



Классификация изображений на Python

От классического компьютерного
зрения до CNN и AlexNet

Итон Блейк

Итон Блейк

**Классификация изображений
на Python. От классического
компьютерного зрения
до CNN и AlexNet**

«Автор»

2026

Блейк И.

Классификация изображений на Python. От классического компьютерного зрения до CNN и AlexNet / И. Блейк — «Автор», 2026

В этой книге вы проходите весь маршрут на одном поле боя: от простых миниатюр и KNN/SVM через HOG и SIFT до CNN, AlexNet, transfer learning, GPU и Google Colab. Не только теория, а код, обучение, тестирование, точность, время выполнения и сравнение подходов на одинаковой задаче. Вы увидите, где классические методы еще держатся, где нейросети вырываются вперед и как перейти от признаков, придуманных человеком, к признакам, которые сеть учит сама. Это короткий практический мост между «я знаю Python» и «я реально понимаю, как строится классификатор изображений». Откройте ноутбук и повторяйте по ходу чтения.

© Блейк И., 2026

© Автор, 2026

Содержание

Глава 1: Введение	5
Глава 2: Представление миниатюрных изображений и классификаторы	7
Глава 3: Характеристики и классификаторы на основе гистограмм градиента направления	17
Конец ознакомительного фрагмента.	19

Итон Блейк

Классификация изображений на Python. От классического компьютерного зрения до CNN и AlexNet

Глава 1: Введение

В компьютерном зрении и машинном обучении классификация изображений — одна из основных задач. Она состоит в том, чтобы присвоить входному изображению метку из заранее заданного набора классов. Многие задачи, которые на первый взгляд выглядят совершенно иначе, связаны с классификацией изображений или используют её как составную часть — например, обнаружение объектов и сегментация.

По способу использования размеченных данных задачи классификации обычно делят на два типа: классификацию с учителем и классификацию без учителя.

При классификации с учителем изображения в обучающем наборе имеют заранее известные правильные метки. Алгоритм анализирует изображения вместе с этими метками и строит модель, которая затем используется для классификации новых изображений.

При классификации без учителя изображения в обучающем наборе не имеют заранее заданных меток. Алгоритм самостоятельно выявляет общие закономерности и структуру данных, а затем использует найденные признаки для группировки или анализа новых изображений.

В данной книге все алгоритмы относятся к категории классификации с учителем.

В этой книге алгоритмы классификации будут реализованы на Python ^[1] — широко используемом универсальном языке программирования высокого уровня, созданном Гвидо ван Россумом (Guido van Rossum). Первая версия Python вышла в 1991 году. В языке большое внимание уделяется читаемости кода; в частности, блоки программы выделяются отступами. Python использует динамическую типизацию и автоматическое управление памятью, поддерживает объектно-ориентированный, императивный, функциональный и процедурный стили программирования и располагает обширной стандартной библиотекой.

Версию Python от Anaconda можно скачать по следующему адресу: <https://www.anaconda.com/download>. Для работы с этой книгой потребуются обе версии: 2.7 и 3.7.

Для 64-разрядной Windows сначала необходимо загрузить и установить Anaconda2-2018.12-Windows-x86_64.exe и Anaconda3-2018.12-Windows-x86_64.exe. При установке можно оставить параметры по умолчанию. Затем откройте Anaconda Prompt и установите библиотеки, необходимые для примеров из книги. Для Python 2.7 используйте команду «pip install opencv-contrib-python==3.3.0.9». Для Python 3.7 установите opencv-python командой «pip install opencv-python==3.4.3.18», затем PyTorch; для версии, работающей на CPU, используется команда «pip install https://download.pytorch.org/whl/cpu/torch-1.0.0-cp37-cp37m-win_amd64.whl». Наконец, установите torchvision командой «pip install torchvision».

Для всех вычислений с использованием CPU, описанных в данной книге, конфигурация оборудования следующая: мобильная рабочая станция HP ZBook 15 G2; операционная система Windows 7 Professional, 64-разрядная версия; процессор Intel® Core™ i7-4810MQ с тактовой частотой 2,80 ГГц и 8,0 ГБ оперативной памяти.

