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++

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{}
viod 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
        retrun 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(incode/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；
  在函数中使用 yield，则函数就变为一个生成器 gernerator object，yield 会生成一个序列，但不会全部返回，只有在调用.__next__()方法时才会返回一个值，实际使用时，使用循环将生成器所有数值返回：

```
def func(n):
    for i in range(n):
        yield i
for x in func(5):
    print(x)
```

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

- 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
$	**匹配输入字符串的结尾位置**。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。要匹配 $ 字符本身，请使用 \$。
( )	**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 "foooood" 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 "fooooood" 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;
}
```

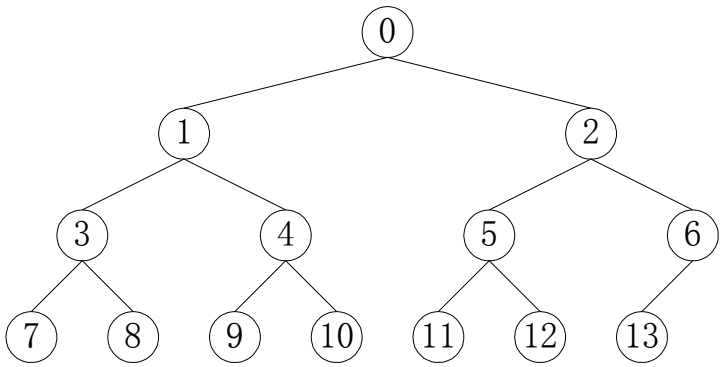
未限定问题规模，2^x = n；得出 x = log(n)，算法复杂度为 O(log(n))；

- 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<=x.key；若 y 为其右子树中的节点，y.key>=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->pLeft 入栈 s.push(tree->pLeft)，再将其左孩子节点 tree->pRight 入栈 s.push(tree->pRight)；

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 || previstited = 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*1 的小矩形横向或者纵向无重叠覆盖一个 2*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 = lambda n: 1 if n<2 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 $\boldsymbol{E(f;D) = \frac{1}{m}\sum_{i=1}^{m} I(f(x_i) \neq y_i)}$；

  分类精度：$\boldsymbol{acc(f;D) = 1 - E(f;D)}$；

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

7.1.3 偏差-误差分解

模型在数据集 ***D*** 上的输出为 $\boldsymbol{f(x;D)}$，**模型期望预测**为 $\boldsymbol{E_D[f(x;D)]}$；

- 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：

  $\boldsymbol{\mathbf{var}[\mathbf{f}(\mathbf{x})] = E_D[(f(x;D) - \mathbf{exp}[f(x)])^2]}$；
- **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：

  $\boldsymbol{\varepsilon^2 = E_D[(y_D - y)^2]}$；
- **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：

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

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：

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

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

Logistic Regression **与最大熵模型有相同的形式，最大熵模型：**

$\boldsymbol{P_w(y|x) = \frac{exp[\sum_{k=1}^{K} w_k f_k(x,y)]}{\sum_y exp[\sum_{k=1}^{K} w_k f_k(x,y)]}}$**;**

统称为对数线性模型，模型的训练方法为极大似然估计；

对数几率回归将线性回归模型应用于分类问题：

$\boldsymbol{z = \omega^T x + b}$;

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

对于二类分类问题：

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

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

极大似然估计法计算参数：

对于数据集$\boldsymbol{\{(x_i, y_i)\}_{i=1}^{N}}$，**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$\boldsymbol{\prod_{i=1}^{N} p(y_i|x_i;\omega,b)}$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：

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

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

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

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

$\boldsymbol{L(\beta)}$是$\boldsymbol{\beta}$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：$\boldsymbol{\omega^T\Sigma_0\omega + \omega^T\Sigma_1\omega}$；

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：

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

定义类内散度矩阵：

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

