计算机视觉总结
SUMMARY OF COMPUTER VISION

(第1版)

LVSHUAILIN

OPEN SOURCE
BEIJING

VERSION 1

一. 数据结构与算法-LeetCode Hot 100

二. PYTHON: 1) NUMPY; 2) PANDAS; 3) PYTHON多进程; 4) PYTHON分布式; 5) PYTHON界面;

三. 深度学习: TensorFlow 2.0; PYTORCH;

四. 图像配准

五. 强化学习

六. OTHERS: 1) Model INFERENCE by EXE; 2) GIT; 3) DOCKER

LVSHUAILIN

2020年2月

目　　录

第1章　LeetCode Hot 100

Goals to Achieve

1. unordered_map.

§ 1.1 两数之和

HOT100 1.1　问题描述

给定一个整数数组**nums** 和一个目标值**target**, 请你在该数组中找出和为目标值的那两个整数, 并返回他们的数组下标. 你可以假设每种输入只会对应一个答案. 但是, 你不能重复利用这个数组中同样的元素.

示例: 给定nums = [2, 7, 11, 15], target = 9; 因为nums[0] + nums[1] = 2 + 7 = 9;

所以返回[0, 1]

https://leetcode-cn.com/problems/two-sum

1.1.1 知识点(unordered_map)

unordered_map内部是一个关联容器, 采用hash 表结构, 有快速检索的功能.

哈希表是通过key关键字直接访问对应value值的数据结构. 特点是键和值一一对应, 查找时间复杂度**O(1)**.

Example_1: unordered_map插入, 迭代遍历.

unordered_map example_1 code

```
1 #include <iostream>
2 #include <unordered_map>
3 #include <string>
4 using namespace std;
5 int main()
6 {
```

```
    unordered_map<string, double> umap;
    umap["PI"] = 3.14;
    umap.insert(make_pair("a", 2.1));

    // find in umap
    string key = "PI";
    if (umap.find(key) == umap.end())
        cout << "cannot find PI" << endl;
    else
        cout << "find " << umap.find(key)->first << " = " << umap.find(key)->second << endl;

    // iterator of umap
    cout << "entire unorded_map is:"<<endl;
    unordered_map<string, double>::iterator itr;
    for (itr = umap.begin(); itr != umap.end(); ++itr)
        cout << "(" << itr->first << "," << itr->second << ")" << endl;
    system("pause");
    return 0;
}
```

output:
find PI = 3
all elements are:
(PI,3.14)
(a,2.1)

Example_2: 利用unordered_map输出一段文字中重复单词的个数

unordered_map example_2 code

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <sstream>

using namespace std;

void printWordFreq(const string& str)
{
    unordered_map<string, int> wordFreq;
    string word;
    stringstream ss(str);
```

```
    while (ss >> word)
        wordFreq[word]++;

    cout << "all elements are:" << endl;
    for (auto u : wordFreq)
        cout << "(" << u.first << "," << u.second << ")" << endl;
}

int main()
{
    string str = "studies very very hard";
    printWordFreq(str);
    return 0;
}
```

output:
all elements are:
(studies, 1)
(very, 2)
(hard, 1)

1.1.2 解题代码

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;

vector<int> twoSum(vector<int>& nums, int target)
{
    unordered_map<int, int> map;
    vector<int> result={};
    int n = (int)nums.size();
    for(int i = 0; i < n; ++i) {
        auto p = map.find(target-nums[i]);
        if(p != map.end()) {
        result.push_back(p->second);
        result.push_back(i);
        }
```

```
        map[nums[i]] = i;
        }
        return result;
}

int main()
{
        vector< int > nums = {2,7,11,15};
        vector<int> result;
        result = twoSum(nums,9);
        cout<<"[ "<<result[0] << " , " <<result[1]<<" ] "<<endl;
        return 0;
}
```

§ 1.2 两数相加

HOT100 1.2 问题描述

给出两个非空的链表用来表示两个非负的整数. 其中, 它们各自的位数是按照逆序的方式存储的, 并且它们的每个节点只能存储一位数字. 如果, 我们将这两个数相加起来, 则会返回一个新的链表来表示它们的和.

您可以假设除了数字0之外, 这两个数都不会以0开头.

示例: 输入$(2->4->3) + (5->6->4)$, 输出: $7->0->8$, 原因: $342 + 465 = 807$

https://leetcode-cn.com/problems/add-two-numbers

1.2.1 知识点(linked list)

这里用c++ 链表来解决

Example_1: 创建链表并初始化

linked list example_1 code

```
#include <iostream>

using namespace std;

class Node{
public:
```

```
    int data;
    Node* next;
};

int main()
{
    Node* head = nullptr;
    Node* second = nullptr;
    Node* third = nullptr;

    head = new Node();
    head->data = 1;

    second = new Node();
    second->data = 2;

    third = new Node();
    third->data = 3;

    cout << head->data << "  " << second->data << "  " << third->data << endl;

    delete head;
    delete second;
    delete third;
    return 0;
}
```

output:
1 2 3

Example_2: 打印链表中的所有元素

linked list example_2 code

```
#include <iostream>

using namespace std;

class Node{
public:
    int data;
    Node* next;
```

```
};

void PrintLinkedList(Node* head)
{
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << "  ";
        temp = temp->next;
    }
    cout << endl;
}

int main()
{
    Node* head = nullptr;
    Node* second = nullptr;
    Node* third = nullptr;

    head = new Node();
    second = new Node();
    third = new Node();

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = third;

    third->data = 3;
    third->next = nullptr;


    PrintLinkedList(head);

    delete head;
    delete second;
    delete third;
    return 0;
}
```

output:
1 2 3

Example_3: 链表插入节点

linked list example_3 code

```
#include <iostream>

using namespace std;

class Node{
public:
    int data;
    Node* next;
};

// 在链表前面插入节点
void push(Node** head_ref, int newData)
{
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = (*head_ref);
    (*head_ref) = newNode;
}

// 在节点后面插入节点
void insertAfter(Node** prev_node, int newData)
{
    if ((*prev_node) == nullptr) {
        cout << "the previous node cannot be nullptr" << endl;
        return;
    }

    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = (*prev_node)->next;
    (*prev_node)->next = newNode;
}

// 在尾节点后插入节点
void append(Node** head_ref, int newData)
```

```
{
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = nullptr;
    if ((*head_ref) == nullptr) {
        (*head_ref) = newNode;
        return;
    }

    Node* move = (*head_ref);
    while (move->next != nullptr) {
        move = move->next;
    }
    move->next = newNode;
}

//打印链表
void PrintLinkedList(Node* head)
{
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << "  ";
        temp = temp->next;
    }
    cout << endl;
}

void destroyLinkedList(Node** head_ref) {
    Node* move = (*head_ref);
    Node* next = nullptr;
    while (move != nullptr) {
        next = move->next;
        delete move;
        move = next;
    }
    (*head_ref) = nullptr;
}

int main()
{
```

```
    Node* head = nullptr;

    append(&head, 6);

    push(&head, 7);

    push(&head, 1);

    append(&head, 4);

    insertAfter(&(head->next), 8);

    cout << "linked list is: ";
    PrintLinkedList(head);
    destroyLinkedList(&head);
    return 0;
}
```

output:
linked list is: 1 7 8 6 4

1.2.2 解题代码

```
#include <iostream>
using namespace std;

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    int len1 = 1;//记录的长度l1
    int len2 = 1;//记录的长度l2
    ListNode* p = l1;
    ListNode* q = l2;
    while (p->next != NULL)//获取的长度l1
    {
```

```
        len1++;
        p = p->next;
    }
    while (q->next != NULL)//获取的长度l2
    {
        len2++;
        q = q->next;
    }
    if (len1 > len2)//较长，在末尾补零l1l2
    {
        for (int i = 1; i <= len1 - len2; i++)
        {
            q->next = new ListNode(0);
            q = q->next;
        }
    }
    else//较长，在末尾补零l2l1
    {
        for (int i = 1; i <= len2 - len1; i++)
        {
            p->next = new ListNode(0);
            p = p->next;
        }
    }
    p = l1;
    q = l2;
    bool count = false;//记录进位
    ListNode* l3 = new ListNode(-1);//存放结果的链表
    ListNode* w = l3;//的移动指针l3
    int i = 0;//记录相加结果
    while (p != NULL && q != NULL)
    {
        i = count + p->val + q->val;
        w->next = new ListNode(i % 10);
        count = i >= 10 ? true : false;
        w = w->next;
        p = p->next;
        q = q->next;
    }
    if (count)//若最后还有进位
```

```
57      {
58          w->next = new ListNode(1);
59          w = w->next;
60      }
61      return l3->next;
62  }
63
64  void printLinkedList(ListNode* head)
65  {
66      ListNode* move = head;
67      while (move != nullptr) {
68          cout << move->val << "  ";
69          move = move->next;
70      }
71  }
72
73  int main()
74  {
75  #if 1
76      ListNode* l1 = new ListNode(2);
77      ListNode* l1_1 = new ListNode(4);
78      ListNode* l1_2 = new ListNode(3);
79
80      l1->next = l1_1;
81      l1_1->next = l1_2;
82
83
84      ListNode* l2 = new ListNode(5);
85      ListNode* l2_1 = new ListNode(6);
86      ListNode* l2_2 = new ListNode(4);
87      l2->next = l2_1;
88      l2_1->next = l2_2;
89  #endif
90
91  #if 0
92      ListNode* l1 = new ListNode(5);
93      ListNode* l2 = new ListNode(5);
94  #endif
95
96      ListNode* result = addTwoNumbers(l1, l2);
```

```
97      printLinkedList(result);
98      return 0;
99  }
```

output:
7 0 8

§ 1.3 无重复字符的最长子串

HOT100 1.3 问题描述

给定一个字符串, 请你找出其中不含有重复字符的最长子串的长度.

示例1:

输入: “abcabcb”

输出: 3

解释: 因为无重复字符的最长子串是“abc”, 所以其长度为3.

示例2:

输入: “bbbbb”

输出: 1

解释: 因为无重复字符的最长子串是“b”, 所以其长度为1.

示例3:

输入: “pwwkew”

输出: 3

解释: 因为无重复字符的最长子串是“wke”, 所以其长度为3. 请注意, 你的答案必须是子串的长度, “pwke”是一个子序列, 不是子串.

https://leetcode-cn.com/problems/longest-substring-without-repeating-characters

1.3.1 知识点(double pointer algorithm和unordered_set)

c++提供两种关联型数据结构, 1) 树型结构, 如: map, set; 2) hash结构, 如: unordered_map, unordered_set. map和set是有序的, 其他两个是无序的.

1.3.2 解题思路

https://cloud.tencent.com/developer/article/1377650

这道题主要用到思路是: 滑动窗口

什么是滑动窗口?

其实就是一个队列, 比如例题中的abcabcbb, 进入这个队列(窗口)为abc满足题目要求, 当再进入a, 队列变成了abca, 这时候不满足要求. 所以, 我们要移动这个队列!

如何移动?

我们只要把队列的左边的元素移出就行了, 直到满足题目要求!

一直维持这样的队列, 找出队列出现最长的长度时候, 求出解!

时间复杂度: **O(n)**

1.3.3 解题代码

```
#include <iostream>
#include <string>
#include <unordered_set>
#include <algorithm> // max, min

using namespace std;

int lengthOfLongestSubstring(string s) {
    if (s.size() == 0) return 0;
    unordered_set<char> lookup;
    int maxStr = 0;
    int left = 0;
    for (int i = 0; i < s.size(); i++) {
        while (lookup.find(s[i]) != lookup.end()) {
            lookup.erase(s[left]);
            left++;
        }
        maxStr = max(maxStr, i - left + 1);
        lookup.insert(s[i]);
    }
    return maxStr;
}

int main()
```

```
{
    string str = "abcabcb";
    cout << lengthOfLongestSubstring(str) << endl;
    return 0;
}
```

output:
3

§ 1.4 寻找两个有序数组的中位数

HOT100 1.4 问题描述

给定两个大小为m和n的有序数组nums1和nums2.

请你找出这两个有序数组的中位数, 并且要求算法的时间复杂度为**O(log(m + n))**.

你可以假设nums1和nums2不会同时为空.

示例1:

nums1 = [1, 3]

nums2 = [2]

则中位数是2.0

示例2:

nums1 = [1, 2]

nums2 = [3, 4]

则中位数是(2 + 3)/2 = 2.5

https://leetcode-cn.com/problems/median-of-two-sorted-arrays

1.4.1 知识点(二分查找算法)

用二分查找算法, 也叫做折半查找算法.

Example_1: 二分查找

二分查找算法*example_1 code*

```
// 二分查找– 折半查找
int search(int arr[], int key, int left, int right)
{
    while (left <= right)
```

```
    {
        int mid = left + (right − left) / 2;
        if (key < arr[mid])
            right = mid − 1;
        else if (key > arr[mid])
            left = mid + 1;
        else
            return mid;
    }
    return −1;
}

int main()
{
    int arr[] = { 0,2 ,3,4};
    int value = 3;

    // left Index of the array
    int left = 0;

    // right Index of the array
    int right = sizeof(arr) / sizeof(arr[0]) − 1;

    cout << "left: " << left << ", right: " << right << endl;

    int ret = search(arr, value, left, right);
    if (ret == −1)
        printf("cannot find the value");
    else
        printf("found the value, the index is: %d\n", ret);
    system("pause");
    return 0;
}
```

output:
left: 0, right: 3
found the value, the index is: 2

1.4.2 解题思路

这道题让我们求两个有序数组的中位数，而且限制了时间复杂度为O(log (m+n))，看到这个时间复杂度，自然而然的想到了应该使用二分查找法来求解。那么回顾一下中位数的定义，如果某个有序数组长度是奇数，那么其中位数就是最中间那个，如果是偶数，那么就是最中间两个数字的平均值。这里对于两个有序数组也是一样的，假设两个有序数组的长度分别为m和n，由于两个数组长度之和m+n 的奇偶不确定，因此需要分情况来讨论，对于奇数的情况，直接找到最中间的数即可，偶数的话需要求最中间两个数的平均值。为了简化代码，不分情况讨论，我们使用一个小trick，我们分别找第(m+n+1) / 2 个，和(m+n+2) / 2 个，然后求其平均值即可，这对奇偶数均适用。假如m+n 为奇数的话，那么其实(m+n+1) / 2 和(m+n+2) / 2 的值相等，相当于两个相同的数字相加再除以2，还是其本身。

这里我们需要定义一个函数来在两个有序数组中找到第K个元素，下面重点来看如何实现找到第K个元素。首先，为了避免产生新的数组从而增加时间复杂度，我们使用两个变量i和j分别来标记数组nums1和nums2的起始位置。然后来处理一些边界问题，比如当某一个数组的起始位置大于等于其数组长度时，说明其所有数字均已经被淘汰了，相当于一个空数组了，那么实际上就变成了在另一个数组中找数字，直接就可以找出来了。还有就是如果K=1的话，那么我们只要比较nums1和nums2的起始位置i和j上的数字就可以了。难点就在于一般的情况怎么处理？因为我们需要在两个有序数组中找到第K个元素，为了加快搜索的速度，我们要使用二分法，对K二分，意思是我们需要分别在nums1和nums2中查找第K/2个元素，注意这里由于两个数组的长度不定，所以有可能某个数组没有第K/2个数字，所以我们需要先检查一下，数组中到底存不存在第K/2个数字，如果存在就取出来，否则就赋值上一个整型最大值。如果某个数组没有第K/2个数字，那么我们就淘汰另一个数字的前K/2个数字即可。有没有可能两个数组都不存在第K/2个数字呢，这道题里是不可能的，因为我们的K不是任意给的，而是给的m+n的中间值，所以必定至少会有一个数组是存在第K/2个数字的。最后就是二分法的核心啦，比较这两个数组的第K/2小的数字midVal1和midVal2的大小，如果第一个数组的第K/2个数字小的话，那么说明我们要找的数字肯定不在nums1中的前K/2个数字，所以我们可以将其淘汰，将nums1的起始位置向后移动K/2个，并且此时的K也自减去K/2，调用递归。反之，我们淘汰nums2中的前K/2个数字，并将nums2的起始位置向后移动K/2个，并且此时的K也自减去K/2，调用递归即可。

1.4.3 解题代码

```
/*分清 起始位置和第几个元素*/
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

int findKthNumber(vector<int>& nums1, int i, vector<int>& nums2, int j, int k) {
    if (i >= nums1.size()) return nums2[j + k - 1];
    if (j >= nums2.size()) return nums1[i + k - 1];
    //if(k ==1) return (double(nums1[i] + nums2[j]));wrong
```

```
11      if (k == 1) return min(nums1[i], nums2[j]);
12      //查找有没有k个元素的位置/2 i + k/2 −1
13      int midVal1 = (i + k / 2 − 1 < nums1.size()) ? nums1[i + k / 2 − 1] : INT_MAX;
14      int midVal2 = (j + k / 2 − 1 < nums2.size()) ? nums2[j + k / 2 − 1] : INT_MAX;
15      if (midVal1 < midVal2)
16          return findKthNumber(nums1, i + k / 2, nums2, j, k − k / 2);
17      else
18          return findKthNumber(nums1, i, nums2, j + k / 2, k − k / 2);
19  }
20  class Solution {
21  public:
22      double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
23          int m = nums1.size(), n = nums2.size();
24          int left = (m + n + 1) / 2, right = (m + n + 2) / 2;
25          return (findKthNumber(nums1, 0, nums2, 0, left) + findKthNumber(nums1, 0, nums2, 0, right)) / 2.0;
26      }
27  };
```

output:
Null

第2章　Bayesian-Python

学习目标与要求

1. .
2. .
3. .
4. .

p(H): Prior

p(D—H): Likelihood

p(H—D): Posterior

p(D): Evidence

prior distribution should reflect what we know about the value of some parameters before see the data D. If we know nothing, use flat priors that do not convey too much information.

likelihood is how we will introduce data in our analysis.