Общая структура книги следующая:

В книге рассматриваются шесть подходов к классификации изображений:

Алгоритм 1: классификатор на основе представления миниатюрных изображений (Tiny Images);

Алгоритм 2: классификатор на основе гистограмм ориентированных градиентов (Histogram of Oriented Gradients, HOG);

Алгоритм 3: представление Bag of SIFT (Bag of Scale-Invariant Feature Transform) в сочетании с классификатором;

Алгоритм 4: свёрточная нейронная сеть (Convolutional Neural Network, CNN), обучаемая с нуля;

Алгоритм 5: тонкая настройка предварительно обученной глубокой нейронной сети AlexNet;

Алгоритм 6: извлечение признаков с помощью предварительно обученной AlexNet в сочетании с классификатором.

Для нескольких алгоритмов, использующих классификаторы, мы будем применять два широко распространённых типа классификаторов: метод k ближайших соседей (k -Nearest Neighbors, сокращённо KNN) и метод опорных векторов (Support Vector Machines, сокращённо SVM).

Для всех этих алгоритмов точность прогнозирования на тестовом наборе данных является основным критерием (чем выше, тем лучше), **а время выполнения — второстепенным** (чем короче, тем лучше). Код всех алгоритмов написан и запущен в Jupyter Notebook. Обращаем внимание читателей на то, что при каждом запуске точность прогнозирования и время выполнения могут отличаться.

Выводы будут приведены в последней главе.

Глава 2: Представление миниатюрных изображений и классификаторы

Представление миниатюрных изображений ^[2] является одной из самых простых форм представления изображений. Оно заключается лишь в приведении каждого изображения к небольшому фиксированному размеру.

Несмотря на то что представление в виде миниатюр прост для понимания, легко реализуем и обеспечивает высокую скорость вычислений, оно не является очень хорошей формой представления. Это связано с тем, что при таком представлении отбрасываются все высокочастотные компоненты изображения, а также с его чувствительностью к изменениям пространства или яркости. Поэтому в данной книге это представление используется лишь в качестве эталона.

Для реализации этого алгоритма нам сначала необходимо импортировать несколько библиотек.

```
# Импорт пакетов # Python # 3.7.0
import cv2 # opencv-python 3.4.3.18
import numpy as np # 1.15.1
import matplotlib.pyplot as plt # 2.2.3
import math
import time
import copy
import random
import glob # Использование glob для получения списка изображений
import pickle # Использование pickle для сохранения данных в файл и загрузки данных
из файла
```

```
import itertools
import skimage.exposure # scikit-image 0.13.1
from sklearn.neighbors import KNeighborsClassifier as KNC
from sklearn.svm import LinearSVC # scikit-learn 0.19.2
from sklearn.cluster import KMeans
from skimage.feature import hog
from sklearn.metrics import confusion_matrix
```

Затем нам нужно извлечь названия всех классов из пути хранения базы данных изображений и присвоить каждому классу уникальный числовой ярлык.

```
c_names = [name[11:] for name in glob.glob('data/Train/*')]
class_names = dict(zip(range(len(c_names)), c_names))
print("Всего { :d } классов:".format(len(class_names)))
print(class_names)
```

В Jupyter Notebook будет выведен следующий результат:

Всего 15 классов:

```
{0: 'Class01', 1: 'Class02', 2: 'Class03', 3: 'Class04', 4: 'Class05', 5: 'Class06', 6: 'Class07',
7: 'Class08', 8: 'Class09', 9: 'Class10', 10: 'Class11', 11: 'Class12', 12: 'Class13', 13: 'Class14', 14:
'Class15'}
```

После получения названий классов и присвоения каждому классу уникального идентификатора следующим шагом для всех алгоритмов, описанных в данной книге, является загрузка обучающих и тестовых данных.

Для Bag of SIFT на этом этапе **достаточно** считать изображения **с помощью cv2.imread()**. Для представления миниатюрных изображений и HOG-признаков данные дополнительно масштабируются, при необходимости сглаживаются, центрируются по среднему и нормируются до единичной длины; для HOG также требуется непосредственно вычислить дескрипторы. Для нейронных сетей нужны дополнительные операции, например формирование мини-пакетов (minibatch). При загрузке данных одновременно считываются метки изображений. Обучающую выборку для всех рассматриваемых алгоритмов следует случайным образом перемешивать: это существенно для корректного обучения модели.

