**Практическая работа № 5-6**

**Метод K ближайших соседей**

Один из самых простых и наиболее употребительных методов — классификация по k ближайшим соседям (KNN). Этот алгоритм сравнивает подлежащий классификации объект (например, вектор признаков) со всеми объектами из обучающего набора с известными метками классов и назначает ему класс, исходя из классов k ближайших соседей. Зачастую метод показывает хорошие результаты, но имеет ряд недостатков. Число K нужно задавать заранее, а от него зависит качество работы. Кроме того, требуется хранить весь обучающий набор, а если он велик, то поиск становится медленным. Для больших обучающих наборов обычно используют ту или иную форму разбиения на интервалы с целью сократить количество необходимых сравнений.

К числу плюсов следует отнести отсутствие ограничений на выбор метрики; подойдет практически любая (правда, это не значит, что результаты окажутся хорошими). К тому же, алгоритм тривиально распараллеливатся.

Реализация KNN в простейшей форме вполне прямолинейна. Показанный ниже код решает задачу, если известны обучающие примеры и соответствующие им метки. Примеры и метки можно передать в виде строк массива или просто списками. Это могут быть числа, строки — вообще все, что угодно.

Библиотеки, которые следует подключить:

import numpy

import scipy

import math

import matplotlib

Добавьте следующий класс в файл knn.py:

class KnnClassifier (object) :

def \_\_init\_\_(self, labels, samples) :

self.labels = labels

self.samples = samples

def classify(self, point, k=3) :

# Вычислить расстояние до всех точек обучающего набора

dist = np.array([self.l2dist(point,s) for s in self.samples])

# отсортировать их

ndx = dist.argsort ()

# сохранить К ближайших в словаре

votes = {}

for i in range(k):

label = self.labels[ndx[i]]

votes.setdefault (label, 0)

votes[label] += 1

return max(votes)

def l2dist(self, p1,p2):

return m.sqrt(sum((p1-p2)\*\*2))

Проще всего определить класс и инициализировать его обучающими данными. Тогда нам не придется хранить и передавать обучающие данные в виде аргументов всякий раз, как нужно будет что-то классифицировать. Если для хранения K ближайших меток используется словарь, то сами метки могут быть строками или числами. В данном случае в качестве метрики мы взяли евклидово расстояние (L2). Другие метрики можно добавить в виде дополнительных функций.

**Простой двумерный пример**

Создадим несколько простых наборов точек на плоскости и наглядно покажем, как работает классификатор. Следующий скрипт создает два набора точек, принадлежащих двум классам, и сохраняет данные с помощью модуля pickle:

from numpy.random import randn

import pickle

# создать демонстрационные наборы точек на плоскости

n = 200

# два нормальных распределения

class\_1 = 0.6 \* randn(n,2)

class\_2 = 1.2 \* randn(n,2) + np.array([5,1])

labels = np.hstack((np.ones(n),-np.ones(n)))

# сохранить с помощью pickle

with open('points\_normal.pkl', 'wb') as f:

pickle.dump(class\_1, f)

pickle.dump(class\_2, f)

pickle.dump (labels, f)

# нормальное распределение и кольцо вокруг него

class\_1 = 0.6 \* randn(n,2)

r= 0.8 \* randn(n,1) + 5

angle = 2\*np.pi\* randn(n,1)

class\_2 = np.hstack((r\*np.cos (angle) ,r\*np.sin(angle) ))

labels = np.hstack((np.ones(n),-np.ones(n)))

# сохранить с помощью pickle

with open('points\_ring.pkl', 'wb') as f:

pickle.dump(class\_1,f)

pickle.dump(class\_2, f)

pickle. dump (labels, f)

Выполните скрипт дважды с разными именами файлов, например, сначала points\_normal\_test.pkl, а затем points\_ring\_test.pkl. Теперь у вас четыре файла с наборами данных, по два для каждого распределения. Один будем использоваться для обучения, другой — для тестирования.

Посмотрим, как решить задачу с помощью KNN-классификатора.

Введите такой скрипт:

#загрузить точки с помощью pickle

with open('points\_normal.pkl', 'rb') as f:

class\_1 = pickle.load(f)

class\_2 = pickle.load(f)

labels = pickle.load(f)

model = KnnClassifier(labels, np.vstack((class\_1,class\_2)))

Он создает модель KNN -классификатора на основе данных в pickle-файле. Добавьте далее такой код:

# загрузить тестовые данные с помощью pickle

with open('points\_normal1.pkl', 'rb') as f:

class\_l = pickle.load(f)

class\_2 = pickle.load(f)

labels = pickle.load(f)

# протестировать на первой точке

print (model.classify(class\_1[0]))

Здесь мы загружаем другой набор данных (тестовый) и печатаем на консоли предсказанную метку класса для первой точки.