If we ignore the **evidence**, we can write Bayes's theorem as a proportionality:

$$p(H|D) \propto p(D|H)p(H)$$

§ 2.1 Probalilities

A common and useful conceptualization in statistics is to think that data was generated from some probability distribution with unobserved parameters.

2.1.1 Gaussian Distribution

$$pdf(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

mu_params = [−1, 0, 1]
sd_params = [0.5, 1, 1.5]
x = np.linspace(−7, 7, 100)
f, ax = plt.subplots(len(mu_params), len(sd_params), sharex=True, sharey=True)
for i in range(3):
    for j in range(3):
        mu = mu_params[i]
        sd = sd_params[j]
        y = stats.norm(mu, sd).pdf(x)
        ax[i, j].plot(x,y)
        ax[i, j].plot(0, 0, label="$\\mu$ = {:3.2f}\n$\\sigma$={:3.2f}".format (mu, sd), alpha=0)
        ax[i, j].legend(fontsize=12)
    ax[2,1].set_xlabel('$x$', fontsize=16)
    ax[1,0].set_ylabel('$pdf(x)$', fontsize=16)
    plt.tight_layout()
```

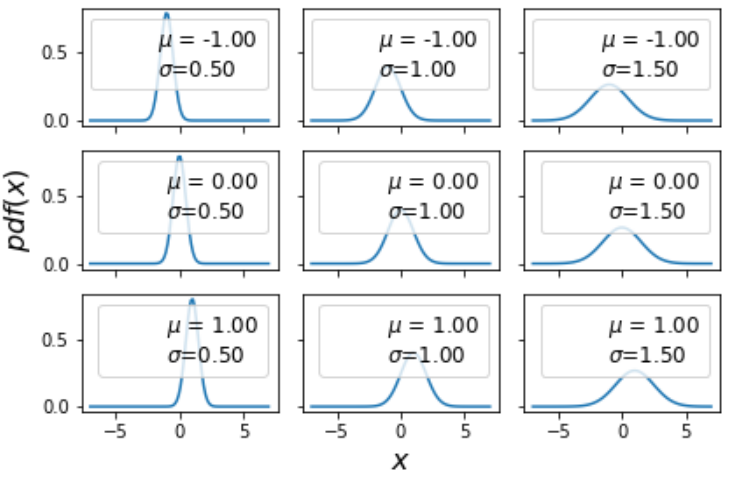

图 2.1 Gaussion Distribution

2.1.2 Coin-Flipping Problen

We will answer this question in a Bayesian setting. We will need data and a probabilistic model.

Data: we will assume that we have already tossed a coin a number of times and we have recorded the number of observed head, so the data part is done.

Model will be discussed soon.

2.1.2.1 The general model

The firs thing we will do is generalize the concept of bias. We will say that a coin with a bias of 1 will always land heads, one with a bias of 0 will always land tails, and one with a bias of 0.5 will land half of the time heads and half of the time tails. To represent the bias, we will use the parameter θ, and to represent the total number of heads for an N number of tosses, we will use the variable y. According to Bayes' theorem we have the following formula:

$$p(\theta|y) \propto p(y|\theta)p(\theta)$$

Notice that we need to specify which prior $p(\theta)$ and likelihood $p(y|\theta)$ we will use. Let's start with the likelihood.

2.1.2.2 Choosing the likelihood

Let's assume that a coin toss does not affect other tosses, that is, we are assuming coin tosses are independent of each other. Let's also assume that only two outcomes are possible, heads or tails. Given these assumptions, a good candidate for the likelihood is the binomial distribution (二项分布):

$$p(y|\theta) = \frac{N!}{N!(N-y)!}\theta^y(1-\theta)^{N-y}$$

This is a discrete distribution returning the probability of getting y heads (or in general, success) out of N coin tosses (or in general, trials or experiments) given a fixed value of θ. The following code generates 9 binomial distributions; each subplot has its own legend indicating the corresponding parameters:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

n_params = [1,2,4]
p_params = [0.25, 0.5, 0.75]
x = np.arange(0, max(n_params) + 1)
f, ax = plt.subplots(len(n_params), len(p_params), sharex=True, sharey=True)
for i in range(3):
    for j in range(3):
        n = n_params[i]
        p = p_params[j]
        y = stats.binom(n=n, p=p).pmf(x)
        ax[i,j].vlines(x, 0, y, colors='b', lw=5)
```

```
            ax[i,j].set_ylim(0,1)
            ax[i,j].plot(0,0,label="n = {:3.2f}\np = {:3.2f}".format(n,p), alpha=0)
            ax[i,j].legend(fontsize=12)
ax[2,1].set_xlabel('$\\theta$', fontsize=14)
ax[1,0].set_ylabel('$p(y|\\theta)$', fontsize=14)
ax[0,0].set_xticks(x)
plt.savefig('binomial_distribution.png')
```

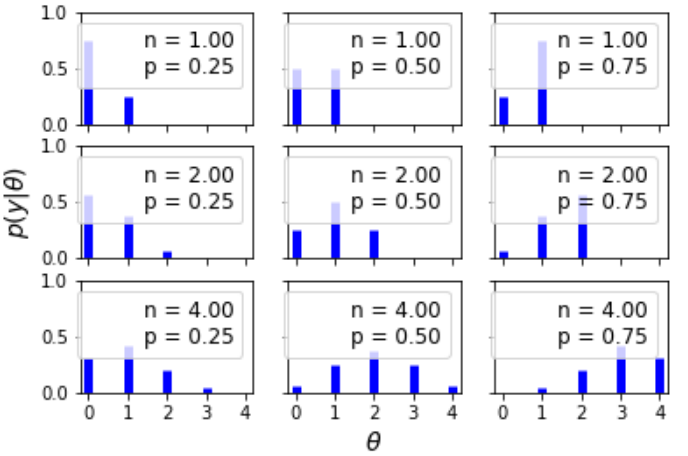

图 2.2 binomial_distribution

The binomial distribution is also a reasonable choice for the likelihood. Intuitively, we can see that θ indicates how likely it is that we will obtain a head when tossing a coin, and we have observed that event y times. Following the same line of reasoning we get that $1 - \theta$ is the chance of getting a tail, and that event has occurred $N - y$ times.

OK, so if we know θ, the binomial distribution will tell us the expected distribution of head. The only problem is that we do not kno θ!. But do not despair; in Bayesian statistics, every time we do not know the value of a parameter, we put a prior on it, so let's move on and choose a prior.

2.1.2.3 Choosing the prior

As a prior we will use a beta distribution (贝塔分布), which is a very common distribution in Bayesian statistics and looks like this:

$$p(\theta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1}(1-\theta)^{\beta-1}$$

The first term is a normalization constant that ensures the distribution integrates to 1.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns
```

```
5
6  params = [0.5, 1, 2, 3]
7  x = np.linspace(0, 1, 100)
8  f, ax = plt.subplots(len(params), len(params), sharex=True, sharey=True)
9  for i in range(4):
10     for j in range(4):
11         a = params[i]
12         b = params[j]
13         y = stats.beta(a, b).pdf(x)
14         ax[i,j].plot(x, y)
15         ax[i,j].plot(0,0,label="$\\alpha$ = {:3.2f}\n$\\beta$={:3.2f}".format(a,b), alpha=0)
16         ax[i,j].legend(fontsize=12)
17 ax[3,0].set_xlabel('$\\theta$', fontsize=14)
18 ax[0,0].set_ylabel('$p(\\theta)$', fontsize=14)
19 plt.savefig('beta_distribution.png')
```

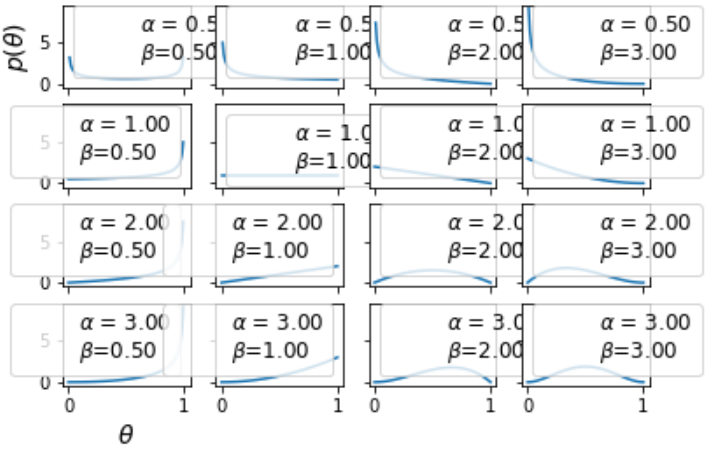

图 2.3 beta_distribution

Why are we using the beta distribution for our model? 1) One reason is that the beta distribution is restricted to be between 0 and 1, in the same way our parameter θ is. 2) Another reason is its versatility (通用性). As we can see in the preceding figure, the distribution adopts several shapes, including a uniform distribution, Gaussian-like distributions, U-like distributions, and so on. 3) A third reason is that the beta distribution is the conjugate prior (共轭先验) of the binomial distribution (which we are using as the likelihood). A conjugate prior of a likelihood is a prior that, when used in combination with the given likelihood, returns a posterior with the same functional form as the prior. There are other pairs of conjugate priors, for example, the Gaussion distribution is the conjugate prior of itself.

For many years, Bayesian analysis was restricted to the use of conjugate priors. Conjugacy ensures mathematical tractability of the posterior, which is important given that common problem in Bayesian statics is to end up with

a posterior we cannot solve analytically. This was a deal beaker before the development of suitable computational methods to solve any possible posterior.

However, modern computational methods to solve Bayesian problems whether we choose conjugate priors or not.

2.1.2.4 Getting the posterior

The Bayes' theorem says that the posterior is proportional to the likelihood times the prior:

$$p(\theta|y) \propto p(y|\theta)p(\theta)$$

which turns out to be

$$p(\theta|y) \propto \frac{N!}{N!(N-y)!}\theta^y(1-\theta)^{N-y}\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$$

To our practical concerns we can drop all the terms that do not depend on θ and our results will still be valid. So we can write the following:

$$p(\theta|y) \propto \theta^y(1-\theta)^{N-y}\theta^{\alpha-1}(1-\theta)^{\beta-1}$$

Reordering it, we have

$$p(\theta|y) \propto \theta^{\alpha-1+y}(1-\theta)^{\beta-1+N-y}$$

We will see that this expression has the same functional form of a beta distribution (except for the normalization) with $\alpha_{posterior} = \alpha_{perior} + y$ and $\beta_{posterior} = \beta_{prior} + N - y$, which means that the posterior for our problem is the beta distribution:

$$p(\theta|y) = Beta(\alpha_{prior} + y, \beta_{prior} + N - y)$$

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from scipy import stats
4  import seaborn as sns
5
6  theta_real = 0.35
7  trials = [0, 1, 2, 3, 4, 8, 16, 32, 50, 150]
8  data = [0, 1, 1, 1, 1, 4, 6, 9, 13, 48]
9
10 beta_params = [(1,1), (0.5,0.5), (20,20)]
11 dist = stats.beta
12 x = np.linspace(0, 1, 100)
13
14 for idx, N in enumerate(trials):
15     if idx == 0:
16         plt.subplot(4, 3, 2)
17     else:
18         plt.subplot(4, 3, idx+3)
```

```
    y = data[idx]
    for (a_prior, b_prior), c in zip(beta_params, ('b', 'r', 'g')):
        p_theta_given_y = dist.pdf(x, a_prior+y, b_prior+N-y)
        plt.fill_between(x, 0, p_theta_given_y, color=c, alpha=0.6)

    plt.axvline(theta_real, ymax=0.3, color='k')
    plt.plot(0,0,label="{:d} experiments\n{:d} heads".format(N,y),alpha=0)
    plt.xlim(0,1)
    plt.ylim(0,12)
    plt.xlabel(r'$\theta$')
    plt.legend()
    plt.gca().axes.get_yaxis().set_visible(False)
plt.tight_layout()
plt.savefig("posterior.png")
```

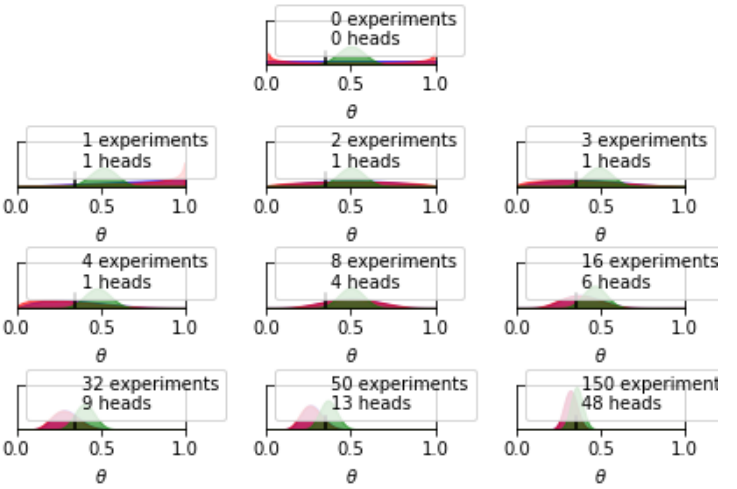

图 2.4 posterior

2.1.2.5 Model notation and visualization

A common notation to succinctly represent probabilistic is as follows:

$$\checkmark \theta \sim Beta(\alpha, \beta)$$

$$\checkmark y \sim Bin(n = 1, p = \theta)$$

2.1.3 Inference engines

There are several methods to compute the posterior even when it is not possible to solve it analytically. Some of the methods are:

1. Non-Markovian methods:
1.1 Grid computing
1.2 Quadratic approximation
1.3 Variation methods

2. Markovian methods:
2.1 Metropolis-Hastings
2.2 Hamiltonian Monte Carlo/No Y-Turn Sampler

NOwadays, Bayesian analysis is performed maninly by using Markov Chain Monte Carlo (MCMC) methods, with variational methods gaining momentum for bigger datasets. We do not need to really understand these methods to perform Bayesian analysis, that's the whole point of probabilistic programming languages, but konwing at least how they work at a conceptual level is often vary useful.

