкой. Во многих расчетах удобно перейти от целых чисел без знака к числам с плавающей точкой, чтобы упростить обработку округления и переполнения.

Mahotas может работать с различными системами ввода-вывода. К сожалению, ни одна из них не понимает все существующие форматы (их сотни, и у многих по несколько вариантов). Однако загрузку изображений в форматах PNG и JPEG поддерживают все. Мы ограничимся только ими, а за сведениями о том, как работать с менее распространенными форматами, отсылаем читателя к документации по mahotas.

Чтобы показать изображение на экране, воспользуемся библиотекой matplotlib, к которой уже неоднократно прибегали:

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

Как показано на рисунке ниже, этот код предполагает, что первое измерение – высота, а второе – ширина. Цветные изображения он также обрабатывает правильно. При использовании Python для числовых расчетов нам помогает тщательно спроектированная экосистема: mahotas работает с массивами NumPy, которые умеет отображать matplotlib; впоследствии мы извлечем из изображений признаки, которые обработаем посредством scikit-learn.



Бинаризация

Бинаризация – очень простая операция: все пиксели, большие некоторого порогового значения, заменяются единицей, а меньшие этого значения – нулем (или, если использовать булевы величины, преобразуются в `True` и `False`). Важно решить, как именно выбрать хорошее пороговое значение. В mahotas реализовано несколько способов выбора порога по изображению. Один из них называется **Otsu**, по имени своего изобретателя Оцу. Прежде всего, нужно перейти к полутоновому изображению с помощью функции `rgb2gray` из подмодуля `mahotas.colors`.

Вместо использования `rgb2gray` можно было бы вычислить среднее значение красного, зеленого и синего каналов, вызвав `image.mean(2)`. Но результат получился бы другой, потому что `rgb2gray` назначает цветам разные веса для получения более приятного для глаза изображения. Чувствительность наших глаз к трем основным цветам неодинакова.

```
>>> image = mh.colors.rgb2grey(image, dtype=np.uint8)
>>> plt.imshow(image) # Вывести изображение на экрана
```

По умолчанию matplotlib отображает такое одноканальное изображение как псевдоцветное: для больших значений берется красный

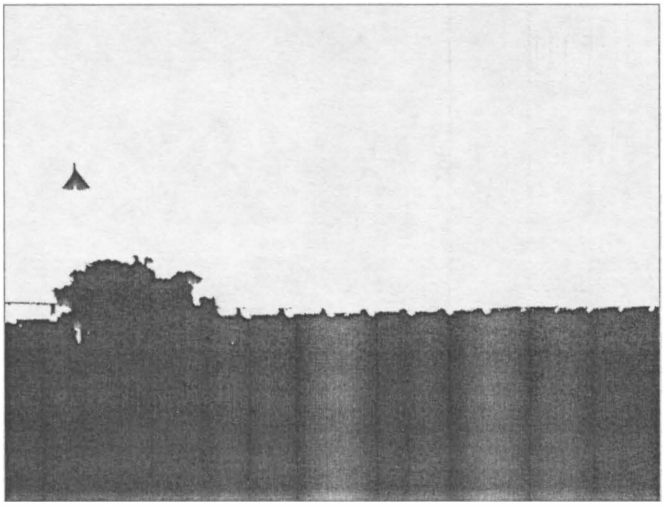
цвет, а для малых – синий. Но для естественных изображений предпочтительнее полутоновая гамма. Этот режим задается так:

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

Теперь получается полутоновое изображение. Отметим, что изменились только интерпретация и способ показа пикселей, сами данные изображения остались в неприкосновенности. Продолжим обработку и вычислим пороговое значение:

```
>>> thresh = mh.thresholding.otsu(image)
>>> print('Порог Оцу равен {}.'.format(thresh))
Порог Оцу равен 138.
>>> plt.imshow(image > thresh)
```

Для показанного выше изображения этот метод вычисляет пороговое значение 138, при этом разделяются земля и небо:



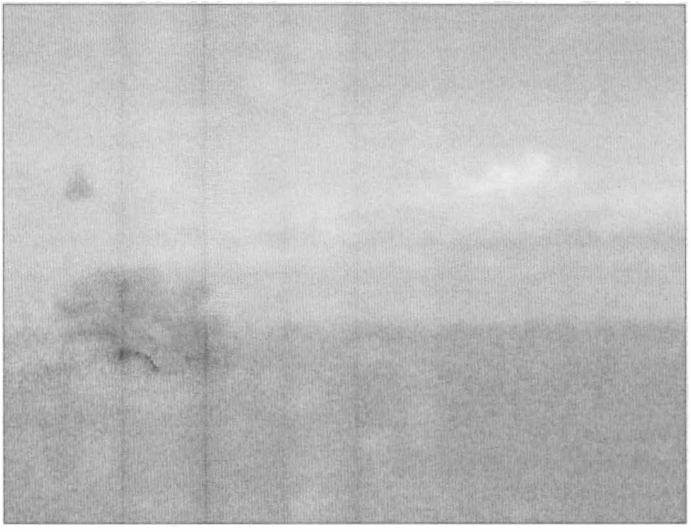
Гауссово размывание

На первый взгляд, непонятно, зачем размывать изображение, но часто это позволяет уменьшить шум и тем самым упростить последующую обработку. В mahotas для этого нужен всего один вызов:

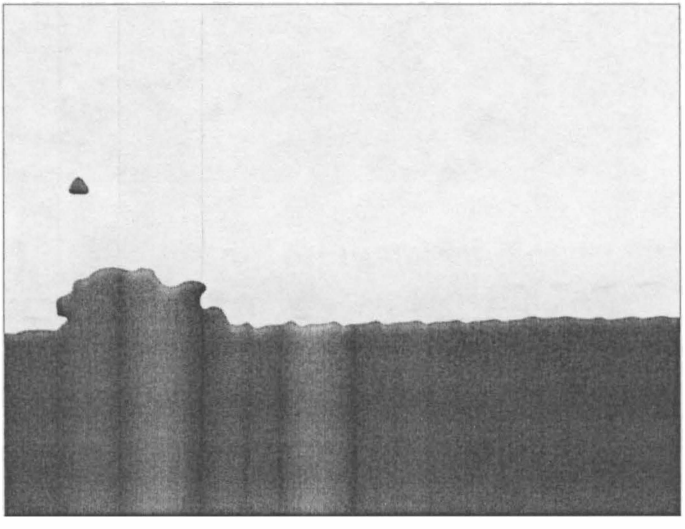
```
>>> im16 = mh.gaussian_filter(image, 16)
```

Отметим, что мы не преобразовывали значения пикселей полутонового изображения в целые числа без знака, а воспользовались

полученным результатом, где пиксели представлены числами с плавающей точкой. Второй аргумент функции `gaussian_filter` – размер фильтра (его стандартное отклонение). Чем он больше, тем сильнее размывание (см. рисунок ниже):



Если применить бинаризацию Оцу к размытому изображению, то границы будут более плавными:



Помещение центра в фокус

В следующем примере показано, как, объединив операторы NumPy с фильтрацией, можно добиться интересного результата. Начнем с фотографии Лены и выделим из него цветовые каналы:

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

Эта фотография девушки часто используется для демонстрации обработки изображений:

Для выделения красного, зеленого и синего каналов напишем такой код:

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

Теперь профильтруем все три канала порознь и построим из них новое изображение с помощью функции `mh.as_rgb`. Она принимает три двумерных массива, производит растяжение контрастности, преобразуя каждый в массив 8-разрядных целых, а затем накладывает их друг на друга и возвращает цветное RGB-изображение:

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

Теперь сошьем оба изображения от центра к краям. Сначала построим массив весов `w`, который будет содержать в позиции каждого пикселя нормированное значение – расстояние пикселя от центра:

```
>>> h, w = r.shape # высота и ширина
>>> Y, X = np.mgrid[:h,:w]
```