В каждой категории имеется по 100 обучающих изображений и по 100 тестовых изображений. Таким образом, в базе данных изображений всего 3000 изображений.

Для загрузки миниатюрных изображений и HOG-признаков определена функция **load_data()**. Она принимает семь параметров: *path*, *img_size*, *flatten*, *mean*, *norm*, *shuffle* и *is_hog*. Параметр *path* задаёт относительный путь к обучающим или тестовым данным. *img_size* — размер изображения после масштабирования; значение по умолчанию (0, 0) означает, что размер не меняется. Параметры *flatten*, *mean*, *norm*, *shuffle* и *is_hog* — логические и по умолчанию равны *False*. Если *flatten=True*, двумерное представление изображения или HOG-дескриптор преобразуется в одномерный массив. При *mean=True* данные центрируются относительно нулевого среднего, а при *norm=True* нормируются до единичной длины. Эти операции могут повысить точность классификации. Если *shuffle=True*, изображения случайным образом перемешиваются. Если *is_hog=True*, для изображений вычисляются HOG-признаки. Функция также измеряет время загрузки обучающих или тестовых данных.

def load_data (path, img_size=(0,0), flatten=False, mean=False, norm=False, shuffle=False, is_hog=False):

```

    start_time = time.time() # Время начала
    n = 0 # Счётчик всех загруженных изображений
    data = [] # Для сохранения всех данных
    labels = [] # Для хранения всех меток
    if is_hog: # Получить HOG-описания
        cv2.HOGDescriptor()
    # Чтение изображений и их обработка
    for id, class_name in class_names.items():
        img_path_class = glob.glob(path + '/' + class_name + '/*.bmp')
        labels.extend([id]*len(img_path_class))
        for filename in img_path_class:
            img = cv2.imread(filename, 0) # Чтение изображения
            if img_size[0] and img_size[1]: # Изменить размер изображения
                img = cv2.resize(img, img_size, cv2.INTER_LINEAR)
            if is_hog: # Получить признаки HOG
                _, img = hog(img, block_norm='L2-Hys', visualise=True)
            if flatten: # Сглаживание изображения
                img = img.flatten()
            if mean: # Привести изображение к нулевому среднему значению
                img = np.subtract(img, np.mean(img))
            if norm: # Нормализовать изображение
                img = cv2.normalize(img, img)
            data.append(img)
            n = n + 1
    # Случайная перестановка (очень важно для обучения)
    if shuffle:

```

```

bundle = list(zip(data, labels))
random.shuffle(bundle)
data, labels = zip(*bundle)
print ("Завершена загрузка {:d} изображений за {:.2f} секунд для {:s}ing.".format(n,
time.time()-start_time, path[5:]))
return data, labels # Возвращаем данные и метки

```

Теперь вызовем эту функцию для загрузки данных, включая откорректированные «изображения» и их метки.

```

# Получаем миниатюрные изображения и их метки из обучающего и тестового наборов
данных
img_size_tiny = (16, 16)
train_d_tiny, train_l_tiny = load_data('data/Train', img_size_tiny, flatten=True, mean=True,
norm=True, shuffle=True)
test_d_tiny, test_l_tiny = load_data('data/Test', img_size_tiny, flatten=True, mean=True,
norm=True, shuffle=True)

```

Завершена загрузка 1500 изображений за 1,34 секунды для обучения.

Загрузка 1500 изображений для тестирования завершена за 1,35 секунды.

Имея загруженные данные, мы будем использовать два классификатора для решения задачи классификации изображений. Этими классификаторами являются: метод *k* ближайших соседей (*k*-Nearest Neighbors, сокращённо KNN) и метод опорных векторов (Support Vector Machines, сокращённо SVM).

Классификатор ближайших соседей (Nearest Neighbor) ^{[3], [4]} прост и понятен: когда необходимо отнести тестовый экземпляр к определенному набору классов, этому классификатору достаточно найти «ближайший» экземпляр из обучающего набора, а затем присвоить тестовому экземпляру метку этого ближайшего экземпляра. Этот классификатор обладает многими идеальными свойствами: он не требует обучения, способен обучаться границам принятия решений любой сложности, а также легко поддерживает задачи с несколькими классами. Однако этот классификатор очень чувствителен к шуму в обучающих данных. Решить эту проблему можно с помощью голосования на основе *k* ближайших соседей (*k*-Nearest Neighbors, сокращённо KNN). С увеличением размерности признаков как классификатор ближайших соседей, так и классификатор *k* ближайших соседей страдают от негативного влияния, поскольку ни один из них не располагает механизмом, позволяющим определить, какие размерности не имеют отношения к принятию решения.