2.1.3.1 Non-Markovian methods

Let's start our discussion of inference engines with the non-Markovian methods. These methods are in general faster than Markovian ones.

Grid computing

Grid computing is a brute-force approach. Even if you are not able to compute the whole posterior, you may be able to compute the prior and the likelihood for a given number of points. Let's assume we want to compute the posterior for a single parameter model. The grid approximation is as follows:

1. Define a reasonable interval for the parameter (the prior should give you a hint).
2. Place a grid of points (generally equidistant) on that interval.
3. For each point in the grid we multiply the likelihood and the prior.

Optionally, we may normalize the computed values (divide the result at each point by the sum of all points).

It is easy to see that a larger number of points (or equivalently a reduced size of the grid) will result in a better approximation. In fact, if we take an infinite number of points we will get the exact posterior. The grid approach does not scale well for many parameters (also referred as dimensions); as you increase the number of parameters the volume of the posterior gets relatively smaller compared with the sampled volume. In other words, we will spend most of the time computing values with an almost null contribution to be posterior, making this approach unfeasible for many statistical and data problems.

The following code implements the grid approach to solve the coin-flipping problem.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
```

```
from scipy import stats
import seaborn as sns

def posterior_grid_approx(grid_points=100, heads=6, tosses=9):
    """
    A grid implementation for the coin-flip problem
    """
    grid = np.linspace(0, 1, grid_points)
    prior = np.repeat(5, grid_points)
    likelihood = stats.binom.pmf(heads, tosses, grid)
    unstd_posterior = likelihood * prior
    posterior = unstd_posterior / unstd_posterior.sum()
    return grid, posterior

# Assuming we made 4 tosses and we observe only 1 head we have the following:

points = 15
h, n = 1, 4
grid, posterior = posterior_grid_approx(points, h, n)
plt.plot(grid, posterior, 'o-', label='heads = {}\ntosses = {}'.format(h, n))
plt.xlabel(r'$\theta$')
plt.legend(loc=0)
plt.savefig('grid_approach.png')
```

$$\checkmark \theta \sim Beta(\alpha, \beta)$$

$$\checkmark y \sim Bin(n = 1, p = \theta)$$

Quadratic method The quadratic approximation, also known as the Laplace method or the normal approximation, consists of approximating the posterior with a Gaussion distribution. This method often works because in general the region close to the mode of the posterior distribution is more or less normal, and in fact in many cases in actually a Gaussian distribution. This method consists of two steps. First, find the mode of the posterior distribution. This method consists of two steps. Fist, find the mode of the posterior distribution. This can be done using optimization methods; that is, methods to find the maximum or minimum of a function, and there are many off-the-shelf methods for this purpose. This will be the mean of the approximating Gaussion. Then we can estimate the curvature of the function near the mode. Based on this curvature, the standard deviation of the approximating Gaussian can be computer. We are going to apply this method once we have introduced PyMC3.

Variational methods Most of modern Bayesian statics is done using Markovian methods (see the next section), but for some problems those methods can be too slow and they do not necessarily parallelize well. The naive approach

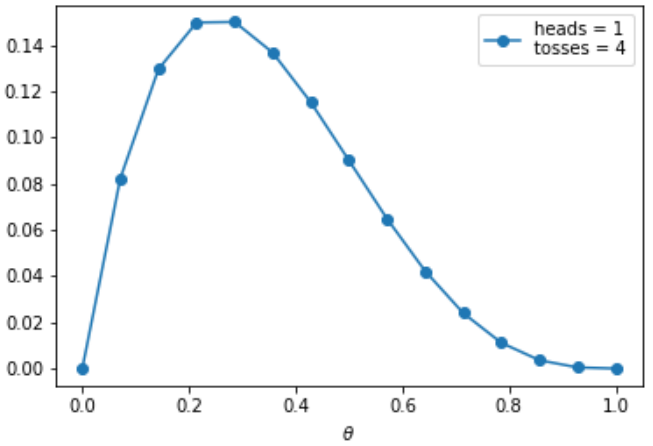

图 2.5 grid_approach to solve the coin flipping

is to simply run n chains in parallel and then combine the results, but for many problems this is not a really good solution. Finding effective ways of parallelizing them is an active research area.

Variational methods could be a better choice for large datasets (think big data) and/or for likelihoods that are too expensive to compute. In addition, these methods are useful for quick approximations to the posterior and as starting points for MCMC methods.

The general idea of variational methods is to approximate the posterior with a simpler distribution; this may sound similar to the Laplace approximation, but the similarities vanish when we check the details of the method. The main drawback of variational methods is that we must come up with a specific algorithm for each model, so it is not really a universal inference engine, but a model-specific one.

Of course, lots of people have tried to automatize variational methods. A recently proposed method is the automatic differentiation variational inference (ADVI) (see http://arxiv.org/abs/1603.00788). At the conceptual level, ADVI works in the following way:

1. Transform the parameters to make them live in the real line. For example, taking the logarithm of a parameter restricted to positive values we obtain an unbounded parameter on the interval $[-\infty, \infty]$.

2. Approximate the unbounded parameters with a Gaussian distribution. Notice that a Gaussian on the transformed parameter space is non-Gaussian on the original parameter space, hence this is not the same as the Laplace approximation.

3. Use an optimization method to make the Gaussian approximation as close as possible to the posterior. This is done by maximizing a quantity known as the Evidence LowerBound (ELBO). How we measure the similarity of two distributions and what ELBO is exactly, at this point, is a mathematical detail.

Markovian methods There is a family of related methods collectively know as MCMC methods. As with the grid computing approximation, we need to be able to compute the likelihood and prior for a given point and we want to approximate the whole posterior distribution. MCMC methods outperform the grid approximation because they are

designed to spend more time in higher probability regions than in lower ones. In fact, a MCMC method will visit different regions of the parameter space in accordance with their relative probabilities. If region A is twice as likely as region B, then we are going to get twice the samples from A as from B. Hence, even if we are not capable of computing the whole posterior analytically, we could use MCMC methods to take samples from it, and the larger the sample size the better the results.

What is in a name? Well, sometimes not much, sometimes a lot. To understand what MCMC methods are we are going to split the method into the two MC parts, the Monte Carlo part and the Markov Chain part.

Monte Carlo The use of random numbers explains the Monte Carlo part of the name. Monte Carlo methods are a very broad family of algorithms that use random sampling to compute or simulate a given process. Monte Carlo is a very famous casino located in the Principality of Monaco. One of the developers of the Monte Carlo method, Stanislaw Ulam, had an uncle who used to gamble there. The key idea Stan had was that while many problems are difficult to solve or even formulate in an exact way, they can be effectively studied by taking samples from them, or by simulating them. In fact, as the story goes, the motivation was to answer questions about the probability of getting a particular hand in a solitary game. One way to solve this problem was to follow the analytical combinatorial problem. Another way, Stan argued, was to play several games of solitaire and just count how many of the hands we play match the particular hand we are interested in! Maybe this sounds obvious, or at least pretty reasonable; for example, you may have used re-sampling methods to solve your statistical problems. But remember this mental experiment was performed about 70 years ago, a time when the first practical computers began to be developed. The first application of the method was to solve a problem of nuclear physics, a problem really hard to tackle using the conventional tools at that time. Nowadays, even personal computers are powerful enough to solve many interesting problems using the Monte Carlo approach and hence these methods are applied to a wide variety of problems in science, engineering, industry, arts, and so on.

A classic pedagogical example of using a Monte Carlo method to compute a quantity of interest is the numerical estimation of . In practice there are better methods for this particular computation, but its pedagocial value still remains. We can estimate the value of π with the following procedure:

1. Throw N points at random into a square of side $2R$.
2. Draw a circle of radius R inscribed in the square and count the number of points that are inside that circle.
3. Estimate $\hat{\pi}$ as the ration $\frac{4 \times inside}{N}$.

A couple of notes: We know a point is inside a circle if the following relation is true: $\sqrt{(x^2 + y^2)} \leq R$

The area of the square is $(2R)^2$ and the area of the circle is πR^2. Thus we know that the ratio of the area of the square to be the area of the circle is $\frac{4}{\pi}$, and the area of the circle and square are proportional to the number of points inside the circle and the total N points, respectively.

Using a few line of Python we can run this simple Monte Carlo simulation and compute π and also the relative error of our estimate compared to the rule value of π:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy import stats
4 import seaborn as sns
```

```
5
6  N = 10000
7  x,y = np.random.uniform(−1, 1, size=(2, N))
8  inside = (x**2 + y**2) <= 1
9  pi = inside.sum() * 4 / N
10 error = abs((pi − np.pi) / pi ) * 100
11
12 outside = np.invert(inside)
13
14 plt.plot(x[inside], y[inside], 'b.')
15 plt.plot(x[outside], y[outside], 'r.')
16 plt.plot(0, 0, label='$\hat \pi$ = {:4.3f}\nerror = {:4.3f}'.format(pi, error), alpha=0)
17 plt.axis('square')
18 plt.legend(frameon=True, framealpha=0.9, fontsize=16)
19
20 plt.savefig('monte_carlo.png')
```

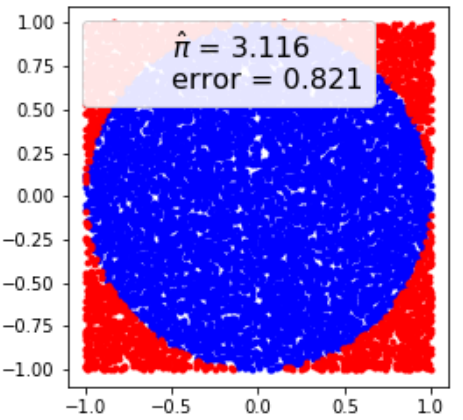

图 2.6 monte_carlo

In the preceding code we can see that the outside variable is only used to get the plot; we do not need it for computing $\frac{4 \times inside}{N}$. Another clarification: because our computation is restricted to the unit circle we can omit computing the square root from the computation of the inside variable.

Markov chain A Markov chain is a mathematical object consisting of a sequence of states and a set of probabilities describing the transitions among those states. A chain is Markovian if the probability of moving to other states depends only on the current state. Given such a chain, we can perform a random walk by choosing a starting point and moving to other states following the transition probabilities. If we somehow find a Markov chain with transitions

proportional to the distribution we want to sample from (the posterior distribution in Bayesian analysis), sampling becomes just a matter of moving between states in this chain. So, how do we find this chain if we do not know the posterior in the first place? Well, there is something known as **detailed balance condition**. Intuitively, this condition says that we should move in a reversible way (a reversible process is a common approximation in physics). That is, the probability of being in state i and moving to state j should be the same as the probability of being in state j and moving towards state i.

In summary, all this means that if we manage to create a Markov Chain satisfying detailed balance we can sample from that chain with the guarantee that we will get samples from the correct distribution. This is a truly remarkable result! The most popular method that guarantees detailed balance is the **Metropolis-Hasting algorithm**.

Metropolis-Hastings To conceptually understand this method, we are going to use the following analogy. Suppose we are interested in finding the volume of water a lake contains and which part of the lake has the deepest point. The water is really muddy so we can't estimate the depth just by looking to the bottom, and the lake is really big, so a grid approximation does not seem like a very good idea. In order to develop a sampling strategy, we seek help from two of our best friends, Markovia and Monty. After some discussion they come up with the following algorithm that requires a boat; nothing fancy, we can even use a wooden raft, and a very long stick. This is cheaper than a sonar and we have already spent all our money on the boat, anyway!

1. Initialize the measuring by choosing a random place in the lake and move the boat there.
2. Use the stick to measure the depth of the lake.
3. Move the boat to some other point and take a new measurement.
4. Compare the two measures in the following way:

4.1. If the new spot is deeper than the old one, write down in your notebook the depth of the new spot and repeat from 2.

4.2 If the spot is shallower than the old one, we have two options: to accept or reject. Accepting means to write down the depth of the new spot and repeat from 2. Rejecting means to go back to the old spot and write down (again) the value for the depth of the old spot.

How do we decide to accept or reject a new spot? Well, the trick is to apply the Metropolis-Hastings criteria. This means to accept the new spot with a probability that is proportional to the ratio of the depth of the new and old spots.

If we follow this iterative procedure, we will get not only the total volume of the lake and the deepest point, but we will also get an approximation of the entire curvature of the bottom of the lake. As you may have already guessed, in this analogy the curvature of the bottom of the lake is the posterior distribution and the deepest point is the mode. According to our friend Markovia, the larger the number of iterations the better the approximation.

Indeed, theory guarantees that under certain general circumstances, we are going to get the exact answer if we get an infinite number of samples. Luckily for us, in practice and for many, many problems, we can get a very accurate approximation using a relatively small number of samples.

Let's look at the method now in a little bit more formal way. For some distributions, like the Gaussian, we have very efficient algorithms to get samples from, but for some other distributions such as many of the posterior distributions, we are going to find this is not the case. Metropolis-Hastings enables us to obtain samples from any distribution with probability $p(x)$ given that we can compute at least a value proportional to it. This is very useful

since in a lot of problems like Bayesian statistics the hard part is to compute the normalization factor, the denominator of the Bayes' theorem. The Metropolis-Hastings algorithm has the following steps:

1. Choose an initial value for our parameter x_i. This can be done randomly or by using some educated guess.

2. We choose a new parameter value x_{i+1}, sampling from an easy-to-sample distribution such as a Gaussian or uniform distribution $Q(x_{i+1}|x_i)$. We can think of this step as perturbing the state x_i somehow.

3. We compute the probability of accepting a new parameter value by using the Metropolis-Hastings criteria $p_\alpha(x_{i+1}|x_i) = \min\left(1, \frac{p(x_{i+1})q(x_i|x_{i+1})}{p(x_i)q(x_{i+1}|x_i)}\right)$.

4. If the probability computed on 3 is larger than the value taken from a uniform distribution on the interval $[0, 1]$ we accept the new state, otherwise we stay in the old state.

5. We iterate from 2 until we have enough samples. Later we will see what enough means.

A couple of things to take into account:

1. If the proposal distribution $Q(x_{i+1}|x_i)$ is symmetric we get $p_\alpha(x_{i+1}|x_i) = \min\left(1, \frac{p(x_{i+1})}{p(x_i)}\right)$, often referred to as **Metropolis criteria** (we drop the **Hastings** apart).

2. Steps 3 and 4 imply that we will always accept or move to a most probable state, to a most probable parameter value. Less probable parameter values are accepted probabilistically given the ratio between the probability of the new parameter value x_{i+1} and the old parameter value x_i. This criteria for accepting steps gives us a more efficient sampling approach compared to the grid approximation, while ensuring a correct sampling.

3. The target distribution (the posterior distribution in Bayesian statistics) is approximated by saving the sampled (or visited) parameter values. We save a sampled value x_{i+1} if we accept moving to a new state x_{i+1}. If we reject moving to x_{i+1}, we save the value of x_i.