类间散度矩阵：

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

优化目标为最大化广义瑞利商：

$\boldsymbol{J = \frac{\omega^T S_b \omega}{\omega^T S_w \omega}}$；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；

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

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$\boldsymbol{P(c_k)}$：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$\boldsymbol{P(c_k)=\frac{|D_k|}{|D|}}$ ***(k = 1,2,…,K)***的估计，其中 ***K = |C|***为类别总数；
- 似然$\boldsymbol{P(x|c_k)}$：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$\boldsymbol{D_k}$(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$\boldsymbol{\hat{\theta}_k=arg\ max_{\theta_k}L(\theta_k)}$；

关于概率***(Probability)***和似然***(Likelihood)***的区别：当模型参数$\boldsymbol{\theta}$已知时，求样本 ***X*** 出现的可能性，此时$\boldsymbol{P(X|\theta)}$**叫做概率函数，概率是已知模型和参数，求数据分布；**

当样本已知，模型参数$\boldsymbol{\theta}$**作为变量，**此时$\boldsymbol{P(X|\theta)}$**叫做似然函数，描述的是对于不同的模型参数，出现样本 *X* 的概率，似然是已知数据分布，求模型和参数；**

贝叶斯公式的物理意义：

$\boldsymbol{P(c|x)=\frac{P(x,c)}{P(x)}=\frac{P(c)P(x|c)}{P(x)}=\frac{P(c)P(x|c)}{P(c)P(x|c)+P(\sim c)P(x|\sim c)}}$ ；

$\boldsymbol{c}$作为$\boldsymbol{x}$出现的证据，有多大的可能性值得相信$\boldsymbol{x}$的出现与其有关；

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：$\boldsymbol{P(c_k|x)=\frac{P(c_k)P(x|c_k)}{P(x)}}$；

贝叶斯公式的物理意义：样本属性$\boldsymbol{x_i}$**独立于其类别标签**$\boldsymbol{y_i=c_k}$**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$\boldsymbol{c^*=arg\,max_c\,P(c_k|x)}$**；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，也可是连续变量；**

类条件概率$\boldsymbol{P(x|c_k)}$代表在类别$\boldsymbol{c_k}$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$\boldsymbol{x=(x_1,\ x_2,\dots,x_d)}$，$x_i$为样本 ***x*** 在属性 ***i*** 上的取值***(i = 1, … , 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$\boldsymbol{P(c_k)}$可依照大数定理求解，$\boldsymbol{P(c_k)=\frac{|D_k|}{|D|}}$；

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

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

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$\boldsymbol{P(x_{null}|c_k)=0}$，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：

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

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

7.7 常用的损失函数

7.7.1 损失函数的数学解释

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：

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

其中$\sum_{i=1}^{N} loss[f(x_i), y_i]$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 $loss_{0/1}(z) = \begin{cases} 1; if\ z<0 \\ 0; else \end{cases}$；

  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)： $loss_{log}(z) = \log(1+\exp(-z))$；

7.7.2 Tensorflow 中的损失函数

神经网络模型使用**交叉熵*(Cross Entropy)***度量网络输出向量与期望向量之间的距离：$H(p,q) = -\sum_x p(x) log[q(x)]$，**其中$p(x)$和$q(x)$是随机变量 *X* 的概率分布；通常神经网络的输出向量并不是一个规范化的概率分布，要使用 Softmax Regression 将输出向量 $y = (y_1, y_2, \dots, y_m)^T$归一化处理：**

$\hat{y}_i = Softmax(y_i) = \frac{exp(y_i)}{\sum_{j=1}^{m} exp(y_j)}$, $i = 1, 2, \dots, m$；

在神经网络的训练过程中，得到经过 *softmax* 层处理后的输出：

$\hat{y} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m)^T$**是一个概率分布，将一个 *Batch* 的样本输出组织成$N \times m$矩阵 $Y_{N\times m}$，将神经网络输出$Y_{N\times m}$逐个元素取自然对数并与样本标记矩阵$Y_{N\times m}$做哈达马乘积*(Hadamard Product)*，即对应元素相乘得到矩阵$[Y * Y_]_{N\times m}$，逐行对元素求和得到 *N* 个样本的交叉熵，再对 *N* 个交叉熵求均值即可得到一个 *Batch* 的平均交叉熵；**

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_)
```

而当应用于分类问题的神经网络输出只有一个正确答案时，可以使用加速计算交叉熵方法：

cross_entropy =

tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y, labels=argmax(y_, 1))