Машины опорных векторов (SVM) представляют собой набор методов обученного обучения, используемых для классификации, *n*.

Машины опорных векторов (Support Vector Machines, сокращённо SVM) ^{[5], [6], [7]} представляют собой набор методов обученного обучения, используемых для классификации, регрессии и обнаружения выбросов. Они эффективны в высокоразмерных пространствах. Более того, они остаются эффективными даже в тех случаях, когда размерность превышает количество образцов. Кроме того, они отличаются высокой эффективностью использования памяти, что обусловлено использованием в функции принятия решений подмножества обучающих точек (называемых опорными векторами). Однако, если количество признаков значительно превышает количество образцов, крайне важно тщательно выбирать ядровую функцию и параметры регуляризации, чтобы избежать переобучения. Кроме того, эти методы не дают вероятностных оценок напрямую, а для их вычисления используется ресурсоемкая пятикратная перекрестная проверка.

Используемый в данной книге классификатор SVM «один к многим» ^[8] представляет собой относительно простую модель обучения. Его пространство признаков разделяется обу-

ченной гиперплоскостью, а классификация тестовых данных осуществляется в зависимости от того, на какой стороне гиперплоскости они находятся. Несмотря на то что такой классификатор гораздо менее ресурсоемкий, чем классификатор KNN, он часто демонстрирует лучшие результаты классификации.

Значения входных параметров классификаторов KNN и SVM фактически оказывают значительное влияние на точность прогнозирования. Чтобы получить оптимальные значения входных параметров (значения, обеспечивающие максимальную точность), мы определили функцию `get_best_parameter()`, которая находит оптимальное значение параметра `n_neighbors` для `KNeighborsClassifier()`, а также оптимальное значение параметра `lambda` (или `C`, который является параметром наказания за ошибки) для `sklearn.svm()`, с целью достижения максимально возможной точности прогнозирования.

Функция принимает шесть параметров: `train_data`, `train_label`, `test_data`, `test_label`, `model` и `n_max`. Первые четыре — соответственно обучающие данные, их метки, тестовые данные и метки тестового набора. Параметр `model` задаёт классификатор: KNN или SVM. В библиотеке `sklearn.neighbors` KNN реализован классом `KNeighborsClassifier()`, поэтому далее также используется сокращение KNC. Параметр `n_max` определяет, сколько значений настраиваемого параметра классификатора будет проверено; по умолчанию `n_max=10`.

В общих чертах функция сначала создаёт классификатор, а затем перебирает значения его настраиваемого параметра. Для каждого значения она обучает модель на обучающем наборе, прогнозирует метки тестового набора, сравнивает прогноз с истинными метками и вычисляет точность. Значение, давшее максимальную точность, сохраняется как оптимальное. Затем строится кривая зависимости точности от параметра, а функция возвращает найденное оптимальное значение. Одновременно измеряется общее время подбора.

