

Hsingmin.lee@yahoo.com

机器学习速查

Machine Learning Lite

黎兴民

2018/3/5 Monday

目录

1. C/C++.....	5
1.1 const 关键字.....	5
1.2 函数传值与传引用.....	10
1.3 字符串转整型问题.....	11
1.4 操作符重载.....	11
2. Python.....	12
2.1 可变类型与非可变类型.....	12
2.2 静态方法，实例方法与类方法.....	12
2.3 类变量与实例变量.....	13
2.4 Python 自省特性.....	15
2.5 列表推导式与字典推导式.....	15
2.6 深拷贝与浅拷贝.....	16
2.7 类方法__new__()与__init__().....	18
2.8 字符串格式化:%与.format.....	18
2.9 Python 的*args 和**kwargs.....	19
2.10 常见的几种设计模式.....	20
2.11 Python 变量的作用域.....	21
2.12 GIL 线程全局锁及协程.....	22
2.13 闭包.....	22
2.14 Lambda 与函数式表达式.....	23
2.15 编码与解码(encode/decode).....	23
2.16 迭代器与生成器.....	24
2.17 装饰器.....	25
2.18 Python 中的重载.....	26
2.19 Python 新式类与旧式类.....	27
2.20 邮箱地址正则表达式.....	27
2.21 Python 内置类型 list/dictionary/tuple/string.....	32
2.22 Python 中的 is.....	33
2.23 read/readline 和 readlines.....	34
2.24 垃圾回收.....	34
3. Algorithm.....	34
3.1 时间复杂度计算.....	34
3.2 二叉树.....	35
3.2.1 二叉树的数据结构.....	35
3.2.2 二叉树的生成.....	36
3.2.3 二叉树的遍历算法.....	36
3.3 最大堆.....	41
3.4 红黑树.....	41
3.5 B 树.....	41
3.6 线性结构-栈与队列.....	41
3.7 线性结构-链表.....	41
3.8 寻找链表倒数第 k 个节点.....	41

3.9	快速排序.....	42
3.10	堆排序.....	42
3.11	无序数字列表寻找所有间隔为 d 的组合.....	42
3.12	列表[a1, a2, a3, ..., an]求其所有组合.....	42
3.13	一行 python 代码实现 1+2+3+...+10**8.....	43
3.14	长度未知的单向链表求其是否有环.....	43
3.15	单向链表应用快速排序.....	43
3.16	长度为 n 的无序数字元素列表求其中位数.....	43
3.17	遍历一个内部未知的文件夹.....	43
3.18	台阶问题/斐波那契.....	43
3.19	变态台阶问题.....	43
3.20	矩形覆盖问题.....	44
4.	OS.....	45
4.1	多线程与多进程的区别.....	45
4.2	协程.....	45
4.3	进程间通信方式.....	45
4.4	虚拟存储系统中缺页计算.....	45
4.5	并发进程不会引起死锁的资源数量计算.....	45
4.6	常用的 Linux/git 命令和作用.....	45
4.7	查看当前进程的命令.....	45
4.8	45
5.	网络.....	46
5.1	TCP/IP 协议.....	46
5.2	OSI 五层协议.....	46
5.3	Socket 长连接.....	46
5.4	Select 与 epoll.....	46
5.5	TCP 与 UDP 协议的区别.....	46
5.6	TIME_WAIT 过多的原因.....	46
5.7	http 一次连接的全过程描述.....	46
5.8	http 连接方式——get 与 post 的区别.....	46
5.9	restful.....	47
5.10	http 请求的状态码 200/403/404/504.....	47
6.	数据库.....	48
6.1	MySQL 锁的种类.....	48
6.2	死锁的产生.....	48
6.3	MySQL 的 char/varchar/text 的区别.....	48
6.4	Join 的种类与区别.....	48
6.5	A LEFT JOIN B 的查询结果中, B 缺少的部分如何显示.....	48
6.6	索引类型的种类.....	48
6.7	BTree 索引与 hash 索引的区别.....	48
6.8	如何对查询命令进行优化.....	48
6.9	NoSQL 与关系型数据库的区别.....	48
6.10	Redis 常用的存储类型.....	49
7.	机器学习.....	50

7.1	模型评估方法.....	50
7.1.1	数据集划分.....	50
7.1.2	模型的性能度量.....	50
7.1.3	偏差-误差分解.....	50
7.2	LR 与极大似然估计.....	51
7.3	LDA.....	51
7.4	类别不平衡问题.....	52
7.5	判别/生成模型与先验/后验概率.....	52
7.6	朴素贝叶斯分类器.....	53
7.7	常用的损失函数.....	53
7.8	决策树.....	54
7.9	连续值与缺失值处理.....	55
7.10	BP 算法.....	55
7.11	梯度下降法.....	57
7.12	IS 与牛顿法/拟牛顿法.....	57
7.13	局部最小与全局最小.....	58
7.14	支持向量机与拉格朗日乘子法.....	59
7.15	正则化.....	62
7.15.1	LASSO 与 Ridge.....	62
7.15.2	L1 正则化求解.....	63
7.16	EM 算法.....	64
7.17	Boosting 与 Bagging.....	64
7.17.1	Boosting 方法.....	64
7.17.2	Bagging 方法.....	68
7.18	监督学习与无监督学习.....	68
7.19	聚类.....	69
7.20	特征选择与稀疏学习.....	70
7.20.1	维度灾难.....	70
7.20.2	特征选择.....	70
7.21	PCA.....	71
7.21.1	特征值分解.....	71
7.21.2	奇异值分解.....	72
7.21.3	主成分分析.....	73
7.22	Apriori 算法与 FP-Growth.....	74
7.22.1	关联规则.....	74
7.22.2	Apriori 算法.....	75
7.23	词频-逆文本词频.....	75
7.24	概率图模型.....	75
7.24.1	隐马尔科夫模型.....	76
7.24.2	马尔科夫随机场.....	78
7.24.3	条件随机场.....	79
7.24.4	隐狄利克雷分配模型.....	83
7.25	从伯努利到狄利克雷分布.....	85
7.25.1	伯努利分布.....	85

7.25.2	二项分布.....	85
7.25.3	<i>Beta</i> 分布	86
7.25.4	多项式分布.....	86
7.25.5	<i>Dirichlet</i> 分布	87

1.C/C++

1.1 const 关键字

const 关键字常用于数组边界和 switch 条件分支标号:

const type variable;	//常变量等价于 type const variable
const type &reference;	//常引用等价于 type const& variable
ClassName const Object;	//常对象等价于 const ClassName Object
ClassName::func(signature) const;	//常成员函数
Type const ArrayName[size];	//常数组等价于 const type ArrayName[size]
const type* pointer;	//常指针定义方法 1
type const* pointer;	//常指针定义方法 2

C 标准中, const 变量定义为全局作用域, 而 C++中则要视其定义位置而定;
使用指针时涉及到两个关键要素: 指针本身(地址)和指针所指向的对象(数据类型);

将一个指针声明为 const 类型, 是限制其指向的对象为指定类型;

将指针本身声明为 type *const pointer, 则是将指针(地址)本身声明为常量;

可以将一个非 const 对象的地址赋给 const 对象的指针, 反之不可;

void Func(const A *arg); //传指针传递参数

void Func(const A &arg); //传引用传递参数

将函数传入参数声明为 const, 提高函数运行效率且禁止修改其内容;

在 C++中使用 const 关键字修饰成员函数, 则 const 对象只能访问 const 成员函数, 而非 const 对象可以访问所有成员函数; const 对象的成员不可修改, const 成员函数不可修改对象的数据, 无论对象是否具有 const 限定;

常量指针与指针常量:

const int *p1 = new int(10); //指向常量的指针, 不可通过 p1 修改其指向的内容

int *const p2 = new int(10); //指向 int 类型的指针常量, 不可指向其他内存

程序载入内存时, 会分配常量区存储常量, 可以通过直接修改变量数值或通过另外一个非常量指针进行修改:

int a = 10;

const int *p = &a;

// error

// *p = 100;

//直接修改变量

a = 100;

```
//
int *pi = (int*) p;           //第三方指针
*pi = 100;
```

函数参数中的指针常量表示不允许此指针指向其他内容：

```
void func(int *const pt){
    int *p = new int(10);
    pt = p;                       //error cannot point to a new pointer
}
```

函数参数中的常量指针表示此参数不允许被修改：

```
void func(const int *pt){
    *pt = 100;                   //error cannot be modified
}
```

若将参数中的指针赋给一个新的指针，则会修改其指向的内容：

常量与引用的关系：引用就是变量的别名，常量引用即不允许此引用成为其他变量的别名；

```
int a = 10;
const int& ra = a;              //常量引用，不可通过此引用改变其对应的内容
// int& const ra = a;          //error
若不希望函数调用者修改参数本身的值，最可靠的方法是传递引用；
void func(const int& arg){
    // arg = 100;               //error cannot be modified
}
```

系统在加载程序时，会将内存分为 4 个区域：堆、栈、数据段和代码段，使用常量的方式保护数据是通过编译器语法规则限定来实现的，但是仍然可以通过以下方式完成修改：

```
const int a = 10;
int *pa = (int*) &a;           //可以通过定义其他指针的方法修改常量数值
*pa = 100;
```

常量函数在 C++ 中可防止类的 数据成员被非法访问，将类的成员函数分为两类：常量成员函数和非常量成员函数；

```
class Test{
public:
    void func() const;
private:
    int intValue;
};

void Test::func() const{
    intValue = 100;              // raise error 常量函数尝试改变数
```

据成员 intValue 的数值

```
//编译时引发异常
}
```

```
class Fred{
    public:
    void inspect() const;
    void mutate();
};

void UserCode(Fred& changeable, const Fred& unChangeable){
    changeable.mutate();           //非常量对象调用非常量函数
    changeable.inspect();          //非常量对象调用常量函数
    unChangeable.mutate();         //常量对象调用非常量函数，错误
    unChangeable.inspect();        //常量对象只能调用常量函数
}
```

常量函数包含一个 this 的常量指针：

```
void inspect(const Fred* this) const;
void mutate(Fred* this);
对于常量函数，不能通过 this 指针修改对象对应的内存，但可以通过重新定义指针来修改内存中的内容；
void func(const int *pt){
    int *ptr = (int*) pt;
    *ptr = 100;
}
```

通过常量对象调用非常量函数将产生语法错误；对于常量函数，不能通过 this 指针修改对象对应的内存块；但是可以通过 this 指针重新定义一个指向同一内存单元的指针；

```
void Fred::inspect() const{
    Fred *pFred = (Fred*) this;
    pFred->intValue = 50;
}
```

对于常量对象，可以构造新指针，指向常量对象所在的内存单元；

C++允许在类的数据成员定义前加上 mutable 关键字以实现成员在常量函数中可修改：

```
class Fred{
    public:
    void inspect() const;
    private:
    mutable int intValue;
};
```


常量函数的重载问题:

```
class Fred{
    public:
    void func() const;
    void func();
}
void Fred::func() const{}
void Fred::func(){}

void UserCode(Fred& fred, const Fred& cFred){
    fred.func();           //call func()
    cFred.func();          //call func() const
}

int main(int argc, char** argv){
    Fred fred;
    UserCode(fred, fred);
    return 0;
}
```

当存在 同名同参数相同返回值的函数重载时，具体调用哪个函数取决于调用对象是常量对象还是非常量对象；

常量返回值：不希望函数调用者修改函数的返回值，将函数返回一个常量。

- **Const 常量与宏定义的区别：**const 常量有数据类型(编译器做类型检查)，宏定义没有数据类型(编译器仅作简单的字符替换)；
- **Const 修饰类的数据成员，**const 数据成员只在对象的生命周期内是常量，而对整个类而言是可变的，因此，const 数据成员的初始化只能在类的构造函数的初始化列表中进行，不能在类中指定；

```
class A{
    const int size = 100;           //错误定义
    int array[size];               //错误定义，未知的 size
}
```

要在类中定义常量，应当使用 enum 类型：

```
class A{
    enum{size1=100, size2=200};
    int array1[size1];
    int array2[size2];
}
```

枚举常量不占用对象的内存空间，在编译时全部求值，枚举常量隐含使用整型类型，最大数值有限，不能表示浮点型数据；

- **const 初始化：**

```
T b;
const T a = b;           //非指针 const 常量初始化
```

```

T* p = new T();
const T* pc = p;           //指针常量初始化
const T* pb = new T();     //同上

T r;
const T& rf = r;           //rf 只能访问 const 成员函数

```

```

const T* c = new T();
T* e = c;                 //声明指针的目的是要修改其指向的内容，但此处指向常量

```

```

T* const c = new T();
T* e = c;                 //声明指针指向的内容可变

```

- 参数 `const` 通常用于参数为指针或引用的情况下使用，若参数为值传递，则函数会自动产生临时变量复制，保护被传递对象的属性；
- 对于非内部数据类型的参数传递，将参数值传递改为 `const` 引用传递可以提高效率；而对于内部数据类型的参数传递，避免改为 `const` 引用传递，以免降低程序的可理解性；
- 修饰返回值的 `const` 关键字，`const T func(); const T* func();` 对函数返回值进行保护；
- 函数的返回值声明为 `const`，通常用于操作符重载，若使用 `const` 修饰函数的返回值类型，则返回的 `const Object` 只能访问类中的公有数据成员和 `const` 成员函数，且无法对其进行赋值操作；指针传递函数返回值加 `const` 修饰符，则函数返回内容(指针)不能被修改，只能赋给同样加 `const` 修饰符的同类型指针；

```

const char* GetString(void);
char *str = GetString();    //compile error
const char* str = GetString(); //compile success

```

- 函数返回值引用传递通常用于类的赋值函数中，实现链式表达：

```

class A{
    A &operator = (const A& other);    //赋值操作符
}
A a, b, c;                          //class A Object a, b, c
...
a = b = c;                          //all right
(a = b) = c;                        //legal but unusual

```

`a.operator = (b)` 的返回值是 `const` 类型的引用，不可再次赋值给 `c`；
返回值的内容不允许被修改；

- 类成员函数 `const` 关键字的使用：

```

class Stack{
public:
    void Push(int elem);

```

```

        int Pop(void);
        int GetCount(void) const;           //const 成员函数
    private:
        int m_num;
        int m_data[100];
};
//公有成员函数定义在类外
int Stack::GetCount(void) const{
    ++m_num;           //编译错误，企图修改数据成员
    Pop();             //编译错误，企图调用非 const 成员函数
    return m_num;
}

```

- 在 C 中，const 是一个不能被改变的普通变量，需要占用存储空间，编译器不知道编译时的数值，且数组下标必须为常量；而在 C++中，将 const 看做编译时的常量，不为其分配存储空间，只是在编译时将其数值存储在名字列表中；
- 在 C 语言中，const int size; 语句正确，在 C++中不正确；C 编译器默认使用外部连接，将其看做声明，可在其他地方分配内存空间；C++编译器默认使用内部链接，必须在声明时初始化，将 const 对象默认看做文件的局部变量；
- 在 C++中，是否为 const 分配内存空间取决于是否添加 extern 关键字或者取 const 变量地址；

1.2 函数传值与传引用

函数传值：压栈的是参数的副本，函数对传递参数的操作作用在参数副本之上，不会修改原始值本身；

函数传指针(地址)：压栈的是地址的副本，但指针指向的是同一内存单元，修改操作会作用在原内存之上；

函数传引用：压栈的是引用的副本，但引用指向同一个变量地址，即同变量的两个别名，对引用的操作即对其指向的变量的操作；

值传递与引用传递的区别：

值传递(passed-by-value)：被调函数的形式参数作为被调函数的局部变量处理，为其开辟内存空间，存放传递进来的主调函数的实参数值，成为实参的一个副本，对被调函数局部变量的操作即对实参副本的操作，因此不会改变主调函数中实参的原始数值；

引用传递(passed-by-reference)：被调函数的形式参数也被当做被调函数的局部变量处理，并在堆栈中开辟内存空间，但其存储的是主调函数实参的地址，因此，被调函数实际上是对主调函数实参的间接引用，操作作用在实参原始数值之上，会改变实参的数值；

1.3 字符串转整型问题

将一个字符串转换成整数，如"0295"转换成 0295 (Microsoft)，容易写出以下代码：

```
int StrToInt(char* string){
    int number = 0;
    while(*string != 0){
        number = number * 10 + *string - '0';
        ++string;
    }
    return number;
}
```

需要考虑以下问题：

- 程序鲁棒性：函数传入空指针判断；
- 输入字符串正负号判断；
- 最大正整数，最小负整数及溢出问题；
- 非法字符输入；

1.4 操作符重载

2. Python

2.1 可变类型与非可变类型

类型属于对象而非变量即对象才有类型，而变量没有类型；对象分为可变对象 (mutable) 和非可变对象 (immutable) 两种，常见的可变对象有 list/dict/set，常见的不可变对象有 strings/tuples/numbers；

当不可变对象的引用传递给函数时，函数自动生成一份引用的拷贝，函数内部的操作均在拷贝上进行，不会改变外部对象；

当可变对象的引用传递给函数时，函数内的引用指向可变对象，如同指针一样，对其操作即对定位的指针地址一样，在内存中完成对对象的修改；

```
a = 1
def fun(a):
    a = 2
fun(a)
print a    # 1
a = []
def fun(a):
    a.append(1)
fun(a)
print a    # [1]
```

2.2 静态方法，实例方法与类方法

Python 中有三种方法：静态方法 @staticmethod 类方法 @classmethod 和实例方法：

```
def foo(x):
    print "executing foo(%s)"%(x)

class A(object):
    def foo(self,x):
        print "executing foo(%s,%s)"%(self,x)

    @classmethod
    def class_foo(cls,x):
        print "executing class_foo(%s,%s)"%(cls,x)

    @staticmethod
```

```

def static_foo(x):
    print "executing static_foo(%s)"%x

a=A()

```

函数参数 **self**: 对实例(object)的绑定, 在类中每次定义方法都要绑定实例 **self**, **foo(self, x)**调用可修改实例自身;
 函数参数 **cls**: 对类(class)的绑定, 类方法 **A.class_foo(x)**传递的是类 **class** 而非实例;
 静态方法与普通方法一样, 不需要绑定类或实例, 仅需要通过实例调用 **a.static_foo(x)**或通过类调用 **A.static_foo(x)**;

2.3 类变量与实例变量

类变量可在类间共享, 不会单独分配给每个实例;
 实例变量是指实例化后, 实例单独拥有的变量;

```

class Test(object):
    num_of_instance = 0
    def __init__(self, name):
        self.name = name
        Test.num_of_instance += 1

if __name__ == '__main__':
    print Test.num_of_instance    # 0
    t1 = Test('jack')
    print Test.num_of_instance    # 1
    t2 = Test('lucy')
    print t1.name , t1.num_of_instance    # jack 2
    print t2.name , t2.num_of_instance    # lucy 2

```

```

class Person:
    name="aaa"

p1=Person()
p2=Person()
p1.name="bbb"
print p1.name    # bbb
print p2.name    # aaa
print Person.name    # aaa

```

参数传递问题, **p1.name** 最初指向的是类变量 **name = "aaa"**, **string** 为不可变对象, 实例单独产生拷贝, 不会改变类变量本身, 而 **list []**为可变对象, 每个实例均会产生对类变量的引用, 实例方法对类变量的操作通过间接引用作用在类变量上, 在实例的作用域内将类变量的引用改变为实例变量;

```

class Person:
    name=[]

p1=Person()
p2=Person()
p1.name.append(1)
print p1.name    # [1]
print p2.name    # [1]
print Person.name # [1]

# -*- coding: utf-8 -*-
# -*- versison: python 3.4.5 -*-

# class_method.py

class Test(object):
    num_of_instance = 0
    name_of_instance = "Object1"
    queue_of_instance = []

    def __init__(self, name):
        self.name = name
        Test.num_of_instance += 1
        Test.name_of_instance = name
        Test.queue_of_instance.append(name)

if __name__ == '__main__':
    print("Class attribute : ")
    print("Test.num_of_instance = ", Test.num_of_instance)    # 0
    print("Test.name_of_instance = ", Test.name_of_instance)  # "Object"
    print("Test.queue_of_instance = ", Test.queue_of_instance) # []

    # Create object named 'jack'
    t1 = Test('jack')
    print("After instantiation : ")
    print("Test.num_of_instance = ", Test.num_of_instance)    # 1
    print("Test.name_of_instance = ", Test.name_of_instance)  # "jack"
    print("Test.queue_of_instance = ", Test.queue_of_instance) # ['jack']

    # Object t1 shares class variable
    print("t1.num_of_instance = ", t1.num_of_instance)        # 1
    print("t1.name_of_instance = ", t1.name_of_instance)      # "jack"
    print("t1.queue_of_instance = ", t1.queue_of_instance)    # ['jack']

```

```

# Create object named 'tom'
t2 = Test('tom')
print("Test.num_of_instance = ", Test.num_of_instance)    # 2
print("Test.name_of_instance = ", Test.name_of_instance)  # "tom"
print("Test.queue_of_instance = ", Test.queue_of_instance) # ['jack', 'tom']

# Object t2 shares class variable
print("t2.num_of_instance = ", t2.num_of_instance)        # 2
print("t2.name_of_instance = ", t2.name_of_instance)      # "tom"
print("t2.queue_of_instance = ", t2.queue_of_instance)    # ['jack', 'tom']
# number, string as immutable variable
# list as mutable variable
t1.name_of_instance = "jack"
print("t1.num_of_instance = ", t1.num_of_instance)        # 2
print("t1.name_of_instance = ", t1.name_of_instance)      # "jack"
print("t1.queue_of_instance = ", t1.queue_of_instance)    # ['jack', 'tom']

```

2.4 Python 自省特性

面向对象语言可在运行时获得对象的类型，常用自省函数：
`type()/dir()/getattr()/hasattr()/isinstance()`

2.5 列表推导式与字典推导式

列表推导式与字典推导式具有高效简短的特性：

```

list = [element for element in iterable]
dict = {key: value for (key, value) in iterable}
使用内建函数 enumerate()赋予元素下标：
{i: el for i, el in enumerate(["one", "two", "three"])}

```

```

# 不使用内建函数 enumerate()
lst = ["one", "two", "three"]
i = 0
for e in lst:
    lst[i] = '%d: %s' % (i, lst[i])
    i += 1

```

将列表推导式中的[]改为(), 其数据结构发生变化：

```

L = [x*x for x in range(10)]
# L = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
g = (x*x for x in range(10))
# g = <generator object <genexpr> at 0x0000028F8B774200>

```

即由列表对象变为生成器对象：

在 Python 中，常使用边循环边计算的 generator 机制，以节省内存空间；

2.6 深拷贝与浅拷贝

Python 引用和 copy()/deepcopy()的关系:

```
import copy
```

```
a = [1, 2, 3, 4, ['a', 'b']] #原始对象
```

```
b = a #赋值, 传对象的引用
```

```
c = copy.copy(a) #对象拷贝, 浅拷贝
```

```
d = copy.deepcopy(a) #对象拷贝, 深拷贝
```

```
a.append(5) #修改对象 a
```

```
a[4].append('c') #修改对象 a 中的['a', 'b']数组对象
```

```
print 'a = ', a
```

```
print 'b = ', b
```

```
print 'c = ', c
```

```
print 'd = ', d
```

```
id(a)      # 12757640
```

```
id(b)      # 12757640
```

```
id(c)      # 12765576
```

```
id(d)      # 12757704
```

输出结果:

```
a = [1, 2, 3, 4, ['a', 'b', 'c'], 5]
```

```
b = [1, 2, 3, 4, ['a', 'b', 'c'], 5]
```

```
c = [1, 2, 3, 4, ['a', 'b', 'c']]
```

```
d = [1, 2, 3, 4, ['a', 'b']]
```

在 Python 中, 用一个变量给另一个变量赋值, 就是给内存中的对象增加一个标签; 浅拷贝与深拷贝都是针对组合对象来说的, 组合对象是指包含了其他对象的对象, 如列表、类实例等, 整型、浮点型、字符串等“原子”类型则没有拷贝一说, 都是对原有对象的引用;

- 浅拷贝是指创建一个新的对象, 其内容是对原对象中元素的引用(拷贝组合对象不拷贝原子对象), list 对象 a 浅拷贝得到 b, a 和 b 都指向内存中共同的 int 对象:

```
>>>a = [1,2,3]
```

```
>>>b = a
```

```
>>>print(id(a), id(b))
```

```
12757064 12765320
```

```
>>>for x, y in zip(x,y):
```

```
    print(id(x), id(y))
```

```
1431568848 1431568848
```

```
1431568880    1431568880
1431568912    1431568912
```

- 深拷贝是指创建一个对象，递归拷贝原对象包含的子对象，深拷贝出来的对象与原对象没有任何关系：

```
>>> import copy
>>> a = [1, 2, 3]
>>> b = copy.deepcopy(a)
>>> print(id(a), id(b))
140601785065840 140601785066200
>>> for x, y in zip(a, b):
...     print(id(x), id(y))
...
140601911441984 140601911441984
140601911442016 140601911442016
140601911442048 140601911442048
```