其中 logits=y 必须是前向传播未经过 Softmax Regression 的输出结果，否则会产生错误结果，而 labels=argmax(y_, 1)是从 BATCH_SIZE*10 维样本标记向量中选出每行中最大值作为其类别标签；

回归问题使用均方误差***(Mean Squared Error)MSE*** 度量输出预测值与样本标签之间的偏离程度：

mse = tf.reduce_mean(tf.square(y_ - y))

神经网络优化

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：

  $\boldsymbol{g(D|A) = H(D) - H(D|A)}$；

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

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

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

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

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

  其中$\boldsymbol{g(D, A) = H(D|A)}$；

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

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

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

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

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：

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

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

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

$\boldsymbol{\sum_{t=1}^{|T|} N_t H_t(T)}$为模型的经验风险，代表决策树对训练数据集的拟合程度，$\boldsymbol{\alpha|T|}$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$\boldsymbol{C_\alpha(T)}$；

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$T_a = \{\frac{a^i+a^{i+1}}{2} \mid 1 \le i \le n-1\}$ 进行划分，检查此划分对样本集合的信息增益：

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

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：$\boldsymbol{v} \leftarrow \boldsymbol{v} + \Delta \boldsymbol{v}$；

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

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

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

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

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

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

用到 ***Sigmoid*** 函数的一个重要特性$\boldsymbol{f'(x)} = \boldsymbol{f(x)}(\mathbf{1} - \boldsymbol{f(x)})$ ；
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：

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

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

$\boldsymbol{\theta_{\mathrm{j}}} \leftarrow \boldsymbol{\theta_{\mathrm{j}}} + \Delta\boldsymbol{\theta_{\mathrm{j}}}$；$\Delta\boldsymbol{\theta_{\mathrm{j}}} = -\boldsymbol{\eta}\frac{\partial \boldsymbol{E_k}}{\partial \boldsymbol{\theta_{\mathrm{j}}}}$；

$\boldsymbol{\gamma_{\mathrm{h}}} \leftarrow \boldsymbol{\gamma_{\mathrm{h}}} + \Delta\boldsymbol{\gamma_{\mathrm{h}}}$；$\Delta\boldsymbol{\gamma_{\mathrm{h}}} = -\boldsymbol{\eta}\frac{\partial \boldsymbol{E_k}}{\partial \boldsymbol{\gamma_{\mathrm{h}}}}$；

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***)：

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

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：

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

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_pro = KEEP_PROB)**

7.11 无约束最优化问题求解

凸优化问题按照有无约束条件可分为约束优化问题和无约束优化问题，约束优化问题按照其约束条件是等式还是不等式可分为等式约束优化问题和不等式约束优化问题，**等式约束优化**可直接使用**拉格朗日乘子法求解，**而不等式约束优化使用 ***KKT*** **条件**下的拉格朗日乘子法将原始的**极小-极大问题**转换为求解**对偶问题极大-极小**问题；

常用的无约束最优化问题求解方法有**梯度下降法**和**牛顿法/拟牛顿法：**

- 梯度下降法使用目标函数的**一阶偏导数**求解参数的梯度向量，并沿着负梯度方向进行参数更新；
- 牛顿法/拟牛顿法使用目标函数的二阶偏导数生成目标函数的 ***Hessian*** **矩阵，**通过其逆矩阵进行参数更新求解；

7.11.1 拉格朗日乘子法与 KKT 条件

不等式约束的拉格朗日乘子法求解及 ***KKT*** 条件详见 Chapter 7.14 支持向量机与拉格朗日乘子法；

7.11.2 梯度下降法

梯度下降法即沿着目标函数 $\boldsymbol{J(\theta)}$的参数$\boldsymbol{\theta}$的负梯度方向以学习速率 $\boldsymbol{\eta}$ 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 $\boldsymbol{J(\theta; x_i, y_i)}$，沿着负梯度方向以学习速率 $\boldsymbol{\eta}$ 更新参数$\boldsymbol{\theta}$，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)

7.11.3 牛顿法/拟牛顿法

对于无约束最优化问题求解：

$\boldsymbol{min_{x\in\mathcal{R}^n}\ f(x)}$；

$\boldsymbol{f(x)}$目标函数存在极小值的必要条件是其梯度向量在极值点处为 ***0***，而当其 ***Hessian*** 矩阵为正定矩阵时，此极值点为极小值点；