Чтобы визуализировать классификацию всех тестовых точек и показать, насколько хорошо классификатор разделяет два класса, добавим такие строчки:

# определить функцию построения графика

def classify (x,y,model=model) :

return np.array([model.classify([xx,yy]) for (xx,yy) in zip(x,y)])

# нанести на график границу между графиками

plot\_2D\_boundary([-6, 6,-6, 6], [class\_1, class\_2],classify, [1,-1])

mt.show()

Здесь мы создали простенькую вспомогательную функцию, которая принимает массивы х и у, содержащие координаты точек, а также классификатор и возвращает массив предсказанных меток классов. Эту функцию можно передать в качестве аргумента функции, которая строит график.

Добавьте показанную ниже функцию:

def plot\_2D\_boundary(plot\_range, points, decisionfcn, labels, values=[0]):

"""Plot\_range – это диапозон (xmin,xmax,ymin, ymax), points — список точек, decisionfcn – функция приниающая решение, labels – массив меток классов, который возвращает decisionfcn, values — список подлежащих показу изолиний решающей функции"""

clist = ['b','r','g','k', 'm', 'y'] # цвета соответствующие классам

# вычислить и нанести на сетку изолинию решающей функции

x=np.arange(plot\_range[0],plot\_range[1],.1)

y = np.arange(plot\_range[2],plot\_range[3],.1)

xx, yy = np.meshgrid(x,y)

xxx, yyy = xx.flatten(),yy.flatten() # списки x,y на сетке

zz = np.array(decisionfcn(xxx, yyy) )

zz = zz.reshape(xx.shape)

# нанести на график изолинии из списка values

mt.pyplot.contour(xx, yy, zz, values)

# Для каждого класса нанести на график точки обозначив '\*' правильные', 'o' — не правильные

for i in range(len(points)):

d = decisionfcn(points[i][:,0],points[i] [:,1])

correct\_ndx = labels[i]==d

incorrect\_ndx = labels[i]!=d

mt.pyplot.plot(points [i] [correct\_ndx,0],points[i] [correct\_ndx,1],'\*', color=clist[i])

mt.pyplot.plot (points [i] [incorrect\_ndx, 0], points [i] [incorrect\_ndx,1],'o', color=clist[i])

mt.axis ('equal')

Эта функция принимает решающую функцию (классификатор) и вычисляет ее в узлах сетки с помощью функции meshgrid(). На график можно нанести изолинии решающей функции, чтобы показать, где проходят границы. По умолчанию подразумевается нулевая изолиния. Получается график, показанный на рис. 1. Как видим, решающая граница KNN-классификатора может адаптироваться к распределению классов без явного построения модели.