深拷贝得到的列表元素的 id 之所以相同是因为对于不可变对象，当需要一个新对象时，Python 可能会返回已经存在的某个类型和值都一致的对象的引用，但这种机制并不影响 a 和 b 的相互独立性，当两个元素指向同一个不可变对象时，对其中一个赋值不影响另一个；

```
>>> import copy
>>> a = [[1, 2],[5, 6], [8, 9]]
>>> b = copy.copy(a)           # 浅拷贝得到 b
>>> c = copy.deepcopy(a)       # 深拷贝得到 c
>>> print(id(a), id(b))        # a 和 b 不同
139832578518984 139832578335520
>>> for x, y in zip(a, b):      # a 和 b 的子对象相同
...     print(id(x), id(y))
...
139832578622816 139832578622816
139832578622672 139832578622672
139832578623104 139832578623104
>>> print(id(a), id(c))        # a 和 c 不同
139832578518984 139832578622456
>>> for x, y in zip(a, c):      # a 和 c 的子对象也不同
...     print(id(x), id(y))
...
139832578622816 139832578621520
139832578622672 139832578518912
139832578623104 139832578623392
```

2.7 类方法__new__()与__init__()

```
>>> class MyClass():
...     def __init__(self):
...         self.__superprivate = "Hello"
...         self._semiprivate = ", world!"
...
>>> mc = MyClass()
>>> print mc.__superprivate
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: myClass instance has no attribute '__superprivate'
>>> print mc._semiprivate
, world!
>>> print mc.__dict__
{'_MyClass__superprivate': 'Hello', '_semiprivate': ', world!'}
```

- 形如__foo__: Python 语言内部的一种命名约定,用以区别其他用户自定义命名,防止命名冲突,类似__init__(), __del__(), __call__(), __new__()这些特殊方法;
 - 形如foo: 程序员指定变量私有,不能使用 from module import *导入,其他方面使用与公有变量相同;
 - 形如foo: 解析器使用_classname__foo 替代,以区别和其他类相同的命名,无法直接像公有成员一样访问,只能通过 objectname._classname__foo()访问;
- __new__()是一个静态方法,返回值是一个创建的实例,只有在__new__()返回一个cls实例之后,__init__()才能被调用;
- __init__()是一个实例方法,无返回值,创建新实例时调用__new__(),初始化新实例时调用__init__();
- __metaclass__在创建类时起作用,因而可以调用__metaclass__(), __new__(), __init__()方法,分别在类创建、实例创建和实例初始化的代码中进行改动;

2.8 字符串格式化:%与.format

```
"hi there %s" % name          # 若 name 为(1,2,3), Raise TypeError Exception
# not all arguments converted during string formatting
格式化符号%无法同时传递一个变量和元组,应当使用以下代码完成正常输出:
"hi there %s" % (name,)
此外,还可以使用.format()格式化输出:
# 使用位置参数
>>> li = ['hoho',18]
>>> 'my name is {},age {}'.format('hoho',18)
'my name is hoho ,age 18'
>>> 'my name is {1} ,age {0}'.format(10,'hoho')
'my name is hoho ,age 10'
```

```

>>> 'my name is {1} ,age {0} {1}'.format(10,'hoho')
'my name is hoho ,age 10 hoho'
>>> 'my name is {} ,age {}'.format(*li)
'my name is hoho ,age 18'

# 使用关键字参数
>>> hash = {'name':'hoho','age':18}
>>> 'my name is {name},age is {age}'.format(name='hoho',age=19)
'my name is hoho,age is 19'
>>> 'my name is {name},age is {age}'.format(**hash)
'my name is hoho,age is 18'

# 填充格式化 :[填充字符][对齐方式<^>][宽度]
>>> '{0:*>10}'.format(10)  ##右对齐
'*****10'
>>> '{0:*<10}'.format(10)  ##左对齐
'10*****'
>>> '{0:*^10}'.format(10)  ##居中对齐
'****10****'

# 数字精度与进制
>>> '{0:.2f}'.format(1/3)
'0.33'
>>> '{0:b}'.format(10)      #二进制
'1010'
>>> '{0:o}'.format(10)      #八进制
'12'
>>> '{0:x}'.format(10)      #16 进制
'a'
>>> '{:,}'.format(12369132698)  #千分位格式化
'12,369,132,698'

# 使用索引
>>> li
['hoho', 18]
>>> 'name is {0[0]} age is {0[1]}'.format(li)
'name is hoho age is 18'

```

2.9 Python 的*args 和**kwargs

当不确定函数内将要传递多少参数时，使用*args:

```

>>> def print_everything(*args):
...     for count, thing in enumerate(args):
...         print '{0}. {1}'.format(count, thing)

```

```
...
>>> print_everything('apple', 'banana', 'cabbage')
0. apple
1. banana
2. cabbage

**kwargs 允许使用事先未定义的参数名:
>>> def table_things(**kwargs):
...     for name, value in kwargs.items():
...         print '{0} = {1}'.format(name, value)
...
>>> table_things(apple = 'fruit', cabbage = 'vegetable')
cabbage = vegetable
apple = fruit
```

2.10 常见的几种设计模式

单例模式：核心结构中只包含一个被称为单例类的特殊类，通过单例模式可以保证系统中一个类只有一个实例而且该实例易于外界访问，从而方便对实例个数进行控制并节约系统资源，若希望系统中某个类的对象只能存在一个，则单例模式是最适合的解决方案；

方法__new__()在__init__()之前调用，用于生成实例对象，单例模式是指创建唯一的对象，单例模式设计的类只能实例化一次：

- 使用__new__()方法实现：

```
class Singleton(object):
    def __new__(cls, *args, **kw):
        if not hasattr(cls, '_instance'):
            orig = super(Singleton, cls)
            cls._instance = orig.__new__(cls, *args, **kw)
        return cls._instance

class MyClass(Singleton):
    a = 1
```

- 共享属性实现：创建实例时将实例的__dict__指向同一个字典，则实例具有相同的属性和方法；

```
class Borg(object):
    _state = {}
    def __new__(cls, *args, **kw):
        ob = super(Borg, cls).__new__(cls, *args, **kw)
        ob.__dict__ = cls._state
        return ob

class MyClass1(Borg):
    a = 1
```

- 装饰器版本实现：

```
def singleton(cls, *args, **kw):
```

```

instances = {}
def getinstance():
    if cls not in instances:
        instances[cls] = cls(*args, **kw)
    return instances[cls]
return getinstance

```

```

@singleton
class MyClass:

```

```

...

```

- import 方法实现，作为 python 的模块是天然的单例模式；

```

# mysingleton.py
class My_Singleton(object):
    def foo(self):
        pass
my_singleton = My_Singleton()

# to use
from mysingleton import my_singleton
my_singleton.foo()

```

2.11 Python 变量的作用域

- 决定 Python 变量作用域最小单位的是关键字 def，类似于 Java 中的{...}，Python 中能够改变变量作用域的代码段是 def、class、lambda；
- if/elif/else、try/except/finally、for/while 并不能涉及变量作用域的更改，即其内部代码块中的变量在外部也可访问；
- 变量的搜索路径是 Local Variable → Global Variable

```

def scopetest():
    var = 6
    print(var)          # 6
    def innerfunc():
        print(var)      #6
    innerfunc()
var = 5
print(var)              # 5
scopetest()
print(var)              # 5

```

调用顺序：Local → Global，def 作为变量作用域标示符，innerfunc()中的 var 首先在其定义域内部进行搜索，没有找到则上溯到其主调函数的作用域内搜索，找到 var = 6，使用其值；

2.12 GIL 线程全局锁及协程

线程全局锁(Global Interpreter Lock)是 Python 为了保护线程安全而采取的独立线程运行的限制机制，即一个核在同一时刻只能运行一个线程，对于 IO 密集型任务，Python 的多线程机制起到作用，对于 CPU 密集型任务，python 多线程任务可能会因为争夺资源而变慢，解决办法即使用多进程和协程：

协程即用户自己控制内核态与用户态的切换，不用陷入系统的内核态，减少切换时间，Python 中的 yield 即使用协程的思想；

在进程中可同时存在一个或多个线程，每个进程拥有独立的地址空间、内存、堆栈和其他数据段；

线程具有开始、顺序执行和结束三个阶段，线程是 CPU 调动的，没有独立的资源，多有线程共享统一进程中的资源，在 Python 中，为了解决多线程访问共享资源的数据保护问题，产生了线程全局锁(GIL)；

线程锁：CPU 执行任务时，线程间的调度是随机进行的，并且每个线程可能只是执行 n 条语句就要转而执行其他线程，由于进程中多个线程间共享数据和资源很容易产生资源抢夺和脏数据，于是需要使用线程锁(GIL)限制对指定数据的访问；Python 在解释器的层面限制了程序在同一时刻只有一个线程被 CPU 实际执行，从而导致多线程编程效率过低，计算密集型任务推荐使用多进程，IO 密集型任务推荐多线程，防止同一资源被占用的情况。

2.13 闭包

如果在一个内部函数中，对外部作用域(非全局作用域)的变量进行引用，则内部函数被认为是一个闭包(closure)：

```
>>>def addx(x):
>>>    def adder(y): return x + y
>>>    return adder
>>> c = addx(8)
>>> type(c)
<type 'function'>
>>> c.__name__
'adder'
>>> c(10)
18
```

adder(y)作为内部函数，对在外部作用域(非全局作用域)的变量 x 进行引用，x 在外部作用域 addx 内，不在全局作用域内，adder(y)就是一个闭包；

闭包 = 函数块 + 定义函数时的环境，adder 是函数块，x 即环境；

闭包无法修改外部作用域的局部变量：

```
def foo():
    a = 1
    def bar():
        a = a+1          # 将 a 看做 bar()的局部变量，赋值时出错
```

```

        return a
    return bar
>>> c = foo()
>>> print c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in bar
UnboundLocalError: local variable 'a' referenced before assignment

```

在 Python 3 中，可以在增 1 语句 `a = a+1` 之前添加语句 `nonlocal a` 声明 `a` 非闭包局部变量即可；

2.14 Lambda 与函数式表达式

- filter 过滤器函数：


```

>>>a = [1,2,3,4,5,6,7]
>>>b = filter(lambda x: x>5, a)
>>>list(b)
[6,7]

```
- map 依次执行函数：


```

>>> a = map(lambda x:x*2,[1,2,3])
>>> list(a)
[2, 4, 6]

```
- reduce 迭代调用函数,python 3.0.0 及更高版本中，`reduce()` 已经被移出内建函数，需要从 `from functools import reduce` 导入：


```

>>>from functools import reduce
>>>reduce(lambda x, y: x+y, range(1, 9))
40320

```

2.15 编码与解码(encode/decode)

编码的目的是兼容字符集之间的通用性，通常用到编码的环境有：

- 系统默认编码
- 程序运行环境编码
- 源文件自身编码
- 源代码中字符串编码

通常中文操作系统中的字符编码是 `gbk` 格式，Python 运行时的字符编码默认为 `ASCII`，源文件编码格式可指定 `# -*- coding: utf-8 -*-`

程序内容的编码可以通过 Python 提供的函数进行转换(Python2 中字符串有两种类型 `unicode` 字符串和非 `unicode` 字符串，而 Python3 中只有 `unicode` 字符串)，可以使用 `str.encode('utf-8')` 将指定字符串编码为 `'utf-8'` 或其他格式，使用 `s.decode('utf-8')` 解码字符串；

指定的编码字符只能用指定的解码方式进行解码，否则无法还原原有的字符串；在 Python3 中，取消了 unicode 类型，使用 unicode 字符类型的字符串替代，编码后变成了字节类型(bytes)：bytes → Unicode string → bytes

u = u'中文' #显示指定 unicode 类型对象 u

str = u.encode('gb2312') #以 gb2312 编码对 unicode 对像进行编码

str1 = u.encode('gbk') #以 gbk 编码对 unicode 对像进行编码

str2 = u.encode('utf-8') #以 utf-8 编码对 unicode 对像进行编码

u1 = str.decode('gb2312')#以 gb2312 编码对字符串 str 进行解码，以获取 unicode

u2 = str.decode('utf-8')#如果以 utf-8 的编码对 str 进行解码得到的结果，将无法还原原来的 unicode 类型

Python 提供了包 codecs 进行文件读取，codec.open()函数可以指定编码类型：

import codecs

f = codecs.open('text.text','r+',encoding='utf-8')#必须事先知道文件的编码格式，这里文件编码是使用的 utf-8

content = f.read()#如果 open 时使用的 encoding 和文件本身的 encoding 不一致的话，那么这里将将会产生错误

f.write('你想要写入的信息')

f.close()

2.16 迭代器与生成器

迭代器可以减少内存开销：迭代器属于临时区，安排一些元素在其中，使用时才创建临时区，一旦遍历结束即清空临时区，再次遍历时临时区失效。

```
# -*- coding: utf-8          -*-
# -*- version:    python 3.5.4 -*-
# _iter.py
import sys
i = iter(range(10000))
print("id(i.__next__()) = ", id(i.__next__()))
print("sys.getsizeof(i) = ", sys.getsizeof(i))
print("sys.getsizeof(i.__next__()) = ", sys.getsizeof(i.__next__()))
# 若一次性将 list 对象全部加载进来需要 90112bytes
# 使用迭代器仅需要 28bytes
e = range(10000)
print("sys.getsizeof(e) = ", sys.getsizeof(e))
print("sys.getsizeof(list(e))", sys.getsizeof(list(e)))
```

可以使用 next()函数不断返回下一个值的对象称为迭代器 iterator，生成器都是迭代器对象；list dict string 虽然具有可迭代属性，但不是 Iterator，可以使用 iter()转换成 iterator，验证是否是迭代器的方法即查看是否具有.__next__()方法；

生成器是一种特殊的迭代器，常在性能限制的条件下应用，readline()/readlines()就是一种常用的生成器，可在循环读取时不断处理，节省内存空间；生成生成器的方法：

- 方法一： `g = (x*x for x in range(10))` # 类似迭代器
- 在函数中加入 `yield` 关键字用来返回值，`yield` 遇到 `next()` 方法即返回值，再次执行时从上次 `yield` 返回处继续执行；

查看一个函数是否是生成器：

使用自省函数 `dir(g)`，返回：

```
[ '__class__',
....
'__next__',
...
'__repr__',
'__setattr__',
]
```

其中，魔术方法 `__next__()` 是生成器特有的属性，可通过调用 `g.__next__()` 或 `next(g)` 获得下一个生成的对象；

2.17 装饰器

装饰器是一种设计模式，常被应用于有切面需求的场景：插入日志、性能测试、事务处理等，装饰器的作用即为已经存在的对象添加额外的功能；

```
#-*- coding: UTF-8 -*-
```

```
import time
```

```
def foo():
```

```
    print 'in foo()'
```

```
# 定义一个计时器，传入一个，并返回另一个附加了计时功能的方法
```

```
def timeit(func):
```

```
    # 定义一个内嵌的包装函数，给传入的函数加上计时功能的包装
```

```
    def wrapper():
```

```
        start = time.clock()
```

```
        func()
```

```
        end = time.clock()
```

```
        print 'used:', end - start
```

```
    # 将包装后的函数返回
```

```
    return wrapper
```

```
foo = timeit(foo)
```

```
foo()
```

在定义函数 `foo()` 之后，调用之前，加上 `foo = timeit(foo)` 语句，即可达到计时的目的，在这个例子中，函数进入和退出时均需要计时，成为一个横切面 **Aspect**，这种编程方式叫做 **Aspect-Oriented Programming**；

使用语法糖@实现，可实现在函数定义之前进行装饰：

```
import time
def timeit(func):
    def wrapper():
        start = time.clock()
        func()
        end =time.clock()
        print 'used:', end - start
    return wrapper

@timeit
def foo():
    print 'in foo()'

foo()
```

在函数之后进行装饰；
使用装饰器实现单例模式；

2.18 Python 中的重载

函数重载为了解决两个问题：1. 可变的参数类型；2. 可变的参数个数；函数重载的设计原则是：仅当两个函数除了参数类型和参数个数不同之外，其功能完全相同，此时需要函数重载；

对于 1：Python 可接受任何类型的参数，若函数**功能相同而参数类型不同**，则在 Python 中代码极有可能完全相同，没有必要使用重载函数实现；

对于 2：函数功能相同而参数个数不同，在 Python 中，使用不定长参数*args 实现，对于缺少的参数，将其设定为不定长参数即可解决问题。

函数参数类型：

- 必备参数：必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样；
- 关键字参数：关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值，使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值；
- 缺省参数：调用函数时，缺省参数的值如果没有传入，则被认为是默认值；
- 不定长参数：处理必声明时更多的参数，在声明时不命名也不固定长度，语法表示为 **def functionname([format_args], *var_args_tuple):**

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 可写函数说明
def printinfo( arg1, *vartuple ):
    "打印任何传入的参数"
```

```

        print "输出: "
        print arg1
        for var in vartuple:
            print var
        return;

    # 调用 printinfo 函数
    printinfo( 10 );
    printinfo( 70, 60, 50 );

```

2.19 Python 新式类与旧式类

新式类在 Python2.2 之前就出现了，故旧式类问题完全是一个兼容问题，Python3 中的类全部是新式类，新式类加载采用广度优先策略，旧式类加载采用深度优先策略；

```

class A():
    def foo1(self):
        print "A"
class B(A):
    def foo2(self):
        pass
class C(A):
    def foo1(self):
        print "C"
class D(B, C):
    pass

d = D()
d.foo1()
# in Python2.2 or lower version : A
# in Python2.3 or higher version : C

```

按照经典类的查找顺序，从左到右深度优先，创建实例 `d`，调用方法 `foo1()`，类 `D` 中没有方法 `foo1()`，查找 `class B`，`B` 中也没有方法 `foo1()`，深度优先，继续查找类 `A`，而非广度优先查找类 `C`，调用类 `A` 的方法 `foo1()`，输出“A”，而在新式类中，则会查找类 `C`，调用重写的函数 `foo1()`，输出“C”。

2.20 邮箱地址正则表达式

正则表达式是一种字符串匹配模式，可以用来检查一个子串是否含有某个子串，将匹配子串替换或从某个串中取出符合条件的子串等，例如：

- `runoo+b`，可以匹配 `runoob`、`runooob`、`runooooooooob` 等，`+` 号代表前面的字符必须至少出现一次（1 次或多次）；

- `runoo*b`，可以匹配 `runob`、`runoob`、`runooooooooob` 等，`*` 号代表字符可以不出现，也可以出现一次或者多次（0 次、或 1 次、或多次）；
- `colou?r` 可以匹配 `color` 或者 `colour`，`?` 问号代表前面的字符最多只可以出现一次（0 次、或 1 次）；

正则表达式的组件可以是单个字符串、字符集合、字符范围、字符间的选择或以上组件的任意组合，正则表达式有普通字符(`a~z`, `0~9`)及特殊字符(元字符)组成的文字模式，模式描述在搜索文本时要匹配一个或者多个字符串，正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配；

- 普通字符：普通字符包含没有显式指定为元字符的所有可打印和不可打印字符(所有大小写字母、所有数字、所有标点符号以及一些其他符号)；
- 非打印字符(转义字符)：非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

字符	描述
<code>\cx</code>	匹配由 <code>x</code> 指明的控制字符。例如， <code>\cM</code> 匹配一个 Control-M 或回车符。 <code>x</code> 的值必须为 A-Z 或 a-z 之一。否则，将 <code>c</code> 视为一个原义的 'c' 字符。
<code>\f</code>	匹配一个换页符。等价于 <code>\x0c</code> 和 <code>\cL</code> 。
<code>\n</code>	匹配一个换行符。等价于 <code>\x0a</code> 和 <code>\cJ</code> 。
<code>\r</code>	匹配一个回车符。等价于 <code>\x0d</code> 和 <code>\cM</code> 。
<code>\s</code>	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 <code>[\f\n\r\t\v]</code> 。
<code>\S</code>	匹配任何非空白字符。等价于 <code>[^\f\n\r\t\v]</code> 。
<code>\t</code>	匹配一个制表符。等价于 <code>\x09</code> 和 <code>\cI</code> 。
<code>\v</code>	匹配一个垂直制表符。等价于 <code>\x0b</code> 和 <code>\cK</code> 。

- 特殊字符：具有特殊含义的字符，要在字符串中进行查找，应当对这些字符进行转义(加“\”)，如要在字符串中查找“*”，应当在对*进行转义：`runo*ob`；

特别字符	描述
<code>\$</code>	匹配输入字符串的结尾位置。如果设置了 <code>RegExp</code> 对象的 <code>Multiline</code> 属性，则 <code>\$</code> 也匹配 <code>\n</code> 或 <code>\r</code> 。要匹配 <code>\$</code> 字符本身，请使用 <code>\\$</code> 。
<code>()</code>	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这

特别字符	描述
	些字符，请使用 \ (和 \)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。
[标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。 例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\\ 匹配 "\，而 \" 则匹配 \"。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 \{。
	指明两项之间的一个选择。要匹配 ，请使用 \ 。

- 限定符：限定符用来指定正则表达式中某一给定组件必须出现多少次才能匹配，共有 * + ? {n} {n,} {n,m} 六种，正则表达式的限定符有：

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。 + 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。? 等价于 {0,1}。

{n}	n 是一个非负整数。匹配确定的 n 次。例如, 'o{2}' 不能匹配 "Bob" 中的 'o', 但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如, 'o{2,}' 不能匹配 "Bob" 中的 'o', 但能匹配 "foooooo" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数, 其中 n <= m 。最少匹配 n 次且最多匹配 m 次。例如, "o{1,3}" 将匹配 "foooooo" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

由于章节编号在大的输入文档中会超过 9, 限定符可以使用 `/Chapter [1-9][0-9]*/` 表达式匹配任何位数的章节号; *、+ 限定符都是贪婪的, 总是尽可能多的匹配文字, 只有在其后面加上一个 ? 才能实现非贪婪或最小匹配;

原始文档: `<H1>Chapter 1 – Introduce to Regular Expression</H1>`

`/<.*>/` : 贪婪匹配 `H1>Chapter 1 – Introduce to Regular Expression</H1`

`/<.*?>/`: 非贪婪匹配 `H1`

`/<\w+?>/`: 非贪婪匹配 `H1`

- 定位字符: 将正则表达式固定到一个单词内、单词开头或单词结尾处, 定位符用来描述字符串或单词的边界;

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性, ^ 还会与 \n 或 \r 之后的位置匹配。
\$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性, \$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个字边界, 即字与空格间的位置。
\B	非字边界匹配。

`/^Chapter [1-9][0-9]{0,1}$/`可匹配在章节号而不匹配交叉引用;

`/ter\b/`匹配字符串 Chapter 的子串"ter";

`/\Bapt/`匹配字符串 Chapter 的子串"apt";

- 选择: 将()把所有选择项括起来, 相邻选择项之间用|分隔, 产生的匹配会被存储到一个临时缓冲区中, 所捕获的每一个子匹配都按照在正则表达式模式中从左向右出现的顺序存储, 缓冲区编号范围 1~99, 每个缓冲区均可使用

\index 访问，其中；index 为缓冲区编号；

- 反向引用：对()产生的匹配缓冲区，可以使用?:、?=或?!重写捕获，忽略对相关匹配的保存；

```
var str = "Is is the cost of of gasoline going up up";
```

```
var patt1 = /\b([a-z]+) \1\b/ig;
```

```
document.write(str.match(patt1));
```

捕获的表达式由[a-z]+指定，包含一个或多个字母，\b 字边界符匹配两个空格之间的子串即单词；

\1 是对捕获的子匹配项的引用，即单词的第二个匹配项恰好由()表达式匹配，

\1 指定第一个子匹配项；

字边界元字符确保只检测整个单词；

正则表达式后面的全局标记 g 指定将该表达式应用到输入字符串中查找尽可能多的匹配；

结尾处的不区分大小写标记 i 指定不区分大小写；

```
var str = "http://www.runoob.com:80/html/html-tutorial.html";
```

```
var patt1 = /(\\w+):\\/(\\/([^/:]+)(:\\d*)?([\\# ]*)\\/);
```

```
arr = str.match(patt1);
```

```
for (var i = 0; i < arr.length ; i++) {
```

```
    document.write(arr[i]);
```

```
    document.write("<br>");
```

```
}
```

(\\w+)子表达式捕获 Web 地址的协议部分，"://"之前的任意字符，匹配"http"；

(\\/([^/:]+))子表达式捕获端口号":80"之前的部分，匹配"www.runoob.com"；

(:\\d*)?子表达式捕获端口号，匹配":"之后的零个或多个数字，且只能重复一次；

([\\#]*)子表达式捕获 Web 地址指定的路径和/或页信息，能匹配除"#"或空格字符之外的任何字符序列，匹配"/html-tutorial.html"；

正则表达式匹配邮箱地址：