At the end of the process we will have a list of values sometimes refereed to as a **sample chain or trace**. If everything was done the right way these samples will be an approximation of the posterior. The most frequent values in our trace will be the most probable values according to the posterior. An advantage of this procedure is that analyzing the posterior is simple. We have effectively transformed integrals (of the posterior) into just summing values in our vector of sampled values. The following code illustrates a very basic implementation of the Metropolis algorithm. Is not meant to solve any real problem, only to show it is possible to sample from a function if we know how to compute its value at a given point. Notice also that the following implementation has nothing Bayesian in it; there is no prior and we do not even have data! Remember that the MCMC methods are very general algorithms that can be applied to a broad array of problems. For example, in a (non-Bayesian) molecular model, instead of func.pdf(x) we would have a function computing the energy of the system for the state x.

The first argument of the metropolis function is a SciPy distribution; we are assuming we do not know how to directly get samples from this distribution.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy import stats
4 import seaborn as sns
5
```

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats
import seaborn as sns

def metropolis(func, steps=10000):
    """
    A very simple Metropolis implementation.
    """
    samples = np.zeros(steps)
    old_x = func.mean()
    old_prob = func.pdf(old_x)

    for i in range(steps):
        new_x = old_x + np.random.normal(0, 0.5)
        new_prob = func.pdf(new_x)
        acceptance = new_prob / old_prob
        if acceptance >= np.random.random():
            samples[i] = new_x
            old_x = new_x
            old_prob = new_prob
        else:
            samples[i] = old_x
    return samples

"""
In the next example we have defined func as a beta function,
simply because is easy to change their parameters and get
different shapes. We are plotting the samples obtained by
metropolisas a histogram and also the True distribution as
a red line:
"""

func = stats.beta(0.4, 2)
samples = metropolis(func=func)
x = np.linspace(0.01, .99, 100)
y = func.pdf(x)
plt.xlim(0, 1)
plt.plot(x, y, 'r-', lw=3, label='True distribution')
plt.hist(samples, bins=30, normed=True, label='Estimated distribution')
```

```
plt.xlabel('$x$', fontsize=14)
plt.ylabel('$pdf(x)$', fontsize=14)
plt.legend(fontsize=14)
plt.savefig('metropolis_algorithm.png')
```

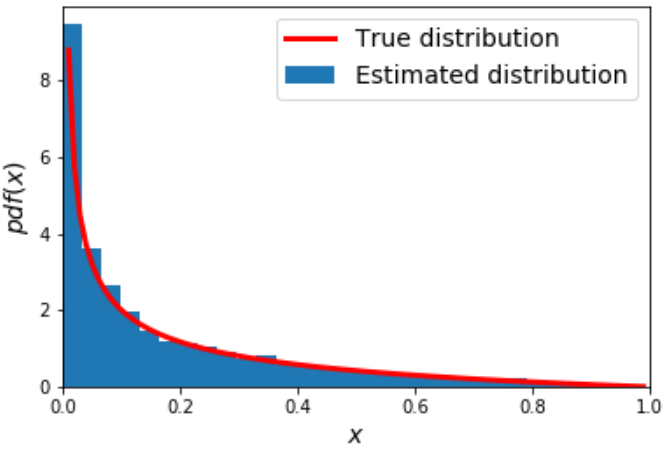

图 2.7 metropolis_algorithm

Hamiltonian Monte Carlo/NUTS MCMC methods, including Metropolis-Hastings, come with the theoretical guarantee that if we take enough samples we will get an accurate approximation of the correct distribution. However, in practice it could take more time than we have to get enough samples. For that reason, alternatives to the general Metropolis-Hastings algorithm have been proposed. Many of those alternative methods such as the Metropolis-Hastings algorithm itself, were developed originally to solve problems in statistical mechanics, a branch of physics that studies properties of atomic and molecular systems. One such modification is known as **Hamiltonian Monte Carlo or Hybrid Monte Carlo (HMC)**. In simple terms a Hamiltonian is a description of the total energy of a physical system. The name Hybrid is also used because is was originally conceived as a hybridization of molecular mechanics, a widely used simulation technique for molecular systems, and Metropolis-Hastings. The HMC method is essentially the same as Metropolis-Hastings except that instead of proposing random displacements of our boat we do something more clever instead; we move the boat following the curvature of the lake's bottom. Why is this clever? Because in doing so we try to avoid one of the main problems of Metropolis-Hastings: the exploration is slow and samples tend to be autocorrelated, since most of the proposed moves are rejected.

So, how can we try to understand this method without going into mathematical details? Imagine we are in the lake with our boat. In order to decide where to move next we let a ball roll at the bottom of the lake, starting from our current position. Remember that this method was brought to us by the same people that treat horses as spheres, so our our ball is not only perfectly spherical, it also has no friction and thus is not slowed down by the water or mud. Well, we throw a ball and we let it roll for a short moment, and then we move the boat to where the ball is. Now we accept or reject this step using the Metropolis criteria just as we saw with the Metropolis-Hastings method. The whole

procedure is repeated a number of times. This modified procedure has a higher chance of accepting new positions, even if they are far away relative to the previous position.

Out of our Gedanken experiment and back to the real world, the price we pay for this much cleverer Hamiltonian-based proposal is that we need to compute gradients of our function. A gradient is just a generalization of the concept of derivative to more than one dimension. We can use gradient information to simulate the ball moving in a curved space. So, we are faced with a trade-off; each HMC step is more expensive to compute than a Metropolis-Hastings one but the probability of accepting that step is much higher with HMC than with Metropolis. For many problems, this compromise turns in favor of the HMC method, especially for complex ones. Another drawback with HMC methods is that to have really good sampling we need to specify a couple of parameters. When done by hand it takes some trial and error and also requires experience from the user, making this procedure a less universal inference engine than we may want. Luckily for us, PyMC3 comes with a relatively new method known as **No-U-Turn Sampler (NUTS)**. This method has proven very useful in providing the sampling efficiency of HMC methods, but without the need to manually adjust any knob.

Other MCMC methods There are plenty of MCMC methods out there and indeed people keep proposing newmethods, so if you think you can improve sampling methods there is a wide range of persons that will be interested in your ideas. Mentioning all of them and their advantages and drawbacks is completely out of the scope of this book. Nevertheless, there are a few worth mentioning because you may hear people talk about them, so it is nice to at least have an idea of what are they talking about.

Another sampler that has been used extensively for molecular systems simulations is the **Replica Exchange** method, also known as **parallel tempering** or **Metropolis Coupled MCMC** (or MC3; maybe that's too many MCs). The basic idea of this method is to simulate different replicas in parallel. Each replica follows the Metropolis-Hastings algorithm. The only difference between replicas is that the value of a parameter called temperature (physics influence once more time!) controls the probability of accepting less probable positions. From time to time, the method attempts a swap between replicas. The swapping is also accepted/rejected according to the Metropolis-Hastings criteria, but this time taking into account both replicas' temperatures. The swapping between chains can be attempted between random chains but it is generally preferable to do it for neighboring replicas; that is, replicas with similar temperatures and hence a higher probability-of-acceptance ratio. The intuition for this method is that as we increase the temperature the probability of accepting the new proposed position increases, and decreases with lower and lower temperatures. Replicas at higher temperatures explore the system more freely; for these replicas the surface becomes effectively flatter and thus easier to explore. For a replica with infinite temperature, all states are equally likely. The exchange between replicas avoids replicas at low temperatures getting trapped in local minima. This method is well suited for exploring systems with multiple minima.

2.1.4 PyMC3 instroduction

第3章　深度学习

Goals to Achieve

1. pytorch basics
2. pytorch projects

§ 3.1 Pytorch

打印模型结构

pip install torchsummary

summary(model, (3, 32, 32))

print(model)

第4章　SuperResolution

Goals to Achieve

1.
2.

§ 4.1 On single image scale-up using sparse-representation

4.1.1 问题描述

用$\mathbf{y}_h \in \mathbf{R}^{N_h}$表示高清图; 用$\mathbf{H} : \mathbf{R}^{N_h} \to \mathbf{R}^{N_h}$表示blur操作，用$\mathbf{S} : \mathbf{R}^{N_h} \to \mathbf{R}^{N_l} (N_l < N_h)$表示decimation(抽取)操作. 假设$\mathbf{H}$表示低通滤波器, $\mathbf{S}$用整数s执行抽取操作(by discarding rows/columns from the input image). 用$\mathbf{z}_l$表示低分辨噪声图

$$\mathbf{z}_l = \mathbf{SHy}_h + \mathbf{v}$$

$\mathbf{v}$表示高斯噪声, $\mathbf{v} \sim \mathcal{N}(0, \sigma^2\mathbf{I})$.

现在有$\mathbf{z}_l$, 目标是找到$\hat{\mathbf{y}} \in \mathbf{R}^{N-h}$使得$\hat{\mathbf{y}} \approx \mathbf{y}_h$. 可用最大似然估计的方法来优化$\|\mathbf{SH}\hat{\mathbf{y}} - \mathbf{z}_l\|_2$.

下面将用Sparse-Land local model来解决这问题. 该模型假设高清图可用字典中的稀疏向量来表示.

4.1.2 Sparse-Land Prior

假设$\mathbf{z}_l$可用简单的插值操作$\mathbf{Q} : \mathbf{R}^{N_l} \to \mathbf{R}^{N_h}$得到高清尺度$\mathbf{y}_h$. 这一过程可以表示为$\mathbf{y}_l$

$$\mathbf{y}_l = \mathbf{Qz}_l = \mathbf{Q}\left(\mathbf{SHy}_h + \mathbf{v}\right) = \mathbf{QSHy}_h + \mathbf{Qv} = \mathbf{L}^{all}\mathbf{y}_h + \tilde{\mathbf{v}}$$

目标是处理$\mathbf{y} \in \mathbf{R}^{N_h}$, 得到近似高清图的$\hat{\mathbf{y}} \in \mathbf{R}^{N_h}$使得$\hat{\mathbf{y}} \approx \mathbf{y}_h$.

针对从$\mathbf{y}_l$上面提取的patch, 我们的目标是估计$\mathbf{y}_h$上面相对应的patch. Let $\mathbf{p}_k^h = \mathbf{R}_k\mathbf{y}_h \in \mathbf{R}^n$表示从高清图上面提取的$\sqrt{n} \times \sqrt{n}$的patch, $\mathbf{R}_k : \mathbf{R}^{N_h} \to \mathbf{R}^n$表示从高清图$\mathbf{y}_h$位置$k$处进行提取的操作.

假设$\mathbf{p}_k^h$ can be represented sparsely by $\mathbf{q}_k \in \mathbf{R}^m$ over the dictionary $\mathbf{A}_h \in \mathbf{R}^{n \times m}$, namely, $\mathbf{p}_k^h = \mathbf{A}_h\mathbf{q}_k$.

现在考虑从低分辨率图$\mathbf{y}_l$的位置k提取尺寸为$\sqrt{n}\times\sqrt{n}$的图像patch $\mathbf{p}_k^l=\mathbf{R}_k\mathbf{y}_l$. 由于能$\mathbf{L}^{all}=\mathbf{QSH}$ 把高清图$\mathbf{y}_h$ 变为低清图$\mathbf{y}_l$, 我们假设$\mathbf{p}_k^l=\mathbf{L}\mathbf{p}_k^l+\widetilde{\mathbf{v}}_k$.

由于$\mathbf{p}_k^h=\mathbf{A}_h\mathbf{q}_k$, 和$\mathbf{L}$相乘得到

$$\mathbf{L}\mathbf{p}_k^h=\mathbf{L}\mathbf{A}_h\mathbf{q}_k$$

利用上面假设的低清和高清patch之间的关系$\mathbf{p}_k^l=\mathbf{L}\mathbf{p}_k^l+\widetilde{\mathbf{v}}_k$, 得到

$$\mathbf{L}\mathbf{A}_h\mathbf{q}_k=\mathbf{L}\mathbf{p}_k^h=\mathbf{p}_k^l-\widetilde{\mathbf{v}}_k$$

也即

$$\left\|\mathbf{p}_k^l-\mathbf{L}\mathbf{A}_h\mathbf{q}\right\|_2\leq\epsilon$$

where ϵ is related to the noise power σ of $\widetilde{\mathbf{v}}$.

The key observation from the above derivations is that the low-resolution patch should be represented by the same sparse vector $\mathbf{q}_k$ over the effective dictionary $\mathbf{A}_l=\mathbf{L}\mathbf{A}_h$, with a controlled error ϵ_l. This implies that for a given low-resolution patch $\mathbf{p}_k^l$, we should find its sparse representation vector, $\mathbf{q}_k$, and then we can recover $\mathbf{q}_k^h$ by simply multiplying this representation by the dictionary $\mathbf{A}_h$.

4.1.3 The single-Image Scale-Up Algorithm

4.1.3.1 Overall Structure

1. 准备高清和低清的patch对: $\left\{\mathbf{p}_k^h,\mathbf{p}_k^l\right\}_j$

2. 字典训练: 分别训练得到$\mathbf{A}_l$和$\mathbf{A}_h$.

训练集构建

§ 4.2 Pytorch

pip install torchsummary

summary(model, (3, 32, 32))

print(model)

第5章　Machine Learning

5.0.1 项目实例

当有一个新问题时：

1) 分析问题

2) 获取数据

3) 可视化数据以便更好的了解数据

4) 为ML准备数据

5) 选择模型并训练

6) Fine-tune模型

7) 展示结果

8) 启动/监视/维护你的ML系统

5.0.2 California housing prices

获取数据

```
import os
import tarfile
import warnings
warnings.filterwarnings("ignore", message="numpy.dtype size changed")
from http import HTTPStatus
import pandas as pd

class Data:
    source_slug = ".\\data\\california-housing-prices\\"
    target_slug = ".\\data_temp\\california-housing-prices\\"
    url = "https://github.com/ageron/handson-ml/raw/master/datasets/housing/
        housing.tgz"
```

表 5.1 top five rows in the dataset

None	*longitude*	*latitude*	*housingmedianage*	*totalrooms*	*totalbedrooms*
0	−122.23	37.88	41.0	880.0	322.0
1	−122.22	37.88	21.0	7099.0	2401.0
2	−122.24	37.85	52.0	1467.0	496.0
3	−122.25	37.85	52.0	1274.0	558.0
4	−122.25	37.85	52.0	1627.0	565.0

```
    source = source_slug + "housing.tgz"
    target = target_slug + "housing.csv"
    chunk_size = 128
def get_data():
    """Gets the data from github and uncompresses it"""
    if os.path.exists(Data.target):
        return

    os.makedirs(Data.target_slug, exist_ok=True)
    os.makedirs(Data.source_slug, exist_ok=True)
    response = requests.get(Data.url, stream=True)
    assert response.status_code == HTTPStatus.OK
    with open(Data.source, "wb") as writer:
        for chunk in response.iter_content(chunk_size=Data.chunk_size):
            writer.write(chunk)
    assert os.path.exists(Data.source)
    compressed = tarfile.open(Data.source)
    compressed.extractall(Data.target_slug)
    compressed.close()
    assert os.path.exists(Data.target)
    return

get_data()
housing = pd.read_csv(Data.target)
print('housing.head-----------------------------------', housing.head())
```

程序输出每行表示一个地区。有10个属性，分别是: longitude/latitude/housing_median_age/total_rooms/total_bed_rooms/population/households/median_income/median_house_value/ocean_proximity. 如下:

info() 方法能打印数据总行数, 以及每个属性的类型和non-null数据的个数

```
print(housing.info())
```

output:

```
RangeIndex: 20640 entries, 0 to 20639
Data columns total 10 columns:
longitude 20640 non−null float64
latitude 20640 non−null float64
housing median age 20640 non−null float64
total rooms 20640 non−null float64
total bedrooms 20433 non−null float64
population 20640 non−null float64
households 20640 non−null float64
median income 20640 non−null float64
median house value 20640 non−null float64
ocean proximity 20640 non−null object
dtypes: float64 (9), object (1)
memory usage: 1.6 MB
```

可以看出, 一共有20640条数据实例, 其中total_bed_rooms属性只有20433个非空值, 说明有207个地区缺失该属性. 另外, ocean_proximity列是object类型, 可以用如**valud_counts()**查看都包含哪些object, 以及每个object出现的次数.

```
print(housing["ocean_proximity"].value_counts())
```

可用**describe()**方法查看数据的numerical attributes.

```
print(housing.describe())
```

程序将输出每条属性对应的count(数值数字出现的次数), mean(该列数据的均值), std, min, 25%(有25%的数是该值), 50%, 75%, max.