Получить `n_neighbors` для `KNeighborsClassifier()` или `lambda` для `sklearn.svm()` с максимальной точностью

```
def get_best_parameter(train_data, train_label, test_data, test_label, model, n_max = 10):
    start_time = time.time() # Время начала
    if model == "KNC":
        param = "n_neighbors"
    elif model == "SVM":
        param = "lambda"
    else:
        return -1
    nn_max = 0.0 # Запись испробованных параметров
    ac_max = 0.0 # Запись максимальной точности для определения n_neighbors или lambda
    nn_all = [] # Для сохранения всех значений n_neighbors или lambda
    ac_all = [] # Для сохранения точностей прогнозирования
    for nn in range(1, n_max+1): # Выбор наивысшей точности путем настройки значения
        n_neighbors в KNeighborsClassifier()
        if model == "KNC": # Вызов KNeighborsClassifier
            mod = KNC(nn)
        elif model == "SVM": # Вызов LinearSVC
            nn = nn * 0.1
            mod = LinearSVC(C=nn, random_state=0)
        mod.fit(train_data, train_label) # Обучение модели на обучающих данных
        pred_label = mod.predict(test_data) # Прогнозирование с помощью модели
        ac = 100.0 * np.sum(pred_label == test_label) / len(test_label) # Получение точности
        ac_all.append(ac)
        nn_all.append(nn)
```

```

print("Точность модели составляет {:.2f}% ({:s} = {:.2f}).".format(ac, param, nn))
if (ac > ac_max): # Обновить максимальную точность и записать n_neighbors или lambda
    ac_max = ac
    nn_max = nn
print("\nМаксимальная точность модели составляет {:.2f}% ({:s} = {:.2f}).".format(ac_max, param, nn_max))
print("Общее время выполнения: {:.2f} секунд.".format(time.time() - start_time))
plt.plot(nn_all, ac_all, '-ob')
plt.title("Точность при различных значениях {:s}".format(param))
plt.xlabel("{:s}".format(param))
plt.ylabel("Точность (%)")
plt.grid(True)
plt.show()
return nn_max

```

Теперь вызовем эту функцию, чтобы получить оптимальные значения входных параметров классификатора KNN.

```

# Получить n_neighbors для KNeighborsClassifier() с наибольшей точностью
n_neighbors_tiny = get_best_parameter(train_d_tiny, train_l_tiny, test_d_tiny, test_l_tiny,
"KNC")

```

Вывод в Jupyter Notebook выглядит следующим образом. Кривая изменения точности прогнозирования в зависимости от различных значений входных параметров классификатора KNN показана на рисунке 1.

```

Точность модели составляет 55,20 % (n_neighbors = 1,00).
Точность модели составляет 50,60 % (n_neighbors = 2,00).
Точность модели составляет 50,87% (n_neighbors = 3,00).
Точность модели составляет 52,60 % (n_neighbors = 4,00).
Точность модели составляет 53,53% (n_neighbors = 5,00).
Точность модели составляет 51,80% (n_neighbors = 6,00).
Точность модели составляет 51,27% (n_neighbors = 7,00).
Точность модели составляет 51,07% (n_neighbors = 8,00).
Точность модели составляет 50,20 % (n_neighbors = 9,00).
Точность модели составляет 49,40 % (n_neighbors = 10,00).
Наивысшая точность модели составляет 55,20 % (n_neighbors = 1,000000).
Общее время выполнения: 9,73 секунды.

```

Ниже вызывается функция **get_best_parameter()** для получения оптимальных значений входных параметров классификатора SVM. При этом параметр **n_max** необходимо установить равным 20:

```

# Получить значение lambda для LinearSVC() с максимальной точностью
lambda_tiny = get_best_parameter(train_d_tiny, train_l_tiny, test_d_tiny, test_l_tiny, "SVM",
20)

```

Вывод в Jupyter Notebook выглядит следующим образом. Кривая изменения точности прогнозирования в зависимости от различных значений входных параметров классификатора SVM показана на рисунке 2.

```

Точность модели составляет 41,20% (lambda = 0,10).
Точность модели составляет 41,67 % (lambda = 0,20).
Точность модели составляет 41,73 % (lambda = 0,30).
Точность модели составляет 41,47 % (lambda = 0,40).
Точность модели составляет 40,80 % (lambda = 0,50).
Точность модели составляет 40,47 % (lambda = 0,60).

```

Точность модели составляет 40,00% ($\lambda = 0,70$).
 Точность модели составляет 40,20 % ($\lambda = 0,80$).
 Точность модели составляет 40,27 % ($\lambda = 0,90$).
 Точность модели составляет 40,07 % ($\lambda = 1,00$).
 Точность модели составляет 40,27% ($\lambda = 1,10$).
 Точность модели составляет 40,00 % ($\lambda = 1,20$).
 Точность модели составляет 39,73% ($\lambda = 1,30$).
 Точность модели составляет 39,73% ($\lambda = 1,40$).
 Точность модели составляет 39,80 % ($\lambda = 1,50$).
 Точность модели составляет 39,73% ($\lambda = 1,60$).
 Точность модели составляет 39,47 % ($\lambda = 1,70$).
 Точность модели составляет 39,33 % ($\lambda = 1,80$).
 Точность модели составляет 39,07 % ($\lambda = 1,90$).
 Точность модели составляет 39,13 % ($\lambda = 2,00$).
 Наивысшая точность модели составляет 41,73% ($\lambda = 0,300000$).
 Общее время выполнения: 11,41 секунды.