```
re = /^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*(\\.\\w{2,3})+$/
```

(/)斜杠之间的所有内容都是正则表达式的组成部分；

脱字符(^)表示要用这个表达式检查以特定字符串(此处为\\w+)开头的字符串；

(\\w)表示任意单一字符，含 a~z、A~Z、0~9 或下划线这些电子邮箱地址开头合法字符；

(+)表示要寻找的前面的条目(单一字符)多于一次的出现；

()表示一个组，表达式后面的内容可能要引用()中的内容，因此将其放入一个组内；

[]表示可以出现其中任意的字符，[\\.-]允许邮箱地址中出现"."点号或连字符"-"

因为"."在正则表达式中代表单字符匹配，应当对其进行转义，此处[]表示，在邮箱地址中可以有"."或"-"其中一个存在，但不能同时存在；

(?)表示前面的条目可以出现一次或者不出现，即电子邮箱地址的第一部分中可以有一个"."或者"-"，也可以没有；

(?)后面的\\w+表示"."或者"-"后面必须要有其他字符；

()后出现的"*"表示([\\.-]?\\w+)组内的条目可以出现 0 次或者多次；

"@"为其本义，邮箱地址必须有，位于电子邮箱地址和域名之间；

"@"后再次出现"\\w+"，表明"@"后必须出现字符，([\\.-]?\\w+)表明在邮箱域名中

允许出现"."或者"-";

(\.\w{2,3})表示在邮箱域名中, 能够匹配到一个"."其后跟随 2~3 个字符, 而"+"表明(\.\w{2,3})组内的模式必须出现一次或者多次, 形如".com" ".cn" ".havard.edu";
"\$"表示匹配字符串到此处结束;

2.21 Python 内置类型 list/dictionary/tuple/string

- list 列表, 即动态数组, 对应于 C++的 vector, 可以包含不同类型的元素在同一个 list 中;

按下标读写, 当做数组处理, list[0] list[-1]...;

len(list)获得 list 的长度;

创建连续的 list : list = range(1, 5) # list = [1,2,3,4], 不含右边界

list 常用方法:

L.append(var)	#追加元素
L.insert(index, var)	#插入元素
L.pop(var)	#返回最后一个元素, 并从 list 中删除
L.remove(var)	#删除第一次出现的该元素
L.count(var)	#统计该元素在 list 中出现的次数
L.index(var)	#该元素出现的位置
L.extend(list)	#追加 list 到 L 上
L.sort()	#排序
L.reverse()	#倒序
L[1:]	#片段操作符, 从 1~最后
[1,2] + [3,4]	#作用同 extend()
[2]*4	#[2,2,2,2]
del L[index]	#删除指定下标的元素
del L[l, h]	#删除指定下表范围的元素
L1 = L	#L1 是 L 的别名, 指向相同的地址
L1 = L[:]	#L1 为 L 的拷贝, 深拷贝
[<expr1> for k in L if <expr2>]	#list 遍历

- dictionary 字典, 对应于 C++中的 Map, 键值对的格式存储 dict = {'ob1': 'str1', 'ob2': 'str2', 'ob3': 'str3', ...};

常用的 dict 方法:

D.get(key, 0)	#同 D[key], 没有则返回默认值 0
D.has_key(key)	#有此键则返回 TRUE, 否则返回 FALSE
D.keys()	#返回字典键的列表
D.values()	#返回字典值的列表
D.items()	#返回字典键值对全列表
D.update(dict2)	#增加合并字典
D.popitem()	#弹出第一个 item 并从字典中删除
D.clear()	#清空字典, 同 del D
D.copy()	#拷贝字典, 浅拷贝
D.cmp(dict1, dict2)	#比较字典, 比较优先级元素个数→key→value, #第一个大则返回 1, 小则返回-1

- ```
dict1 = dict #dict1 只是别名，指向同一块内存
dict2 = dict.copy() #拷贝
```
- tuple 元组即常量数组，tuple = ('a', 'b', 'c', 'd', 'e')，可以使用 list 的[index]，：操作符提取元素，但不能修改(常量)；
  - string 字符串不可变对象，无法直接修改字符串中的子串，但是可以提取 str[:6]，常用的字符串方法：
    - substr in str                      #判断子串是否在字符串中
    - substr not in str
    - S.find(substring, [start [,end]])                      #指定范围查找子串，返回索引
    - S.rfind(substring, [start [,end]])                      #反向查找
    - S.index(substring, [start [,end]])                      #同 find()，查找不到则抛出异常
    - S.rindex(substring, [start [,end]])
    - S.lowercase()
    - S.capitalize()                      #首字母大写
    - S.lower()
    - S.upper()
    - S.swapcase()
    - S.split(string, ' ')                      #以空格切分 string 成 list
    - S.join(list, ' ')                      #将 list 转成 string，以空格连接
    - S.strip('c')                      #移出头尾指定字符 c 并返回
    - len(str)                      #串长度
    - cmp("my friend", str)                      #字符串比较，第一个大返回 1
    - max('abcxyz')                      #寻找字符串中最大的字符
    - min('abcxyz')                      #寻找字符串中最小的字符
    - oat(str)                      #变成浮点数 float("1e-1") 对应 0.1
    - int(str)                      #变成整型 int("12")对应 12
    - int(str,base)                      #变成 base 进制整型数，int("11",2) 对应 2
    - long(str)                      #变成长整型
    - long(str,base)                      #变成 base 进制长整型
    - str\_format % (参数列表)                      #参数列表是以 tuple 的形式定义的，即不可运行中改变
    - >>>print "%s's height is %dcm" % ("My brother", 180)
      - #结果显示为 My brother's height is 180cm
    - list 和 tuple 的相互转化
    - tuple(ls)
    - list(ls)

## 2.22 Python 中的 is

“is”方法是对比对象的地址，“==”对比的是对象的值。

## 2.23 read/readline 和 readlines

- read()读取整个文件;
- readline()读取下一行, 使用生成器方法;
- readlines()读取整个文件到一个迭代器中供遍历使用;  

```
fh = open('c:\\autoexec.bat')
for line in fh.readlines(): # 将文件读入缓冲区
 print(line)
```

## 2.24 垃圾回收

Python Garbage Collection 通过引用计数 Reference Counting 来跟踪和回收垃圾, 在引用计数的基础上, 通过“标记-清除”(Mark and Sweep)解决容器对象可能产生的循环引用问题, 通过分代回收(Generation Collection)以空间换时间的方法提高垃圾回收的效率;

- 引用计数: PyObject 是每个对象必有的内容, 其中 ob\_refcnt 作为引用计数, 当一个对象有新的引用时, ob\_refcnt 计数会增加, 当引用对象被删除时, ob\_refcnt 计数会减少, 当 ob\_refcnt 清零时, 该对象的生命周期结束;
- 标记-清除机制: 按需分配, 当没有空闲内存时, 从寄存器和栈上的引用出发, 遍历以对象为节点, 以引用为边构成的图, 将所有可以访问到的对象打上标记, 清扫内存, 将没有标记的对象释放;
- 分代回收: 将系统中所有内存块根据其存活时间划分为不同的集合, 每个集合成为一个“代”, 垃圾收集频率随着“代”的存活时间增大而减小, 存活时间通常利用记过几次垃圾回收来度量。

# 3.Algorithm

## 3.1 时间复杂度计算

待解决问题的规模为  $n$ , 基本操作被重复执行的次数为  $n$  的函数, 时间复杂度记作:  $T(n) = O[f(n)]$ , 表示随着问题规模  $n$  的增长, 算法执行的时间增长率与  $f(n)$  的增长率相同;

```
for(i=1; i<=n; i++){
 for(j=1; j<=i; j++){
 ++x; //执行频度为 1+2+3+...+n=(n+1)n/2
 a[i,j] = x;
 }
}
```

常见的复杂度:

- 常数阶  $O(1)$ : 没有循环(for/while), 仅对变量做常数范围以内的操作;
- 线性阶  $O(n)$ : 一层循环

```
int i;
for(i=0; i<n; i++){
 //operations
}
```

- 对数阶  $O(\log(n))$ :

```
int count = 1;
while(count < n){
 count = count * 2;
}
```

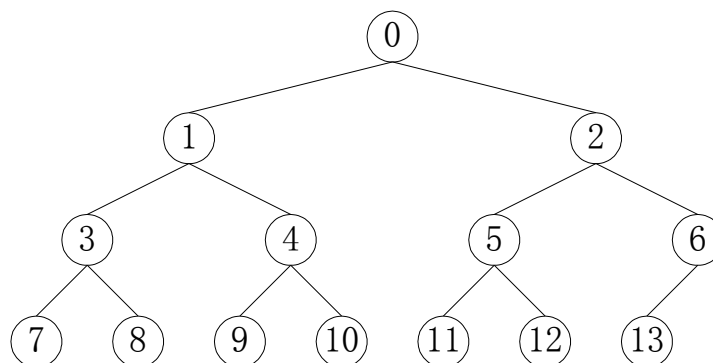
未限定问题规模,  $2^x = n$ ; 得出  $x = \log(n)$ , 算法复杂度为  $O(\log(n))$ ;

- 平方阶  $O(n^2)$ : 两重循环

```
int i,j;
for(i=0; i<n; i++){
 for(j=0; j<n; j++){
 //operations
 }
}
```

## 3.2 二叉树

### 3.2.1 二叉树的数据结构



二叉树的三种遍历:

- 先序遍历: ParentNode—LeftNode—RightNode;
- 中序遍历: LeftNode—ParentNode—RightNode;
- 后序遍历: LeftNode—RightNode—ParentNode;
- 层序遍历: 逐层遍历;

二叉树常用作二叉查找树、二叉堆和二叉排序树;

//二叉树节点数据结构

```
typedef struct BiTreeNode{
```

```

// Data Field
char data;
// Child pointer
struct BiTreeNode *lchild, *rchild;
}BiTreeNode, *BiTree;

```

### 3.2.2 二叉树的生成

```

// 先序生成创建二叉树
int CreateBiTree(BiTree &T){
 char data;
 // 按先序顺序输入二叉树节点中的数值
 InputNode(&data);
 if(CHARP_VALUE == data){
 T = NULL;
 }
 else{
 T = (BiTree)malloc(sizeof(BiTreeNode));
 T->data = data;
 CreateBiTree(T->lchild);
 CreateBiTree(T->rchild);
 }
 return 0;
}

```

### 3.2.3 二叉树的遍历算法

三种遍历均有递归实现与循环实现两种方法(广度优先/深度优先):

广度优先: 逐层遍历, 从左至右依次访问, 可利用 FIFO 队列实现广度优先搜索;

深度优先: 先访问根节点, 然后遍历左子树后遍历右子树, 可利用栈 FILO 的特点, 将右子树压栈后将左子树压栈;

可以使用堆(单数组加上堆的最末节点的下标)来表示完全二叉树;

二叉搜索树上的基本操作花费的时间与树的高度成正比, 完全二叉树操作的时间复杂度为  $O(\log(n))$ , 二叉搜索树的性质: 对二叉搜索树中的节点  $x$ , 若  $y$  为其左子树中的节点,  $y.key \leq x.key$ ; 若  $y$  为其右子树中的节点,  $y.key \geq x.key$ ;

- 二叉树遍历
- 二叉树查找
- 二叉树插入与删除

中序遍历的递归写法:

```

void InorderTreeWalk(pTree* tree){
 if(NULL != tree){

```

```

 InorderTreeWalk(tree->pLeft);
 StoreTree(tree->key);
 InorderTreeWalk(tree->pRight);
 }
}

```

先序遍历的递归写法:

```

void PreorderTreeWalk(pTree* tree){
 if(NULL != tree){
 StoreTree(tree->key);
 PreorderTreeWalk(tree->pLeft);
 PreorderTreeWalk(tree->pRight);
 }
}

```

后序遍历的递归写法:

```

void PostorderTreeWalk(pTree* tree){
 if(NULL != tree){
 PostorderTreeWalk(tree->pLeft);
 PostorderTreeWalk(tree->pRight);
 StoreTree(tree->key);
 }
}

```

先序遍历的非递归方法: 先使根节点 `tree` 入栈 `s.push(tree)`, 只要栈不为空 `!s.empty()`, 即可使栈中元素出栈 `s.pop()`, 每次弹出一个节点, 都要将其右孩子节点 `tree->pRight` 入栈 `s.push(tree->pRight)`, 再将其左孩子节点 `tree->pLeft` 入栈 `s.push(tree->pLeft)`;

访问 `tree->key` 后, 将 `tree` 入栈, 遍历 `tree->pLeft`, 遍历左子树结束后, 栈顶元素为 `tree`, `tree` 出栈, 遍历右子树;

```

void PreorderNonrecursive(BiTree T){
 if(! T){
 return ;
 }
 stack<BiTree> s; // STL
 s.push(T); // push root node

 while(! s.empty()){
 BiTree temp = s.top(); // top element in stack
 StoreNode(temp->data);
 s.pop(); // pop root
 if(NULL != temp->rchild){ // right child push
 s.push(temp->rchild);
 }
 if(NULL != temp->lchild){ // left child push
 s.push(temp->lchild);
 }
 }
}

```

```

 }
}
}

```

中序遍历的非递归方法:

将 tree 入栈, 遍历左子树 tree->pLeft, 栈顶元素为 tree, tree 出栈, 访问 tree->key, 遍历右子树 tree->pRight;

引入指向当前节点的指针 curr, 以 curr 指向非 NULL 或栈非空为外循环条件, 先将左子树全部入栈; 若栈非空, 则将 curr 指向栈顶元素, 并将其出栈, 访问该节点, 将 curr 指针指向当前节点的右孩子节点;

```

void InorderNonrecursive(BiTree T){
 if(! T){
 return ;
 }
 BiTree curr = T; //point to current node
 stack<BiTree> s;
 while(NULL != curr || ! s.empty()){
 //左子树全部入栈
 while(NULL != curr){
 s.push(curr);
 curr = curr->lchild;
 }//while
 if(! s.empty()){
 curr = s.top(); // point to top element in stack
 s.pop(); // pop element out
 StoreNode(curr->data); // store element
 curr = curr->rchild; // point to right child
 }
 }//while
}

```

后序遍历的非递归方法:

将 tree 入栈, 遍历左子树 tree->pLeft, 栈顶元素为 tree, 将 tree 出栈, 遍历右子树 tree->pRight;

当前节点指针 curr, 节点已访问标记 previsited, 以 curr 指针非 NULL 或栈非空为外循环条件, 将左子树全部入栈; curr 指向栈顶元素, 若 curr 节点的右孩子节点为 NULL 或 curr 节点的右孩子节点已被访问:

T: 访问当前节点, 并标记当前节点的 previsited 标记, curr 节点出栈, 并置 curr 为 NULL;

F: 否则访问 curr 的右孩子节点;

```

void PostorderNonrecursive(BiTree T){
 stack<BiTree> s;
 BiTree curr = T; // point to current node
 BiTree previsited = NULL;

```

```

while(NULL != curr || ! s.empty()){
 while(NULL != curr){
 s.push(curr); // 左子树全部入栈
 curr = curr->lchild;
 } //while
 curr = s.top(); //左子树叶子节点
 /* 右子树为空或已被访问过, 则访问当前节点并标记当前节点已被访问
 * 当前节点出栈, curr 指针指空;
 */
 if(NULL == curr->rchild || previsted = curr->rchild){
 StoreNode(curr->data); //store current node
 previsted = curr; //flag pointer
 s.pop(); //current node pop out
 curr = NULL; //current pointer to NULL
 } //if
 else{
 curr = curr->rchild; //访问右子树中的左子树
 } //else
} //while
}

```

后续遍历的非递归遍历，双栈法：

```

void PostOrderDoubleStack(BiTree T){
 stack<BiTree> s1, s2;
 BiTree curr; //指向当前节点
 s1.push(T);
 while(! s1.empty()){
 curr = s1.top();
 s1.pop();
 s2.push(curr);
 if(curr->lchild != NULL){
 s1.push(curr->lchild);
 }
 if(curr->rchild != NULL){
 s1.push(curr->rchild);
 }
 } //while

 while(! s2.empty()){
 StoreNode(s2.top()->data);
 s2.pop();
 }
}

```



二叉树的先序遍历，中序遍历和后序遍历均是 DFS 深度优先，易于使用递归方法实现；层序遍历是 BFS 广度优先，易于使用队列实现非递归方法：

```
int visit(BiTree T){
 if(T){
 StoreNode(T->data);
 return 1;
 }
 else
 return 0;
}
```

使用队列(FIFO)实现二叉树的层序遍历具有结构优势：

```
void LevelOrderWalk(BiTree T){
 queue<BiTree> Q;
 BiTree p; //pointer to BiTree
 p = T;
 if(1 == visit(p)){
 Q.push(p);
 }
 while(! Q.empty()){
 p = Q.front();
 Q.pop();
 if(1 == visit(p->lchild)){
 Q.push(p->lchild);
 }
 if(1 == visit(p->rchild)){
 Q.push(p->rchild);
 }
 }
}
```

二叉树求深度操作：

```
int BiTreeDepth(BiTNode *T){
 if(! T){
 return 0;
 }
 int d1, d2;
 d1 = BiTreeDepth(T->lchild);
 d2 = BiTreeDepth(T->rchild);

 return (d1>d2? d1:d2) + 1;
}
```

二叉树求节点数操作：

```

int BiTreeNodeCount(BiTreeNode *T){
 if(NULL == T){
 return 0;
 }
 return (1 + BiTreeNodeCount(T->lchild) + BiTreeNodeCount(T->rchild));
}

```

### 3.3 最大堆

最大堆是一种完全二叉树，其父节点的关键字不小于其左右子节点的关键字；

### 3.4 红黑树

### 3.5 B 树

### 3.6 线性结构-栈与队列

栈与队列都是动态集合，栈是 LIFO 结构，只有栈顶指针指示当前所在位置，入栈操作 PUSH 和出栈操作 POP 只能在栈顶进行，栈有上溢出和下溢出错误；队列是 FIFO 结构，分别有队头指针和队尾指针，入队操作 ENQUEUE 在队尾进行，出队操作 DEQUEUE 在队头；

### 3.7 线性结构-链表

### 3.8 寻找链表倒数第 k 个节点

寻找链表中倒数第 k 个节点，定义两个指针 pAhead 和 pBehind，使 pAhead 先走 k-1 步，然后两个指针一起移动，当 pAhead 移动到链表尾部时，pBehind 所在位置即倒数第 k 个节点；容易写出以下代码：

```

ListNode* FindKthToTail(ListNode* pListHead, unsigned int k){
 Unsigned int i;
 if(NULL == pListHead){
 return nullptr;
 }
}

```

```

 }

 ListNode* pAhead = pListHead;
 ListNode* pBehind = nullptr;

 for(i=0; i<k-1; i++){
 pAhead = pAhead->next;
 }

 pBehind = pListHead;

 while(nullptr != pAhead->pNext){
 pAhead = pAhead->pNext;
 pBehind = pBehind->pNext;
 }

 return pBehind;
}

```

需要考虑的问题：

- 输入参数为空指针，即 pListHead 为 nullptr；
- 输入参数 k 非法，如 k 为 0；
- 链表节点总数小于 k；

## 3.9 快速排序

复杂度

最坏情况如何优化

## 3.10 堆排序

复杂度

最坏情况如何优化

## 3.11 无序数字列表寻找所有间隔为 d 的组合

## 3.12 列表[a1, a2, a3, ..., an]求其所有组合

### 3.13 一行 python 代码实现 $1+2+3+...+10^{**}8$

### 3.14 长度未知的单向链表求其是否有环

### 3.15 单向链表应用快速排序

### 3.16 长度为 n 的无序数字元素列表求其中位数

### 3.17 遍历一个内部未知的文件夹

### 3.18 台阶问题/斐波那契

青蛙上台阶，一次可以跳 1 级台阶也可以跳 2 级台阶，求上 n 层台阶共有多少种跳法：

```
fib = lambda n: 2 if n<=2 else fib(n-1)+fib(n-2)
```

### 3.19 变态台阶问题

青蛙上台阶，一次可以跳 1 级台阶，2 级台阶，...，n 级台阶，求上 n 层台阶共有多少种跳法：