将目标函数$\boldsymbol{f(x)}$在第 ***k*** 次迭代的取值$\boldsymbol{x^{(k)}}$处进行二阶泰勒展开：

$$\boldsymbol{f(x)\cong f\left(x^{(k)}\right)+g\left(x^{(k)}\right)\left(x-x^{(k)}\right)+\frac{1}{2}\left(x-x^{(k)}\right)^{T}H(x^{(k)})\left(x-x^{(k)}\right)};$$

其中：$\boldsymbol{g\left(x^{(k)}\right)=\nabla f\left(x^{(k)}\right)=\frac{\partial f(x)}{\partial x}|_{x=x^{(k)}}}$ 为目标函数的梯度向量在$\boldsymbol{x^{(k)}}$处的取值；

$\boldsymbol{H\left(x^{(k)}\right)=[\frac{\partial^2 f(x)}{\partial x_i\partial x_j}|_{x=x^{(k)}}]_{n\times n}}$ 为目标函数的 Hessian 矩阵在$\boldsymbol{x^{(k)}}$处的取值；

对目标函数的二阶泰勒展开式求梯度并令其为 ***0***：

$\boldsymbol{\nabla f(x)=\frac{\partial f(x)}{\partial x}=\ g\left(x^{(k)}\right)+\ H\left(x^{(k)}\right)\left(x-x^{(k)}\right)=0}$，由此可以得到参数更新式：

$$\boldsymbol{x^{(k+1)}=x^{(k)}-H_k^{-1}g_k}\ ;$$

类比于梯度下降法参数更新式$\boldsymbol{x^{(k+1)}=x^{(k)}-\eta g_k}$，牛顿法在负梯度之上还考虑了目标函数的 ***Hessian*** 矩阵的逆矩阵作为参数更新方向，而学习率需要一维搜索实时更新：$\boldsymbol{\lambda_k=arg\ min_\lambda\ f(x^{(k)}-\lambda H_k^{-1}g_k)}$；

得到牛顿法求解无约束最优化问题的步骤：

(1) 输入目标函数$\boldsymbol{f(x)}$及其梯度向量$\boldsymbol{g(x)=\nabla f(x)}$和 ***Hessian*** 矩阵$\boldsymbol{H(x)=[\frac{\partial^2 f(x)}{\partial x_i\partial x_j}]_{n\times n}}$，精度$\boldsymbol{\varepsilon}$；

(2) 初始化变量点$\boldsymbol{x^{(0)}}$，设置 ***k=0***；

(3) 计算 $g(x^{(k)}) = \nabla f(x^{(k)})$ 和 $H(x^{(k)}) = [\frac{\partial^2 f(x)}{\partial x_i \partial x_j}|_{x=x^{(k)}}]_{n\times n}$ ，若梯度满足 $||g(x^{(k)})|| \le \varepsilon$，则停机，否则更新参数：

$x^{(k+1)} = x^{(k)} - H_k^{-1} g_k$；

牛顿法中目标函数的 ***Hessian*** 矩阵求逆运算较难实现，因此引入拟牛顿方法，使用一个 ***n*** 阶矩阵 $G_k = G(x^{(k)})$近似H_k^{-1}，也可使用矩阵B_k逐步逼近H_k：

由 $\nabla f(x) = \frac{\partial f(x)}{\partial x} = g(x^{(k)}) + H(x^{(k)})(x - x^{(k)})$，令$x = x^{(k+1)}$ 可得：

$g(x^{(k+1)}) - g(x^{(k)}) = H(x^{(k)})(x^{(k+1)} - x^{(k)})$；即有拟牛顿条件：

$H_k^{-1}(g_{k+1} - g_k) = x^{(k+1)} - x^{(k)}$；选择一个正定矩阵$G_k = G(x^{(k)})$近似$H_k^{-1}$可保证参数更新的方向是向下的，从而完成参数和矩阵$G_k$的更新：