Как показано на рис. 1 и рис. 2, при размере миниатюрных изображений (16, 16) точность варьируется в диапазоне от 49,40 % до 55,20 %, если значение входного параметра *n_neighbors* классификатора KNN увеличивается от 1 до 10. Что касается классификатора SVM, то при изменении параметра наказания за ошибку (*lambda* или *C*) в диапазоне от 0,1 до 2,0 с шагом 0,1 точность прогнозирования колеблется в пределах от 39,07 % до 41,73 %. Для всех алгоритмов, использующих эти два классификатора в данной книге, мы будем проводить одинаковые вычисления с целью получения максимальной точности прогнозирования: 10 различных значений *n_neighbors*, изменяющихся от 1 до 10 с шагом 1; 20 различных значений *lambda*, изменяющихся от 0,1 до 2,0 с шагом 0,1.

Для вывода точности прогнозирования и затраченного времени мы определили функцию **print_stats()**. Эта функция имеет шесть входных параметров: *label_pred*, *label_true*, *time_train*, *time_test*, *model* и *n_class*. Первые четыре входных параметра легко понять: это прогнозируемый ярлык, истинный ярлык, время обучения и время тестирования. Параметр *model* указывает, какой классификатор используется в модели. *n_class* — это количество классов в данной задаче классификации, по умолчанию равное 15. Эта функция не только выводит общую точность прогнозирования и затраченное время, но и отображает точность прогнозирования для каждого класса, если *n_class* не равен нулю.

```

def print_stats(label_pred, label_true, time_train, time_test, model, n_class = 15):
    accuracy = 100.0 * np.sum(label_pred == label_true) / len(label_true)
    print("Точность модели \"{s}\" составляет {:.2f} %.".format(model, accuracy))
    print("Обучение:\tВремя обучения: {:.7.2f} секунд.".format(time_train))
    print("Тестирование:\tВремя выполнения: {:.7.2f} секунд.\n".format(time_test))
    # отобразить точность по каждому классу
    if n_class:
        class_correct = list(0. for i in range(n_class))
        class_total = list(0. for i in range(n_class))
        for i in range(len(label_true)):
            label = label_true[i]
            class_total[label] += 1
            if label_pred[i] == label:
                class_correct[label] += 1
        for i in range(n_class):
            acc = 100.0 * class_correct[i] / class_total[i]

```

```
print("Точность составляет {:.2f}% для {:s}.".format(acc, class_names[i]))
return accuracy
```

Теперь мы создадим классификатор, используя оптимальные значения входных параметров, обучим его на обучающей выборке и ее метках, используем обученный классификатор для прогнозирования меток тестовой выборки, сравним истинные метки тестовой выборки с прогнозируемыми метками, чтобы получить точность прогнозирования, а затем вызовем функцию **print_stats()**, чтобы вывести результаты прогнозирования классификаторов KNN и SVM, как показано ниже.

```
# Обучение и оценка с помощью классификатора KNN
time_start = time.time() # Время начала
model_best_tiny_KNC = KNC(n_neighbors_tiny)
model_best_tiny_KNC.fit(train_d_tiny, train_l_tiny) # Обучение модели на обучающих
данных
time_mid = time.time()
time_train = time_mid - time_start # Время, затраченное на обучение
label_pred_tiny_KNC = model_best_tiny_KNC.predict(test_d_tiny) # Прогнозирование с
помощью модели
time_test = time.time() - time_mid # Время, затраченное на тестирование
acc_tiny_KNC = print_stats(label_pred_tiny_KNC, test_l_tiny, time_train, time_test, "Tiny
Images + Nearest Neighbor")
```

Точность модели «Tiny Images + Nearest Neighbor» составляет 55,20%.

Обучение: Время выполнения: 0,02 секунды.

Тестирование: Время выполнения: 0,73 секунды.

Точность для класса Class01 составляет 21%.

Точность для класса Class02 составляет 38%.

Точность для класса Class03 составляет 76%.