可以快速查看每条属性的直方图.代码如下:

```
import matplotlib.pyplot as plt
housing.hist(bins=50,figsize=(20,15))
plt.savefig('housing_price_hist.png')
```

一些直方图tail heavy. 左右两侧的数据偏离中心较大. 这将使得机器学习算法难以detect patterns. 后面我们将对这些tail heavy数据进行transform, 使得直方图更像钟形分布(bell-shaped distributions).

5.0.3 准备测试集

在选择算法之前, 需要先选出一些测试数据. 因为如果在全部数据上面选择算法, 人的大脑可能被某些pattern所吸引, 然后选择一个特定的机器学习算法, 这样的算法效果不好.

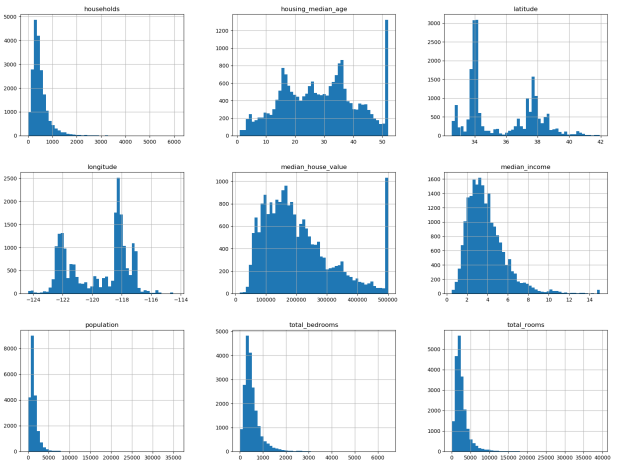

图 5.1　A histogram for each numerical attribute

从数据集中选出20%的数据作为测试集即可.

```
import numpy as np
def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]

train_set, test_set = split_train_test(housing, 0.2)
print(len(train_set))
print(len(test_set))
```

这样我们就把数据集拆分为了16512条训练集和4128条测试集. 然而当你下次启动程序时候, 又将重新选择新的测试集. 可选的做法是在np.random.permutation()前面加上np.random.seed(42), 那样每次就输出一样的shuffled indices了.

一个更实用的做法是用每条实例的identifier去选择测试集. 这样能保证每次执行程序, 选择出的测试集都是一样的. 即便当你添加新的数据到数据集中, 新的测试集将从新加数据中选出20%的数据加到自己的样本中, 并且不包括之前测试集中的数据.

```
import numpy as np
from zlib import crc32
def test_set_check(identifier, test_ratio):
```

```
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32
def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]

# 在数据集前面加上一列, 用赋值index, 该列作为indentifier id
housing_with_id = housing.reset_index() # adds an 'index' column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")

print(len(train_set)) # 16512
print(len(test_set)) # 4128
```

当你用数据的行号作为id时候, 每次新加数据需要加到数据集的后面, 并且前面的数据不能被删除某条. 这样很不方便. 我们应该选择一个不变的id, 一个地区的经纬度是不变的, 可以把他们2个合并起来作为一个id就能解决上面的问题.

```
housing_with_id['id'] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

假设专家告诉你说median income对房屋价格影响关系比较大, 那么你需要使得测试集里的median income也和整个数据集中的median income分布类似. 让我们再看一下之前的median income直方图: 大多数值在2和5($20,000-$50,000)附近, 也有一些值远超出了6(i.e., $60,000). It is important to have a sufficient number of instances in your data-set for each stratum, or else the estimate of the stratum's importance may be biased. 下面的代码创建了一个income category 属性, 该属性是median income除以1.5(为了限制income种类), and rounding up using ceil (to have discrete categories), 然后保留低于5的种类并且把其他类变成5. income categories直方图也画出来了.

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
ax = housing["income_cat"].hist()
fig = ax.get_figure()
fig.savefig('income_hist.png')
```

下面就可以根据柱状图的比例来取样了, 用到的函数是**StratifiedShuffleSplit**.

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]

```

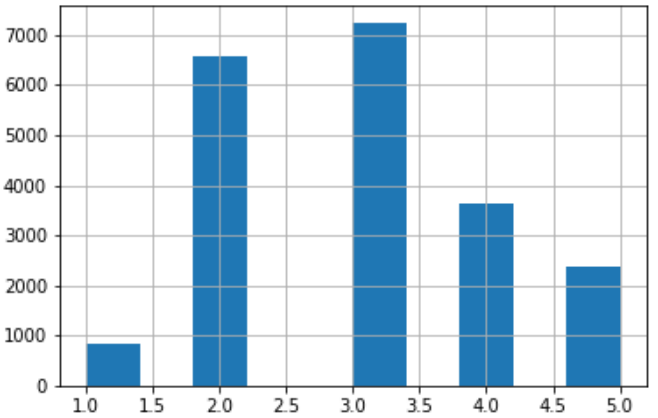

图 5.2　income hist

```
print('test:',strat_test_set["income_cat"].value_counts() / len(strat_test_set))
print('train:', strat_train_set["income_cat"].value_counts() / len(strat_train_set))
```

二者输出基本一样. 可以看出, 测试集的数据分布和训练集基本一样.

下面去掉income_cat属性, 这样数据就恢复到之前的样子了.

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

5.0.4 数据可视化分析

首先把测试集放在一边, 现在只考虑训练集. 如果训练集比较大, 可以取小样本进行操作. 首先copy一下数据.

```
housing = strat_train_set.copy()
```

5.0.4.1 数据的地理信息

```
ax = housing.plot(kind="scatter", x="longitude", y="latitude")
fig = ax.get_figure()
fig.savefig('geographical_visualization.png')
```

看起来像加利福尼亚的版图. 但是上图看不出有用的信息. 可以通过把alpha设置为0.1, 这样能看清楚哪个地方数据密度大.

```
ax = housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

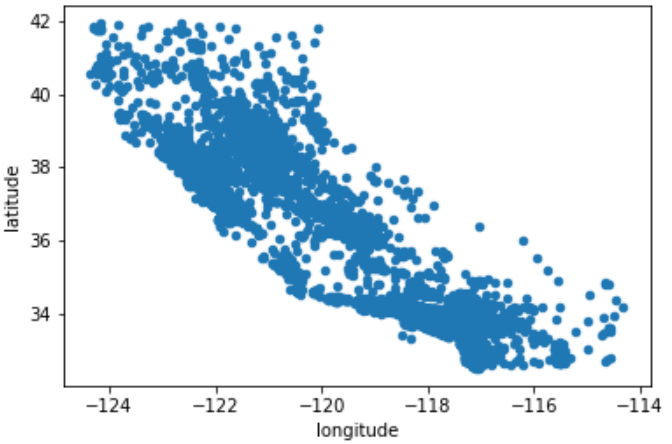

图 5.3 geographical visualization

```
fig = ax.get_figure()
fig.savefig('geographical_visualization_high_density_areas.png')
```

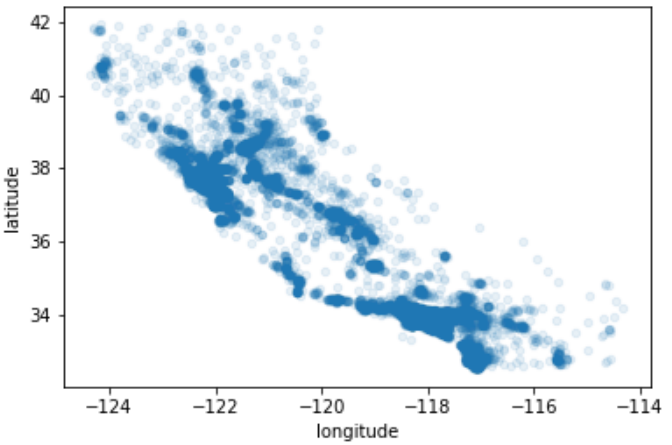

图 5.4 geographical visualization with high density areas

这样能看出一些有用信息: Los Angeles和San Diego附近的bay area密度大, Central Valley附近也密度大.

The radius of each circle represents the district’ s population (option s), and the color represents the price (option c). We will use a predefined color map (option cmap) called jet, which ranges from blue (low values) to red (high prices):

```
ax = housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.4,
s=housing["population"]/100, label="population", figsize=(10,7),
```

```
c="median_house_value", cmap=plt.get_cmap("jet"), colorbar=True,)
plt.legend()
fig = ax.get_figure()
fig.savefig('  California_housing_prices.png')
```

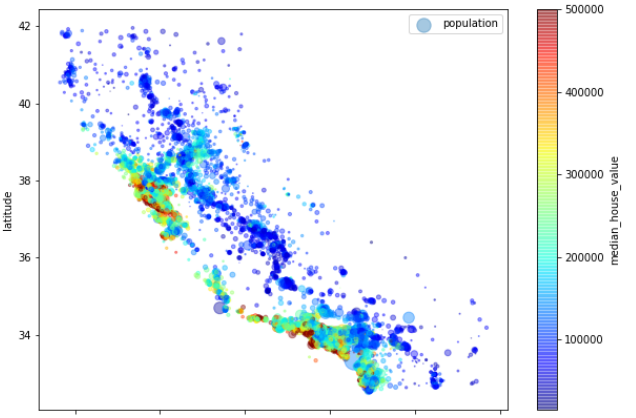

图 5.5　California housing prices

5.0.5 找相关性

计算其他属性和某属性之间的标准相关系数(standard correlation coefficient). 相关系数变化范围-1到1.

```
corr_matrix = housing.corr()
print(corr_matrix["median_house_value"].sort_values(ascending=False))
```

这样就打印出来其他属性和median_house_value之间的相关性了. 可以看出, median_house_value最相关, 相关系数为1, median_income次相关, 相关系数0.687170. 用Pandas'的**scatter_matrix**函数可以画每个属性和其他属性之间的相关性图. 现在的数据集有11条属性, 可以画出121 条关系图. 现在我们只考虑和median housing value最相关的属性.

```
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms","housing_median_age
    "]
ax = scatter_matrix(housing[attributes], figsize=(12, 8))
plt.savefig(r"figure_1.png")
```

我们知道和房屋价格最相关的属性是median income, 现在来画一下他们之间的散点图.

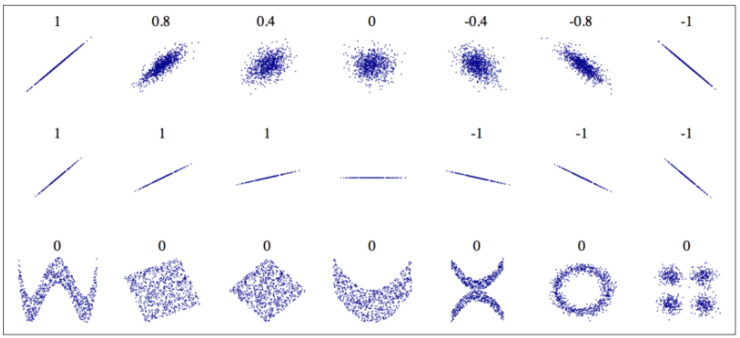

图 5.6 Standard correlation coefficient of various datasets

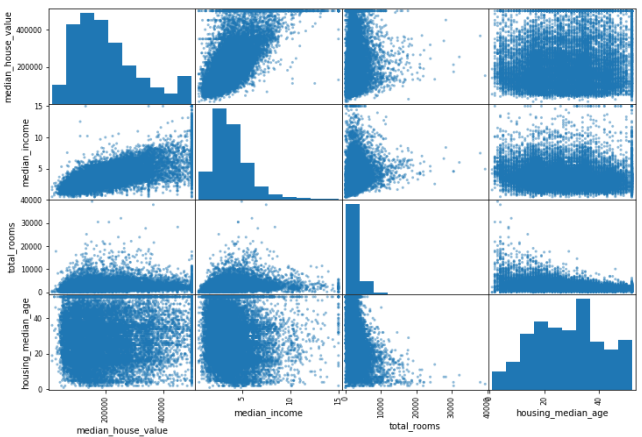

图 5.7 scatter matrix all

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",alpha=0.1)
plt.savefig(r"median_income_versus_median_house_value.png")
```

5.0.6 属性联合

如果不知道某地区的家庭总数, 那么知道该地区的room总数意义不大, 我们真正需要的是number of rooms per household. 同样, 知道总的卧室数意义也不大, 我们需要拿他和房间总数比较. 每个家庭的人口数也有意义.

```
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"]=housing["population"]/housing["households"]

corr_matrix = housing.corr()
```

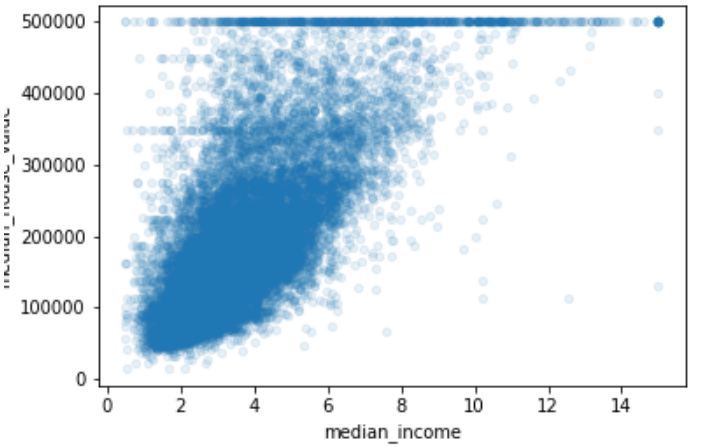

图 5.8　median income versus median house value

```
print('----------------------------------')
print(corr_matrix["median_house_value"].sort_values(ascending=False))
```

可以看出, 相比总room数和总bedroom数来说, 新的bedrooms_per_room属性和房屋价格更相关. 明显地, bedroom/room比值越小房屋价格越贵. 和总房间数相比, 每个家庭的房间数也和房屋价格更相关. 明显地, 房子越大, 房子越贵.

5.0.7 为机器学习算法准备数据

drop复制了一份数据, 并不影响strat_train_set

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

5.0.7.1 数据清洗

total_bedrooms属性有一些数据缺失. 有三种方法处理:

option1: 把对应的地区从数据中删除

option2: 删除整个属性

option3: 设置为某值(zero, mean, the median)

可用下面的代码实现

```
housing.dropna(subset=["total_bedrooms"]) # option 1
housing.drop("total_bedrooms", axis=1) # option 2
median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

5.0.8 处理文本属性

ocean_proximity属性是文本, 不能用来计算.

```
housing_cat = housing[['ocean_proximity']]
print(housing_cat.head(10))
```

需要对这些文本进行编码, 使用sklearn-learn的**OridinalEncoder**类

```
from sklearn.preprocessing import OrdinalEncoder
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
print(housing_cat_encoded[:10])
print(ordinal_encoder.categories_) # 分别对应0, 1, 2, 3, 类4
```

机器学习算法会假设相邻的两个值比距离较远的两个值更相似. 这对于”bad” ”average”, ”good”, ”excellent”这样的类别适用. 但是对ocean_proximity属性不适用, 举例来说, 0 (OCEAN)和4 (OCEAN NEARBY)更相似, 而不是0(OCEAN)和1(INLAND)更相似. 为解决这一问题, 使用one-hot 编码.