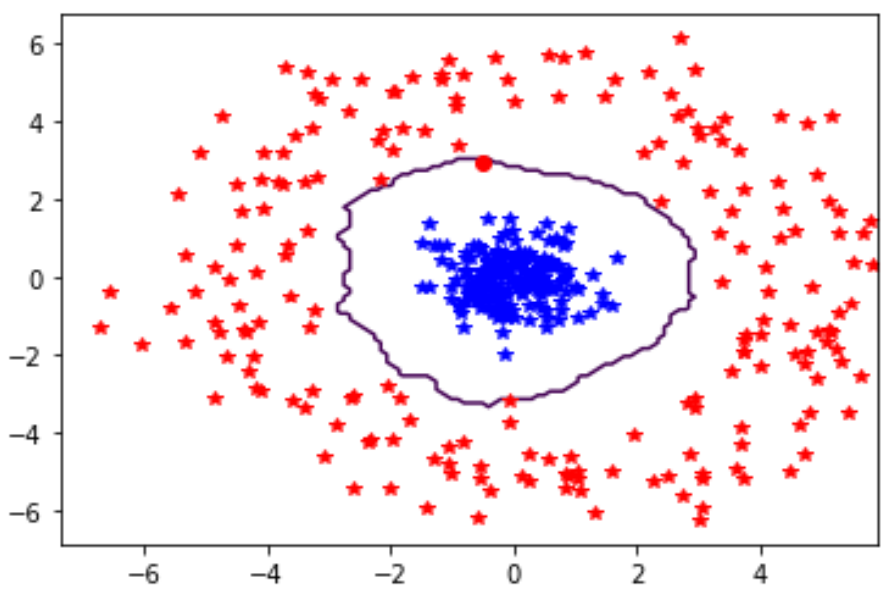
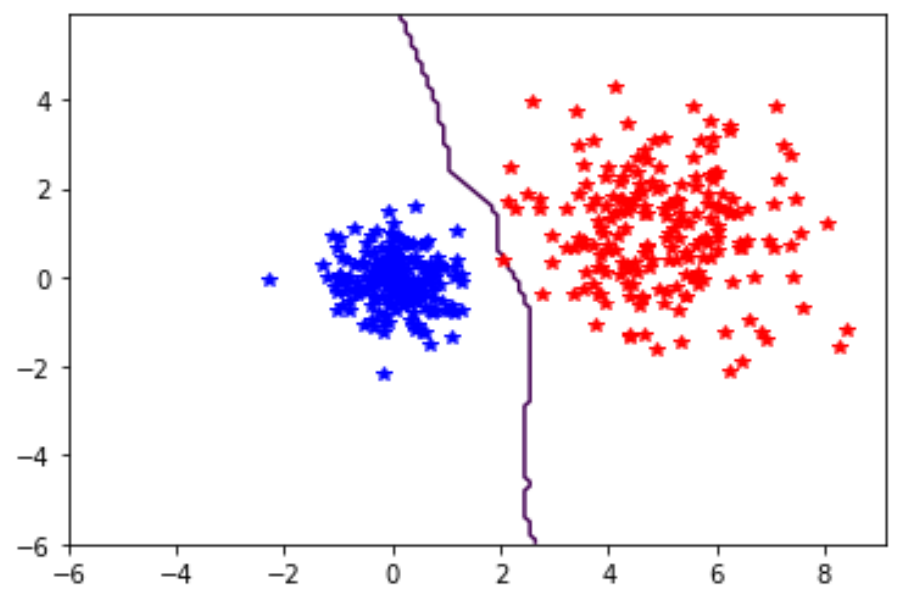


Рис. 1. Классификация точек на плоскости с помощью классификатора по К ближайшим соседям. Цветом кодируются классы меток. Правильно классифицированные точки показаны звездочками, неправильные - кружочками. Кривая на графике - решающая граница

**Плотные SIFT-дескрипторы в качестве признаков изображения**

Рассмотрим теперь классификацию изображений. Нам понадобится вектор признаков изображения. Введем так называемое плотное **SIFT** -представление.

Плотное **SIFT** - представление создается посредством применения дескрипторной части алгоритма **SIFT** к регулярной сетке, наложенной на все изображения. Мы можем воспользоваться кодом из практической работы № 2 и получить **SIFT**-признаки на плотной сетке, добавив дополнительные параметры. Напишите следующую функцию функцию:

import sift

def process\_image\_dsift (imagename, resultname, size=20, steps=10, force\_orientation=False, resize=None):

""" Обработать изображение с помощью **SIFT** – дескрипторов на плотной сетке и сохранить результаты в файле. Необязательные параметры: размер признаков, количество шагов между точками, принудительное вычисление ориентации дескриптора (False означает, что все дескрипторы направлены вверх), кортеж с новыми размерами изображения. """

im = Image.open(imagename).convert ('L')

if resize!=None:

im = im.resize (resize)

m,n = im.size

if imagename[-3:]!= 'pgm':

# Создать pgm-файл

im.save('tmp.pgm')

imagename = 'tmp.pgm!'

# Создать фреймы и сохранить во временныом файле

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)

Сравните эту функцию с функцией process\_image() из практической работы № 2. Мы вызываем функцию savetxt () для сохранения массива frame в текстовом файле для последующей обработки в пакетном режиме.

Последний параметр этой функции можно использовать для изменения размера изображения перед выделением дескрипторов. Например, если передать imsize=(100, 100), то изображения будут преобразованы в квадратные размером 100х100 пикселей. Наконец, если параметр force\_orientation равен True, то дескрипторы нормируются в направлении доминирующего локального градиента. В противном случае все дескрипторы просто направлены вверх.

Воспользуемся этой функцией для вычисления плотных **SIFT** - дескрипторов и визуализации места их расположения:

import sift

process\_image\_dsift ('v1.jpg', 'v1.sift', 90, 40, True)

l,d = sift.read\_features\_from\_file('v1.sift')

im = array(Image.open('empire.jpg'))

sift.plot\_features(im,l,True)

show()

Этот код вычисляет **SIFT** - дескрипторы на плотной сетке, покрывающей все изображение, ориентируя их вдоль направления локального градиента (поскольку параметр force\_orientation равен True).

Места расположения дескрипторов показаны на рис. 2.