```
n = 1 : fib(1) = 1 ;
```

```
n = 2 : fib(2) = fib(1) + 1 ;
```

```
n = 3 : fib(3) = fib(2) + fib(1) + 1 ;
```

```
...
```

```
n = n : fib(n) = fib(n-1) + fib(n-2) + ... + fib(1) + 1
```

```
fib = lambda n : 1 if n<2 else 2*fib(n-1)
```

### 3.20 矩形覆盖问题

可以使用  $2 \times 1$  的小矩形横向或者纵向无重叠覆盖一个  $2 \times n$  的大矩形, 求问总共有多少种覆盖方法(与上台阶问题相同):

$f = \text{lambda } n: 1 \text{ if } n < 2 \text{ else } f(n-1) + f(n-2)$

## **4.OS**

### **4.1 多线程与多进程的区别**

CPU 密集型适合使用多线程还是多进程；

### **4.2 协程**

### **4.3 进程间通信方式**

### **4.4 虚拟存储系统中缺页计算**

### **4.5 并发进程不会引起死锁的资源数量计算**

### **4.6 常用的 Linux/git 命令和作用**

### **4.7 查看当前进程的命令**

### **4.8**

## **5.网络**

### **5.1 TCP/IP 协议**

### **5.2 OSI 五层协议**

### **5.3 Socket 长连接**

### **5.4 Select 与 epoll**

### **5.5 TCP 与 UDP 协议的区别**

### **5.6 TIME\_WAIT 过多的原因**

### **5.7 http 一次连接的全过程描述**

从用户发起 request 到用户接收到 response

### **5.8 http 连接方式——get 与 post 的区别**

## **5.9 restful**

## **5.10 http 请求的状态码 200/403/404/504**



## **6.数据库**

### **6.1 MySQL 锁的种类**

### **6.2 死锁的产生**

### **6.3 MySQL 的 char/varchar/text 的区别**

### **6.4 Join 的种类与区别**

### **6.5 A LEFT JOIN B 的查询结果中，B 缺少的部分如何显示**

### **6.6 索引类型的种类**

### **6.7 BTree 索引与 hash 索引的区别**

### **6.8 如何对查询命令进行优化**

### **6.9 NoSQL 与关系型数据库的区别**

## 6.10 Redis 常用的存储类型

# 7.机器学习

## 7.1 模型评估方法

### 7.1.1 数据集划分

泛化误差最小，训练误差过小易导致过拟合，将数据集划分为训练集和测试集，并使训练集与测试集尽量互斥，训练/测试集生成方法：

- 留出法：保留类别比例，分层采样，单次留出得到的评估结果不够稳定可靠，使用时，常采用若干次随机划分重复试验评估取平均值作为最终的评估结果；
- 交叉验证法：分层采样将原数据集划分为  $k$  个子集， $k-1$  个子集作为训练集，1 个子集作为验证集，成为  $k$  折交叉验证，在数据集  $D$  上采用  $n$  次交叉验证法，成为  $n$  次  $k$  折交叉验证；
- 自助法：对  $|D| = m$  的数据集进行有放回采样，得到训练数据集  $D'$ ， $|D'| = m$ ，将初始数据集作为测试数据集测试模型的泛化性能，其中有 36.8% 的样本没有在训练数据集中出现，称为包外估计；

### 7.1.2 模型的性能度量

模型的好坏不仅取决于算法和数据，还取决于任务需求：

- 回归任务：均方误差；
- 分类任务：分类错误率  $E(f; D) = \frac{1}{m} \sum_{i=1}^m I(f(x_i) \neq y_i)$ ；

分类精度： $acc(f; D) = 1 - E(f; D)$ ；

**P 查准率(准确率)**：预测正例中客观正例所占的比率

**R 查全率(召回率)**：客观正例中能够预测为正例所占的比率

**F1 指数**： $F1 = \frac{2 \times P \times R}{P + R}$ ；

### 7.1.3 偏差-误差分解

模型在数据集  $D$  上的输出为  $f(x; D)$ ，模型期望预测为  $E_D[f(x; D)]$ ；

- 方差来源于数据集的波动，方差刻画了数据扰动造成的影响，使用不同样本集(样本容量相同)产生的模型输出方差为：

$var[f(x)] = E_D[(f(x; D) - \exp[f(x)])^2]$ ；

- 噪声来源于数据集自身的缺陷，噪声代表模型在当前数据集上所能达到的期望泛化误差下界，代表问题的难度，数据集自身固有的噪声为：

$\epsilon^2 = E_D[(y_D - y)^2]$ ；

- 模型的偏差来源于模型的性能缺陷，刻画了模型算法的拟合能力，期望输出与样本真实标记之间的偏差为：

$bias^2(\exp[f(x)]) = [\exp[f(x)] - y]^2$ ；

而模型最终的泛化误差可以分解为数据集波动方差、模型预测偏差与数据集自身噪声之和：

$$E(f; D) = \text{bias}^2(\exp[f(x)]) + \text{var}[f(x)] + \varepsilon^2;$$

当模型拟合能力较弱时，训练数据集带来的扰动不足以引起模型发生显著的变化，此时模型泛化误差主要来源于偏差，此时，模型处于欠拟合阶段；

随着训练程度加深，模型的拟合能力增强，数据集的轻微扰动都能够引起模型发生显著变化，此时，方差主导了泛化误差的变化，若训练数据自身的非全局的特征被学习到，模型就发生了过拟合；

因此，很多算法会引入策略避免发生过拟合：决策树剪枝，集成学习(Bagging)减少基学习器数量，神经网络引入早停等。

## 7.2 LR 与极大似然估计

对数几率回归将线性回归模型应用于分类问题：

$$z = \omega^T x + b;$$

$$y = \frac{1}{1 + \exp(-z)};$$

对于二类分类问题：

$$p(y = 1|x) = \frac{\exp(\omega^T x + b)}{1 + \exp(\omega^T x + b)};$$

$$p(y = 0|x) = \frac{1}{1 + \exp(\omega^T x + b)};$$

极大似然估计法计算参数：

对于数据集 $\{(x_i, y_i)\}_{i=1}^N$ ，使每个样本属于其真实标记的概率最大，由于样本是

独立同分布的，因此使得 $\prod_{i=1}^N p(y_i|x_i; \omega, b)$ 最大，但是连续乘积运算会导致下溢，因此常采用对数求和形式作为对数似然函数，对模型进行极大似然估计，即有：

$$L(\omega, b) = \sum_{i=1}^N \ln(p(y_i|x_i; \omega, b));$$

此处，令 $\beta = (\omega; b)$ ， $\hat{x} = (x; 1)$ ，采用后验概率的幂指数形式作为似然项：

$$L(\beta) = \ln(\prod_{i=1}^N [p(y_i = 1|x_i; \beta)]^{y_i} [p(y_i = 0|x_i; \beta)]^{1-y_i});$$

求解 $\beta^* = \arg \min_{\beta} L(\beta)$ ;

$L(\beta)$ 是 $\beta$ 的高阶连续可导凸函数，采用数值方法求解(梯度下降法/牛顿法);

## 7.3 LDA

线性判别分析：将训练数据集投影到一条直线上，使得同类样本点尽可能接近，异类样本点尽可能远离，当有新的样本点加入时，根据其投影的位置判断其所属类别；

模型学习算法即使得其同类样本点的投影尽可能接近，即其协方差尽可能小：

$$\omega^T \Sigma_0 \omega + \omega^T \Sigma_1 \omega;$$

使得异类样本点投影尽可能远离，即使得两类样本点中心距离尽可能大：

$$\|\omega^T \mu_0 - \omega^T \mu_1\|_2^2;$$

定义类内散度矩阵：

$$S_w = \Sigma_0 + \Sigma_1 = \sum_{x \in X_0} (x - \mu_0)(x - \mu_0)^T + \sum_{x \in X_1} (x - \mu_1)(x - \mu_1)^T;$$

类间散度矩阵：

$$S_b = (\mu_0 - \mu_1)(\mu_0 - \mu_1)^T;$$

优化目标为最大化广义瑞利商：

$$J = \frac{\omega^T S_b \omega}{\omega^T S_w \omega}; \text{ 令其分母为 1 作为约束条件, 分子取负号作为优化目标得到等式约}$$

束最优化问题，拉格朗日乘子法求解；

## 7.4 类别不平衡问题

类别不平衡问题是指训练样本中不同类别样本数量不同的情况，处理类别不平衡的策略有：

- 再缩放： $\frac{y'}{1-y'} = \frac{y}{1-y} \times \frac{m^-}{m^+}$ ，即使用预测正例/预测负例>正例数量/负例数量作为模型的预测几率；
- 欠采样：取出训练样本集中的过多的反例，使得正负样例数量接近；
- 过采样：增加训练集中的正例样本，使得政府样例数量接近；
- 阈值迁移：使用现有训练集训练模型，将再缩放接过嵌入预测结果；

## 7.5 判别/生成模型与先验/后验概率

以概率框架来理解机器学习，机器学习所要实现的任务是基于有限的训练样本极可能准确的估计后验概率  $P(c|x)$ ，主要有两种策略：

- 判别式模型：给定样本  $x$ ，直接建模  $P(c|x)$  预测样本所属分类  $c$ ；
- 生成式模型：先对联合概率  $P(x,c)$  建模，由此得到后验概率  $P(c|x)$  预测样本分类；

$$\text{对于生成模型, 应用贝叶斯公式: } P(c|x) = \frac{P(x,c)}{P(x)} = \frac{P(c)P(x|c)}{P(x)};$$

其中  $P(c)$  为类别的先验概率，是对模型假设空间的描述，与输入样本集没有关系；

$P(x)$  为输入样本属性的先验概率，是对输入样本集的描述，与样本分类无关；

$P(x|c)$  表示在给定类别的情况下，样本在数据集中出现的概率，称为似然；

- 先验概率  $P(c_k)$ ：根据大数定理，可以求各个类别(category)的样本在训练集中出现的频率作为先验概率  $P(c_k) = \frac{|D_k|}{|D|}$  ( $k = 1, 2, \dots, K$ ) 的估计，其中  $K = |C|$  为类别总数；

- 似然  $P(x|c_k)$ ：通常首先假设其具有某一种形式的概率分布，再基于训练样

本子集 $D_k$ (训练样本集中属于类别  $k$  的样本子集)估算其参数, 参数估计方法即极大似然估计 $\hat{\theta}_k = \arg \max_{\theta_k} L(\theta_k)$ ;

## 7.6 朴素贝叶斯分类器

先验概率、后验概率与似然概率见 **Chapter 7.5**;

贝叶斯公式:  $P(c_k|x) = \frac{P(c_k)P(x|c_k)}{P(x)}$ ;

贝叶斯公式的物理意义: 样本属性 $x_i$ 独立于其类别标签 $y_i = c_k$ 被观察到的概率越大, 表明其对类别的支持度越小;

样本所属分类 $c^* = \arg \max_c P(c_k|x)$ ; 即贝叶斯分类器并不将样本直接指派给某一分类, 而是给出其所属分类的一个概率, 在分类过程中, 所有属性都会起到作用, 属性可以是离散变量, 也可能是连续变量;

类条件概率 $P(x|c_k)$ 代表在类别 $c_k$ 上的所有属性上的联合概率, 通常样本具有多个属性(feature), 将其表示成特征向量的形式, 即 $x = (x_1, x_2, \dots, x_d)$ ,  $x_i$ 为样本 $x$ 在属性 $i$ 上的取值( $i = 1, \dots, d$ ), 则贝叶斯模型求解的是样本各个特征取值对其所属分类的决定作用;

先验概率 $P(c_k)$ 可依照大数定理求解,  $P(c_k) = \frac{|D_k|}{|D|}$ ;

而类条件概率(似然概率)较难求解, 朴素贝叶斯分类器引入属性条件独立性假设:

则有:  $P(c_k|x) = \frac{P(c_k)}{P(x)} \prod_{i=1}^d P(x_i|c_k)$ ;

其中类别-属性条件概率 $P(x_i|c_k) = \frac{|D_{k,i}|}{|D_k|}$ ;

由条件独立性假设的连乘运算带来的问题: 训练样本集中的未出现属性值的类条件概率 $P(x_{null}|c_k) = 0$ , 会将其他属性的类条件概率抹去, 因而引入拉普拉斯平滑策略:

$\hat{P}(c_k) = \frac{|D_k|+1}{|D|+K}$ ; 其中  $K=|C|$  为数据集中类别的总数;

$\hat{P}(x_i|c_k) = \frac{|D_{k,i}|+1}{|D_k|+|X_i|}$ ; 其中 $|X_i|$ 为第 $i$ 个属性的取值个数( $x_i = x_i^l, l = 1, \dots, |X_i|$ );

## 7.7 常用的损失函数

损失函数 **Loss Function** 用来度量模型输出与样本标签之间的差异, 在模型的训练过程中, 以最小化损失函数为目标, 选择优化策略, 调整模型参数, 将训练过程写作:

$$\min_{\omega} (\sum_{i=1}^N \text{loss}[f(x_i), y_i]) + \lambda \Omega(\omega);$$

其中 $\sum_{i=1}^N \text{loss}[f(x_i), y_i]$ 为模型的损失函数, 衡量模型在训练数据集上的经验风险;

$\lambda\Omega(\omega)$ 是为了避免模型过拟合引入的正则化项，衡量模型的结构风险；

- 回归问题常用的损失函数为均方误差；
- 分类问题常用的损失函数有：

0/1 损失  $loss_{0/1}(z) = \begin{cases} 1; & \text{if } z < 0 \\ 0; & \text{else} \end{cases}$ ；

0/1 损失函数非凸不连续，数学性质不好，不易于求解，通常选用其替代函数作为替代损失函数：

hinge 损失(hinge loss):  $loss_{hinge}(z) = \max(0, 1 - z)$ ；

指数损失(exponential loss):  $loss_{exp}(z) = \exp(-z)$ ；

对率损失(logistic loss):  $loss_{log}(z) = \log(1 + \exp(-z))$ ；

## 7.8 决策树

决策树的数据结构：

- 根节点：根节点包含全部样本集；
- 内部节点：对样本集进行属性测试，根据属性测试的结果将样本集划分到其左右子树；
- 叶子节点：对应于样本集的分类结果；

决策树划分的停机条件：

- 当前节点包含的样本全部属于同一类别；
- 当前属性集为空，或所有样本在所有属性集上评价指标相同；
- 当前节点包含样本集为空；

决策树划分过程要度量按照某一属性划分后其样本集合的纯度，常采用指标有：

- **ID3(Iterative Dichotomiser)**算法使用的性能指标为信息增益：

$$g(D|A) = H(D) - H(D|A)；$$

其中  $H(D) = -\sum_{k=1}^K \frac{|C_k|}{|D|} \log_2 \frac{|C_k|}{|D|}$ ；其中  $K$  为样本所属类别总数

$$H(D|A) = \sum_{i=1}^n \frac{|D_i|}{|D|} H(D_i)；其中 n 为当前待划分的属性取值个数$$

$$H(D_i) = -\sum_{k=1}^K \frac{|D_{i,k}|}{|D_i|} \log_2 \frac{|D_{i,k}|}{|D_i|}；$$

- **C4.5(Classifier)**算法使用的性能指标为信息增益比：

$$g_R(D, A) = \frac{g(D, A)}{H_A(D)}；$$

其中  $g(D, A) = H(D|A)$ ；

$$H_A(D) = -\sum_{i=1}^n \frac{|D_i|}{|D|} \log_2 \frac{|D_i|}{|D|}；$$

- **CART(Classification and Regression Tree)**算法使用的性能指标是基尼指数：

$$Gini(D) = 1 - \sum_{k=1}^K \left(\frac{|C_k|}{|D|}\right)^2；$$

$$Gini(D, A) = \sum_{i=1}^n \frac{|D_i|}{|D|} Gini(D_i);$$

决策树的剪枝处理，在决策树的学习过程中，为了尽可能正确地对训练数据集进行分类，节点的划分过程或不断进行，可能会导致过拟合，因此需要对生成的决策树进行剪枝处理：

- **预剪枝**：在决策树生成过程中，对每个节点在划分之前进行估计，若按照当前划分策略进行划分能够带来决策树整体的泛化性能提升，则进行划分，否则停止划分并将当前节点设置为叶子节点，是一种贪心策略，会导致决策树欠拟合；
- **后剪枝**：对生成的决策树从下向上进行考察，若将当前节点对应的子树替换为叶子节点能够带来决策树整体泛化性能的提升，则将当前子树替换为叶子节点，即不按照原划分策略划分数据集；

决策树的剪枝目标是极小化树的整体损失函数：

$$C_{\alpha}(T) = \sum_{t=1}^{|T|} N_t H_t(T) + \alpha |T|;$$

其中， $T$  为决策树叶子节点， $|T|$  为决策树叶子节点的个数，代表决策树模型的复杂程度；

$N_t$  为当前叶子节点上样本的数量， $H_t(T) = -\sum_{k=1}^K \frac{N_{t,k}}{N_t} \log_2 \frac{N_{t,k}}{N_t}$ ，其中  $K$  为样本所属种类数量；

$\sum_{t=1}^{|T|} N_t H_t(T)$  为模型的经验风险，代表决策树对训练数据集的拟合程度， $\alpha |T|$  为模型结构风险，代表决策树模型的复杂程度；剪枝过程的目标就是最小化整体损失函数  $C_{\alpha}(T)$ ；

后剪枝决策树欠拟合风险较小，泛化性能比预剪枝决策树好得多，但是训练过程及其时间开销较大。

## 7.9 连续值与缺失值处理

属性的连续值处理：给定样本集  $D$  和连续属性  $a$  在  $D$  上出现了  $n$  个取值  $\{a^1, a^2, \dots, a^n\}$ ，选择划分点  $T_a = \{ \frac{a^i + a^{i+1}}{2} \mid 1 \leq i \leq n-1 \}$  进行划分，检查此划分对样本集合的信息增益：

$$g(D|a) = \max_{t \in T_a} g(D|a, t) = \max_{t \in T_a} [H(D) - \sum_{\lambda \in \{-1, +1\}} \frac{|D_t^\lambda|}{|D|} H(D_t^\lambda)];$$

选择使得信息增益最大的  $t$  作为连续属性  $a$  的划分点；

属性的缺失值处理：将缺失数值的样本以不同的概率划分到不同的子节点中去；

## 7.10 BP 算法

神经网络的误差反向传播算法(**Error Back Propagation**)，基于**梯度下降法**，以目标函数的负梯度方向对参数进行调整：

参数更新式： $v \leftarrow v + \Delta v$ ；



目标函数： $E_k = \frac{1}{2} \sum_{j=1}^l (\hat{y}_j^k - y_j^k)^2$  均方误差；

对于训练样例 $(x^k, y^k)$ ，神经网络输出为 $\hat{y}_k = (y_1^k, y_2^k, \dots, y_l^k)$ ，代表经过 **softmax** 层处理后得到的一个概率分布；

而 $y_j^k = f(\beta_j - \theta_j)$ 代表第  $j$  个输出层神经元的输出， $f(z) = \frac{1}{1+\exp(-z)}$  **sigmoid** 函数作为其激活函数， $\theta_j$ 为其激活阈值；

$\beta_j = \sum_{h=1}^q \omega_{hj} b_h$ 代表第  $j$  个输出层神经元的输入， $\omega_{hj}$ 为连接权重；

$b_h$ 为第  $h$  个隐含层神经元的输出， $b_h = f(\alpha_h - \gamma_h)$ ；

$\alpha_h$ 为第  $h$  个隐含层神经元的输入 $\alpha_h = \sum_{i=1}^d v_{ih} x_i$ ；

用到 **Sigmoid** 函数的一个重要特性 $f'(x) = f(x)(1 - f(x))$ ；

依次对各个权值和激活阈值求偏导数，得到各个参数的更新式；

**BP 算法**的工作流程：将输入样例提供给输入层神经元，逐层向前传递，直至产生输出，计算输出与样例标签的误差，再将误差向前传递至隐含层；  
参数更新列表：

$$\omega_{hj} \leftarrow \omega_{hj} + \Delta\omega_{hj}; \Delta\omega_{hj} = -\eta \frac{\partial E_k}{\partial \omega_{hj}}; \text{沿着负梯度方向以学习速率更新参数}$$

$$v_{ih} \leftarrow v_{ih} + \Delta v_{ih}; \Delta v_{ih} = -\eta \frac{\partial E_k}{\partial v_{ih}};$$

$$\theta_j \leftarrow \theta_j + \Delta\theta_j; \Delta\theta_j = -\eta \frac{\partial E_k}{\partial \theta_j};$$

$$\gamma_h \leftarrow \gamma_h + \Delta\gamma_h; \Delta\gamma_h = -\eta \frac{\partial E_k}{\partial \gamma_h};$$

标准 BP 算法只针对单个样例进行误差反向传播，参数更新较频繁，通常使用累计误差反向传播(按照样例的 **Batch**):

$$E_m = \frac{1}{m} \sum_{k=1}^m E_k;$$

**BP** 神经网络防止过拟合的措施：

- 早停：将数据集划分为训练集和验证集，训练集误差用来计算梯度、更新网络连接权值和激活函数阈值，验证集用来估算当前模型产生的验证误差，若模型在训练集上误差降低而验证集误差上升，则停止参数更新，并返回拥有最小验证集误差的参数作为最终的模型；
- 正则化：在误差目标函数中增加用于描述网络复杂程度的正则化项：

$$\min_{\omega} (\sum_{i=1}^N \text{loss}[f(x_i), y_i]) + \lambda \Omega(\omega);$$

**Tensorflow** 中的 **tf.contrib.layers.l2\_regularizer(REGULARIZATION\_RATE)**即对连接权重采用了 **L2** 正则化项；

此外，**LSTM** 的 **dropout** 方法在训练时随机丢弃多层之间的连接权重也可以有效避免过拟合：  
`lstm_cell = tf.nn.rnn_cell.DropoutWrapper (lstm_cell, output_keep_prob = KEEP_PROB)`

## 7.11 梯度下降法

梯度下降法即沿着目标函数  $J(\theta)$  的参数  $\theta$  的负梯度方向以学习速率  $\eta$  逐步更新模型参数以达到目标函数的极小值；深度学习中使用的三种梯度下降算法框架的区别在于每次学习使用的样本个数不同，从而导致学习的效果不同(正确率与运行时间)：

- **批量梯度下降 *Batch Gradient Descent***：使用全部的训练样本累积误差更新模型参数，保证能够收敛于极小值点(凸函数收敛于全局极小值点，非凸函数收敛于局部极小值点)

```
for i in range(max_epochs):
```

```
 Grads = evaluate_gradient(loss_function, dataset, params)
```

```
 Params = params - learning_rate * grads
```

批量梯度下降方法每次计算整个数据集的累计误差，学习时间过长，内存开销大，无法进行参数在线更新；

- **随机梯度下降 *Stochastic Gradient Descent***：每次从训练集中随机选择一个样本计算目标函数值  $J(\theta; x_i, y_i)$ ，沿着负梯度方向以学习速率  $\eta$  更新参数  $\theta$ ，学习速度快，并且可以在线更新参数

```
for i in range(max_epochs):
```

```
 np.random.shuffle(dataset)
```

```
 for example in dataset:
```

```
 grads = evaluate_gradient(loss_function, example, params)
```

```
 params = params - learning_rate * grads
```

**SGD** 算法的训练过程不会一直向着正确方向进行，会产生较大的波动，但是也利于算法跳出局部最优，对于凸函数来说，可能会收敛于全局最优，非凸函数会收敛于一个较好的局部最优值，**SGD** 算法收敛速度较慢，学习过程漫长；

- **小批量梯度下降 *Mini-batch Gradient Descent***：综合 **BGD** 和 **SGD** 算法的特点，每次从训练数据集中随机选择一个 **batch** 进行训练( $batch\_size < dataset\_size$ )，既保证算法的稳定性，又减小训练过程的内存开销，缩短训练过程

```
for i in range(max_epochs):
```

```
 np.random.shuffle(dataset)
```

```
 for batch in get_batches(dataset, batch_size = BATCH_SIZE):
```

```
 grads = evaluate_gradient(loss_function, batch, params)
```

```
 params = params - learning_rate * grads
```

**tensorflow** 中使用 **API** 实现的 **SGD** 算法实际上是 **Mini-batch Gradient Descent**，依靠训练数据集的 **batch** 划分实现小批量样本训练过程的随机梯度下降；