(1) 输入目标函数$f(x)$及其梯度向量$g(x) = \nabla f(x)$，精度ε；

(2) 初始化变量点$x^{(0)}$，选择正定对称矩阵B_0，设置 ***k=0***；

(3) 计算$g(x^{(k)}) = \nabla f(x^{(k)})$，若$||g(x^{(k)})|| \le \varepsilon$则停机，否则进行参数更新：

一维搜索更新学习率$\lambda_k = arg\ min_\lambda\ f(x^{(k)} - \lambda B_k^{-1} g_k)$；

更新变量$x^{(k+1)} = x^{(k)} - \lambda_k B_k^{-1} g_k$；

计算$g_{k+1} = g(x^{(k+1)}) = \nabla f(x^{(k+1)})$，若$||g(x^{(k+1)})|| \le \varepsilon$则停机，并置近似解$x^* = x^{(k+1)}$，否则更新矩阵$B_{k+1} = B_k + \frac{(g_{k+1}-g_k)(g_{k+1}-g_k)^T}{(g_{k+1}-g_k)^T(x^{(k+1)}-x^{(k)})} - \frac{B_k \delta_k \delta_k^T B_k}{\delta_k^T B_k \delta_k}$；

置 ***k=k+1*** 转下一次迭代；

7.12 最大熵模型与 IIS

7.12.1 最大熵模型

最大熵模型原理可表述为：在满足约束条件的模型集合中选取具有最大熵的模型作为待求模型的解；
模型集合：$\mathcal{C} = \{P \in \mathcal{P} | E_{\tilde{P}}[f_k] = E_P[f_k],\ k = 1, 2, \dots, K\}$；
其中$f_k(\mathrm{x}, \mathrm{y})$是描述随机变量 x 和 y 关系的特征函数：

$E_{\tilde{P}}[f_k] = \sum_{x,y} \tilde{P}(x, y) f_k(x, y), k = 1, 2, \dots, K$ 表示特征函数关于经验分布 $\tilde{P}(x, y)$的期望；

$E_P[f_k] = \sum_{x,y} \tilde{P}(x) P(y|x) f_k(x, y),\ k = 1, 2, \dots, K$ 表示特征函数关于模型和经验分布$\tilde{P}(x)$的期望；
将$E_{\tilde{P}}[f_k] = E_P[f_k]$作为满足模型要求的约束条件，而模型$P(Y|X)$的条件熵为：

$H(P) = -\sum_{x,y} \tilde{P}(x) P(y|x) log P(y|x)$；

因此得到最大熵模型的目标就是要在模型集合$\mathcal{C}$中寻找$H(P)$最大的模型：

$max_{P \in \mathcal{C}}\ H(P) = -\sum_{x,y} \tilde{P}(x) P(y|x) log P(y|x)$ ；

$s.t. E_{\tilde{P}}[f_k] = E_P[f_k],\ k = 1, 2, \dots, K$；

$\sum_y P(y|x) = 1$；

输入数据集$\mathbf{D} = \{(x_i, y_i)\}_{i=1}^{N}$ ，特征函数集$\{f_k(x, y)\}_{k=1}^{K}$ ；

最大熵模型的求解：

$min_{P \in \mathcal{C}} -H(P) = \sum_{x,y} \tilde{P}(x) P(y|x) log P(y|x)$ ；

$s.t. E_{\tilde{P}}[f_k] - E_P[f_k] = 0,\ k = 1, 2, \dots, K$；

$1 - \sum_y P(y|x) = 0$；

构造拉格朗日函数：

$L(P, w) = \sum_{x,y} \tilde{P}(x) P(y|x) log P(y|x) + \sum_{k=1}^{K} w_k \left[\sum_{x,y} \tilde{P}(x, y) f_k(x, y) - \right.$

$\left. \sum_{x,y} \tilde{P}(x) P(y|x) f_k(x, y)\right] + w_0 [1 - \sum_y P(y|x)]$；

原始最优化问题是极小-极大问题 $min_P\ max_w\ L(P, w)$，对偶问题为：
$max_w\ min_P\ L(w, P)$在满足 KKT 条件的情况下成立；
先求解极小化问题$\psi(w) = min_P\ L(w, P) = L(w, P_w)$称为对偶函数，对偶函数的极大化等价于模型条件概率$P(y|x)$的极大似然估计：