Точность для класса Class04 составляет 50%.

Точность для Class05 составляет 93%.

Точность для класса Class06 составляет 28%.

Точность для класса Class07 составляет 74%.

Точность для класса Class08 составляет 40%.

Точность для класса 09 составляет 53%.

Точность для класса Class10 составляет 87%.

Точность для класса Class11 составляет 29%.

Точность для класса 12 составляет 62%.

Точность для класса Class13 составляет 40%.

Точность для класса Class14 составляет 65%.

Точность для класса 15 составляет 72%.

```
# Обучение и оценка с помощью классификатора SVM
```

```
time_start = time.time() # Время начала
```

```
model_best_tiny_SVM = LinearSVC(C=lambda_tiny, random_state=0) # Вызов LinearSVC
```

```
model_best_tiny_SVM.fit(train_d_tiny, train_l_tiny) # Обучение SVM
```

```
time_mid = time.time()
```

```
time_train = time_mid - time_start # Время обучения
```

```
label_pred_tiny_SVM = model_best_tiny_SVM.predict(test_d_tiny) # Вызов clf.predict
```

```
time_test = time.time() - time_mid # Время тестирования
```

```
acc_tiny_SVM = print_stats(label_pred_tiny_SVM, test_l_tiny, time_train, time_test, "Tiny
Images + Linear SVC")
```

Точность модели «Tiny Images + Linear SVC» составляет 41,73%.

Обучение: Время выполнения: 0,19 секунды.

Тестирование: Время выполнения: 0,02 секунды.

Точность для Class01 составляет 26%.

Точность для Class02 составляет 22%.

Точность для Class03 составляет 75%.

Точность для Class04 составляет 11%.

Точность составляет 78% для Class05.

Точность для класса Class06 составляет 21%.

Точность для класса Class07 составляет 63%.

Точность для класса 08 составляет 18%.

Точность для класса Class09 составляет 44%.

Точность для класса Class10 составляет 64%.

Точность для класса Class11 составляет 26%.

Точность для класса Class12 составляет 42%.

Точность для класса Class13 составляет 22%.

Точность для класса Class14 составляет 42%.

Точность для класса 15 составляет 72%.

Последний шаг — использование матрицы путаницы ^[9] для отображения точности прогнозирования. Нам нужно вызвать функцию **plot_confusion_matrix()**, чтобы получить две матрицы путаницы.

```
# Вывод и построение графика матрицы путаницы
def plot_confusion_matrix(cm, classes, normalize=True, title=' ', cmap=plt.cm.Blues):
    if normalize: # Нормализация
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.figure(figsize=(10, 8))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title('Матрица путаницы: ' + title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
    fmt = '.2f', если normalize, иначе 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center", color="white", если cm[i, j]
> thresh, иначе "black")
    plt.ylabel("Истинные метки")
    plt.xlabel("Предсказанные метки")
    plt.tight_layout()

# Алгоритм 1: Представление миниатюрных изображений
# Матрица путаницы (метод ближайшего соседа)
cm11 = confusion_matrix(test_1_tiny, label_pred_tiny_KNC)
plot_confusion_matrix(cm11, c_names, title='Миниатюрные изображения + метод ближай-
шего соседа')
# Матрица путаницы (линейный SVC)
cm12 = confusion_matrix(test_1_tiny, label_pred_tiny_SVM)
plot_confusion_matrix(cm12, c_names, title='Миниатюрные изображения + линейный
SVC')
```

Матрицы путаницы для обоих классификаторов показаны на рисунках 3 и 4 соответственно.

Диагональ матрицы путаницы показывает точность распознавания каждого класса. Несмотря на то что представление миниатюрных изображений используется здесь лишь как простая базовая модель, для классов 03, 05, 07, 10 и 15 точность превышает 60 % при обоих классификаторах.

Размер миниатюрных изображений последовательно задавался равным (8, 8), (16, 16), (32, 32), (64, 64) и (128, 128). Результаты приведены в таблице 1. В целом при увеличении размера изображения точность растёт как для KNN, так и для SVM; однако при каждом удвоении линейного размера объём вычислений увеличивается примерно в четыре раза, поскольку площадь изображения возрастает в четыре раза.