```
optimizer = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)
```

## 7.12 IIS 与牛顿法/拟牛顿法

## 7.13 局部最小与全局最小

梯度下降法是沿着负梯度方向寻找最小值,对于非凸函数可能会陷于局部最小点,甚至有可能是鞍点,无法得到期望的全局最小值,解决方法:

- 多组参数值初始化神经网络,按照标准方法训练后,取其中验证误差最小的模型作为最终的模型;
- 模拟退火:每一步训练都以一定概率接受比当前解更差的结果,但接受次优解的概率要随着训练过程的进行逐步下降,以保证算法收敛;
- 随机梯度下降: **SGD** 算法,利用单个样例训练模型引起的波动跳出局部最小;为了解决 **SGD** 算法求解非凸函数易陷于局部最优的情况,可采用以下方法:
- 在目标函数的峡谷地区,某些反向上比较陡峭, **SGD** 算法会产生较大的震荡,此时,可以采用 **Momentum** 方法对参数进行更新:

$$v_t = \gamma v_{t-1} + \Delta_{\theta} J(\theta);$$

$$\theta \leftarrow \theta - \eta v_t;$$

除目标函数的梯度以外,给参数更新式增加了动量因素,若当前更新方向与上次更新方向相同,则加速更新,若方向相反则减缓更新,从而有效避免在峡谷地区的震荡并加速收敛;

- 涅斯捷罗夫梯度加速 **NAG**: 在 **Momentum** 的基础之上,在损失函数中减去动量部分,能够预估下一次参数更新的位置

$$v_t = \gamma v_{t-1} + \nabla_{\theta} J(\theta - \gamma v_{t-1});$$

$$\theta \leftarrow \theta - \eta v_t;$$

- 自适应学习速率梯度下降 **Adagrad**: 对学习速率 $\eta$ 进行约束

$$n_t = n_{t-1} + [\nabla_{\theta} J(\theta)]^2;$$

$$\Delta \theta_t = -\frac{\eta}{\sqrt{n_t + \epsilon}} \nabla_{\theta} J(\theta); \quad \epsilon \text{ 保证分母不为 } 0$$

$$\theta \leftarrow \theta + \Delta \theta_t ;$$

对梯度 $g_t = \nabla_{\theta} J(\theta)$ 从 **1** 到 **t** 形成一个对学习率的正则化约束项 **regularizer**

$$-\frac{\eta}{\sqrt{(g_t)^2 + \epsilon}};$$

训练前期梯度较小时, **regularizer** 能够放大当前梯度,加速更新;

训练后期梯度较大时, **regularizer** 能够缩小当前梯度,减缓更新;

可以辅助早停,适合处理稀疏梯度,但是依赖于预设定的全局学习速率 $\eta$ ;

- **Adadelat** 扩展了 **Adagrad** 方法:

$$n_t = \gamma n_{t-1} + (1 - \gamma) [\nabla_{\theta} J(\theta)]^2;$$

$$\Delta \theta_t = -\frac{\eta}{\sqrt{n_t + \epsilon}} \nabla_{\theta} J(\theta);$$

继续处理如下:

$$E|g^2|_t = \rho E|g^2|_{t-1} + (1 - \rho) g_t^2;$$

$$\Delta \theta_t = -\frac{\sqrt{\sum_{r=1}^{t-1} \Delta \theta_r}}{\sqrt{E|g^2|_t + \epsilon}} ;$$

不再依赖于全局设定的学习速率 $\eta$ ,训练前期收敛较快,后期在局部会产生抖动;

- **RMSprop** 算法: 介于 **Adadelta** 和 **Adagrad** 之间  
 $E|g^2|_t = \rho E|g^2|_{t-1} + (1 - \rho)g_t^2$ ; 令  $\rho = 0.5$ , 则变成对梯度  $g_t^2$  求算数均值,  
 对  $E|g^2|_t$  再度求根式, 可得到  

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E|g^2|_t + \epsilon}}$$
; 依赖于全局学习速率  $\eta$ , 适合处理非平稳目标, 如 **RNN**  
 等;
- **Adam** 算法 **Adaptive Moment Estimator**, 带有动量项的 **RMSprop**, 经过偏置校正后, 每一次迭代学习速率都有一个确定的范围, 使得参数较平稳:  
 $m_t = \mu m_{t-1} + (1 - \mu)g_t$ ; 对  $g_t$  的一阶矩估计  
 $n_t = \rho n_{t-1} + (1 - \rho)g_t^2$ ; 对  $g_t$  的二阶矩估计  

$$\hat{m}_t = \frac{m_t}{1 - \mu^t};$$
  

$$\hat{n}_t = \frac{n_t}{1 - \rho^t};$$
  

$$\Delta\theta_t = -\frac{\hat{m}_t}{\sqrt{\hat{n}_t + \epsilon}} \eta;$$

结合了 **Adagrad** 善于处理稀疏梯度和 **RMSprop** 善于处理非平稳目标的优点, 对内存开销较小, 适用于大多数非凸函数的优化;

对于稀疏数据集, 尽量使用自适应优化方法, 无需手动调节, 最好采用默认值;  
**SGD** 训练时间较长, 但在好的初始数值和学习率调整方案的情况下, 结果更为可靠;

训练深度网络时, 使用学习率自适应优化方法;

## 7.14 支持向量机与拉格朗日乘子法

支持向量机问题就是寻找一个分类超平面  $\omega^T x + b = 0$ , 使得距离其最近的异类样本点的间隔最大, 其中距离分类超平面距离最近的点叫做支持向量, 异类支持向量使得:

$$\omega^T x_i + b = +1; y_i = +1$$

$$\omega^T x_i + b = -1; y_i = -1$$

异类支持向量的间隔成为分类间隔:  $\frac{2}{\|\omega\|}$ ; 因此求分类超平面问题变为不等式约束的最优化问题(支持向量机的基本型):

$$\min_{\omega} \frac{1}{2} \|\omega\|^2 ;$$

$$\text{s.t. } y_i(\omega^T x_i + b) \geq 1; i = 1, 2, \dots, N$$

构造拉格朗日函数:

$$L(\omega, b, \alpha) = \frac{1}{2} \|\omega\|^2 + \sum_{i=1}^N \alpha_i (-y_i(\omega^T x_i + b) + 1);$$

对模型参数  $\omega, b$  求偏导, 并令其为 0, 得到:

$$\omega = \sum_{i=1}^N \alpha_i y_i x_i ;$$

$$\sum_{i=1}^N \alpha_i y_i = 0 ;$$

带入拉格朗日函数得到：

$$L(\omega, b, \alpha) = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i x_j + \sum_{i=1}^N \alpha_i ;$$

原问题为  $\min_{\omega, b} \max_{\alpha} L(\omega, b, \alpha)$  在满足 KKT 条件的前提下，原始问题与其对

偶问题  $\max_{\alpha} \min_{\omega, b} L(\alpha, \omega, b)$  具有相同解，问题转换为：

$$\max_{\alpha} -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i x_j + \sum_{i=1}^N \alpha_i ;$$

$$\text{s.t. } \sum_{i=1}^N \alpha_i y_i = 0 ;$$

$$\alpha_i \geq 0 ; i = 1, 2, \dots, N$$

并满足 KKT 条件：

$$\alpha_i \geq 0 ; i = 1, 2, \dots, N$$

$$y_i(\omega^T x_i + b) \geq 1 ;$$

$$\alpha_i(-y_i(\omega^T x_i + b) + 1) = 0 ;$$

KKT 条件限制了非支持向量点不会出现在求和式  $\sum_{i=1}^N \alpha_i(-y_i(\omega^T x_i + b) + 1)$  中，

出现在其中的点都是支持向量，此时不等式的等号成立，不等式约束取最大值， $\alpha_i > 0$ ，即模型训练完成后，大多数样本点不需要保留，仅保留落在分类间隔上的支持向量点；

$$\max_{\alpha} -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i x_j + \sum_{i=1}^N \alpha_i ;$$

$$\text{s.t. } \sum_{i=1}^N \alpha_i y_i = 0 ;$$

$$\alpha_i \geq 0 ; i = 1, 2, \dots, N$$

原始问题的对偶问题是一个二次规划问题，经常使用 **SMO(Sequential Minimal Optimization)** 算法求解：

每次只更新两个参数而固定其他参数，利用等式约束条件有

$$\alpha_i y_i + \alpha_j y_j = \text{const} = -\sum_{k \neq i, j} \alpha_k y_k ;$$

选择两个间隔最远的样本，使得目标函数有最大更新，带入原式，就会得到一个单变量的函数求极值问题，具有闭式解；

对于偏移项 **b** 的求解：对于所有的支持向量具有  $y_s(\sum_{i \in S} \alpha_i y_i x_i^T x_s + b) = 1$ ；

**s** 是所有支持向量的下标集，更加鲁棒的结果是  $b = \frac{1}{|S|} (y_s - \sum_{i \in S} \alpha_i y_i x_i^T x_s)$ ；

**SVM** 按照其应用是线性分类问题还是非线性分类问题可分为线性支持向量机和非线性支持向量机；线性支持向量机按照其数据集是否线性可分也分为线性可分支持向量机和线性支持向量机；

线性可分支持向量机应用于数据集严格线性可分的情况下，学习目标是分类间隔最大化；

线性支持向量机应用于数据集非严格线性可分的情况下，学习目标是使得分类间隔尽量大同时保证误分类点尽量少，引入松弛变量  $\xi_i$ ， $i = 1, 2, \dots, N$ ，得到优化问题：

$$\min_{\omega, b, \xi} \frac{1}{2} \|\omega\|^2 + C \sum_{i=1}^N \xi_i ;$$

$$\begin{aligned} \text{s.t. } & y_i(\omega^T x_i + b) \geq 1 - \xi_i; i = 1, 2, \dots, N \\ & \xi_i \geq 0; \end{aligned}$$

构造拉格朗日函数，对参数 $\omega, b, \xi$ 求偏导并令其为0，并与原始问题联立，得到对偶问题描述为：

$$\max_{\alpha} -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j x_i x_j + \sum_{i=1}^N \alpha_i ;$$

$$\begin{aligned} \text{s.t. } & \sum_{i=1}^N \alpha_i y_i = 0 ; \\ & C - \alpha_i - \mu_i = 0 ; \\ & \mu_i \geq 0 ; \\ & \alpha_i \geq 0 ; i = 1, 2, \dots, N \end{aligned}$$

满足 KKT 条件：

$$\begin{aligned} & \alpha_i \geq 0 ; \mu_i \geq 0 ; \\ & y_i(\omega^T x_i + b) \geq 1 - \xi_i ; \\ & \alpha_i [y_i(\omega^T x_i + b) - (1 - \xi_i)] = 0 ; \\ & \xi_i \geq 0 ; \mu_i \xi_i = 0 ; \end{aligned}$$

称线性支持向量机算法的优化目标为**软间隔最大化**，此时，支持向量可能落在分类间隔上，对应 $\xi_i^* = 0$ ，支持向量可能落在分类间隔与分离超平面之间，对应 $0 < \xi_i^* < 1$ ；支持向量可能落在分离超平面上，对应 $\xi_i^* = 1$ ；支持向量还有可能落在分离超平面的误分类一侧，对应 $\xi_i^* > 1$ ；

**对于非线性分类问题，常采用非线性支持向量机：**

将低维空间中的不可分数据集映射到高维空间中会得到线性可分数据集，非线性映射：

$\phi(x): \mathcal{X} \rightarrow \mathcal{H}$ ；将样本集从输入空间映射到高维希尔伯特空间；

假设对于输入样例  $x_i, x_j \in \mathcal{X}$ ，存在核函数 $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$  为样例在高维空间中的内积，则可将核函数应用于非线性支持向量机问题中：

$$\max_{\alpha} -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(x_i, x_j) + \sum_{i=1}^N \alpha_i ;$$

$$\begin{aligned} \text{s.t. } & \sum_{i=1}^N \alpha_i y_i = 0 ; \\ & C \geq \alpha_i \geq 0 ; i = 1, 2, \dots, N \end{aligned}$$

得到最优解 $\alpha^* = (\alpha_1^*, \alpha_2^*, \dots, \alpha_m^*)^T$ ；选择其中一个正分量 $C \geq \alpha_j \geq 0$  对应的

样本点 $(x_j, y_j)$ ，求偏置量 $b^* = y_j - \sum_{i=1}^N \alpha_i^* y_i K(x_i, x_j)$ ；

最终得到的决策函数为：

$$f(x) = \text{sign} \left( \sum_{i=1}^N \alpha_i^* y_i K(x_i, x_j) + b^* \right)$$

当核函数 $K(x_i, x_j)$ 正定，非线性支持向量机模型为凸二次规划问题，也使用 SMO 算法求解。

## 7.15 正则化

### 7.15.1 LASSO 与 Ridge

考虑模型的优化问题

$$\min_{\omega} (\sum_{i=1}^N \text{loss}[f(x_i), y_i]) + \lambda \Omega(\omega);$$

其中  $\text{loss}[f(x_i), y_i]$  为损失函数:

- 若  $\text{loss}[f(x_i), y_i] = [f(x_i) - y_i]^2$  平方误差, 模型为最小二乘;
- 若  $\text{loss}[f(x_i), y_i] = \{0, 1 - y_i f(x_i)\}$  **hinge** 损失, 模型为支持向量机;
- 若  $\text{loss}[f(x_i), y_i] = \exp(-y_i f(x_i))$  指数损失, 模型为 **Boosting**, 基分类器

的指数损失函数为:  $\ell_{\exp}(\alpha_t h_t | D_t) = E_{x \sim D_t} [\exp(-y \alpha_t h_t(x))]$

按照分类结果与样本标签是否相同, 将指数损失函数变换为:

$$\ell_{\exp}(\alpha_t h_t | D_t) = E_{x \sim D_t} [\exp(-\alpha_t) I(y = h_t(x)) + \exp(\alpha_t) I(y \neq h_t(x))],$$

$$\ell_{\exp}(\alpha_t h_t | D_t) = \exp(-\alpha_t) P_{x \sim D_t}(y = h_t(x)) + \exp(\alpha_t) P_{x \sim D_t}(y \neq h_t(x)),$$

以  $\varepsilon_t$  表示基分类器  $h_t(x)$  的分类错误率, 则将上式表示为:

$$\ell_{\exp}(\alpha_t h_t | D_t) = \exp(-\alpha_t)(1 - \varepsilon_t) + \exp(\alpha_t)\varepsilon_t ;$$

- 若  $\text{loss}[f(x_i), y_i] = -\ln \left[ \frac{1}{1 + \exp(-f(x_i))} \right]^{y_i} \left[ 1 - \frac{1}{1 + \exp(-f(x_i))} \right]^{1-y_i}$  对数损失, 模型为 **logistic regression** ;

$\lambda \Omega(\omega)$  为正则化项,  $L_p$  范数是常用的正则化项, 参数越少, 模型越简单, 越不容易发生过拟合 (**Occam Razor**), 模型泛化能力越强:

若  $\omega = (\omega^1, \omega^2, \dots, \omega^m)$ , 则  $L_0$  范数  $\|\omega\|_0$  为其各个分量中非 0 元素的个数, 希望最终得到的模型参数向量是稀疏的, 此时模型的可解释性(直观地表示出预测输出与哪些关键特性相关)最好, 但是  $L_0$  范数非凸、不连续, 数学性质不好, 难于求解;

$L_1$  范数  $\|\omega\|_1 = \sum_{i=1}^m |\omega^i|$ , 称为稀疏规则算子, 对  $L_1$  范数的回归称为岭回归(**Ridge Regression**), 也叫做权值衰减(**Weight Decay**), 得到的模型具有稀疏性;

$L_2$  范数  $\|\omega\|_2 = \sqrt{\sum_{i=1}^m (\omega^i)^2}$ , 作为正则化项, 倾向于使得权值的分量接近于 0,

而非直接为 0, 对  $L_2$  范数的回归成为 **LASSO Regression**, 得到的模型较平滑; 系统权值矩阵的 **ill-condition number** 计算方法:

$\kappa(W) = \|W\| \cdot \|W^{-1}\|$  ; 用来衡量系统的可信度, 即当输入发生微小变化时, 输出的变化的程度, 若  $\kappa(W)$  接近于 1, 则系统是 **well-conditioned**, 距离 1 越远, 其病态就越严重;

对于系统权值向量的求解问题:

$X_{n \times m} \omega_{m \times 1} = y_{n \times 1}$  ; 为其增加一个  $L_2$  规则项, 得到问题的解:

$\omega^* = (X^T X + \lambda I)^{-1} X^T y$ , 若没有规则项引入, 解线性方程会产生不稳定解, 引入规则项会改善系统的 **condition number**;

规则项的引入, 实际上将目标函数变成了具有 $\lambda$ 强凸性质的函数:

$$f(x_0 + \tau) \geq f(x_0) + [\nabla_x f(x_0)]^T \cdot \tau + \frac{\lambda}{2} \|\tau\|^2 ;$$

而一般的凸函数定义为:

$$f(x_0 + \tau) \geq f(x_0) + [\nabla_x f(x_0)]^T \cdot \tau + o(\|\tau\|) ;$$

仅依靠 **convex** 性质不能保证在梯度下降和有限的迭代次数的情况下得到全局最优解的近似 $\omega^*$ , 若全局最优解附近比较平坦, 不停机的情况下, 可能会得到一个

距离较远的点; 因此, 引入 $L_2$  范数作为正则化项  $\frac{\lambda}{2} \|\omega\|^2$ , 将目标函数变为 $\lambda$ 强

凸, 从而拥有一个二次下界, 得到较稳定的解和较快的收敛速度;

观察 $L_1$ 和 $L_2$ 规则化的优化问题:

$$LASSO : \min_{\omega} \frac{1}{n} \|y - X\omega\|^2, \quad s.t. \|\omega\|_1 \leq C ;$$

$$Ridge : \min_{\omega} \frac{1}{n} \|y - X\omega\|^2, \quad s.t. \|\omega\|_2 \leq C ;$$

即 **LASSO** 回归使用 **L1-ball** 超矩形与等高线相交, 大多数相交点均在超矩形的角上, 会有多数权值的分量为 0, 因此会得到稀疏解;

而 **Ridge** 回归使用 **L2-ball** 超球面与等高线相交, 不易产生稀疏解, 因而模型比较平滑;

### 7.15.2 L1 正则化求解

**L1** 正则化求解使用近端梯度下降方法(**Proximal Gradient Descent**)PGD:

$$\min_{\omega} f(\omega) + \lambda \|\omega\|_1 ;$$

若 $f(\omega)$ 可导, 且 $\nabla f(\omega) = \frac{\partial f(\omega)}{\partial \omega}$ 满足 **L-Lipschitz** 条件:

$$\exists L > 0, \|\nabla f(\omega_1) - \nabla f(\omega_0)\|_2^2 \leq L \|\omega_1 - \omega_0\|_2^2 \quad (\forall \omega_1, \omega_0) ;$$

将 $f(\omega)$ 二阶泰勒展开:

$$\begin{aligned} \hat{f}(\omega) &\cong f(\omega_k) + [\nabla f(\omega_k)]^T \cdot (\omega - \omega_k) + \frac{L}{2} \|\omega - \omega_k\|_2^2 \\ &= \frac{L}{2} \left\| \omega - \left[ \omega_k - \frac{1}{L} \nabla f(\omega_k) \right] \right\|_2^2 + const; \end{aligned}$$

即可得到最优化问题:

$$\min_{\omega} \frac{L}{2} \left\| \omega - \left[ \omega_k - \frac{1}{L} \nabla f(\omega_k) \right] \right\|_2^2 + \lambda \|\omega\|_1 ;$$

令 $z = \omega_k - \frac{1}{L} \nabla f(\omega_k)$ , 可得参数更新式:

$$\omega_{k+1} = \operatorname{argmin}_{\omega} \frac{L}{2} \|\omega - z\|_2^2 + \lambda \|\omega\|_1 ;$$



## 7.16 EM 算法

期望最大化算法(**Expectation-Maximization**)算法是对含有未观测属性(隐变量)的概率模型参数的极大似然估计方法, 是一种坐标下降优化算法;

对于待解决问题 $P(X|\theta)$ , 其中 $X$ 是已观测属性,  $\theta$ 是模型参数, 此外还有未观测属性 $R$ 作为隐含变量;  $P(X|\theta)$ 是不完全数据 $X$ 的边缘概率:

$$P(X|\theta) = \sum_R P(X, R|\theta) = \sum_R P(R|\theta)P(X|R, \theta);$$

由不完全数据边缘概率的对数似然函数 $L(\theta) = \log[P(X|\theta)]$ 极大化的迭代求解可以导出完全数据的对数似然函数 $\log[P(X, R|\theta)]$ 关于在给定观测数据 $X$ 和当前参数估计 $\theta^{(t)}$ 下对未观测属性(隐含变量) $R$ 依条件概率 $P(R|X, \theta^{(t)})$ 的数学期望:

$$E_R[\log[P(X, R|\theta)] | X, \theta^{(t)}] = \sum_R \log[P(X, R|\theta)] P(R|X, \theta^{(t)});$$

称此期望为  $Q$  函数, 即有:

$$Q(\theta, \theta^{(t)}) = E_R[\log[P(X, R|\theta)] | X, \theta^{(t)}] = \sum_R \log[P(X, R|\theta)] P(R|X, \theta^{(t)});$$

可证明对数似然函数 $L(\theta)$ 是 $Q(\theta, \theta^{(t)})$ 的下界;

EM 算法分为两步进行:

- **E** 步求期望:  $Q(\theta, \theta^{(t)}) = \sum_R \log[P(X, R|\theta)] P(R|X, \theta^{(t)});$
- **M** 步最大化  $Q$  函数求参数:  $\theta^{(t+1)} = \operatorname{argmax}_{\theta} Q(\theta, \theta^{(t)});$

迭代 **E** 步和 **M** 步, 直至收敛使得  $\theta - \vartheta < \varepsilon$ ;

## 7.17 Boosting 与 Bagging

集成学习(**Ensemble Learning**)方法可以分为两种: 以 **Boosting** 为代表的串行化集成学习方法和以 **Bagging**、**Random Forest** 为代表的并行化集成学习方法;

当样本集的个体之间存在较强的依赖关系时, 使用串行化学习方法;

当样本集的个体之间不存在较强的依赖关系时, 使用并行化学习方法;

### 7.17.1 Boosting 方法

**Boosting** 对特定的数据分布进行学习, 在每一轮训练中对数据集中的样例  $x_i, i = 1, 2, \dots, N$  赋一个权重  $w_i, i = 1, 2, \dots, N$ , 在上一轮训练中正确分类的样本点的权重会降低, 而误分类点的权重会得到相应的上升, 每一轮均使用一个弱分类器  $h_t(x)$  作为基分类器对全部样本进行分类, 同时生成当前基分类器的权重  $\alpha_t, t = 1, 2, \dots, T$ ; 最终得到一个线性加性模型  $H(x) = \sum_{t=1}^T h_t(x)$  作为集成分类器;

用于二分类问题的 **AdaBoost(Adaptive Boost)**算法:

- (1) 对原始数据集中的样本分配初始权值  $w_{1i} = \frac{1}{N}$ , 并训练第一个基分类器

$h_1(x)$ ;

- (2) 迭代式：对第  $t$  ( $t=1,2,\dots,T$ ) 轮迭代，得到基分类器  $h_t(x)$ ，此时有组合分类器  $G_t(x) = \sum_{\tau=1}^t h_\tau(x)$ ，对训练数据集中样本的分类误差为  $\varepsilon_t = \sum_{i=1}^N w_{t,i} I(G_t(x_i) \neq y_i)$ ，而基分类器  $h_t(x)$  的权重为  $\alpha_t = \frac{1}{2} \ln(\frac{1-\varepsilon_t}{\varepsilon_t})$ ，由

最小化指数损失函数  $\ell_{exp}(\alpha_t h_t | D_t) = \exp(-\alpha_t)(1 - \varepsilon_t) + \exp(\alpha_t)\varepsilon_t$  求得；

更新样本集中的样例权重  $w_{t+1,i} = \frac{w_{t,i}}{Z_t} \exp(-\alpha_t y_i G_t(x_i))$  ;  $i = 1, 2, \dots, N$

- (3) 最终得到分类器  $G_T(x) = \sum_{t=1}^T h_t(x)$  ;

常采用的基分类器有决策树桩等；

基于梯度下降的梯度提升 (**Gradient Boosting**) 算法：与 **AdaBoost** 不同，**GB** 算法是对损失函数 **Loss Function** 的负梯度的拟合，其中，

负梯度  $r_{t,i} = - \left[ \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} \right] |_{G(x)=G_{t-1}(x)}$  又称为伪残差，是回归问题中的残差

$r_{t,i} = y_i - G_{t-1}(x_i)$  推广到分类、排序等一般性问题中的近似；

**GBDT** 算法的一般过程：

- (1) 初始化基学习器为常数：  $G_0(x) = \gamma_0 h_0(x)$  ;  $h_0(x) = 1$  ;

$$\gamma_0 = \arg \min_{\gamma} \sum_{i=1}^N \ell(y_i, \gamma);$$

- (2) 迭代式：对于第  $t$  ( $t=1,2,\dots,T$ ) 步，计算伪残差：

$$r_{t,i} = - \left[ \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} \right] |_{G(x)=G_{t-1}(x)}$$

拟合伪残差序列  $\{(x_i, r_{t,i})\}_{i=1}^N$  学习一个基分类器  $h_t(x)$ ，并求其权重：

$$\gamma_t = \arg \min_{\gamma} \sum_{i=1}^N \ell(y_i, G_{t-1}(x_i) + \gamma h_t(x_i));$$

更新模型  $G_t(x) = G_{t-1}(x) + \gamma_t h_t(x)$  ;

- (3) 最终得到线性加性分类器模型：  $G(x) = \sum_{t=0}^T \gamma_t h_t(x)$  ;

使用决策树作为基分类器的 **GBDT(Gradient Boosting Decision Tree)** 算法：同 **GB** 算法，其主要的关注问题变为，使用 **CART** 拟合伪残差序列，经常使用深度 **depth<5**，叶节点数 **leafnodes<10** 的决策树作为基学习器，此时衡量分类效果好坏的标准不再是 Gini 指数，而使用均方误差，因而，此时的决策树是一棵最小二乘树，最小二乘树生成过程中切分变量及切分点的选择：

$$\min_{j,s} [\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2];$$

根据选定的切分特征  $x^{(j)}$  及其切分点取值  $s$ ，将数据集划分为两个子区域：

$$R_1(j, s) = \{x | x^{(j)} \leq s\};$$

$$R_2(j, s) = \{x | x^{(j)} > s\};$$

而对于这两个子区域中样本标签的估计为：

$$\hat{c}_1 = \frac{1}{|R_1|} \sum_{x_i \in R_1(j, s)} y_i;$$

$$\hat{c}_2 = \frac{1}{|R_2|} \sum_{x_i \in R_2(j, s)} y_i;$$

最小二乘回归树生成算法的停机条件：

- (1) 达到最大迭代次数；
- (2) 叶子节点中拥有单一的输出；
- (3) 特征空间中没有更多的特征供切分；

最终生成的回归树模型将数据集切分为  $M$  个子区域，表示为：

$$h_t(x) = \sum_{m=1}^M \hat{c}_m I(x \in R_m); t = 1, 2, \dots, T$$

再将以上得到的最小二乘回归树应用于 GB 算法中，就得到了 GBDT；

除了使用 **CART** 还可以使用线性分类器作为基学习器的 **XgBoost** 算法：**XgBoost** 与 **GBDT** 算法步骤相同，都是先初始化一个常数作为基分类器，不同点在于 **GBDT**

算法使用一阶偏导数  $r_{t,i} = - \left[ \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} \right] |_{G(x)=G_{t-1}(x)}$ ,  $i = 1, 2, \dots, N$  生成伪残差序列，而 **XgBoost** 算法使用一阶偏导数和二阶偏导数生成伪残差：

$$r_{t,i} = - \left( \left[ \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} \right] |_{G(x)=G_{t-1}(x)} f_t(x_i) + \frac{1}{2} \left[ \frac{\partial^2 \ell(y_i, G(x_i))}{\partial G^2(x_i)} \right] |_{G(x)=G_{t-1}(x)} f_t^2(x_i) \right), i = 1, 2, \dots, N$$

给损失函数增加一个正则化项描述决策树的复杂程度：

$$\mathcal{L}_t(y, G(x)) = \sum_{i=1}^N \ell(y_i, G_{t-1}(x_i) + f_t(x_i)) + \sum_{t=1}^t \Omega(f_t);$$

在第  $t$  步迭代中，拟合残差得到的最优决策树  $f_t(x)$  就是使得目标函数最小的决策树，则损失函数可以表示成：

$$\mathcal{L}_t(y, G(x)) = \sum_{i=1}^N \ell(y_i, G_{t-1}(x_i) + f_t(x_i)) + \Omega(f_t) + \text{const};$$

其中 **const** 表示已获得的  $t-1$  棵决策树的正则化损失为常数，将损失函数进行二阶泰勒展开：

$$\begin{aligned} \mathcal{L}_t(y, G(x)) &= \sum_{i=1}^N \left[ \ell(y_i, G_{t-1}(x_i)) + \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} |_{G(x)=G_{t-1}(x)} f_t(x_i) \right. \\ &\quad \left. + \frac{1}{2} \frac{\partial^2 \ell(y_i, G(x_i))}{\partial G^2(x_i)} |_{G(x)=G_{t-1}(x)} f_t^2(x_i) \right] + \Omega(f_t) + \text{const} \end{aligned}$$

对于第  $t$  轮迭代计算当前最有决策树的过程来说， $\ell(y_i, G_{t-1}(x_i))$  也是常数项，

最小化损失函数不需要考虑常数项，因此将目标损失函数重新写成：

$$L_t(y, G(x)) = \sum_{i=1}^N \left[ \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} \Big|_{G(x)=G_{t-1}(x)} f_t(x_i) + \frac{1}{2} \frac{\partial^2 \ell(y_i, G(x_i))}{\partial G^2(x_i)} \Big|_{G(x)=G_{t-1}(x)} f_t^2(x_i) \right] + \Omega(f_t)$$

将每一次迭代得到的决策树  $f_t(x)$  表示为：

$$f_t(x) = w_{q(x)}; w \in \mathcal{R}^M, q(x): \mathcal{R}^d \rightarrow \{1, 2, \dots, M\};$$

$M$  代表决策树的叶子节点个数， $M$  个叶子节点上的输出值组成决策树  $f_t(x)$  的输出空间  $\mathcal{R}^M$ ， $q(x)$  代表将样本从原始特征空间  $\mathcal{R}^d$  到决策树输出空间的映射， $w$  是一个  $M$  维向量， $w_{q(x)}$  是对样本的预测值，因此 **XgBoost** 算法使用了正则化项：

$$\Omega(f_t) = \gamma M + \frac{\lambda}{2} \sum_{j=1}^M w_j^2;$$

$$\text{定义 } g(x_i) = \frac{\partial \ell(y_i, G(x_i))}{\partial G(x_i)} \Big|_{G(x)=G_{t-1}(x)}, \quad h(x_i) = \frac{\partial^2 \ell(y_i, G(x_i))}{\partial G^2(x_i)} \Big|_{G(x)=G_{t-1}(x)};$$

则有：

$$L_t(y, G(x)) = \sum_{i=1}^N [g(x_i) w_{q(x_i)} + \frac{1}{2} h(x_i) w_{q(x_i)}^2] + \gamma M + \frac{\lambda}{2} \sum_{j=1}^M w_j^2;$$

决策树  $f_t(x) = w_{q(x)}$  的作用是将原始数据集中的  $N$  个样例划分到  $M$  个叶子节点

中去，因此可以将  $\sum_{i=1}^N [g(x_i) w_{q(x_i)} + \frac{1}{2} h(x_i) w_{q(x_i)}^2]$  改写成按照叶子节点求和，

因此得到变形：

$$L_t(y, G(x)) = \sum_{j=1}^M [(\sum_{x_i \in I_j} g(x_i)) w_j + \frac{1}{2} (\lambda + \sum_{x_i \in I_j} h(x_i)) w_j^2] + \gamma M;$$

得到一个简单的二次式，可求解最优解：

$$w_j^* = - \frac{\sum_{x_i \in I_j} g(x_i)}{\lambda + \sum_{x_i \in I_j} h(x_i)};$$

$$L_t^* = - \frac{1}{2} \sum_{j=1}^M \frac{[\sum_{x_i \in I_j} g(x_i)]^2}{\lambda + \sum_{x_i \in I_j} h(x_i)} + \gamma M;$$

即第  $t$  次迭代得到的最优决策树仅与决策树的结构  $q(x)$  相关，而与叶子节点的取值无关，即此时选择决策树只涉及到 **特征(key)** 的选择，还没有涉及 **切分点(value)**，

以上作为决策树生成过程的特征选择过程，即划分子区域(叶子节点)  $I_j, j =$

$1, 2, \dots, M$ ；而特征切分点的选择需要考察增益值 **Gain**，**Gain** 的意义是指当前节点被切分后的损失函数的减少，此时将带切分节点作为根节点，切分后得到其左右子树，考察其  $L_t^*$  函数：

$$E_t = -\frac{1}{2} \frac{[\sum_{x_i \in I} g(x_i)]^2}{\lambda + \sum_{x_i \in I} h(x_i)} + \gamma - (-\frac{1}{2} \sum_{j=1}^2 \frac{[\sum_{x_i \in I_j} g(x_i)]^2}{\lambda + \sum_{x_i \in I_j} h(x_i)} + 2\gamma)$$

$E_t$ 越大，表明切分效果越好，选择使 $E_t$ 取值最大的样本标签作为切分点；  
 重复上述过程，先选择切分特征，得到最好的决策树结构，再选择切分点，逐步得到最优的决策树；其中的 $\gamma$ 参数用来控制决策树的复杂度，在生成决策树的过程中就考虑了过拟合问题，因此后续不需要剪枝处理；  
 此外，在每一步迭代中，生成决策树 $f_t(x)$ 使用到的 $g(x_i)$ 和 $h(x_i)$ 两个梯度都可以对每一个样本分开计算，使用并行化处理可以加速训练过程；  
 以上 **Boosting** 算法主要关注降低模型的偏差；

### 7.17.2 Bagging 方法

**Bagging (Bootstrap Aggregating)** 使用自助采样法(有放回)从数据集中抽取  $T$  个样本容量为  $N$  的子集作为训练数据集，此时原始数据集中含有 36.8% 的样本没有被抽到，称为包外估计，基于每个子集训练基分类器(常见于不剪枝决策树，神经网络等)最终  $T$  个分类器以投票法决定测试样本的分类结果，以样本  $x$  的包外预测衡量算法的分类错误率：

$$H^{oop}(x) = \arg \max_{y \in \mathbb{Y}} \sum_{t=1}^T I(h_t(x) = y) \cdot I(x \notin D_t);$$

$$\varepsilon^{oop} = \frac{1}{|D|} \sum_{(x,y) \in D} I(H^{oop}(x) \neq y);$$

扩展算法为**随机森林(Random Forest)**：RF 算法不仅沿用了 **Bagging** 算法在训练数据集生成过程中的样本扰动(**Bootstrap Sampling**)还增加了属性扰动：

从样本集  $D$  的属性空间  $C_d$  中随机选择  $k$  个属性，得到属性子空间  $C_k$ ，用于基决策树的生成  $|C_k| \leq |C_d|$ ，**RF** 组合基决策树得到最终的分类模型，通过增加个体学习器之间的差异提升最终模型的泛化能力；

**Bagging** 算法关注降低由于样本扰动带来的方差；

## 7.18 监督学习与无监督学习

按照训练数据集中的样例是否具有标签，将学习任务分成监督学习和无监督学习两类，其中样本的标签包含了其所归属的分类(**category**)信息；最常见的无监督学习就是**聚类**；

监督学习的问题分为三大类：

- 分类问题：  $P(Y|X; \theta)$ ；样例的标签信息即输出变量  $Y$  的取值为有限个离散数值时，对样例标签的预测问题即分类问题，此时样例的属性信息即输入变量  $X$  可以是离散的(**SVM, Decision Tree, Naïve Bayes, EM, KNN** 等)，也可以是连续的(**Logistic Regression**)；
- 标注问题：  $P(Y^1, Y^2, \dots, Y^T | X^1, X^2, \dots, X^T; \theta)$ ；作为分类问题的推广，标注问题的输入  $X = (X^1, X^2, \dots, X^T)$  是一个观测序列，输出  $y = (Y^1, Y^2, \dots, Y^T)$  是一个标记序列或状态序列，标注问题的学习目标就是找到一个模型，能够对给定的观测序列预测其对应的状态序列；

- 回归问题:  $\hat{y} = f(x; \omega)$ ; 预测输入变量  $x$  与输出变量  $y$  之间的关系, 回归问题的学习等价于函数拟合;

## 7.19 聚类

聚类算法是典型的无监督学习算法, 在缺少样本标签信息的情况下, 借助样本的特征信息或其他信息将其划分到若干个不相交的子集中去;

- 原型聚类: 原型聚类的意义在于寻找样本空间中具有代表性的点完成类别划分; 常使用 **K-Means** 聚类算法, 将样本空间划分成  $K$  个簇, 学习目标为最小化均方误差:

$$\ell(x; C) = \sum_{k=1}^K \sum_{x_i \in C_k} \|x_i - \mu_k\|_2^2;$$

其中,  $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i$  称为均值向量;

(1) 从数据集  $D$  中随机挑选  $k$  个样本作为初始均值向量  $\{\mu_1, \mu_2, \dots, \mu_K\}$ ;

(2) 迭代式: 置  $C_k = \emptyset$ ,  $k = 1, 2, \dots, K$ ;

**for**  $i = 1, 2, \dots, N$ ; 对每一个样本计算其与各个均值向量之间的距离

$$d_{i,k} = \|x_i - \mu_k\|_2; \quad k = 1, 2, \dots, K;$$

则样本  $x_i$  的簇标记为  $\lambda(x_i) = \arg \min_k d_{i,k}$ , 将样本划入对应簇:

$$C_{\lambda(x_i)} = C_{\lambda(x_i)} \cup \{x_i\};$$

遍历结束后, 更新各个簇的均值向量  $\mu_k = \frac{1}{|C_k|} \sum_{x_i \in C_k} x_i; k = 1, 2, \dots, K$

(3) 重复上述过程, 直至均值向量不再更新或达到最大迭代次数;

- 密度聚类: **DBSCAN** 算法, 基于样本点的密度分布进行聚类, 选择数据集中的核心对象作为种子, 由此出发确定相应的聚类簇:

(1) 初始化核心对象集合  $\Omega = \emptyset$ , 从原始数据集中挑选出邻域内样本点数量大于 **MinPts** 的样例作为核心对象, 并加入核心对象集合;

(2) 寻找所有核心对象的密度可达的样本点, 通过核心对象将密度相连的样本点聚集成簇;

初始化聚类簇数  $k=0$ , 未访问样本集合  $\Gamma = D$ ;

**while**  $\Omega \neq \emptyset$  对核心对象集合进行遍历:

备份未访问样本集合  $\Gamma_b = \Gamma$ ;

随机选取核心对象点并初始化待聚类队列  $o \in \Omega, Q = \langle o \rangle, \Gamma = \Gamma \setminus \{o\}$ ;

**while**  $Q \neq \emptyset$ :

$q = Q.head$ ; 取出队列的 **head** 元素;

若  $q$  是一个核心对象, 即  $|N_\epsilon(q)| > \text{MinPts}$ , 则将其邻域中未被访问过

得样本点均加入队列，并将其从  $\Gamma$  中剔除：

$Q.append(N_\epsilon(q) \cap \Gamma); \Gamma = \Gamma \setminus (N_\epsilon(q) \cap \Gamma);$

直到队列  $Q$  为空停止，生成聚类簇  $C_k = \Gamma_b \setminus \Gamma$ ，更新核心对象集合

$\Omega = \Omega \setminus C_k;$

$k = k + 1;$

核心对象  $o$  出发得到其临域内所有样本点，这些样本点中有的核心对象，有的不是核心对象，从这些核心对象出发继续访问其临域...，直到将所有从  $o$  出发密度可达的点纳入一个聚类簇中  $C_k$ ，同时将簇中的元素从未访问集合中剔除，并从核心对象集合中剔除，从而得到  $K$  个聚类簇划分；

- 层次聚类：在不同的层次对样例进行聚类，典型应用 **AGNES** 算法，自底向上层次聚类，初始时将所有样本点看做一个单点簇，两两求其距离度量(最大距离，最小距离，平均距离)，合并距离最近的簇并重新为聚类簇编号，重复进行以上过程，直至达到预设的聚类簇数量为止；常用的集合间距离度量：

最小距离： $d_{min}(C_i, C_j) = \min_{x \in C_i, y \in C_j} \|x - z\|_p$

最大距离： $d_{max}(C_i, C_j) = \max_{x \in C_i, y \in C_j} \|x - z\|_p$

平均距离： $d_{avg}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{z \in C_j} \|x - z\|_p$

## 7.20 特征选择与稀疏学习

### 7.20.1 维度灾难

给定数据集  $\{(x_i, y_i)\}_{i=1}^N$ ，其中  $m$  维行向量  $x_i = (x_i^{(1)}, x_i^{(2)}, \dots, x_i^{(m)})$  定义了样本  $i$  的

属性信息，包含  $m$  个特征， $x_i^{(l)} \in \mathcal{R}_l$ ， $\mathcal{R}_l$  为第  $l$  个特征的取值空间 ( $l = 1, 2, \dots, m$ )，

则分类器要完成所有特征的学习并找出其与样本标签  $y_i$  的关系，至少需要  $\prod_{l=1}^m |\mathcal{R}_l|$  个样本，当  $m$  数值较大时， $\prod_{l=1}^m |\mathcal{R}_l|$  数值极大，即出现了组合爆炸问题，而实际给定的数据集不可能完全覆盖所有可能，称其为样本稀疏问题，这种由特征维度过大带来的组合爆炸与样本稀疏问题统称为维度灾难，维度灾难还会导致模型过拟合；

### 7.20.2 特征选择

在数据预处理过程中，保留样本的相关特征 (**Relevant Feature**)，去除无关特征 (**Irrelevant Feature**) 的过程叫做特征选择，即从原始特征集  $A = \{a_i\}_{i=1}^m$  中选择特征子集并进行评价：

- 子集搜索与子集评价：贪心策略，选择特征  $A_s = \{a_1\}$  作为初始特征子集，逐步选择候选特征  $a_l$  for  $l \text{ in range}(m)$ ，在已划分子集  $D_s$  ( $D = \cup_s D_s$ ) 上计算以新增特征  $a_l$  进行划分的信息增益  $g(D_s | a_l) = H(D_s) - \sum_v^{|\mathcal{R}_l|} \frac{|D_s^{(l),v}|}{|D_s|} H(D_s^{(l),v})$ ，若  $g(D_s | a_l) > 0$  则在当前节点上以候选特征  $a_l$  进行划分得到子集  $\{D_s^{(l),v}\}_v^{|\mathcal{R}_l|}$ ，并更新特征子集  $A_s = A_s \cup \{a_l\}$  和划分子集  $D_s = \{D_s^{(l),v}\}_v^{|\mathcal{R}_l|}$ ，进入下一轮更新；
- 类似于决策树的生成策略，除信息增益之外，还可以选择信息增益比和基尼指数作为子集评价标准；
- 特征选择方法：
    - 过滤式选择方法：先对数据集进行特征选择 (**Filter**) 再训练学习器，二者分开进行；
    - 包裹式选择方法：将最终要得到的学习器的性能衡量作为特征选择的评价标准；
    - 嵌入式选择方法：目标函数求解时加入  $L_1$  正则化，易于得到稀疏解，仅选择权值向量  $\omega$  的非零分量对应的特征进行学习，自动实现特征选择；

## 7.21 PCA

### 7.21.1 特征值分解

特征值分解 (**Eigenvalue Decomposition**) 只能对方阵进行：

$$Av = \lambda v;$$

求解特征值  $|\lambda I - A| = 0$ ，通过初等行变换和列变换求解特征值，例如：

$$A = \begin{bmatrix} 1 & -3 & 3 \\ 3 & -5 & 3 \\ 6 & -6 & 4 \end{bmatrix};$$

$$|\lambda I - A| = \begin{vmatrix} \lambda - 1 & 3 & -3 \\ -3 & \lambda + 5 & -3 \\ -6 & 6 & \lambda - 4 \end{vmatrix} \xrightarrow{c1+c2} \begin{vmatrix} \lambda + 2 & 3 & -3 \\ \lambda + 2 & \lambda + 5 & -3 \\ 0 & 6 & \lambda - 4 \end{vmatrix} \rightarrow (\lambda +$$

$$2) \begin{vmatrix} 1 & 3 & -3 \\ 1 & \lambda + 5 & -3 \\ 0 & 6 & \lambda - 4 \end{vmatrix} \xrightarrow{r2-r1} (\lambda + 2) \begin{vmatrix} 1 & 3 & -3 \\ 0 & \lambda + 2 & 0 \\ 0 & 6 & \lambda - 4 \end{vmatrix} \rightarrow (\lambda + 2)^2 \begin{vmatrix} 1 & 3 & -3 \\ 0 & 1 & 0 \\ 0 & 6 & \lambda - 4 \end{vmatrix}$$

$$\xrightarrow{r3-6*r2} (\lambda + 2)^2 \begin{vmatrix} 1 & 3 & -3 \\ 0 & 1 & 0 \\ 0 & 0 & \lambda - 4 \end{vmatrix} = (\lambda + 2)^2 (\lambda - 4) = 0;$$

得到矩阵  $A$  的特征值为  $\lambda_1 = \lambda_2 = -2$ ， $\lambda_3 = 4$ ；

解线性方程组  $(\lambda E - A)v = 0$ ，得到二重根  $\lambda = -2$  对应的基础解系为：



$$\xi_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad \xi_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix};$$

$$\lambda=4 \text{ 对应的特征向量为 } \xi_3 = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix};$$

特征值分解可以将矩阵写成  $A = Q\Sigma Q^{-1}$ , 其中  $\Sigma = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_d)$  是由矩阵  $A$  的特征值组成的对角阵,  $Q = [v_1, v_2, \dots, v_d]$  是对应的特征向量组成的矩阵;

可以将矩阵看做对向量的线性变换, 则矩阵的特征向量代表变换的方向, 而特征值就代表变换方向的主次重要程度, 要想描述一个线性变化, 只需要描述主要的变换方向即可;

在 **Python** 中使用 **numpy** 进行矩阵特征值分解运算:

```
import numpy as np
A = np.array([[1,-3,3],[3,-5,3],[6,-6,4]])
d, v = np.linalg.eig(A) # d 是特征值, v 是特征向量组成的矩阵
>>d
array([4.,-2.,-2.])
>>v
array([[...,...,...],[...,...,...],[...,...,...]])
X = np.diag(d)
>>X
array([[4.,0.,0.],
 [0.,-2.,0.],
 [0.,0.,-2.]])
X = np.mat(X) #将 array 形式转换成 matrix
v = np.mat(v)
v*X*(v.I) # v.I 求其逆矩阵
>> matrix([[1.,-3.,3.],
 [3.,-5.,3.],
 [6.,-6.,4.]])
```

### 7.21.2 奇异值分解

不同于特征值分解只能对方阵提取其特征, 奇异值分解 (**Singular Value Decomposition**) **SVD** 适用于任意矩阵:

$$A_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T ;$$

$\Sigma_{m \times n}$  矩阵中对角线上的元素是矩阵  $A_{m \times n}$  的奇异值, 除对角线意外其他元素均为 0, 方阵  $U_{m \times m}$  中的向量两两正交, 称为左奇异向量,  $V_{n \times n}$  中的向量也是两两正交, 称为右奇异向量;

奇异值分解通过构造方阵并进行特征值分解完成:

$$(A^T A) v_i = \lambda_i v_i$$

得到其奇异值为  $\sigma_i = \sqrt{\lambda_i}$ , 对应右奇异向量为  $v_i$ , 对应左奇异向量为

$$u_i = \frac{1}{\sigma_i} A v_i ;$$

在 **Python** 中对矩阵进行奇异值分解:

```
import numpy as np
A = np.array([[1,1,1,0,0],
 [2,2,2,0,0],
 [3,3,3,0,0],
 [5,5,3,2,2],
 [0,0,0,3,3],
 [0,0,0,3,3],
 [0,0,0,6,6]])
u, sigma, v_t = np.linalg.svd(A)
>>u
matrix([[..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...]])
>>sigma
array([10.98, 8.79, 1.04, 1.18e-15, 2.13e-32])
>>v_t #右奇异矩阵的转置
matrix([[..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...],
 [..., ..., ..., ..., ...]])
```

在矩阵  $\Sigma_{m \times n}$  中奇异值从大到小排列, 且具有较快的衰减速度, 通常前面 **10%** 的奇异值之和能够占到全部奇异值之和(矩阵的全部特征)的 **99%** 以上, 即可以使用这 **10%** 的奇异值近似描述矩阵:

$$A_{m \times n} \cong U_{m \times r} \Sigma_{r \times r} V_{r \times n}^T, r \ll n ;$$

要存储一个较大的矩阵  $A_{m \times n}$ , 通过 **SVD** 运算只需要存储得到的三个较小的矩阵  $U_{m \times r}$ ,  $\Sigma_{r \times r}$ ,  $V_{r \times n}^T$  即可, 极大地减少了存储空间;

### 7.21.3 主成分分析

主成分分析(**Principal Component Analysis**)**PCA** 的作用是将给定数据集  $D$

$$D = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_N^{(1)} & \cdots & x_N^{(m)} \end{bmatrix}_{N \times m}$$

通过线性变换  $P_{m \times m}$  转换成  $\tilde{D}$ :  $\tilde{D}_{N \times m} = D_{N \times m} P_{m \times m}$ , 使得数据集在新线性空间中的各个正交坐标系中拥有最大的方差, 线性变换  $P_{m \times m}$  就是这样的一组正交坐标系, 第 1 个坐标轴具有最大的方差, 第 2 个坐标轴与第 1 个坐标轴正交, 且拥

有最大的方差.....

样本属性数据较大的方差利于模型的训练,降低模型偏差提高泛化能力,而主导方差的方向才是数据的有效特征,要想获得数据的有效特征,选择前面  $r$  ( $r < m$ ) 个坐标轴对原始数据集  $D$  进行变换:  $\tilde{D}_{N \times r} = D_{N \times m} P_{m \times r}$ , 可得到主导样本方差的  $r$  个属性即数据集的主成分;

通过变换  $P_{m \times r}$  后新得到的  $\tilde{D}_{N \times r}$  在样本的属性方向上进行了压缩,忽略了方差较小的  $m-r$  个属性(噪声),这一线性变换的过程可借助 **SVD** 实现:

$$\text{SVD: } D_{N \times m} \cong U_{N \times r} \Sigma_{r \times r} V_{r \times m}^T ;$$

$$\text{令: } P_{m \times r} = V_{m \times r} ;$$

$$D_{N \times m} P_{m \times r} \cong U_{N \times r} \Sigma_{r \times r} (V_{r \times m}^T V_{m \times r});$$

$$\text{其中 } V_{r \times m}^T V_{m \times r} = I_{r \times r} ;$$

$$\text{即有 } D_{N \times m} P_{m \times r} \cong U_{N \times r} \Sigma_{r \times r} ;$$

即: 通过将原始数据集组成的矩阵  $D$  (行索引代表样本, 列索引代表属性) 进行奇异值分解得到其全部奇异值并排序  $\sigma_1 > \sigma_2 > \sigma_3 > \dots > \sigma_d$ , 从中选择前面  $r$  个

奇异值, 使得  $\frac{\sum_{j=1}^r \sigma_j}{\sum_{j=1}^d \sigma_j} > 99\%$ , 以  $\{\sigma_1, \sigma_2, \dots, \sigma_r\}$  对应的左奇异向量  $\{u_j = \frac{1}{\sigma_j} D v_j\}_{j=1}^r$

构成变换矩阵  $U_{N \times r} = (u_1, u_2, \dots, u_r)$ , 并生成对角矩阵:

$$\Sigma_{r \times r} = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_r);$$

进而得到对原数据集的主成分抽取:

$$D_{N \times m} P_{m \times r} \cong U_{N \times r} \Sigma_{r \times r} ;$$

类似于对属性的主成分提取, 若原始数据集中存在过多相似的样本, 要将其压缩, 则可对矩阵  $D_{N \times m}$  进行行压缩:

$$Q_{r \times N} D_{N \times m} = \tilde{D}_{N \times r} ;$$

$$\text{观察 SVD: } D_{N \times m} \cong U_{N \times r} \Sigma_{r \times r} V_{r \times m}^T ;$$

$$\text{令 } Q_{r \times N} = U_{r \times N}^T \text{ 得到:}$$

$$Q_{r \times N} D_{N \times m} \cong (U_{r \times N}^T U_{N \times r}) \Sigma_{r \times r} V_{r \times m}^T, \text{ 其中: } U_{r \times N}^T U_{N \times r} = I_{r \times r} ;$$

$$\text{则有 } Q_{r \times N} D_{N \times m} \cong \Sigma_{r \times r} V_{r \times m}^T ;$$

从矩阵  $D_{N \times m}$  的奇异值中选取前  $r$  个后, 以其对应的右奇异向量  $\{v_j\}_{j=1}^r$  构成矩阵

$$\text{并转置得到变换矩阵 } V_{r \times m}^T = (v_1, v_2, \dots, v_r)^T ;$$

对角矩阵  $\Sigma_{r \times r} = \text{diag}(\sigma_1, \sigma_2, \dots, \sigma_r)$ , 进而完成对原始数据集矩阵的样本提取(行压缩);

## 7.22 Apriori 算法与 FP-Growth

### 7.22.1 关联规则

有事件  $A$  和  $B$ , 关联规则  $A \rightarrow B$  的支持度:  $\text{support}(A \rightarrow B) = P(AB)$ ;

关联规则  $A \rightarrow B$  的置信度:  $\text{confidence}(A \rightarrow B) = P(B|A) = \frac{P(AB)}{P(A)}$  ;

满足最小支持度阈值和最小置信度阈值要求的规则为强规则:

$support(A \rightarrow B) > \alpha$ ,  $confidence(A \rightarrow B) > \beta$ , 此时称  $A \rightarrow B$  存在强关联;

## 7.22.2 Apriori 算法

## 7.23 词频-逆文本词频

**TF-IDF**

## 7.24 概率图模型

概率模型的框架下, 以  $X$  表示观测变量(输入), 以  $Y$  表示预测变量(输出), 以  $R$  表示未观测的隐含变量, 判别模型直接计算条件概率  $P(Y, R|X)$ , 生成模型通过  $P(Y, R, X)$  得到条件概率  $P(Y, R|X)$ , 都是为了解决推断问题得到观测数据下对预测变量的条件概率分布  $P(Y|X)$ ;

其中, 判别模型直接考虑条件概率  $P(Y|X) = \sum_R P(Y, R|X)$ ;

生成模型无法直接获得条件概率, 通过贝叶斯公式得到

$$P(Y, R|X) = \frac{P(Y, R, X)}{P(X)} = \frac{P(X, R|Y)P(Y)}{P(X)};$$

在概率框架中, **EM** 算法可以解决含有隐变量的极大似然估计问题  $P(X|\theta)$ , 但是通过概率求和消去隐变量  $R$  会遭遇组合爆炸和样本稀疏问题, 算法复杂度为

$O(|\mathcal{X}|^{|\mathcal{X}|}|\mathcal{R}|^{|\mathcal{R}|})$ , 其中  $\mathcal{X}$  为样本观测特征取值空间,  $\mathcal{R}$  为样本未观测特征取值空

间, 此外样本的属性之间可能存在较强的耦合关系, 因此基于训练样本集预测条件概率分布  $P(Y|X)$  较困难;

概率图模型使用有向图或无向图描述属性之间的相关关系, 对概率图建模解决推断问题; 若变量间存在显式的因果关系, 使用有向图模型(贝叶斯网络)进行描述, 若已知变量间存在相关性, 但是难于使用显式因果关系描述, 此时使用概率无向图模型(马尔科夫网络)进行描述;

**HMM 模型与 CRF 模型对比:**

- **HMM** 模型是有向图生成模型, 给定模型  $\lambda = (A, B, \pi)$  后, 生成状态序列  $R = (r_1, r_2, \dots, r_T)$ , 再由状态序列中的每一时刻状态生成对应的输出观测:  $\lambda \rightarrow r_t \rightarrow x_t, t = 1, 2, \dots, T$ ; 即输出观测(标记)仅与其所处时刻的状态有关, 而此时此刻的状态仅与其前一时刻的状态有关, **HMM** 沿着时序方向进行;
- **CRF** 模型是概率无向图判别模型, 给定输入状态序列  $X = (X_1, X_2, \dots, X_n)$  求其对应的输出标记序列  $Y = (Y_1, Y_2, \dots, Y_n)$  的条件概率  $P(Y|X)$ , 而标记  $Y_i$  不仅与整个输入观测序列  $X = (X_1, X_2, \dots, X_n)$  有关, 还与其它标记  $Y_{j \neq i}$  有关, 当 **CRF**

为线性链式结构时:

$$P(Y_i|X, Y_1, \dots, Y_{i-1}, Y_{i+1}, \dots, Y_n) = P(Y_i|X, Y_{i-1}, Y_{i+1}), i = 1, 2, \dots, n;$$

### 7.24.1 隐马尔科夫模型

隐马尔科夫模型(**Hidden Markov Model**)HMM 是一种有向图模型(动态贝叶斯网络),用于解决标注问题,作用在时序上,由马尔科夫链随机生成不可观测的状态序列 $R = (r_1, r_2, \dots, r_T)$ ,再由这个生成的状态序列产生对应的(每一个时刻的状态产生一个对应的观测)观测序列 $X = (x_1, x_2, \dots, x_T)$ ;此时,模型的不可观测

状态变量 $r_t \in S$ , ( $t = 1, 2, \dots, T$ )与输出观测变量 $x_t \in \mathbb{O}$ , ( $t = 1, 2, \dots, T$ )不再是分类问题中样本的特征与标记,而是在特定的状态空间和观测空间中的取值,其中,状态空间 $S = \{s_1, s_2, \dots, s_N\}$ ,观测空间 $\mathbb{O} = \{o_1, o_2, \dots, o_M\}$ ,HMM 中的马尔科夫链描述了变量之间的依赖关系,即某一时刻  $t$  ( $t=1,2,\dots,T$ ),观测变量 $x_t$ 仅由当前时刻的状态变量 $r_t$ 决定,而状态变量 $r_t$ 仅与前一时刻的状态变量 $r_{t-1}$ 有关,整个时序中的变量联合概率分布为:

$$P(r_1, x_1, r_2, x_2, \dots, r_T, x_T) = P(x_1|r_1)P(r_1) \prod_{t=2}^T P(x_t|r_t)P(r_t|r_{t-1});$$

HMM 由参数 $\lambda = (A, B, \pi)$ 确定,其中 $\pi$ 为初始状态概率向量,表示 HMM 在初始时刻在状态空间中的概率分布; $A$ 为状态转移概率矩阵,表示状态空间中各个取值之间进行转移的概率分布; $B$ 为观测输出概率矩阵,表示由状态空间中取值产生观测空间中取值的概率分布;

$$\pi = (\pi_i)_{N \times 1}; \pi_i = P(r_1 = s_i), i = 1, 2, \dots, N;$$

$A = [a_{ij}]_{N \times N}; a_{ij} = P(r_t = s_j | r_{t-1} = s_i), i, j = 1, 2, \dots, N$ ; 此处的  $t-1$  与  $t$  不是变量,仅代表从上一时刻到当前时刻的状态转移关系,与时序无关;

$B = [b_{jl}]_{N \times M}; b_{jl} = P(x_t = o_l | r_t = s_j), j = 1, 2, \dots, N; l = 1, 2, \dots, M$ ; 此处的  $t$  不是变量,仅代表在当前时刻由某一状态产生输出的关系,与实际时序无关;

HMM 工作过程:

(1)  $t=1$  时刻,根据初始状态概率向量 $\pi = (\pi_i)_{N \times 1}$ 从状态空间 $S = \{s_1, s_2, \dots, s_N\}$ 中选择初始状态 $r_1$ ;

(2) 迭代式:  $t = 2, 3, \dots, T-1$ 时刻,由当前的状态 $r_t$ 按输出观测概率矩阵

$B = [b_{jl}]_{N \times M}$  产生对应的观测 $x_t$ ,并按状态转移概率矩阵 $A = [a_{ij}]_{N \times N}$  产生

下一时刻的状态 $r_{t+1}$ ;

(3) 在  $t=T$  时刻,由此时的状态 $r_T$ 按观测概率矩阵产生输出 $x_T$ ;

HMM 需要关心的三个问题:

● 概率计算问题: 给定模型参数 $\lambda = (A, B, \pi)$ 和观测序列 $X = (x_1, x_2, \dots, x_T)$ 求产生当前观测序列的概率 $P(X|\lambda)$ ; 前向后向算法:

定义了处于时刻  $t$  时的观测序列为 $(x_1, x_2, \dots, x_t)$ 且处于状态 $s_i$ 的概率作为前向概率 $\alpha_t(i) = P(x_1, x_2, \dots, x_t, r_t = s_i | \lambda)$ ;

在初始时刻:  $\alpha_1(i) = \pi_i b_i(x_1), i = 1, 2, \dots, N$ ; 表示从状态 $r_1 = s_i$ 输出观测 $x_1$ 的概率,其中 $x_1$ 从给定的观测序列 $(x_1, x_2, \dots, x_t)$ 中得到;

递推式:  $\alpha_t(i) = [\sum_{j=1}^N \alpha_{t-1}(j) a_{ji}] b_i(x_t), i = 1, 2, \dots, N$ ; 表示从状态 $r_t = s_i$ 输

出观测 $x_t$ 的概率，其中当前状态 $r_t = s_i$ 由前一时刻状态 $r_{t-1} = s_j$ 依状态转移

概率矩阵 $A$ 中的第 $i$ 列 $[a_{\cdot i}]_{N \times 1}$ 转移而来，因此需要求和；

得到最终时刻 $T$ 的前向概率 $\alpha_T(i)$ 后，即可得到 $P(X|\lambda) = \sum_{i=1}^N \alpha_T(i)$ ；

定义在时刻 $t$ 的状态为 $s_i$ ，并由此向后出发得到观测序列 $(x_{t+1}, x_{t+2}, \dots, x_T)$ 的概率为后向概率 $\beta_t(i) = P(x_{t+1}, x_{t+2}, \dots, x_T | r_t = s_i; \lambda)$ ；

从 $T$ 时刻出发： $\beta_T(i) = 1, i = 1, 2, \dots, N$ ；规定在 $T$ 时刻由状态 $r_T = s_i$ 输出给定的观测 $x_T$ 的概率为定值 $1$ ；

递推式： $\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j), i = 1, 2, \dots, N$ ；表示由当前状态

$r_t = s_i$ 得到下一时刻状态 $r_{t+1} = s_j$ 及其对应的输出观测 $x_{t+1}$ 的概率；由于当前状态 $r_t = s_i$ 转移可以得到多个下一时刻的状态，对应状态转移概率矩阵 $A$ 中的第 $i$ 行 $[a_{i\cdot}]_{1 \times N}$ ，因此需要求和；

得到最终时刻 $1$ 的后向概率 $\beta_1(i)$ 后，可求 $P(X|\lambda) = \sum_{j=1}^N \pi_j b_1(x_1) \beta_T(j)$ ；

统一前向后向概率，即可得到：

$$P(X|\lambda) = \sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j), t = 1, 2, \dots, T-1$$

- 学习问题：给定观测序列 $X = (x_1, x_2, \dots, x_T)$ 估计模型参数 $\lambda = (A, B, \pi)$ ，使得观测概率产生的概率 $P(X|\lambda)$ 最大，即模型参数的极大似然估计问题；  
**Baum-Welch** 算法：以状态序列 $R = (r_1, r_2, \dots, r_T)$ 作为隐含变量，则学习问题变为 $P(X|\lambda) = \sum_R P(X|R; \lambda) P(R|\lambda)$ ，即含有隐变量的极大似然估计，使用EM算法，构造 $Q(\lambda, \bar{\lambda}) = \sum_R P(R|X, \bar{\lambda}) \log P(X, R|\lambda)$ ，其中 $P(R|X, \bar{\lambda}) = \frac{P(R, X|\bar{\lambda})}{P(X|\bar{\lambda})}$ ；

其中 $\frac{1}{P(X|\bar{\lambda})}$ 为常数，将 $Q$ 函数变形为 $Q(\lambda, \bar{\lambda}) = \sum_R P(R, X|\bar{\lambda}) \log P(X, R|\lambda)$ ，其

中 $P(R, X|\lambda) = \pi_{r_1} b_{r_1}(x_1) a_{r_1 r_2} b_{r_2}(x_2) \cdots a_{r_{T-1} r_T} b_{r_T}(x_T)$ ， $\bar{\lambda}$ 是HMM模型参数的当前估计值；则有：

$$Q(\lambda, \bar{\lambda}) = \sum_R P(R, X|\bar{\lambda}) \log \pi_{r_1} + \sum_R P(R, X|\bar{\lambda}) \sum_{t=1}^{T-1} \log a_{r_t r_{t+1}} +$$

$$\sum_R P(R, X|\bar{\lambda}) \sum_{t=1}^T \log b_{r_t}(x_t)；$$

逐项最大化求解，得到： $\pi_i = \frac{P(X, r_1 = s_i|\bar{\lambda})}{P(X|\bar{\lambda})}, i = 1, 2, \dots, N$ ；

$$a_{ij} = \frac{\sum_{t=1}^{T-1} P(X, r_t = s_i, r_{t+1} = s_j|\bar{\lambda})}{\sum_{t=1}^{T-1} P(X, r_t = s_i|\bar{\lambda})}, i, j = 1, 2, \dots, N；$$

$$b_{jl} = \frac{\sum_{t=1}^T P(X, r_t = s_j, x_t = o_k|\bar{\lambda})}{\sum_{t=1}^T P(X, r_t = s_j|\bar{\lambda})}, j = 1, 2, \dots, N; k = 1, 2, \dots, M；$$

给定观测序列  $X = (x_1, x_2, \dots, x_T)$  和初始参数值  $\pi_i^{(0)}, a_{ij}^{(0)}, b_{jl}^{(0)}$  得到初始模型  $\lambda^{(0)} = (\pi_i^{(0)}, a_{ij}^{(0)}, b_{jl}^{(0)})$ ;

递推式：依据上一步得到的参数估计  $\bar{\lambda} = \lambda^{(p)} = (\pi_i^{(p)}, a_{ij}^{(p)}, b_{jl}^{(p)})$  计算

$$P^{(p)}(X, r_1 = s_i | \bar{\lambda}), \quad P^{(p)}(X, r_t = s_i, r_{t+1} = s_j | \bar{\lambda}), \quad P^{(p)}(X, r_t = s_i | \bar{\lambda}),$$

$$P^{(p)}(X, r_t = s_j, x_t = o_k | \bar{\lambda});$$

更新参数

$$\pi_i^{(p+1)} = \frac{P^{(p)}(X, r_1 = s_i | \bar{\lambda})}{P^{(p)}(X | \bar{\lambda})}, i = 1, 2, \dots, N;$$

$$a_{ij}^{(p+1)} = \frac{\sum_{t=1}^{T-1} P^{(p)}(X, r_t = s_i, r_{t+1} = s_j | \bar{\lambda})}{\sum_{t=1}^{T-1} P^{(p)}(X, r_t = s_i | \bar{\lambda})}, i, j = 1, 2, \dots, N;$$

$$b_{jl}^{(p+1)} = \frac{\sum_{t=1}^T P^{(p)}(X, r_t = s_j, x_t = o_k | \bar{\lambda})}{\sum_{t=1}^T P^{(p)}(X, r_t = s_j | \bar{\lambda})}, j = 1, 2, \dots, N; k = 1, 2, \dots, M;$$

直至达到最大迭代轮数为止;

- 预测(解码)问题：给定模型参数  $\lambda = (A, B, \pi)$  和观测序列  $X = (x_1, x_2, \dots, x_T)$  求最有可能产生当前观测序列的状态序列  $R = (r_1, r_2, \dots, r_T)$ ，即使得条件概率  $P(R|X)$  最大的状态序列  $R = (r_1, r_2, \dots, r_T)$ ；维特比算法：使用动态规划解决 HMM 的预测问题，从时刻  $t=1$  开始，递推计算在时刻  $t$  状态为  $r_t = s_i$  的每条部分路径的最大概率：

$$\delta_t(s_i) = \max_{r_1, r_2, \dots, r_{t-1}} P(r_t = s_i, r_{t-1}, r_{t-2}, \dots, r_1, x_t, x_{t-1}, \dots, x_1 | \lambda),$$

$$i = 1, 2, \dots, N;$$

由此得到最优路径的递推式：

$$\delta_t(s_i) = \max_{r_1, r_2, \dots, r_{t-1}} P(r_t = s_i, r_{t-1}, r_{t-2}, \dots, r_1, x_t, x_{t-1}, \dots, x_1 | \lambda) =$$

$$\max_j [\delta_{t-1}(s_j) a_{ji}] b_{s_i}(x_t), i, j = 1, 2, \dots, N; t = 2, \dots, T;$$

而在时刻  $t$  状态为  $r_t = s_i$  的所有单个路径  $(r_1, r_2, \dots, r_{t-1}, r_t = s_i)$  中拥有最大概率的路径的第  $t-1$  个节点为：

$$\psi_t(i) = \arg \max_j [\delta_{t-1}(s_j) a_{ji}], j = 1, 2, \dots, N;$$

$\delta_t(s_i)$  表示当前时刻  $t$  由前一时刻  $t-1$  转移得到状态  $r_t = s_i$  并输出观测  $x_t$  的最大概率，而此最大概率  $\delta_t(s_i)$  对应的前一时刻  $t-1$  所处的状态  $r_{t-1} = s_j$  就是  $t$  时刻的最优路径节点  $\psi_t(i)$ ;

#### 7.24.2 马尔科夫随机场

马尔科夫网络(无向图模型)：满足成对马尔科夫性，局部马尔科夫性和全局马尔

科夫性的无向图模型;

$P(x) = \frac{1}{Z^*} \prod_{Q \in C^*} \psi_Q(x_Q)$ ; 其中  $Z^* = \sum_x \prod_{Q \in C^*} \psi_Q(x_Q)$  为概率归一化因子, 表示对

所有随机变量的  $\prod_{Q \in C^*} \psi_Q(x_Q)$  求和;

成对马尔科夫性:  $P(Y_u, Y_v | Y_o) = P(Y_u | Y_o) P(Y_v | Y_o)$ , 其中  $Y_u$  与  $Y_v$  两个节点随机变量没有边连接,  $Y_o$  为图中其他节点;

局部马尔科夫性:  $P(Y_v, Y_o | Y_w) = P(Y_v | Y_w) P(Y_o | Y_w)$ , 其中  $Y_v$  表示节点  $v \in V$  的概率分布,  $Y_w$  表示与  $v$  直接相连的节点集合变量组, 节点  $Y_o$  表示其他变量组;

全局马尔科夫性:  $P(Y_A, Y_B | Y_C) = P(Y_A | Y_C) P(Y_B | Y_C)$ , 其中变量组  $A$  和变量组  $B$  在无向图中被变量组  $C$  分开;

### 7.24.3 条件随机场

条件随机场(**Conditional Random Field**) **CRF** 是一种判别式无向图模型, 是指给定随机变量  $X$  的条件下, 随机变量  $Y$  的马尔科夫随机场;

将输入变量  $X$  作为观测序列, 则输出变量  $Y$  是标记序列(对应于 **HMM** 的状态序列  $R$ ), **CRF** 要解决的两个问题:

- 利用训练数据集通过极大似然估计得到条件概率  $\hat{P}(Y|X)$ ;
- 给定输入观测序列  $x$ , 求使得条件概率  $\hat{P}(y|x)$  最大的输出序列  $y$ ;

考虑线性链条件随机场,  $X = (X_1, X_2, \dots, X_n)$  为输入观测序列,  $Y = (Y_1, Y_2, \dots, Y_n)$  为输出标记序列, 则条件概率  $P(Y|X)$  构成条件随机场, 即有

$P(Y_i | X, Y_1, \dots, Y_{i-1}, Y_{i+1}, \dots, Y_n) = P(Y_i | X, Y_{i-1}, Y_{i+1}), i = 1, 2, \dots, n$ ;

即标记  $Y_i$  仅与条件随机场中其相邻节点对应的随机变量  $Y_{i-1}$  和  $Y_{i+1}$  以及输入观测变量  $X = (X_1, X_2, \dots, X_n)$  有关;

参数化形式:  $P(y|x) = \frac{1}{Z(x)} \exp[\sum_{i,k} \lambda_k t_k(y_{i-1}, y_i, x, i) + \sum_{i,l} \mu_l s_l(y_i, x, i)]$ ;

其中  $t_k(y_{i-1}, y_i, x, i), i = 2, 3, \dots, N$ ; 是转移特征函数, 依赖于当前位置与前一个位置节点, 以及输入观测序列,  $\lambda_k$  是其对应的权值;

$s_l(y_i, x, i), i = 1, 2, \dots, N$ ; 是状态特征函数, 依赖于当前位置和输入观测序列,  $\mu_l$  为其对应的权值;

概率归一化因子  $Z(x) = \sum_y \exp[\sum_{i,k} \lambda_k t_k(y_{i-1}, y_i, x, i) + \sum_{i,l} \mu_l s_l(y_i, x, i)]$ , 将指数形式转换为规范化概率;

**CRF** 中同一个位置  $i (i=1, 2, \dots, n)$  的随机变量对应多个特征函数  $[(t_k, \lambda_k), (s_l, \mu_l)]$ ,

而特征函数  $[(t_k, \lambda_k), (s_l, \mu_l)]_{i=1, 2, \dots, n}$  在每一个位置均有定义, 因此先将特征函数

统一定义称分段形式, 可以得到随机变量  $y$  在 **CRF** 中的  $K = K_1 + K_2$ :

$f_k(y_{i-1}, y_i, x, i) = \begin{cases} t_k(y_{i-1}, y_i, x, i), & k = 1, 2, \dots, K_1 \\ s_l(y_i, x, i), & k = K_1 + l; l = 1, 2, \dots, K_2 \end{cases}$ ;

$w_k = \begin{cases} \lambda_k, & k = 1, 2, \dots, K_1 \\ \mu_l, & k = K_1 + l; l = 1, 2, \dots, K_2 \end{cases}$ ;

将特征函数  $f_k(y_{i-1}, y_i, x, i)$  在各个位置求和:

$f_k(y, x) = \sum_{i=1}^n f_k(y_{i-1}, y_i, x, i), k = 1, 2, \dots, K$ ;



则将随机变量 $y$ 的条件概率表示为:

$$P(y|x) = \frac{1}{Z(x)} \exp[\sum_{k=1}^K w_k f_k(y, x)];$$

$$Z(x) = \sum_y \exp[\sum_{k=1}^K w_k f_k(y, x)];$$

定义向量 $w = (w_1, w_2, \dots, w_K)^T$ ;

$$F(y, x) = (f_1(y, x), f_2(y, x), \dots, f_K(y, x))^T;$$

则有 **CRF** 的内积形式:

$$P_w(y|x) = \frac{1}{Z_w(x)} \exp[w \cdot F(y, x)];$$

$$Z_w(x) = \sum_y \exp[w \cdot F(y, x)];$$

若对特征函数每一个位置上的特征函数求和:

$$W_i(y_{i-1}, y_i|x) = \sum_{k=1}^K f_k(y_{i-1}, y_i, x, i);$$

在位置  $i$  上随机变量 $y_i$ 的取值空间为 $\mathcal{Y}$ ,  $m = |\mathcal{Y}|$ , 则可得到在位置  $i$  上的矩阵:

$$M_i(x) = [M_i(y_{i-1}, y_i|x)]_{m \times m};$$

其中,  $M_i(y_{i-1}, y_i|x) = \exp(W_i(y_{i-1}, y_i|x))$ ;

则有条件概率的矩阵形式:

$$P_w(y|x) = \frac{1}{Z_w(x)} \prod_{i=1}^{n+1} M_i(y_{i-1}, y_i|x);$$