$L_{\tilde{P}}(P) = log[\prod_{x,y} P(y|x)^{\tilde{P}(x,y)}] = \sum_{x,y} \tilde{P}(x, y)\ log\ P(y|x)$：

$\frac{\partial L(P,w)}{\partial P} = \sum_{x,y}\widetilde{P}(x)[logP(y|x)+1] - \sum_{x,y}\widetilde{P}(x)\sum_{k=1}^{K} w_k f_k(x,y) - \sum_y w_0 =$

$\sum_{x,y}\widetilde{P}(x)\left[logP(y|x)+1-\sum_{k=1}^{K} w_k f_k(x,y) - w_0\right]$ ；

令$\frac{\partial L(P,w)}{\partial P} = 0$，得到：$P(y|x) = \frac{exp[\sum_{k=1}^{K} w_k f_k(x,y)]}{exp(1-w_0)}$；归一化处理，分子分母同除以1，由$\sum_y P(y|x) = 1$得到：

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

再求对偶函数$\psi(w) = min_P L(w,P) = L(w,P_w)$的极大化问题，拉格朗日乘子参数的解：

$w^* = arg\,max_w\ \psi(w)$；由求得的参数确定最大熵模型$P^* = P_{w^*}(y|x)$；

7.12.2 IIS

改进的迭代尺度法*(Improved Iterative Scaling)IIS* 用来解决最大熵模型(或其他对数线性模型如 *LR*)的模型学习问题，最大熵模型的学习问题实质上是以对数似然函数为目标函数的最优化问题：

$P_w(y|x) = \frac{exp[\sum_{k=1}^{K} w_k f_k(x,y)]}{\sum_y exp[\sum_{k=1}^{K} w_k f_k(x,y)]}$；令$Z_w(x) = \sum_y exp[\sum_{k=1}^{K} w_k f_k(x,y)]$；

则有：

$L(w) = log[\prod_{x,y} P_w(y|x)^{\widetilde{P}(x,y)}] =$

$\sum_{x,y}\widetilde{P}(x,y)\sum_{k=1}^{K} w_k f_k(x,y) - \sum_x \widetilde{P}(x)\,logZ_w(x)$；

IIS 寻找一组参数更新的增量$\delta = (\delta_1,\delta_2,\dots,\delta_K)^T$，参数向量在一次迭代后更新为$w+\delta = (w_1+\delta_1, w_2+\delta_2,\dots,w_K+\delta_K)^T$，对数似然函数$L(w)$增大，不断迭代求解，直到$L(w)$不再增大为止；

$L(w+\delta) - L(w)$对数似然函数增量式寻找其下界，得到：

$B(\delta|w) =$

$\sum_{x,y}\widetilde{P}(x,y)\sum_{k=1}^{K}\delta_k f_k(x,y) + 1 -$

$\sum_x \widetilde{P}(x)\sum_y P_w(y|x)\sum_{k=1}^{K}[\frac{f_k(x,y)}{f^{\#}(x,y)}]exp[\delta_k f^{\#}(x,y)]$;

其中特征计数$f^{\#}(x,y) = \sum_{k=1}^{K} f_k(x,y)$，令：

$\frac{\partial B(\delta|w)}{\partial \delta_k} = \sum_{x,y}\widetilde{P}(x,y)f_k(x,y) - \sum_x \widetilde{P}(x)\sum_y P_w(y|x)\,f_k(x,y)exp[\delta_k f^{\#}(x,y)] = 0$，得到参数更新方程：

$\sum_x \widetilde{P}(x)\sum_y P_w(y|x)\,f_k(x,y)exp[\delta_k f^{\#}(x,y)] = \sum_{x,y}\widetilde{P}(x,y)f_k(x,y) =$

$E_{\widetilde{P}}[f_k], k = 1,2,\dots,K$；

依次求解参数更新方程即可得到增量向量$\delta = (\delta_1,\delta_2,\dots,\delta_K)^T$；

对于参数更新方程的求解：

若特征计数值为常数$f^{\#}(x,y)=M$，则$\delta_k=\frac{1}{M}log\frac{E_{\tilde{P}}[f_k]}{E_P[f_k]}$；