Что касается представления миниатюрных изображений, максимальная точность прогнозирования классификаторов KNN и SVM составила 59,07 % и 57,13 % соответственно; во всех случаях классификации классификатор KNN показал лучшие результаты, чем SVM.

Табл. 1. Результаты точности прогнозирования при работе с мелкими изображениями.

Размер изображения

(8,8)

(16,16)

(32,32)

(64,64)

(128, 128)

Максимальная точность (%)

KNN

48,73

55,20

55,20

57,47

59,07

SVM

32,93

41,73

48,20

53,40

57,13

Время выполнения (сек.)

KNN

0,25

0,97

3,88

14,97

58,05

SVM

0,19

0,57
1,66
6,10
25,76

Глава 3: Характеристики и классификаторы на основе гистограмм градиента направления

Гистограмма ориентированных градиентов (Histogram of Oriented Gradients, сокращённо HOG) [10], [11] — это полезный инструмент, используемый для извлечения признаков изображения с последующей классификацией изображений.

HOG сначала разбивает изображение на ячейки и блоки, затем вычисляет направления градиентов пикселей и формирует локальные гистограммы направлений. Совокупность этих гистограмм образует дескриптор признаков изображения.

Чтобы использовать HOG-признаки для классификации объектов, сначала при необходимости приводят обучающие и тестовые изображения к заданному размеру. Затем для каждого изображения вычисляют HOG-дескриптор, после чего обучают классификатор, например KNN или SVM, на признаках обучающей выборки и её метках. На заключительном этапе обученный классификатор применяется к HOG-признакам тестовых изображений.

Вызвав функцию `load_data()`, определённую в главе 2, и установив для её входного параметра `is_hog` значение `True`, мы можем извлечь HOG-признаки из каждого изображения в обучающем или тестовом наборе данных, как показано ниже.

```
# Получить миниатюрные изображения из обучающего и тестового наборов данных
img_size_hog = (64, 64)
train_d_hog, train_l_hog = load_data('data/Train', img_size_hog, flatten=True, shuffle=True,
is_hog=True)
test_d_hog, test_l_hog = load_data('data/Test', img_size_hog, flatten=True, shuffle=True,
is_hog=True)
```

Завершена загрузка 1500 изображений за 17,24 секунды для обучения.

Завершена загрузка 1500 изображений для тестирования за 17,21 секунды.

По сравнению с миниатюрными изображениями время загрузки данных в данном случае увеличилось более чем в десять раз, что связано с тем, что вычисление HOG-признаков занимает больше времени.

Теперь мы вызовем функцию `get_best_parameter()` (см. главу 2), чтобы получить оптимальное значение `n_neighbors` для KNN-классификатора и оптимальное значение `lambda` для SVM-классификатора. Результаты вывода приведены ниже. Кривые изменения точности прогнозирования в зависимости от значений входных параметров классификаторов показаны на рисунках 5 и 6.

```
# Получить значение n_neighbors для KNeighborsClassifier() с максимальной точностью
n_neighbors_hog = get_best_parameter(train_d_hog, train_l_hog, test_d_hog, test_l_hog,
"KNC")
```

Точность модели составляет 61,13% (`n_neighbors = 1,00`).

Точность модели составляет 56,73% (`n_neighbors = 2,00`).

Точность модели составляет 56,40% (`n_neighbors = 3,00`).

Точность модели составляет 56,33% (`n_neighbors = 4,00`).

Точность модели составляет 55,13% (`n_neighbors = 5,00`).

Точность модели составляет 53,60 % (`n_neighbors = 6,00`).

Точность модели составляет 52,00 % (`n_neighbors = 7,00`).

Точность модели составляет 51,87% (`n_neighbors = 8,00`).

Точность модели составляет 51,33% (`n_neighbors = 9,00`).

Точность модели составляет 50,60 % (`n_neighbors = 10,00`).

Наивысшая точность модели составляет 61,13% (`n_neighbors = 1,000000`).

Общее время выполнения: 155,58 секунд.

Когда размер изображения установлен на (64, 64), точность классификатора KNN колеблется в диапазоне от 50,60 % до 61,13 %; точность классификатора SVM — от 47,80 % до 48,73 %. Время выполнения составило 156 секунд и 378 секунд соответственно, то есть на каждое значение ушло 15,6 секунд и 18,9 секунд.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.