$$Z_w(x) = [M_1(x)M_2(x) \cdots M_{n+1}(x)]_{start, stop};$$

即规范化因子 $Z_w(x)$ 是以 **start** 为起点, 以 **stop** 为终点状态的所有路径 $y_1 y_2 \cdots y_n$

的非规范化概率 $\prod_{i=1}^{n+1} M_i(y_{i-1}, y_i|x)$ 之和;

同 **HMM** 类似, **CRF** 也要解决三类问题:

- 概率计算问题: 计算状态特征条件概率 $P(Y_i = y_i|x)$ 和转移特征条件概率

$P(Y_{i-1} = y_{i-1}, Y_i = y_i|x)$ , 以 $\alpha_i(y_i|x)$ 前向概率表示在位置  $i$  处状态为

$Y_i = y_i$ , 下标为 $(1, 2, \dots, i-1)$ 的前部分序列的非规范化概率:

$$\alpha_i(y_i|x) = \alpha_{i-1}(y_{i-1}|x) M_i(y_{i-1}, y_i|x), i = 1, 2, \dots, n+1;$$

$$\alpha_0(y_0|x) = \begin{cases} 1, & y_0 = start \\ 0, & else \end{cases};$$

$\beta_i(y_i|x)$ 后向概率表示在位置  $i$  处状态为 $Y_i = y_i$ , 下标为 $(i+1, i+2, \dots, n+1)$ 的后部分序列的非规范化概率:

$$\beta_i(y_i|x) = M_i(y_i, y_{i+1}|x) \beta_{i+1}(y_{i+1}|x), i = 1, 2, \dots, n;$$

$$\beta_{n+1}(y_{n+1}|x) = \begin{cases} 1, & y_{n+1} = stop \\ 0, & else \end{cases};$$

由于随机变量 $y_i$ 的取值空间为 $\mathcal{Y}$ ,  $m = |\mathcal{Y}|$ , 所以 $\alpha_i(y_i|x)$ 和 $\beta_i(y_i|x)$ 都是  $m$  阶向量,  $i=1, 2, \dots, n$ ;

$$P(Y_i = y_i|x) = \frac{\alpha_i^T(y_i|x) \beta_i(y_i|x)}{Z(x)};$$

$$P(Y_{i-1} = y_{i-1}, Y_i = y_i|x) = \frac{\alpha_{i-1}^T(y_{i-1}|x) M_{i-1}(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)};$$

其中  $Z(x) = \alpha_n^T(y_n|x) \cdot [1]_{m \times 1} = [1^T]_{1 \times m} \cdot \beta_1(y_1|x)$  为概率归一化因子；  
特征函数  $f_k(y_{i-1}, y_i, x, i)$  关于条件概率分布  $P(Y|X)$  的期望：

$$E_{P(Y|X)}[f_k] = \sum_{i=1}^{n+1} \sum_{y_{i-1}y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}^T(y_{i-1}|x) M_{i-1}(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)},$$

$k = 1, 2, \dots, K$ ;

特征函数  $f_k(y_{i-1}, y_i, x, i)$  关于联合概率分布  $P(X, Y)$  的期望：

$$E_{P(X,Y)}[f_k] = \sum_x \tilde{P}(x) \sum_{i=1}^{n+1} \sum_{y_{i-1}y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}^T(y_{i-1}|x) M_{i-1}(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)},$$

$k = 1, 2, \dots, K$ ;

$Z(x) = \alpha_n^T(y_n|x) \cdot [1]_{m \times 1} = [1^T]_{1 \times m} \cdot \beta_1(y_1|x)$ ;

- 学习问题：定义在时序上的对数线性模型，可使用改进的迭代尺度法 *IIIS*，梯度下降法，牛顿法、拟牛顿法求解；

学习问题的输入是转移特征函数  $(t_1, t_2, \dots, t_{K_1})$ ，状态特征函数

$(s_1, s_2, \dots, s_{K_2})$ ，以及输入观测序列与输出标记序列的经验分布  $\tilde{P}(x, y)$ ；

输出对权值向量的估计  $\hat{w} = (\hat{w}_1, \hat{w}_2, \dots, \hat{w}_K)^T$  以及对应的模型  $P_{\hat{w}}(y|x)$ ；

训练数据集  $\{(x_i, y_i)\}_{i=1}^n$  的对数似然函数为：

$$L(w) = \log \prod_{x,y} [P_w(y|x)]^{\tilde{P}(x,y)} = \sum_{x,y} \tilde{P}(x,y) \log P_w(y|x);$$

代入 *CRF* 条件概率的向量表示  $P_w(y|x) = \frac{1}{Z_w(x)} \exp[\sum_{k=1}^K w_k f_k(y, x)]$ ，得到：

$$L(w) = \sum_{x,y} [\tilde{P}(x,y) \sum_{k=1}^K w_k f_k(y, x) - \tilde{P}(x,y) \log Z_w(x)];$$

其中  $\sum_{x,y} \tilde{P}(x,y) = 1$ ；

$L(w) = \sum_{j=1}^n \sum_{k=1}^K w_k f_k(y_j, x_j) - \sum_{j=1}^n \log Z_w(x_j)$ ,  $k = 1, 2, \dots, K$ ；不断优化

对数似然函数  $L(w)$  改变量的下界，极大化对数似然函数，可以得到转移特征  $t_k$  与状态特征  $s_l$  的更新方程为：

$$E_{\tilde{P}}[t_k] = \sum_{x,y} [\tilde{P}(x,y) \sum_{i=1}^{n+1} t_k(y_{i-1}, y_i, x, i) \exp(\delta_k T(x, y))], k = 1, 2, \dots, K_1;$$

$$E_{\tilde{P}}[s_l] = \sum_{x,y} [\tilde{P}(x,y) \sum_{i=1}^n s_l(y_i, x, i) \exp(\delta_{K_1+l} T(x, y))], l = 1, 2, \dots, K_2;$$

$$T(x, y) = \sum_k f_k(y, x) = \sum_{k=1}^K \sum_{i=1}^{n+1} f_k(y_{i-1}, y_i, x, i);$$

其中，状态特征与转移特征函数依经验分布  $\tilde{P}(x, y)$  的期望由下式得到：

$$E_{\tilde{P}}[f_k] = \sum_x \tilde{P}(x) \sum_{i=1}^{n+1} \sum_{y_{i-1} y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}^T(y_{i-1}|x) M_{i-1}(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)},$$

$k = 1, 2, \dots, K;$

$$Z(x) = \alpha_n^T(y_n|x) \cdot [1]_{m \times 1} = [1^T]_{1 \times m} \cdot \beta_1(y_1|x);$$

设置权值更新增量  $\delta = (\delta_1, \delta_2, \dots, \delta_K)^T$ , 解特征方程, 求解相应的  $\delta_k$ , 并更新参数:

$$w_k \leftarrow w_k + \delta_k, k = 1, 2, \dots, K;$$

直至得到收敛的更新增量序列  $\delta = (\delta_1, \delta_2, \dots, \delta_K)^T$ ;

不同于迭代尺度法对各个特征函数求和, 拟牛顿法对序列中所有位置求和:

$$P_w(y|x) = \frac{\exp[\sum_{i=1}^n w_i f_i(x, y)]}{\sum_y \exp[\sum_{i=1}^n w_i f_i(x, y)]};$$

其中,  $f_i(x, y) = \sum_{k=1}^K f_k(y_{i-1}, y_i, x, i), i = 1, 2, \dots, n;$

构造  $P_w(y|x)$  的对数似然函数:  $L(w) = \log \prod_{x,y} [P_w(y|x)]^{\tilde{P}(x,y)}$ , 整理得到:

$$L(w) = \sum_{x,y} \tilde{P}(x, y) \log [P_w(y|x)] = \sum_{x,y} \tilde{P}(x, y) \sum_{i=1}^n w_i f_i(x, y) -$$

$$\sum_{x,y} \tilde{P}(x, y) \log \sum_y \exp[\sum_{i=1}^n w_i f_i(x, y)];$$

将对数似然函数极大化转换成目标函数最优化问题即:

$$\min_{w \in \mathcal{R}^n} f(w) =$$

$$\sum_x \tilde{P}(x) \log \sum_y \exp[\sum_{i=1}^n w_i f_i(x, y)] - \sum_{x,y} \tilde{P}(x, y) \sum_{i=1}^n w_i f_i(x, y);$$

求目标函数  $f(w)$  对  $w$  的梯度:

$$g(w) = \sum_{x,y} \tilde{P}(x) P_w(y|x) f(x, y) - E_{\tilde{P}}[f(x, y)];$$

**CRF** 模型的 **BFGS** 算法输入为各个位置上的特征函数(对  $k$  求和)  $f_1, f_2, \dots, f_n$ , 以及训练数据的经验分布  $\tilde{P}(x, y)$ ; 输出为估计参数值  $\hat{w} = (\hat{w}_1, \hat{w}_2, \dots, \hat{w}_K)^T$  以及对应的模型  $P_{\hat{w}}(y|x)$ ;

选定初始点  $w^{(0)} = (w_1^{(0)}, w_2^{(0)}, \dots, w_K^{(0)})^T$ , 取  $B_0$  为正定对称矩阵,  $t=0$ ;

$B_t p_t = -g_t$ ; 求解向量  $p_t$ , 其中  $g_t = g(w^{(t)})$ ;

求  $\lambda_t$ , 满足条件  $f(w^{(t)} + \lambda_t p_t) = \min_{\lambda \geq 0} f(w^{(t)} + \lambda p_t)$ ;

更新  $w^{(t+1)} = w^{(t)} + \lambda_t p_t$ , 计算  $g_{t+1} = g(w^{(t+1)})$ , 更新矩阵

$$B_{t+1} = B_t + \frac{(\Delta g)_t (\Delta g)_t^T}{(\Delta g)_t^T (\Delta w)_t} - \frac{B_t (\Delta w)_t (\Delta w)_t^T B_t}{(\Delta w)_t^T B_t (\Delta w)_t};$$

$$(\Delta g)_t = g_{t+1} - g_t, (\Delta w)_t = w^{(t+1)} - w^{(t)};$$

$t = t + 1$ , 重复上述过程迭代更新参数值, 直至到达最大迭代轮数  $T$  或者在

计算过程中使得梯度  $g_{\tau} = g(w^{(\tau)}) = 0$ ;

- 预测问题，**CRF** 的预测问题是指给定模型  $P(Y|X)$  以及输入观测序列  $x$ ，求解使得条件概率最大的输出观测序列  $y^*$ ，是典型的标注问题；

$y^* = \arg \max_y P_w(y|x) = \arg \max_y \frac{\exp(w \cdot F(y, x))}{Z_w(x)}$ ；再由概率归一化因子与求最大化无关性和指数函数的单调性可得到：

$y^* = \arg \max_y (w \cdot F(y, x))$ ；即求解非规范化概率最大的输出序列；

$w = (w_1, w_2, \dots, w_K)^T$ ;

$F(y, x) = (f_1(y, x), f_2(y, x), \dots, f_K(y, x))^T$ ;

$f_k(y, x) = \sum_{i=1}^n f_k(y_{i-1}, y_i, x_i, i), k = 1, 2, \dots, K$ ;

可得到  $F_i(y_{i-1}, y_i, x) = (f_1(y_{i-1}, y_i, x), f_2(y_{i-1}, y_i, x), \dots, f_K(y_{i-1}, y_i, x))^T$ ;

对将原问题变形为：

$y^* = \arg \max_y \sum_{i=1}^n w \cdot F_i(y_{i-1}, y_i, x)$ ;

首先求位置  $i = 1$  处对应的标记  $y_1 = v_j, v_j \in \mathcal{Y}^m$  的非规范化概率：

$\delta_1(v_j) = w \cdot F_1(y_0 = \text{start}, y_1 = v_j, x), j = 1, 2, \dots, m$ ;

递推式得到最大非规范化概率及其对应的最佳路径为：

$\delta_l(v_l) = \max_{1 \leq j \leq m} [\delta_{l-1}(v_j) + w \cdot F_l(y_{l-1} = v_j, y_l = v_l, x)], l = 1, 2, \dots, m$ ;

$\psi_l(v_l) = \arg \max_{1 \leq j \leq m} [\delta_{l-1}(v_j) + w \cdot F_l(y_{l-1} = v_j, y_l = v_l, x)],$

$l = 1, 2, \dots, m$ ;

直至  $i = n$  处终止，此时得到全路径最大非规范化概率为：

$\max_y (w \cdot F(y, x)) = \max_{1 \leq j \leq m} \delta_n(v_j)$ ;

$y_n^* = \arg \max_{1 \leq j \leq m} \delta_n(v_j)$ ;

回溯得到最优的全路径：

$y_i^* = \psi_{i+1}(y_{i+1}^*), i = n-1, n-2, \dots, 1$ ;

得到最优路径  $y^* = (y_1^*, y_2^*, \dots, y_n^*)^T$ ;

#### 7.24.4 隐狄利克雷分配模型

**LDA(Latent Dirichlet Allocation)** 是一种话题模型，话题模型是生成式有向图模型；长于处理离散型数据的词袋(**bag of words**)表示：

**Document** 对应于样本；

**Word** 对应于样本的特征；

**Topic** 对应于样本的分类，具体表现为一个概念，以及在这个概念下出现频率较高的一系列词语；

设有词典(**dictionary**)，其长度为  $N$ ， $T$  篇文档可以表示为  $T$  个  $N$  维向量，数据集

可表示为:  $W = \{\omega_1, \omega_2, \dots, \omega_T\}, \omega_t \in \mathcal{R}^N$ ,  $\omega_{t,i}$  表示第  $t$  篇文档的第  $i$  个词语出现的频率(词频),  $\sum_{i=1}^N \omega_{t,i} = 1$ ;

使用  $K$  个  $N$  维向量表示话题:  $\beta = \{\beta_1, \beta_2, \dots, \beta_K\}, \beta_k \in \mathcal{R}^N$ ,  $\beta_{k,j}$  表示话题  $k$  中的第  $j$  个词语的词频,  $\sum_{j=1}^N \beta_{k,j} = 1$ ;

$\beta_k$  由以  $\eta$  为参数的 *Dirichlet* 分布得到:

$$Dir(\beta_{k,1}, \beta_{k,2}, \dots, \beta_{k,N} | \eta) = \frac{\Gamma(\sum_{j=1}^N \eta_j)}{\Gamma(\eta_1) \Gamma(\eta_2) \dots \Gamma(\eta_N)} \prod_{j=1}^N \beta_{k,j}^{\eta_j - 1};$$

*LDA* 认为一篇文档中包含多个话题, 以向量  $\theta_t \in \mathcal{R}^K$  表示在文档  $t$  中包含的话题分布, 其中  $\theta_{t,k}$  表示话题  $k$  在文档  $t$  中所占比例,  $\sum_{k=1}^K \theta_{t,k} = 1$ ;

● 根据参数为  $\alpha$  的 *Dirichlet* 分布随机采样一个话题分布  $\theta_t$ :

$$Dir(\theta_{t,1}, \theta_{t,2}, \dots, \theta_{t,K} | \alpha) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\Gamma(\alpha_1) \Gamma(\alpha_2) \dots \Gamma(\alpha_K)} \prod_{k=1}^K \theta_{t,k}^{\alpha_k - 1};$$

● 生成文档中的  $N$  个词语:

根据向量  $\theta_t$  进行话题指派, 得到文档  $t$  中各个词语的所属的话题  $k = z_{t,i}, k = 1, 2, \dots, K$ ;

再根据指派话题对应的分布向量  $\beta_k$  依 *Dirichlet* 分布随机采样生成词语; 文档  $t$  以不同的比例包含  $K$  个话题, 话题  $\theta_t$  由 *Dirichlet* 分布随机采样得到, 文档中的  $N$  个词语分属于不同的话题;

文档词频  $\omega_{t,i}$  可由文档词频统计得到, 是唯一的可观测变量;

$\omega_{t,i}$  依赖于对词语的话题指派  $z_{t,i}$  以及话题词频向量  $\beta_{k,j}, (k = z_{t,i}, j = i)$ ;

话题指派  $z_{t,i}$  依赖于话题分布  $\theta_t$ ;

话题分布  $\theta_t$  依赖于 *Dirichlet* 分布及其参数  $\alpha$ ;

词频统计  $\omega_{t,i}$  依赖于参数  $\eta$ ;

则得到 *LDA* 的概率分布为:

$$p(W, z, \beta, \theta | \alpha, \eta) = \prod_{t=1}^T p(\theta_t | \alpha) \prod_{k=1}^K p(\beta_k | \eta) [\prod_{i=1}^N P(\omega_{t,i} | z_{t,i}) P(z_{t,i} | \theta_t)];$$

其中, 文档  $t$  中的话题先验分布:

$$p(\theta_t | \alpha) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\Gamma(\alpha_1) \Gamma(\alpha_2) \dots \Gamma(\alpha_K)} \prod_{k=1}^K \theta_{t,k}^{\alpha_k - 1}; \text{ 由此产生每个词语的话题指派}$$

$k = z_{t,i}$ ;

其中话题  $k$  中的词频分布:

$$p(\beta_k | \eta) = \frac{\Gamma(\sum_{j=1}^N \eta_j)}{\Gamma(\eta_1) \Gamma(\eta_2) \dots \Gamma(\eta_N)} \prod_{j=1}^N \beta_{k,j}^{\eta_j - 1};$$

由话题词频先验分布和话题先验分布产生的话题指派  $k = z_{t,i}$  共同产生文档词频

$\omega_{t,i}$  ;

给定数据集  $W = \{\omega_1, \omega_2, \dots, \omega_T\}$ ,  $\omega_t \in \mathcal{R}^N$ , LDA 模型参数  $\alpha, \eta$  使用极大似然估计法求解:

$$L(\alpha, \eta) = \ln \prod_{t=1}^T p(\omega_t | \alpha, \eta);$$

## 7.25 从伯努利到狄利克雷分布

### 7.25.1 伯努利分布

伯努利分布 (**Bernoulli Distribution**) 是指对二元随机变量进行一次伯努利试验 (**Bernoulli Trial**) 得到的随机变量分布律:

$Bern(x|\mu) = \mu^x(1-\mu)^{1-x}$ ; 其中随机变量  $x \in \{0, 1\}$ ,  $0 \leq \mu \leq 1$ ;

伯努利分布描述了随机变量的分布  $P(x=1|\mu) = \mu$ ;  $P(x=0|\mu) = 1-\mu$ ;

伯努利分布的均值为:

$$E[x] = E_{Bern}[x] = 1 \cdot P(x=1|\mu) + 0 \cdot P(x=0|\mu) = \mu;$$

方差为:

$$var[x] = (1-E[x])^2 \cdot P(x=1|\mu) + (0-E[x])^2 \cdot P(x=0|\mu) = \mu(1-\mu);$$

伯努利分布的参数估计方法:

基于给定样本数据集  $D = \{x_1, x_2, \dots, x_N\}$ , 估计模型参数  $\mu$ , 使用极大似然估计,

$$P(x_i|\mu) = \mu^{x_i}(1-\mu)^{1-x_i}, x_i \in D;$$

$$\text{构造对数似然函数 } L(\mu|D) = \ln \prod_{i=1}^N P(x_i|\mu) = \ln \prod_{i=1}^N \mu^{x_i}(1-\mu)^{1-x_i};$$

$$\text{对变量 } \mu \text{ 求导, 得到 } \frac{\partial L(\mu|D)}{\partial \mu} = \frac{1}{\mu} \sum_{i=1}^N x_i - \frac{1}{1-\mu} \sum_{i=1}^N (1-x_i);$$

$$\text{令 } \frac{\partial L(\mu|D)}{\partial \mu} = 0, \text{ 得到 } \hat{\mu} = \frac{1}{N} \sum_{i=1}^N x_i \text{ 作为对参数 } \mu \text{ 的估计};$$

### 7.25.2 二项分布

伯努利分布考虑的是随机变量  $x \in \{0, 1\}$ ,  $0 \leq \mu \leq 1$  在伯努利试验中的分布律, 即概率  $Bern(x|\mu) = \mu^x(1-\mu)^{1-x}$ , 而二项分布 (**Binomial Distribution**) 考虑在数据集  $D = \{x_1, x_2, \dots, x_N\}$  中, 随机变量  $x_i$  取值为 1 的个数  $m$  的概率分布, 其中

$$m = \sum_{i=1}^N I(x_i = 1);$$

$$Bin(m|D; \mu) = C_N^m \mu^m (1-\mu)^{N-m};$$

二项分布的均值求解转换成如下问题描述:  $N$  次伯努利试验(相互独立)出现  $x=1$  的次数;

方差可看做  $N$  次伯努利试验(相互独立)的随机变量方差求和;

$E[m] = \sum_{m=0}^N m \cdot \text{Bin}(m|D; \mu) = N \cdot \mu$ ; 独立事件加和的均值等于均值的加和,  
加和的方差等于方差的加和;  
方差为:

$$\text{var}[m] = \sum_{m=0}^N (m - E[m])^2 \cdot \text{Bin}(m|D; \mu) = N \cdot \mu(1 - \mu);$$

对参数 $\mu$ 的估计是数据集中 $x_i = 1, i = 1, 2, \dots, N$ 所占的比例 $\hat{\mu} = \frac{m}{N}$ ;

### 7.25.3 Beta 分布

极大似然估计的局限性: 当数据集的样本量较少时, 使用极大似然估计易发生过拟合, 即对参数的估计容易受到样本波动的影响;

由贝叶斯公式 $P(x|\mu)P(\mu) = P(\mu|x)P(x)$ 得到 $P(\mu|x) = \frac{P(x|\mu)P(\mu)}{P(x)}$ , 即后验概率

$P(\mu|x)$ 正比于似然 $P(x|\mu)$ 与先验概率 $P(\mu)$ 的乘积, 先验概率 $P(x)$ 在数据集给定的情况下可认为是常数:  $f(\mu; D) \propto p(x|\mu)p(\mu)$ ;

由于 $x \sim \text{Bern}$ :  $P(x|\mu) = \mu^x(1 - \mu)^{1-x}$ , 选择 **Beta** 分布:

$\text{Beta}(\mu|a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \mu^{a-1}(1 - \mu)^{b-1}$ ; 其中 **Gamma** 函数:

$$\Gamma(a) = \int_0^{+\infty} t^{a-1} e^{-t} dt;$$

**Beta** 分布的期望为:  $E[\mu] = \frac{a}{a+b}$ ; 方差为:  $\text{var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)}$ ;

可以得到后验概率:

$$P(\mu|x) = P(x|\mu) \cdot \text{Beta}(\mu|a, b) = C_N^m \mu^m (1 - \mu)^{N-m} \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \mu^{a-1}(1 - \mu)^{b-1};$$

即有 $P(\mu|x) \propto \mu^{m+a-1}(1 - \mu)^{N-m+b-1}$ , 令 $l = N - m$ , 再将其进行概率归一化得到拥有新参数的 **Beta** 分布:

$\text{Beta}(\mu|m + a, l + b) = \frac{\Gamma(a+b+m+l)}{\Gamma(a+m)\Gamma(b+l)} \mu^{a+m-1}(1 - \mu)^{b+l-1}$ ; 先验函数关于似然函数共轭;

贝叶斯估计与极大似然估计对比: 在数据集规模有限的情况下, 参数 $\mu$ 的后验均值比其先验分布 $\text{Beta}(\mu|a, b)$ 给出的均值大, 比其极大似然估计均值 $\hat{\mu}$ 小, 当数据集规模趋近于无穷大时, 二者得到相同的结果;

### 7.25.4 多项式分布

不同于伯努利分布和二项分布产生于伯努利试验, 多项式分布 (**Multinomial Distribution**)描述了随机变量 $\mathbf{x}$ 的取值空间 $|\mathcal{X}| > 2$  即 $\mathbf{x}$ 的属性拥有多种取值的情况, 使用 **one-hot** 表示方法:  $\mathbf{x} = [x_k]_{K \times 1}, x_k \in \{0, 1\}$ , 且有 $\sum_{k=1}^K x_k = 1$ , 使用 $\mu_k$ 表

示 $x_k = 1$ 的概率, 则有 $p(\mathbf{x}|D) = \prod_{k=1}^K \mu_k^{x_k}$  表示随机变量的概率分布;

$$\sum_x p(x|\mu) = \sum_{k=1}^K \mu_k = 1;$$

$$E_\mu[x] = \sum_x p(x|\mu) \cdot x = \mu = (\mu_1, \mu_2, \dots, \mu_K)^T;$$

给定数据集  $D = \{x_1, x_2, \dots, x_N\}$ , 似然函数为:

$$p(D|\mu) = \prod_{i=1}^N \prod_{k=1}^K \mu_k^{x_{ik}} = \prod_{k=1}^K \mu_k^{m_k}, m_k = \sum_{i=1}^N x_{ik};$$

构造最优化问题:

$$\min_{\mu_k} - \ln p(D|\mu);$$

$$s. t. \sum_{k=1}^K \mu_k = 1;$$

$$\text{求解得到 } \hat{\mu}_k = \frac{m_k}{N}, k = 1, 2, \dots, K;$$

此时, 考虑变量  $m_k$  在数据集和参数  $\mu = (\mu_1, \mu_2, \dots, \mu_K)^T$  下的概率分布就得到多项式分布:

$$Mult(m_1, m_2, \dots, m_K | \mu, N) = C_{m_1 m_2 \dots m_K}^N \prod_{k=1}^K \mu_k^{m_k};$$

$$C_{m_1 m_2 \dots m_K}^N = \frac{N!}{m_1! m_2! \dots m_K!}; \text{ 且满足限制条件 } \sum_{k=1}^K \mu_k = 1;$$

### 7.2.5.5 Dirichlet 分布

**Beta** 分布作为二项分布的参数  $\mu$  的先验分布, 与似然函数  $p(x|\mu)$  相乘得到其后验分布, 以解决极大似然估计法在有限样本量的条件下的过拟合问题;

**Dirichlet** 分布则是作为多项分布参数  $\mu = \{\mu_k\}_{k=1}^K$  的先验分布, 观察多项分布

$Mult(m_1, m_2, \dots, m_K | \mu, N) = C_{m_1 m_2 \dots m_K}^N \prod_{k=1}^K \mu_k^{m_k}$  可以得到其共轭先验具有以下形式:

$$p(\mu|\alpha) \propto \prod_{k=1}^K \mu_k^{\alpha_k}; \text{ 概率归一化后得到:}$$

$$Dir(\mu|\alpha) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\Gamma(\alpha_1) \Gamma(\alpha_2) \dots \Gamma(\alpha_K)} \prod_{k=1}^K \mu_k^{\alpha_k - 1};$$

类似于二项分布, 参数  $\mu = \{\mu_k\}_{k=1}^K$  的后验分布可以由其先验分布与似然做乘积

得到:  $p(\mu|D) \propto p(D|\mu)p(\mu|\alpha) \propto \prod_{k=1}^K \mu_k^{m_k + \alpha_k - 1}$ ; 再进行概率归一化得到:

$$p(\mu|D; \alpha) = \frac{\Gamma(\sum_{k=1}^K \alpha_k + \sum_{k=1}^K m_k)}{\Gamma(\alpha_1 + m_1) \Gamma(\alpha_2 + m_2) \dots \Gamma(\alpha_K + m_K)} \prod_{k=1}^K \mu_k^{\alpha_k + m_k - 1} = Dir(\mu|\alpha + m); \text{ 仍然是}$$

**Dirichlet** 分布, 其中  $m = (m_1, m_2, \dots, m_K)^T$ , 将参数  $\alpha_k$  看做是  $m_k = 1$  的观测数;