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
print(housing_cat_1hot.toarray())
print(cat_encoder.categories_)
```

5.0.9 Custom Transformers

把变换写成如下类的形式可以使代码无缝嵌套进Scikit-Learn中. 如下是合并类属性的变换

```
from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]
```

```
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

上面的变换有一个超参, add_bedrooms_per_room, 默认是True. 该超参可让你很容易知道添加的这条属性是否对提升精度有帮助.

5.0.10 特征尺度变化

total number of rooms从6变到39,320, 而median incomes只从0变到15. 尺度差异太大, 需要解决.

有两种方法, 1) min-max scaling (normalization, 变到0-1), sklearn对应的函数是**MinMaxScaler**, 有个参数**feature_range**让你选择数据上下界; 2) standardization.

Standardization is quite different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0 – 15 down to 0 – 0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called StandardScaler for standardization.

5.0.11 Transformation pipelines

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),])

housing_num_tr = num_pipeline.fit_transform(housing_num)

from sklearn.compose import ColumnTransformer
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
full_pipeline = ColumnTransformer([("num", num_pipeline, num_attribs),
                                   ("cat", OneHotEncoder(), cat_attribs),])
housing_prepared = full_pipeline.fit_transform(housing)
```

5.0.12 选择和训练模型

线性回归模型.

至此, 先给出一份相对完整的代码

```
import os
import tarfile
import warnings
warnings.filterwarnings("ignore", message="numpy.dtype size changed")
from http import HTTPStatus
import pandas as pd

class Data:
    source_slug = ".\\data\\california-housing-prices\\"
    target_slug = ".\\data_temp\\california-housing-prices\\"
    url = "https://github.com/ageron/handson-ml/raw/master/datasets/housing/
        housing.tgz"
    source = source_slug + "housing.tgz"
    target = target_slug + "housing.csv"
    chunk_size = 128
def get_data():
    """Gets the data from github and uncompresses it"""
    if os.path.exists(Data.target):
        return

    os.makedirs(Data.target_slug, exist_ok=True)
    os.makedirs(Data.source_slug, exist_ok=True)
    response = requests.get(Data.url, stream=True)
    assert response.status_code == HTTPStatus.OK
    with open(Data.source, "wb") as writer:
        for chunk in response.iter_content(chunk_size=Data.chunk_size):
            writer.write(chunk)
    assert os.path.exists(Data.source)
    compressed = tarfile.open(Data.source)
    compressed.extractall(Data.target_slug)
    compressed.close()
    assert os.path.exists(Data.target)
    return

get_data()
```

```
housing = pd.read_csv(Data.target)

import numpy as np
from zlib import crc32
def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32
def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]

housing_with_id = housing.reset_index() # adds an 'index' column

housing_with_id['id'] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")

print(len(train_set))
print(len(test_set))

housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)



from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]

print('test:',strat_test_set["income_cat"].value_counts() / len(strat_test_set))
print('train:', strat_train_set["income_cat"].value_counts() / len(strat_train_set))

for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)


from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
```

```
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]
attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)

housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

housing.dropna(subset=["total_bedrooms"]) # option 1
housing.drop("total_bedrooms", axis=1) # option 2
median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"].fillna(median, inplace=True)

from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median") # 把缺失的值替换为median
housing_num = housing.drop("ocean_proximity", axis=1)

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),])

housing_num_tr = num_pipeline.fit_transform(housing_num)

from sklearn.compose import ColumnTransformer
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
full_pipeline = ColumnTransformer([("num", num_pipeline, num_attribs),
```

```
                                        ("cat", OneHotEncoder(), cat_attribs),])
housing_prepared = full_pipeline.fit_transform(housing)

from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
print(housing_prepared.shape)
print(housing_labels.shape)
lin_reg.fit(housing_prepared, housing_labels)
```

5.0.13 在训练集上面训练和Evaluating

下面用Linear Regression的方法训练模型, 并且查看训练集上面的表现, 发现误差较大. 然后用**mean_squared_erro**查看模型在整个训练集上面的精度为68628, 而大多数地区的median_housing_values 变化范围是$120,000到$265,000, 可看出误差68,628不能令人满意.

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
print(housing_prepared.shape)
print(housing_labels.shape)
lin_reg.fit(housing_prepared, housing_labels) # got the machine model
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions:", lin_reg.predict(some_data_prepared))
print("Labels:", list(some_labels))

from sklearn.metrics import mean_squared_error
housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
print(lin_rmse) # 68628.1981984..
```

模型精度不准的原因在于数据不足, 或者模型不够强大. 解决underfitting的方法在于选择一个更强大的模型, 送入better features.加更多的features (e.g., the log of the population). 下面用决策树训练一个更强大的模型, 决策树能够找到数据之间复杂的非线性关系.

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
housing_predictions = tree_reg.predict(housing_prepared)
```

```
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
print(tree_rmse) # 0
```

MSE输出为0, 模型过拟合. 我们需要用有份训练集来训练, 另一部分用来模型validation.

5.0.14 Cross-Validation

train_test_split可用来把训练集拆分为小的训练集和验证集, 然后在小的训练集上面训练, 在验证集上面做交叉验证.

下面的代码把训练集拆分为10份, 然后在决策树模型上面训练10次, 每次从10份中选一份作为交叉验证, 另外九份用来做模型训练. 结果是十次评估的array.

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
#tree_reg.fit(housing_prepared, housing_labels)
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels, scoring="neg_mean_squared_error", cv
    =10)
tree_rmse_scores = np.sqrt(-scores)
display_scores(tree_rmse_scores)
```

输出的Mean: 71407, Standard deviation: 2439.434, 说明决策树有一个接近71407的得分, ±2439是上下浮动值. (上的决策树模型也过拟合, 模型精度还不如之前的线性回归).

下面试一下**RandomForestRegressor**模型. 随机森林的原理是在随机feature的子集中训练很多决策树. 在很多其他模型基础上构建新模型这叫作*Ensemble Learning*.

5.0.15 Fine-Tune your model

可以用Grid Search的方法或者Randomized Search的方法(**GridSearchCV**和**RandomizedSearchCV**)自动搜索超参空间, 选择最佳组合.

5.0.16 在测试集上面测试模型效果

§ 5.1 Classification

§ 5.2 无监督学习

Yann LeCun famously said that ”if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake”.

假如说产线要实现一套缺陷检测装置, 需要拍摄很多图片, 然后判断图片是好的还是有缺陷的. 有监督的学习方法需要对图片进行标记”normal”和”defective”, 耗费大量人力. 另外, 如果人工标注, 标记的样本如果比较少, 分类结果不佳. 采用无监督学习的方法可有效解决这问题.

这章将介绍:

1. 聚类. 用到的方法有: kmeans, DBSCAN, Gaussian mixture models.
2. Anomaly detection. 异常检测
3. Density estimation.

5.2.1 Clustering

聚类是无监督学习. 有监督的学习方法是: Logistic regression, SVMs, Random Forest

聚类的应用:

1. 客户群体划分. 这样可更好理解他们需要什么, 这对推荐系统也是有用的.
2. 数据分析. 当拿到新的数据集后, 可用先把数据聚类, 然后分开分析不同种类的数据.
3. 降维.
4. 缺陷检测.
5. 用在半监督学习. 假设你有很少的label, 可以通过聚类得到更多的label, 然后用监督学习提升效果.
6. 用在搜索系统中. 搜索系统会推荐你搜索类似的图片. 类似的图片是聚类得到的.

5.2.1.1 K-Means

假设有如下图分布的数据x, 下面的代码将找到每一类的中心, 并且将每类放在闭合空间中.

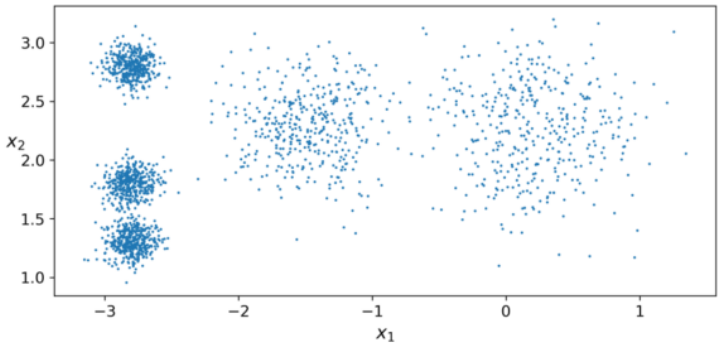

图 5.9 An unlabeled dataset composed of five blobs of instances.

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k) # k must be specified
y_pred = kmeans.fit_predict(x)
# print(y_pred)
# array([4, 0, 1, ..., 2, 1, 0], dtype=int32)

# give the 5 centroids
print(kmeans.cluster_centers_)
```

然后y_pred输出每一个数据的标签label. 以及每类数据中心的坐标值.

然后给一对数据, 就能判断其所属的类别了.

```
x_new = np.array([[0,2], [3,2], [−3,3], [−3,2.5]])
print(kmeans.predict(x_new))
# output is array([1,1,2,2], dtype=int32)
```

上面每类数据预测到的label都是具体的值, 这叫**hard clustering**. 如果给出属于每一类的得分, 这叫**soft clustering**. 这个得分可以是该数据距离类中心的距离, 也可以是相似性得分(Gaussian Radias Basis). Kmeans的transform()方法测量该数据距离中心的距离:

```
print(kmeans.transorm(x_new))
# output is four array, not four number
```

可通过改变kmeans初始化参数来实验得到更好的模型Kmeans(n_clusters=5, init=good_init, n_init=1). 另外, Kmeans默认每次训练执行十次(n_init)默认等10, 10次后Kmean输出精度高的模型, 他是计算每个数据距离离它最近的中心的mean squared distance得到的. 我们可通过Kmeans.inertia_打印出来看看, 该值越小越好.

5.2.1.2 Mini-batch K-Means

该方法每次训练不使用全部数据集, 而是使用部分数据集, 这样可加速训练3-4倍, 同时也解决了KMeans方法在大数据集情况下内存不足的问题.

```
from sklearn.cluster import MiniBatchKmeans
minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(x)
```

MiniBatchKMeans方法可加速模型训练, 但是他的inertia有一些大, 尤其是当分类数越多, inertia越大.

5.2.1.3 how to select K

inertia越小, 精度越高, 那么是不是可用根据inertia来选K呢, 答案是否定的. 他们二者之间的关系曲线如下图, 该曲线也被称为"elbow rule". 从上图可用看出inertia随着k的增加始终在减少, 然而实际情况是对该分类任务来说, k=4才是最佳选择.

实际中我们可通过**silhouette_score()**来选择最佳k.

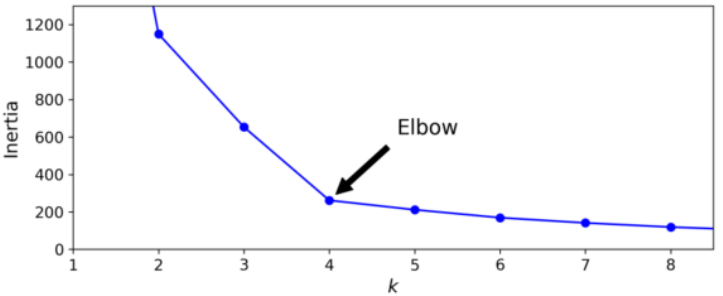

图 5.10 Selecting the number of clusters k using the elbow rule.

```
from sklearn.metrics import silhouette_score
silhouette_score(X, kmeans.labels_)
# output is 0.6555176425
```

多取几组k, 画出silhouette score和k之间的关系 从上图看到k=4才是最佳选择, k=5也是一个不错的选择. 还可

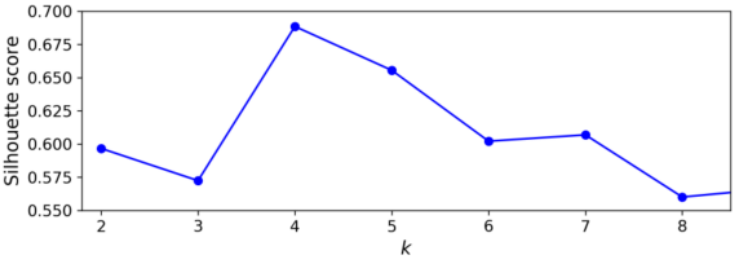

图 5.11 Selecting the number of clusters k using the silhouette score.

以通过画silhouette diagram来选择最佳k, 这里略去.

5.2.2 K-Means的局限

K-Means不适用:

1) the clusters have varying sizes

2) different densities

3) non-spherical shapes

5.2.3 聚类在数据预处理中的应用

在MNIST手写字体识别中, 可提升精度. 提升前流程如下, 精度是96.7%.

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

X_digits, y_digits = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)

result = log_reg.score(X_test, y_test)
# output is: 0.9666666666666667
```

虽然是有0-9十个数, 然而每个数有不同的写法. 下面先进性聚类再进行回归

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([("kmeans", KMeans(n_clusters=50)),("log_reg", LogisticRegression()),])

pipeline.fit(X_train, y_train)

pipeline.score(X_test, y_test)
# output is: 0.982222222222222
```

下面随机选取k使得最终精度最高. 将使用**GridSearchCV**来选择最佳聚类个数k.

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)

# get the best k
print(grid_clf.best_params_)
# output is {'kmeans__n_clusters': 90}

# get the accuracy
print(grid_clf.score(X_test, y_test))
# output is 0.9844444444444445
```

以上, 我们得到了90类, 精度为98.4%.

5.2.4 聚类用于半监督学习

适用于有很多未标注的数据和少量已经标注的数据. 假设digits数据集只有50条标注数据, 其他都是未标注数据. 我们训练一个线性回归模型:

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])

# check mode performance on test set
print(log_reg.score(X_test, y_test))
# output is 0.8266666666667
```

精度只有82.7%, 精度显然没有在全部训练集上面高. 下面先聚50类类, 然后对于每一类找到距离该类中心最近的图片, 我们称之为representative images.

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx
```

这50类代表性图片如下: 然后我们对这50给代表性图片手动加label

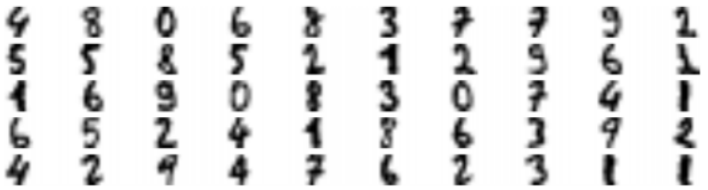

图 5.12 Fifty representative digit images (one per cluster).

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

这样我们就得到了50类带标签的数据. 下面用这50个带标签的数据进行训练, 可发现精度从82.7%提升到了92.4%.

```
log_reg = LogisticRegression()
log_reg.fit(X_representative_digits, y_representative_digits)
print(log_reg.score(X_test, y_test))

# output is: 0.92444444444
```

同样是用50个带标签的数进行训练, 由于先做了聚类, 精度提升了很多.

下面通过label propagation的方式继续提升精度.

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train_propagated)

print(log_reg.score(X_test, y_test))
# output is: 0.92888888888889
```

精度提升一点点. 精度提升没那么大的原因是我们把有代表性图片的标签赋给了该图片所在的类, 此时由于部分图片其实是在各个类的边界线上的, 按照这样的标注方式很可能被标注错误. 下面将propagate 20%的label.