Мы воспользовались объектом `np.mgrid`, который возвращает массив размера `(h, w)`, значения в котором соответствую координатам y и x. Далее проделаем следующие операции:

```
>>> Y = Y - h/2. # центрировать на h/2
>>> Y = Y / Y.max() # привести к диапазону -1 .. +1

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

Теперь воспользуемся гауссовой функцией, чтобы увеличить значения пикселей в центральной области:

```
>>> C = np.exp(-2.*(X**2+ Y**2))
>>> # Снова нормируем: приводим к диапазону 0..1
>>> C = C - C.min()
>>> C = C / C.ptp()
>>> C = C[:,:,None] # Добавляем в массив C третье измерение
```

Отметим, что всё это – операции с массивами NumPy, никакой специфики mahotas здесь нет. Наконец, объединим оба массива, так чтобы центральная часть изображения оказалась в фокусе, а края были более расплывчатыми:

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

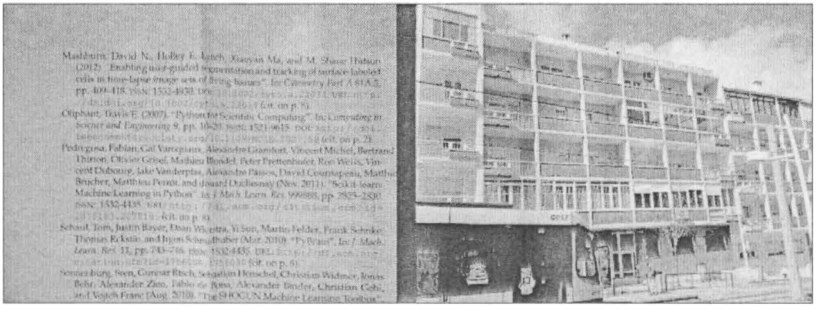
Простая классификация изображений

Начнем с небольшого набора данных, подготовленного специально для этой книги. В нем есть три класса: здания, природные ландшафты и фотографии текста. В каждой категории представлено 30 изображений, все они были сделаны камерой сотового телефона с минимальной композицией. Фотографии похожи на те, что закачивают на современные сайты пользователи, не учившиеся фотографировать. Набор данных можно скачать с сайта книги или из репозитория кода на GitHub. Позже мы рассмотрим более трудный набор данных, в котором и изображений, и категорий больше.

Для классификаций изображений у нас имеется большой прямоугольный массив чисел (значений пикселей). В наши дни типичный размер массив – порядка нескольких миллионов пикселей. Можно, конечно, попробовать загрузить все эти числа в алгоритм обучения в качестве признаков, но это не очень хорошая мысль. Дело в том, что связь каждого отдельного пикселя (и даже небольшой группы пикселей) с изображением очень отдаленная. Кроме того, наличие миллионов пикселей при очень небольшом количестве изображений приводит к чрезвычайно трудной задаче статистического обучения. Это крайнее проявление задач типа «*P* больше *N*», которые мы обсуждали в главе 7 «Регрессия». Гораздо разумнее вычислить признаки по изображению и использовать их для классификации.

И тем не менее, хочу отметить, что существуют методы, работающие непосредственно со значениями пикселей. В них встроены подпрограммы вычисления признаков. Они даже могут попытаться вывести хорошие признаки автоматически. В этой области сейчас ведутся активные исследования. Как правило, такие методы хорошо работают с очень большими наборами данных (содержащими миллионы изображений).

Выше мы уже видели пример изображения ландшафта. А вот ниже показаны примеры изображений текста и здания.



Вычисление признаков по изображению

Mahotas позволяет легко вычислять признаки по изображению. Соответствующие функции находятся в подмодуле `mahotas.features`.

Хорошо известен набор текстурных признаков Haralick. Как и многие алгоритмы обработки изображений, он назван по имени своего изобретателя. Признаки основаны на текстурах, то есть различают структурированные и неструктурированные изображения, а также различные повторяющиеся структуры. С помощью mahotas эти признаки вычисляются очень просто:

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

Функция `mh.features.haralick` возвращает массив 4×13. Первое измерение – четыре возможных направления, по которым вычисляются признаки (вертикаль, горизонталь и две диагонали). Если никакое конкретное направление нас не интересует, то можно усреднить признаки по всем направлениям (в коде выше эта переменная названа `haralick_features_mean`). Или же можно использовать все признаки по отдельности (переменная `haralick_features_all`). Решение зависит от свойств конкретного набора данных. Мы сочли, что в нашем случае признаки по вертикали и по горизонтали нужно хранить порознь, поэтому используем `haralick_features_all`.