**Классификация изображений — распознавание жестов**

Рассмотрим применение плотных **SIFT** - дескрипторов к изображениям жестов руки с целью построения простой системы распознавания жестов. Для иллюстрации воспользуемся изображениями из базы данных статических положений руки (по адресу http://www.idiap.ch/resource/gestures/). Скачайте малый тестовый набор (на веб-странице ссылка на него называется [test set 16.3Mb](https://www.idiap.ch/webarchives/sites/www.idiap.ch/resource/gestures/data/shp_marcel_test.tar.gz), найдите все папки с именем «uniform». Изображения, находящиеся в каждой из них, поровну разложите по папкам «train» и «test» (предварительно создав их).

Обработаем изображения приведенной выше функцией, чтобы получить векторы признаков. В предположении, что имена файлов хранятся в списке imlist, это делается следующим образом:

import dsift

# обработать изображения, приведенные к фиксированному размеру (50,50)

for filename in imlis

featfile = filename[:-3]+'dsift'

dsift.process\_image\_dsift (filename, featfile, 10,5, resize=(50,50))

В результате для каждого изображения создается файл его признаков с расширением «.dsift». Обратите внимание на приведение всех изображений к единому размеру. Это очень важно, иначе количество дескрипторов в изображениях, а значит, и длины векторов признаков были бы различны, что привело бы впоследствии к ошибкам при их сравнении. Нанесенные на график изображения с дескрипторами показаны на рис. 3.

Определим вспомогательную функцию для чтения файлов плотных SIFT-дескрипторов:

import os, sift

def read\_gesture\_ features labels (path):

# создать список всех файлов с расширением .аз1ЕЕ

featlist = [os.path.join(path,f) for f in os.listdir (path)

if f.endswith('.dsift")]

# прочитать признаки

features = []

for featfile in featlist:

1,d = sift.read\_features\_from\_file (featfile)

features.append(d.flatten())

features = array (features)

# создать метки

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

return features, array (labels)

Теперь можно прочитать признаки и метки для тестового и обучающего набора:

features, labels = read\_gesture\_features\_labels('train/')

test\_features, test\_labels=read\_gesture\_features\_labels('test/")

classnames = unique (labels)

Здесь мы создаем метки классов, исходя из первой буквы имени файла. А с помощью функции NumPy unique () получаем отсортированный список уникальных имен классов.

Настало время опробовать наш код классификации по ближайшим соседям на этих данных:

# протестировать &ММ

kel

knn\_classifier = knn.KnnClassifier (labels, features)

res = array({knn\_classifier.classify(test\_features[i],k) for i in

range (len(test\_labels))])

# верность

acc = sum(1.0\*(res==test\_labels)) / len(test\_labels)

print 'Bepxocre:', acc

Сначала создаем объект классификатора, передавая ему обучающие данные и метки. Затем обходим тестовый набор и классифицируем каждое изображение, вызывая метод classify(). Верность классификации вычисляется путем умножения булевого массива на единицу и суммирования. В данном случае истинные значения равны 1, поэтому подсчитать количество правильно назначенных классов нетрудно. Получается такой результат:

Верность: 0.811518324607

т.е. 81 % примеров классифицировано правильно. Значение зависит от выбора k и параметров алгоритма вычисления плотных SIFT-дескрипторов.

Полученное выше значение верности показывает, сколько примеров из тестового набора было классифицировано правильно, но ничего не говорит ни о том, какие жесты оказалось трудно классифицировать, ни о типичных ошибках.

Матрица неточностей содержит информацию о том, как именно были классифицированы примеры из каждого класса. Она показывает, как распределились ошибки и какие классы классификатор чаще всего «путает» с другими.

Следующая функция печатает метки и матрицу неточностей:

def print\_confusion (res, labels, classnames) :

n = len(classnames)

+ матрица неточностей

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 'Marpwya Herowxocred ans"

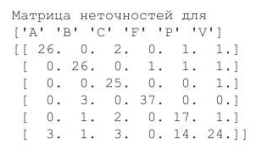
print classnames

print confuse

В результате вызова

print\_confusion (res, test\_labels,classnames)

будет напечатано:



Отсюда видно, к примеру, что «Р» («Point» — указание) часто неправильно классифицируется как «У».

**Упражнения**

1. Качество KNN-классификатора зависит от значения k. Попробуйте поиграть с ним и посмотрите, как меняется верность. Нанесите на график решающие границы множеств точек на плоскости и понаблюдайте за их изменением.

2. Набор данных для распознавания жестов руки так же содержит изображения с более сложным задним планом (в папке complex/). Попробуйте обучить и протестировать классификатор на этих изображениях. Как изменилось качество? Предложите улучшенный дескриптор изображения?