```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]

# train the model
log_reg = LogisticRegression()
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)

print(log_reg.score(X_test, y_test))
# output is: 0.9422222222222222
```

现在我们用了50个标注的实例, 得到了94.2的精度. 用全部标注的数据得到的精度是96.7%. This is because the propagated labels are actually pretty good, their accuracy is very close to 99

```
np.mean(y_train_partially_propagated == y_train[partially_propagated])
```

```
# output is: 0.9896907216494846
```

5.2.5 DBSCAN

K-Means聚类方式的局限在上面已介绍了. DBSCAN聚类方法能够对任意形状的数据进行聚类. 该方法对连续区域内的高密度数据进行聚类. 原理如下:

1. 对每个数据, 算法统计其周围小邻域范围内有多少实例.

2. 如果一个数据至少有min_samples个数据在其邻域内, 那该数据叫做core instance. 也就是说core instance是在dense regions.

3. 在core instance周围的所有数据属于同一类. 这样可能把其他的core instances也包括进去了, 因此很多邻域core instances组成了一类.

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)

# print all the instances' labels
print(dbscan.labels_)

# output is
# array([ 0, 2, −1, −1, 1, 0, 0, 0, ..., 3, 2, 3, 3, 4, 2, 6, 3])
```

label等于-1表示算法认为该数据为异常点. 而core instances可用如下代码获得

```
len(dbscan.core_sample_indices_)
dbscan.core_sample_indices_
```

该算法没有predict()方法. 我们需要用算法得到的label, 然后用其他算法进行训练, 然后才能预测

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])

# use the model to predict
X_new = np.array([[−0.5, 0], [0, 0.5], [1, −0.1], [2, 1]])
knn.predict(X_new)
# outut is: rray([1, 0, 1, 0])

```

```
knn.predict_proba(X_new)
# output is: prop array
```

5.2.6 Gaussian Mixture Model (GMM)

GMM假设数据分布服从参数未知的多种高斯分布函数, 同类数据可聚类形成一个椭圆形的高斯分布. GMM可用于density estimation, clustering and anomaly detection. 适用于数据分布形状为椭圆形的数据.

§ 5.3 Dimensionality Reduction

训练数据特征太多导致训练太慢, 然后使得很难得到好的训练结果. 这问题叫做curse of dimensionality.

实际问题中, 很容易把数据的维度大幅降低. 比如说, 考虑MNIST数据集, 图像周围几乎都是白色, 因此可用把这些像素丢弃而并不会损失很多信息. 这些像素对分类任务不重要. 不仅如此, 相邻的2个像素通常高度相关, 就算把他们合并为1个像素也不会损失很多信息.

有时候降维会使得模型效果变差. 很少时候降维可用过滤掉噪声. 本节主要介绍两种降维方法: projection和manifold learning.

5.3.1 Projection

降维可将下面3维空间数据降到二维平面: 然而降维对于如下数据来说可能效果不好. projection得到的

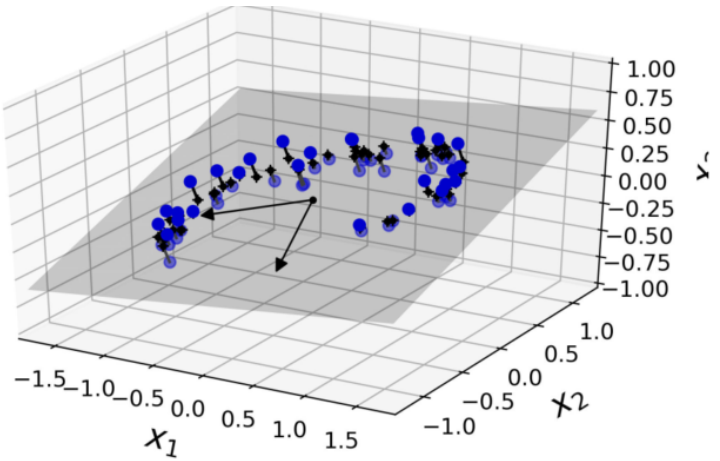

图 5.13 三维空间数据.

结果如下图左侧. 这并不是我们想要的结果, 我们想要的结果在下图右侧.

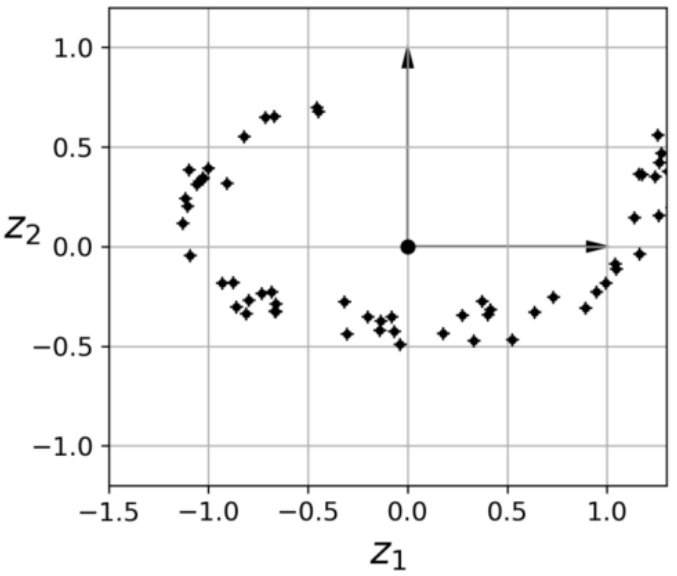

图 5.14 通过projection把三维空间数据降低到2维空间.

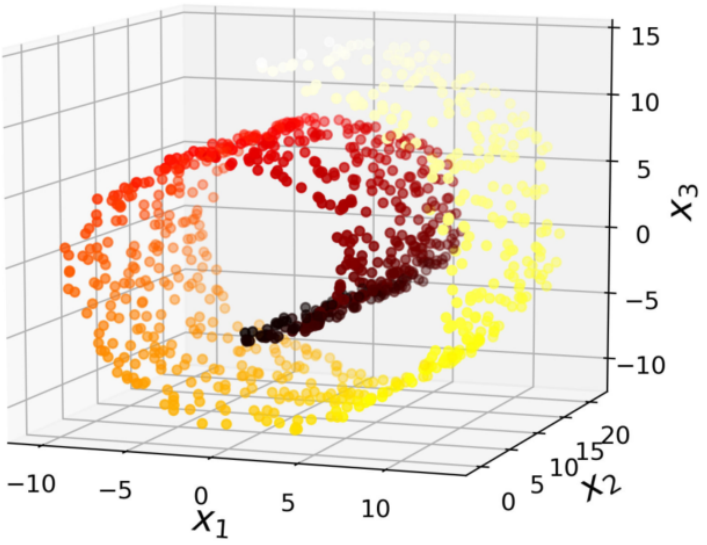

图 5.15 swiss roll dataset.

5.3.2 Manifold learning

现实世界中很多高维度数据都聚集在一个低维度的manifold附近(manifold hypothesis, most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.). 很多高维降维问题通过modeling the manifold.

另外, 如果数据能够在低维度空间表示, 那么手头的任务(比如分类/回归)会更简单. 也有特例.

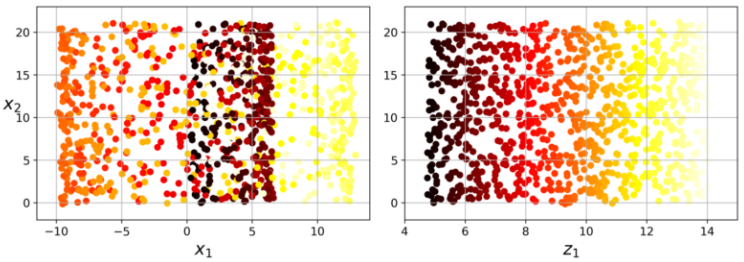

图 5.16 Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right).

§ 5.4 PCA

PCA是最流行的降维方法, 他首先识别出超平面, 然后把数据投影到该平面.

考虑下图左侧数据, 以及三个超平面. 把数据投影到实现可使得variance最大. 把数据投影到点线, variance很小, 投影到另外一条线variance 居中. Vairance最大的投影方式最好. 这就是PCA背后的基本原理.

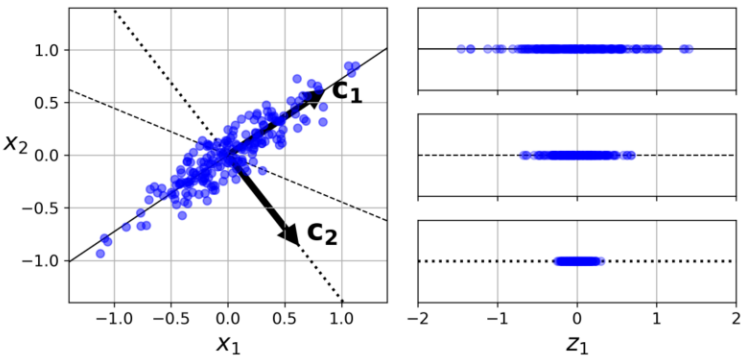

图 5.17 Selecting the subspace onto which to project.

5.4.1 Principal components

对上面的二维情况来说, PCA找到axis c1, 以及垂直它的c2. C1和C2称为数据的i^{th} principal component (PC). 如果数据是三维的, c3将垂直纸面向上或者向下.

我们可用singular value decomposition (SVD)矩阵因式分解的方法得到数据的主成分. SVD返回三个矩阵$\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}^T$, 其中$\mathbf{V}^T$包括所有的主成分.

$$\mathbf{V} = (\mathbf{c_1}, \mathbf{c}_2, \mathbf{c}_3)$$

可用下面的代码得到训练集所有的主成分, 并提取前两个PCs.

```
X_centered = X − X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
```

```
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

有了这些超平面, 我们就能把数据映射到上面了(使得方差最大). 下面代码把训练数据映射到为d维数据.

$$\mathbf{X}_{d-proj} = \mathbf{X}\mathbf{W}_d$$

为最大限度保留训练数据的方差, 比如95%, 可用如下代码

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

还可以画出维度和方差之间的关系曲线如下图

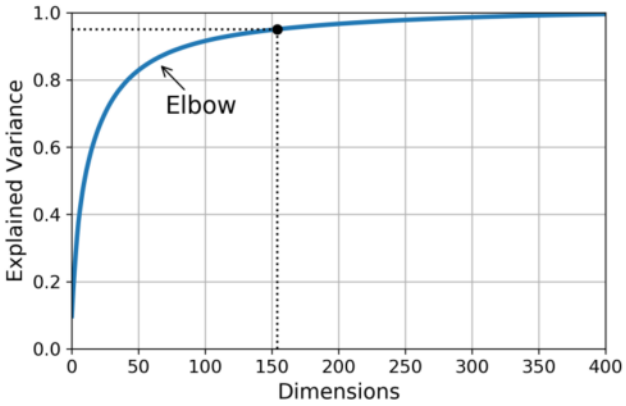

图 5.18 Explained variance as a function of the number of dimensions.

当我们使用PCA保留95%的方差处理MNIST数据集, 每个图片降低到150个feature, 而不是原来的784个feature. 数据的大小变为原来的20%. 这可极大加速模型训练. 压缩后的MNIST数据如下图.

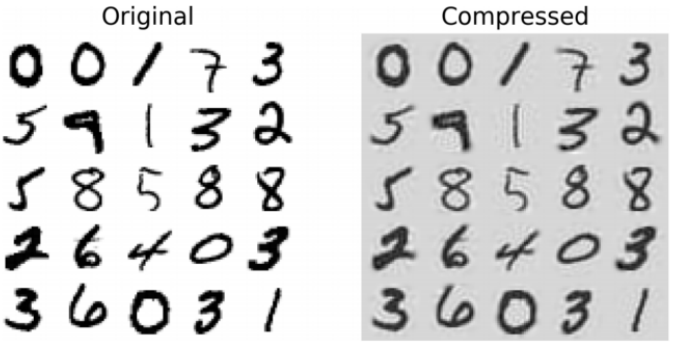

图 5.19 MNIST compression preserving 95% of the variance.

5.4.2 other PCA

Randomized PCA能提速. svd_solver默认为auto.

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

无论是PCA还是RandomizedPCA都是一次把训练数据送进去. 而Incremental PCA可用把数据分批送进去. 该方法适用于大量数据.

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_train)
```

另外, 可用使用kernel trick, 它可把数据映射到到高维度空间(called feature space), 然后就能用非线性分类器或者regression with SVM. 再高维度空间的一个线性分界线其实是低维度空间复杂的非线性分界线.

于是有了Kernel PCA. 常用的kernel有RBF kernel和sigmoid kernel.

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

§ 5.5 Application

kPCA是无监督学习的方法, 可用作有监督学习的预处理步骤. 比如说可用在分类任务中, 先用grid search选择kernel和超参, 然后再进行分类.

下面的代码用kPCA先降维至2维, 然后用线性回归来分类. 然后用Grid SearchCV来找最好的kernel和gamma值.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([("kpca", KernelPCA(n_components=2)),
                ("log_reg", LogisticRegression())])
param_grid = [{"kpca__gamma": np.linspace(0.03, 0.05, 10),
               "kpca__kernel": ["rbf", "sigmoid"]}]
```

```

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

然后可打印出最好的kernel和超参

```
print(grid_search.best_params_)

# output is
# {'kpca__gamma': 0.043333333333333335, 'kpca__kernel':'rbf'}
```

第6章　MongoDB and Multithreading

NTIRE比赛中训练一个模型大概需要3天, 尝试了多种加速策略均对速度没明显提升: 1) Pytorch dataloader 的多进程; 2) NVIDIA的prefetch; 3)DataloaderX. io操作比较耗时, 大量时间用来读取图片. 可通过把数据预存到数据库中来提速. 另外, 把图片数据转化为二进制序列能进一步提速. 用到的技术有MongoDB database, Python的序列化函数Pickle以及Python Package mongo_pickle.

§ 6.1 MongoDB Basics

MongoDB数据库的结构: Database − > Collections − > Documents

6.1.1 Setup

下载ww.mongodb.com/download-center/community, 并安装.

创建目录:

1. 数据存放目录D:\data\db

2. 日志存放目录D:\data\log

3. 配置文件D:\data\mongod.cfg; 内容为:

systemLog:

destination: file

path: d:\data\log\mongod.log

storage:

dbPath: d:\data\db

然后在mongodb安装目录下的bin文件夹下打开cmd, 执行mongodb.exe –config ”D:\data\mongod.cfg” –install

注意: 如果要使用MongoDB服务, 需要打开cmd界面, 启动mongod并始终在后台运行.

§ 6.2 MongoDB的基本命令

1. 创建数据库DB1: use DB1, 若DB1已经存在, 则连接到该数据库

2. 查看当前数据库有哪些: show dbs, 执行该命令后发现并没有看到刚刚创建的DB1, 原因是该数据库为空(没有任何的documents)

3. 向数据库中插入一些简单的数据: db.mycollection.insert('id':0, 'data':0), 此时输入show dbs, 就可以看到该数据库了

4. 删除数据库: db.dropDatabase(). 现在就把这个数据库删除了, 因为当前在该数据库中. 此时输入show dbs, 可看到该数据库被删除了

5. 创建Collection, db.createCollection("collection1")

6. 删除Collection, db.collection1.drop()

7. 开启MongoDB服务, 需要在bin文件夹下执行, net start MongoDB

8. 关闭MongoDB服务, net stop MongoDB

§ 6.3 创建数据库

下面创建2个collection, 并依次删除

```
use DB1
db.createCollection("myCollection")
show collections
db.myCollection2.insert({"name":"Max"}) # 如果不存在则创建myCollection2
show collections

db.myCollection.drop() # 删除这个collection
show collections
db.myCollection2.drop() # 删除这个collection
```

§ 6.4 插入数据

```
use school
db.students.insert(
    [
        {
            "studentNo": "1",
            "FirstName": "Mark",
```

```
            "LastName": "Waugh",
            "Age", "10"
        },
        {
            "studentNo": "2",
            "FirstName": "Shuailin",
            "LastName": "Lv",
            "Age", "28"
        }
)
db.students.find()
db.students.find().pretty()
db.students.find(
    {
        "Age" : {$lt: "15"} # 小于15
    }
)
db.students.find(
    {
        "FirstName" : "Mark", "Age" : "10"
    }
)
```

§ 6.5 MongoDB with Python

6.5.1 Creating a DB

```
# create a DB, collectoin and add data

from pymongo import MongoClient
democlient = MongoClient()
myclient = MongoClient('localhost', 27017)

# initial state: the database is empty
print(myclient.list_database_names())
# output is: ['admin', 'config', 'local']
```

```
# create a database "demo", and a collection "dbtable"
# the databse is still empty, because no data is inserted
mydb = myclient['demo']
mycoll = mydb['dbtable']
print(myclient.list_database_names())
# output still is: ['admin', 'config', 'local']

# the data to be inserted
mylist = [
    {'id': 0, "data":[0,1,2,3]},
    {'id': 1, "data":[4,5,6,7]},
    {'id': 2, "data":[8,9,10,11]}
]

# insert the data to the collection as well as the database
x = mycoll.insert_many(mylist)
print(myclient.list_database_names())
# now the output is ['admin', 'config', 'local', 'demo']
```

6.5.2 find method

从创建好的demo数据库中取出元素.