В mahotas реализовано еще несколько наборов признаков. В частности, локальные бинарные шаблоны весьма устойчивы к изменению освещенности. Есть и другие типы признаков, в том числе локальных, которые мы обсудим ниже в этой главе.

Имея признаки, мы можем воспользоваться каким-нибудь стандартным методом классификации, например логистической регрессией:

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

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

У трех наших классов текстуры сильно различаются. Для зданий характерны резкие края и крупные области, в которых цвета очень близки (значения пикселей редко бывают одинаковыми, но вариации невелики). Для текста характерно много резких переходов от темного к светлому и маленькие островки черного в море белого. Природные ландшафты характеризуются более плавными переходами фрактального типа. Поэтому можно ожидать, что классификатор, основанный на текстурах, будет работать хорошо.

Мы построим классификатор на базе логистической регрессии с предварительной обработкой признаков:

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

Поскольку наш набор данных мал, можно воспользоваться регрессией с исключением по одному:

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

Верность 81% – неплохо для трех классов (случайное угадывание дало бы только 33%). Но можно улучшить этот результат, создав собственные признаки.

Создание собственных признаков

В признаках нет ничего сверхъестественного. Это просто числа, вычисляемые по изображению. В литературе описано несколько наборов признаков. У них зачастую есть то преимущество, что они специально проектировались, чтобы не зависеть от малосущественных факторов. Например, локальные бинарные шаблоны инвариантны относительно умножения значений всех пикселей на одно число или прибавления к ним константы. Поэтому такой набор признаков устойчив к изменению освещенности.

Но не исключено, что в конкретной ситуации лучше будет работать какой-то специальный набор признаков.

Простой признак, не включенный в дистрибутив mahotas, – гистограмма цветов. К счастью, реализовать его совсем нетрудно. Идея

гистограммы цветов состоит в том, чтобы разбить пространство цветов на несколько интервалов, а затем подсчитать, сколько пикселей оказалось в каждом интервале.

Изображения хранятся в формате RGB, то есть каждый пиксель представлен тремя значениями: R (красный), G (зеленый) и B (синий). Каждое значение – это 8-разрядное число, что в сумме дает 17 миллионов цветов. Мы сократим количество цветов до 64, разнеся их по интервалам. Напишем функцию, инкапсулирующую этот алгоритм:

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

Чтобы вычислить интервал цвета, сначала разделим весь массив на 64 с округлением значений:

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

Теперь значения каждого пикселя лежат в диапазоне от 0 до 3, то есть всего получается 64 цвета.

Выделим красный, зеленый и синий канал:

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

Приведем к логарифмической шкале, как показано в следующем фрагменте. Строго говоря, это необязательно, но признаки получаются более качественными. Мы пользуемся функцией `np.log1p`, которая вычисляет $\log(h+1)$. При этом нулевые значения остаются нулевыми (логарифм нуля не определен, и NumPy напечатает предупреждение при попытке вычислить его).

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

Эту функцию несложно включить в написанный ранее код:

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

Применяя тот же код перекрестной проверки, что и раньше, мы получаем верность 90%. Но самые лучшие результаты получаются, если объединить все признаки:

```
>>> features = []
>>> for im in images:
```

```
...  imcolor = mh.imread(im)
...  im = mh.colors.rgb2gray(imcolor, dtype=np.uint8)
...  features.append(np.concatenate([
...          mh.features.haralick(im).ravel(),
...          chist(imcolor),
...  ]))
```

При использовании всех этих признаков верность составит 95.6%:

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

Это еще одно подтверждение того, что хорошие алгоритмы – простая часть работы. Мы всегда можем воспользоваться реализациями самых передовых алгоритмов классификации, имеющимися в библиотеке scikit-learn. А самая-то соль и трудность – проектирование и подготовка признаков. Именно тут оказывается полезно знание о характере набора данных.

Использование признаков для поиска похожих изображений

Принципиальную идею о том, чтобы представить изображение сравнительно небольшим числом признаков, можно применить не только к классификации. Например, с ее помощью можно искать изображения, похожие на предъявленный в запросе образец (как с текстовыми документами).

Мы будем вычислять те же признаки, что и раньше, но с одним важным отличием: приграничные участки изображения игнорируются. Причина в том, что из-за любительской композиции на краях фотографии часто оказываются несущественные детали. Если вычислять признаки по всему изображению, то эти детали вносят вклад. А игнорируя их, удается получить чуть более качественные признаки. В случае обучения с учителем это не важно, потому что алгоритм обучения сможет определить, какие признаки более информативны, и назначить им соответственный вес. В режиме обучения без учителя нужно более внимательно следить за тем, чтобы признаки улавливали важные особенности данных. Эта идея реализована в следующем цикле:

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

```
...    # игнорировать все, что находится не дальше чем в 200
...    # пикселях от границы
...     imcolor = imcolor[200:-200, 200:-200]
...   im = mh.colors.rgb2gray(imcolor, dtype=np.uint8)
...   features.append(np.concatenate([
...       mh.features.haralick(im).ravel(),
...       chist(imcolor),
...   ]))
```

Теперь нормируем признаки и вычислим матрицу расстояний:

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

Выведем только подмножество данных (каждый десятый элемент), расположив образец сверху, а возвращенный «ближайший сосед» снизу:

```
>>> fig, axes = plt.subplots(2, 9)
>>> for ci,i in enumerate(range(0,90,10)):
...   left = images[i]
...   dists_left = dists[i]
...   right = dists_left.argsort()
...   # right[0] - то же, что left[i], поэтому выберем следующий ближайший
...   right = right[1]
...   right = images[right]
...   left = mh.imread(left)
...   right = mh.imread(right)
...   axes[0, ci].imshow(left)
...   axes[1, ci].imshow(right)
```

Результат показан на рисунке ниже:

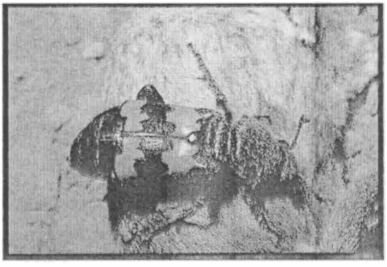
Видно, что система не совершенна, но все же способна находить изображения, которые, по крайней мере, зрительно похожи на предъявленный образец. Во всех случаях, кроме одного, найдены изображения из того же класса, что образец.

Классификация на более трудном наборе данных

Предыдущий набор оказалось легко классифицировать с помощью текстурных признаков. На самом деле, многие задачи, представляющие интерес для бизнеса, сравнительно просты. Но иногда попадаются проблемы потруднее, и тогда для получения приемлемого результата приходится применять более развитые современные методы.

Поэкспериментируем с общедоступным набором данных, имеющим такую же структуру: несколько фотографий, отнесенных к небольшому числу классов: животные, автомобили, транспортные средства и природные ландшафты.

Эти классы труднее различить, чем те три, что мы рассматривали выше. Природные ландшафты, здания и тексты характеризуются совершенно разными текстурами. А теперь текстуры и цвет не могут служить столь же очевидными маркерами класса. Вот пример изображения животного:



А вот – автомобиля:

Оба объекта сняты на фоне природы, а сами не имеют четко выраженной повторяющейся структуры. Это задача потруднее, поэтому понадобятся более сложные методы. Во-первых, мы воспользуемся более мощным классификатором. В библиотеке scikit-learn реализована регуляризованная форма логистической регрессии с настраиваемым параметром C. По умолчанию `C = 1.0`, но это значение не всегда оптимально. Для нахождения хорошего значения параметра можно использовать сеточный поиск:

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

Данные в наборе данных расположены не в случайном порядке: похожие изображения находятся рядом. Поэтому при перекрестной проверке будем перетасовывать данные, чтобы данные в каждой группе были более репрезентативны:

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

Для четырех классов совсем неплохо, но ниже мы покажем, как улучшить результат, взяв другой набор признаков. На самом деле, мы увидим, что для получения оптимального результата необходимо сочетать эти признаки с другими методами.

Локальные представления признаков

Сравнительно недавно в области машинного зрения были разработаны методы на основе локальных признаков. Локальные признаки вычисляются по небольшому участку изображения – в отличие от рассмотренных ранее признаков, вычисляемых по всему изображению. В mahotas поддерживается вычисление признаков типа **SURF** (Speeded Up Robust Features – ускоренно вычисляемые устойчивые признаки). Есть и несколько других, самый известный – первоначально предложенный SIFT. Эти признаки спроектированы с учетом

устойчивости относительно вращения и освещенности (то есть они мало изменяются при изменении освещения).

Нам нужно решить, где вычислять эти признаки. Обычно выбирают одну из трех возможностей:

- в случайных областях;
- на сетке;
- выявить интересные участки изображения (эта техника известна под названием «определение ключевых точек»).

Каждый из этих подходов при определенных обстоятельствах дает хорошие результаты. В mahotas поддерживаются все три. Определение ключевых точек лучше всего работает, если есть основания полагать, что ключевые точки действительно соответствуют наиболее важным участкам изображения.

Мы применим метод ключевых точек. Вычислить признаки в mahotas легко: нужно импортировать подходящий подмодуль и вызвать функцию `surf.surf`:

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

Флаг `descriptors_only=True` означает, что нас интересуют только сами дескрипторы, а не положение пикселей, размер или ориентация. Можно было бы вместо этого воспользоваться методом частой выборки, вызвав функцию `surf.dense`:

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

При этом возвращаются значения дескрипторов, вычисленные в точках, находящихся на расстоянии 16 пикселей друг от друга. Поскольку положения точек фиксированы, метаинформация о ключевых точках не очень интересна и по умолчанию не возвращается. В любом случае результат (дескрипторы) – это массив $n \times 64$, где n – количество точек в выборке. Оно зависит от размера изображения, его содержания и параметров функции. В данном случае мы взяли значения по умолчанию, это дает несколько сотен дескрипторов на одно изображение.

Мы не можем непосредственно загрузить эти дескрипторы в алгоритм опорных векторов, логистической регрессии или еще какой-нибудь алгоритм классификации. Существует несколько способов использовать дескрипторы изображения. Можно их просто усреднить,

но результаты получатся плохонькие, потому что при таком подходе отбрасывается важная информация о местоположении. В нашем примере мы просто получили бы еще один глобальный набор признаков, основанных на измерениях у границ.

Решение – воспользоваться моделью **набора слов** – появилось совсем недавно. Впервые в такой форме оно было опубликовано в 2004 году. Его можно отнести к идеям типа «как же мы раньше-то проглядели»: реализовать очень легко, а результаты получаются замечательные.

На первый взгляд, странно говорить о *словах* применительно к изображениям. Понять эту идею будет проще, если представлять себе не написанные слова, которые легко разделить, а произносимые. Произнесенное слово каждый раз звучит немного иначе, а разные люди произносят одно и то же слово по-разному. Поэтому волновые формы слова не будут повторяться. Тем не менее, применив к волновым формам кластеризацию, мы надеемся восстановить большую часть структуры, так что все варианты произношения данного слова окажутся в одном кластере. Даже если этот процесс не совершенен (а он таки не совершенен), все равно можно говорить о группировке волновых форм в слова.

Ту же самую операцию можно проделать и с данными изображения: кластеризовать похожие участки всех изображений и назвать кластеры **визуальными словами**.

Количество используемых слов не оказывает существенного влияния на качество алгоритма. Естественно, если их количество совсем уж мало (10–20 при нескольких тысячах изображений), то хорошей работы от системы ожидать не приходится. Но и при чрезмерно большом количестве слов (например, если их намного больше, чем изображений) система тоже будет работать не оптимально. Между этими крайностями часто лежит протяженное плато, на котором качество результата слабо зависит от количества слов. Значения 256, 512 или, – при очень большом числе изображений, – 1024 должны дать приемлемый результат.

Начнем с вычисления признаков:

```
>>> alldescriptors = []
>>> for im in images:
...   im = mh.imread(im, as_grey=True)
...   im = im.astype(np.uint8)
...   alldescriptors.append(surf.dense(image, spacing=16))
```

```
>>> # получить все дескрипторы в одном массиве
>>> concatenated = np.concatenate(alldescriptors)
>>> print('Количество дескрипторов: {}'.format(len(concatenated)))
Количество дескрипторов: 2489031
```

Мы получили свыше 2 миллионов локальных дескрипторов. Теперь с помощью кластеризации методом K средних найдем центроиды. Можно было бы использовать все дескрипторы, но для быстроты мы ограничимся только малой частью:

```
>>> # использовать каждый 64-ый вектор
>>> concatenated = concatenated[::64]
>>> from sklearn.cluster import KMeans
>>> k = 256
>>> km = KMeans(k)
>>> km.fit(concatenated)
```

Это займет некоторое время, но по завершении объект `km` будет содержать сведения о центроидах. Теперь вернемся к дескрипторам и построим векторы признаков:

```
>>> sfeatures = []
>>> for d in alldescriptors:
...   c = km.predict(d)
...   sfeatures.append(
...       np.array([np.sum(c == ci) for ci in range(k)])
...   )
>>> # строим один массив и преобразовываем в тип float
>>> sfeatures = np.array(sfeatures, dtype=float)
```

По выходе из цикла элемент `sfeatures[fi, fj]` массива показывает, сколько раз элемент `fj` встречается в изображении `fi`. Этот массив можно было бы вычислить и быстрее, воспользовавшись функцией `np.histogram`, но подготовка аргументов для нее – не вполне тривиальное занятие. Мы преобразуем результат к типу с плавающей точкой, поскольку не хотим возиться с целочисленной арифметикой и семантикой округления.

Теперь каждое изображение представлено одним массивом признаков, и размеры всех массивов одинаковы (равны количеству кластеров, в нашем случае 256). Таким образом, можно применить стандартные методы классификации:

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

Хуже, чем раньше! Мы что же, ничего не выиграли?

На самом деле, выиграли, потому что можем объединить все признаки и получить верность 76.1%:

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

Это лучший из достигнутых результатов – лучше, чем при любом наборе признаков по отдельности. Объясняется это тем, что локальные признаки SURF существенно отличаются от глобальных признаков изображения и потому привносят новую информацию в окончательный результат.

Резюме

Мы познакомились с классическим основанным на признаках подходом к обработке изображений в контексте машинного обучения: перейдя от миллионов пикселей к немногим числовым признакам, мы смогли воспользоваться классификатором на базе логистической регрессии. Все технологии, изученные в предыдущих главах, волшебным образом оказались применимы к задачам машинного зрения. Один из примеров – поиск похожих изображений в наборе данных.

Мы также научились использовать для классификации локальные признаки в виде модели набора слов. Это очень современный подход к машинному зрению, который, с одной стороны, дает отличные результаты, а, с другой, нечувствителен к несущественным деталям изображения, например освещению и даже неравномерному освещению одного и того же изображения. Мы также воспользовались кластеризацией не ради нее самой, а как полезным промежуточным шагом классификации.

Мы работали с mahotas, одной из основных библиотек машинного зрения на Python. Но есть и другие, поддерживаемые ничуть не хуже. Skimage (scikit-image) близка по духу, но строит другой набор признаков. OpenCV – отличная библиотека, написанная на C++ и имеющая интерфейс к Python. Все они могут работать с массивами NumPy, поэтому можно свободно пользоваться функциями из разных библиотек для построения сложных конвейеров машинного зрения.

В следующей главе мы займемся другим видом машинного обучения: понижением размерности. В предыдущих главах мы видели,

что с вычислительной точки зрения сгенерировать много признаков совсем нетрудно. Но часто желательно уменьшить число признаков ради повышения быстродействия, наглядности или улучшения качества результатов. Далее мы узнаем, как это делается.