```
from pymongo import MongoClient
democlient = MongoClient()
myclient = MongoClient('localhost', 27017)
mydb = myclient['demo']
mycoll = mydb['dbtable']

# find one
# x = mycoll.find_one()

# find all
for x in mycoll.find():
    print(x)
```

6.5.3 fetch one data from the db

```
from pymongo import MongoClient
democlient = MongoClient()
myclient = MongoClient('localhost', 27017)
mydb = myclient['demo']
mycoll = mydb['dbtable']

"""
outputs of this script are
{'_id': ObjectId('5e6d120c8bf5e1762ccf110b'), 'id': 0, 'data': [0, 1, 2, 3]}
{'_id': ObjectId('5e6d120c8bf5e1762ccf110c'), 'id': 1, 'data': [4, 5, 6, 7]}
{'_id': ObjectId('5e6d120c8bf5e1762ccf110d'), 'id': 2, 'data': [8, 9, 10, 11]}
"""

myquery = {"id":0}

# find all
mydoc = mycoll.find(myquery)
for x in mydoc:
    print(x)
```

§ 6.6 pickle

创建一个字典并进行序列化保存, 这样可加速数据读取速度

```
import pickle
example_dict = {1:"6", 2:"2", 3:"3"}
pickle_out = open("dict.pickle", "wb")
pickle.dum(example_out, pickle_out) # save the dict to disk
pickle_out.close()

# read the dict back to memory

pickle_in = open("dict.pickle", "rb")
example_dict = pickle.load(pickle_in) # read the pickle file back to memory

print(example_dict)
print(example_dict[2]) # output is 2
```

§ 6.7 Save and Fetch Images With MongoDB

实例. 创建数据库db1, 并设置用户名”lvshuailin”, 密码”root”, 然后创建collection为”collection1”. 在cmd界面打开mongo, 执行命令:

```
use db1

db.createUser(
    {
    user:"lvshuailin",
    pwd:"root",
    roles:["dbOwner"]
    }
    )

db.createCollection("collecton1")
```

然后把遍历到的img写到db1中.

```
import glob
from mongo_pickle.collection import CollectionConfig
from mongo_pickle.model import Model
from bson.binary import Binary
import pickle
import cv2
import numpy as np


class EventsConfig(CollectionConfig):
    host = 'localhost'
    port = 27017

    database = 'db1'
    collection = "collection1"
    username = 'lvshuailin'
    password = 'root'


class Event(Model):
    # Shortcut to get pymongo collection
    COLLECTION = EventsConfig.get_collection()
```

```
23
24      def __init__(self, id, img_name, data):
25          super(Event, self).__init__()
26          self.id = id
27          self.name = img_name
28          self.data = data
29
30
31  if __name__=='__main__':
32      imgs = glob.glob(r".\imgs\\*.*")
33      print('imgs:', imgs)
34
35      count = 0
36
37      for img in imgs:
38
39          img_np = cv2.imread(img)
40          img_name = img.split("\\")[2]
41
42          img_binary = pickle.dumps((img_np))
43
44          python_event = Event(count, img_name, img_binary)
45          print_message = img_name + " has been saved to database " + EventsConfig.database
46          print(print_message)
47
48          python_event.save()
49
50          count += 1
51      print_message = "total " + str(count) + " images have been saved to database " +
            EventsConfig.database
52      print(print_message)
```

下面从数据库中取出数据

```
1  import glob
2  from mongo_pickle.collection import CollectionConfig
3  from mongo_pickle.model import Model
4  from bson.binary import Binary
5  import pickle
6  import cv2
7  import numpy as np
```

```
8
9
10 class EventsConfig(CollectionConfig):
11     host = 'localhost'
12     port = 27017
13
14     database = 'db1'
15     collection = "collection1"
16     username = 'lvshuailin'
17     password = 'root'
18
19
20 class Event(Model):
21     # Shortcut to get pymongo collection
22     COLLECTION = EventsConfig.get_collection()
23
24     def __init__(self, id, img_name, data):
25         super(Event, self).__init__()
26         self.id = id
27         self.name = img_name
28         self.data = data
29
30 if __name__== '__main__':
31
32     count = 0
33
34     for i in range(0, 10):
35         recovery_img_binary = Event.load_objects({'id': count})[0].data
36         recovery_img_name = Event.load_objects({'id': count})[0].name
37         recovery_img_np = pickle.loads(recovery_img_binary)
38         cv2.imwrite(recovery_img_name, recovery_img_np)
39         print_message = recovery_img_name + " has been fetched from the database " +
               EventsConfig.database
40         print(print_message)
41         count += 1
42
43     print_message = "total " + str(count) + " images have been fetched from the database "
           + EventsConfig.database
44     print(print_message)
```

§ 6.8 Multithreading Python

定义一个函数, 并放到Thread函数中.

```
import threading
import time

# thread will execute the funciton
def sleeper(n, name):
    print("Hi, I am {}. Going to sleep for 5 seconds\n".format(name))
    time.sleep(n)
    print("{} has woken up from sleep \n".format(name))

if __name__=="__main__":
    # initialize a thread
    # target: the function we are going to execute
    # name: the name of the thread
    t = threading.Thread(target=sleeper, name="thread1", args=(5, "thread1"))
    t.start()

    # the main program will wait here
    # after the thread1 is finished, the main program will continue
    t.join()
    print("hello")
```

创建5个线程, 并发执行

```
import threading
import time

# thread will execute the funciton
def sleeper(n, name):
    print("Hi, I am {}. Going to sleep for 5 seconds\n".format(name))
    time.sleep(n)
    print("{} has woken up from sleep \n".format(name))

if __name__=="__main__":

    thread_list = []

    start = time.time()
```

```
    # create 5 threads
    for i in range(5):
        t = threading.Thread(target=sleeper, name="thread{}".format(i), args=(5, 'thread{}'.format(i)))
        thread_list.append(t)
        t.start()
        print('{} has started'.format(t.name))

    for t in thread_list:
        t.join()
    end = time.time()
    print('time take: {}'.format(end - start))
    print('All five threads have finished their jobs')
```

6.8.1 Daemon Threads

主程序(main thread)执行结束, 然而主程序中创建的thread还没结束. creator1和creator2执行完后, limitor没必要继续执行

```
import threading
import time

total = 4

def creates_items():
    global total
    for i in range(10):
        time.sleep(2)
        print('add item')
        total += 1
    print('creation is done')

def creates_items_2():
    global total
    for i in range(7):
        time.sleep(1)
        print('added item')
        total += 1
    print('creation is done')

```

```
def limits_items():

    global total
    while True:
        if total > 5:
            print('overload')
            total -= 3
            print('substracted 3')
        else:
            time.sleep(1)
            print('waiting')


if __name__=="__main__":
    creator1 = threading.Thread(target=creates_items)
    creator2 = threading.Thread(target=creates_items_2)
    limitor = threading.Thread(target=limits_items)

    creator1.start()
    creator2.start()
    limitor.start()

    creator1.join()
    creator2.join()
    # limitor.join()

    print('our ending value of total is ', total)
```

加deamon=True, 这样主线程结束后,

```
import threading
import time

total = 4

def creates_items():
    global total
    for i in range(10):
        time.sleep(2)
        print('add item')
        total += 1
```

```
12      print('creation is done')
13
14  def creates_items_2():
15      global total
16      for i in range(7):
17          time.sleep(1)
18          print('added item')
19          total += 1
20      print('creation is done')
21
22  def limits_items():
23
24      global total
25      while True:
26          if total > 5:
27              print('overload')
28              total -= 3
29              print('substracted 3')
30          else:
31              time.sleep(1)
32              print('waiting')
33
34
35  if __name__=="__main__":
36      creator1 = threading.Thread(target=creates_items)
37      creator2 = threading.Thread(target=creates_items_2)
38      limitor = threading.Thread(target=limits_items, daemon=True)
39
40      print(limitor.isDaemon())
41      creator1.start()
42      creator2.start()
43      limitor.start()
44
45      creator1.join()
46      creator2.join()
47      # limitor.join()
48
49      print('our ending value of total is ', total)
```

6.8.2 Locks

先看一下没有锁的情况

```
import threading

x = 0
count = 100000

def adding_2():
    global x

    for i in range(count):
        x += 2

def adding_3():
    global x
    for i in range(count):
        x += 3

def subtracting_4():
    global x
    for i in range(count):
        x -= 4

def subtracting_1():
    global x
    for i in range(count):
        x -= 1

t = threading.Thread(target=adding_2)
t2 = threading.Thread(target=subtracting_4)
t3 = threading.Thread(target=adding_3)
t4 = threading.Thread(target=subtracting_1)

t.start()
t2.start()
t3.start()
t4.start()

```

```
37
38 t.join()
39 t2.join()
40 t3.join()
41 t4.join()
42
43
44 print(x)
```

程序有时候输出0, 有时候输出其他异常值.

现在给程序加锁.

```
1  import threading
2
3  x = 0
4  count = 100000
5  lock = threading.Lock()
6
7  def adding_2():
8      global x
9      with lock:
10         for i in range(count):
11             x += 2
12
13 def adding_3():
14     global x
15     with lock:
16         for i in range(count):
17             x += 3
18
19 def subtracting_4():
20     global x
21     with lock:
22         for i in range(count):
23             x -= 4
24
25 def subtracting_1():
26     global x
27     with lock:
28         for i in range(count):
```

```
                x -= 1

t = threading.Thread(target=adding_2)
t2 = threading.Thread(target=subtracting_4)
t3 = threading.Thread(target=adding_3)
t4 = threading.Thread(target=subtracting_1)

t.start()
t2.start()
t3.start()
t4.start()


t.join()
t2.join()
t3.join()
t4.join()


print(x)
```

现在程序的输出始终是0.

或者写为

```
import threading

x = 0
count = 100000
lock = threading.Lock()

def adding_2():
    global x
    lock.acquire()
    for i in range(count):
        x += 2
    lock.release()

def adding_3():
    global x
    lock.acquire()
```

```
    for i in range(count):
        x += 3
    lock.release()

def subtracting_4():
    global x
    lock.acquire()
    for i in range(count):
        x -= 4
    lock.release()

def subtracting_1():
    global x
    lock.acquire()
    for i in range(count):
        x -= 1
    lock.release()
t = threading.Thread(target=adding_2)
t2 = threading.Thread(target=subtracting_4)
t3 = threading.Thread(target=adding_3)
t4 = threading.Thread(target=subtracting_1)

t.start()
t2.start()
t3.start()
t4.start()


t.join()
t2.join()
t3.join()
t4.join()


print(x)
```

§ 6.9 Queue

三种queue, FIFO, LIFO, Priority

```
import queue

q = queue.Queue() # FIFO

# put items in the queue

q.put(5)

print(q.get())
print(q.empty())
```

```
import queue

q = queue.Queue() # FIFO

for i in range(5):
    q.put(i)

while not q.empty():
    print(q.get(), end=' ') # output is 0 1 2 3 4
```

```
import queue
import threading
import time

def putting_thread(q):
    while True:
        print('starting thread')
        time.sleep(10)
        q.put(5)
        print('put something')

q = queue.Queue()

# when the main thread terminate, the daemon thread will automatically terminate.
t = threading.Thread(target=putting_thread, args=(q,), daemon=True)
```

```
16
17 t.start()
18
19
20 q.put(5)
21 x = q.get()
22 print(x)
23 print('first item gotten')
24
25 print(q.get())
26 print('finished')
```

§ 6.10 Python Multithreading with MongoDB

```
1  import threading
2  import pdb
3  from mongo_pickle.collection import CollectionConfig
4  from mongo_pickle.model import Model
5  import pickle
6  import numpy as np
7  import queue
8  import time
9
10
11 lock = threading.Lock()
12
13 class EventsConfig(CollectionConfig):
14     host = 'localhost'
15     port = 27017
16
17     database = 'db1'
18     collection = "collection1"
19     username = 'lvshuailin'
20     password = 'root'
21
22
23 class Event(Model):
24     # Shortcut to get pymongo collection
```

```
    COLLECTION = EventsConfig.get_collection()

    def __init__(self, id, img_name, data):
        super(Event, self).__init__()
        self.id = id
        self.name = img_name
        self.data = data


def fetch_data_from_db():

    epoch = 20
    count = 0
    for j in range(0, epoch): # number of epoches
        for i in range(0, 10): # number of images
            recovery_img_binary = Event.load_objects({'id': i})[0].data
            # recovery_img_name = Event.load_objects({'id': count})[0].name
            recovery_img_np = pickle.loads(recovery_img_binary)
            lock.acquire()
            que.put(recovery_img_np)
            lock.release()
            count += 1
            if que.qsize() > 1000:
                print("push data to the queue too quickly")# 如果打印的特频繁，把时间变大sleep
                time.sleep(0.01)
    print("total {} put times".format(count))

def push_data_to_queue():
    fetch_data_from_db()
    global b_thread_run
    b_thread_run = False


def pop_data_from_queue():
    if not que.empty():
        lock.acquire()
        data = que.get()
        lock.release()
```

```
    else:
        return False
    return data



if __name__=='__main__':
    que = queue.Queue()

    t_push = threading.Thread(target=push_data_to_queue, daemon=True)
    t_push.start()

    b_thread_run = True

    count = 0
    print("process begin")
    time_begin = time.time()
    while (b_thread_run is True) or (que.qsize() > 0):
        data = pop_data_from_queue()
        if data is not False:
            count += 1
            # print("img shape is {}", data.shape)
        else:
            time.sleep(0.01)
    print("total {} imges".format(count))
    time_end = time.time()
    print('total time is {}'.format(time_end - time_begin))
    # per image about 115 ms
```

pytorch 多进程读取数据

```
def fetch_data_from_db():

    epoch = 20
    for j in range(0, epoch): # number of epoches
        for i in range(0, 10): # number of images
            recovery_img_binary = Event.load_objects({'id': i})[0].data
            recovery_img_np = pickle.loads(recovery_img_binary)
            que.put(recovery_img_np)

def push_data_to_queue():
```

```
    fetch_data_from_db()


def pop_data_from_queue():
    data = que.get()
    return data
```

然后在get_item处写入:

```
while True:
    data = pop_data_from_queue()
    return {'A':data[0], 'B':data[1]}
```

另外还需要创建队列以及启动多线程.

```
# create a que
import multiprocessing as mp
queue = mp.Queue(maxsize=1000)

# create a thread and start it
t_push = threading.Thread(target=push_data_to_queue, daemon=True)
t_push.start()
```