当$f_k(x,y)$不是简单的二值函数$f_k(x,y)=\begin{cases}1,\ x,y\text{满足某条件；}\\0,\ \text{否则；}\end{cases}$，而与变量 x,y 有关，则$f^{\#}(x,y)=\sum_{k=1}^{K}f_k(x,y)$不是常数，则参数更新方程不能显式求解，应用数值方法求解：牛顿法或拟牛顿法；

7.12.3 熵/条件熵/互信息

熵*(Entropy)*：$H(Y)=-\sum_{y\in Y}P(y)\,log\,P(y)$；

其中$P(y)$是随机变量 Y 的概率分布，$H(Y)$表示随机变量 Y 分布的不确定性，要消除这种不确定性，唯一有效的方法就是从外部增加信息；

条件熵*(Conditional Entropy)*：$H(Y|X)=-\sum_{x\in X,y\in Y}P(x,y)\,log\,P(y|x)$；

$H(Y)\geq H(Y|X)$；

不等式代表额外引入的信息 X 使得 Y 的不确定性下降，当引入信息与 Y 毫不相关时，取等号；

互信息*(Mutual Information)*：$I(X,Y)=\sum_{x\in X,y\in Y}P(x,y)\,log\frac{P(x,y)}{P(x)P(y)}$；

互信息描述了两个信息之间的相关性；

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：

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

  $\boldsymbol{\theta \leftarrow \theta - \eta v_t}$；

  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

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

  $\boldsymbol{\theta \leftarrow \theta - \eta v_t}$；
- 自适应学习速率梯度下降 ***Adagrad***：对学习速率$\boldsymbol{\eta}$进行约束

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

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

  $\boldsymbol{\theta \leftarrow \theta + \Delta\theta_t}$ ；

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

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

  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$\boldsymbol{\eta}$；
- ***Adadelta*** 扩展了 ***Adagrad*** 方法：

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

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

  继续处理如下：

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

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

  不再依赖于全局设定的学习速率$\boldsymbol{\eta}$，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
  $\mathbf{E}|g^2|_t = \rho E|g^2|_{t-1} + (1-\rho)g_t^2$；令$\rho = 0.5$，则变成对梯度$g_t^2$求算数均值，对$\mathbf{E}|g^2|_t$再度求根式，可得到

  $\Delta\theta_t = -\frac{\eta}{\sqrt{\mathbf{E}|g^2|_t+\varepsilon}}$ ；依赖于全局学习速率η，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*

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

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

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

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$\boldsymbol{\omega^T x + b = 0}$，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：

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

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

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**)：

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

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

构造拉格朗日函数：

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

对模型参数$\boldsymbol{\omega, b}$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：

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

原问题为 $\boldsymbol{min_{\omega,b}\, max_\alpha\ L(\omega, b, \alpha)}$ 在满足 KKT 条件的前提下，原始问题与其对偶问题 $\boldsymbol{max_\alpha\, min_{\omega,b}\ L(\alpha, \omega, b)}$ 具有相同解，问题转换为：

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

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

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

并满足 KKT 条件：

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

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

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

KKT 条件限制了非支持向量点不会出现在求和式$\boldsymbol{\sum_{i=1}^{N} \alpha_i(-y_i(\omega^T x_i + b) + 1)}$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，$\boldsymbol{\alpha_i > 0}$ ，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；

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

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

$\alpha_i \geq 0$ ； $i = 1,2,\dots,N$

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：

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

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 $y_s\left(\sum_{i\in S}\alpha_i y_i x_i^T x_s + b\right) = 1$ ；

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

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 ξ_i ， $i = 1,2,\dots N$ ，得到优化问题：

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

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

$\xi_i \geq 0$ ；

构造拉格朗日函数，对参数ω, b, ξ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：

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

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

$C - \alpha_i - \mu_i = 0$ ；

$\mu_i \geq 0$ ；

$\alpha_i \geq 0$ ； $i = 1,2,\dots,N$

满足 KKT 条件：

$\alpha_i \geq 0$ ； $\mu_i \geq 0$ ；

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

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

$\xi_i \geq 0$ ； $\mu_i\xi_i = 0$ ；

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：

$\emptyset(\boldsymbol{x})$: $\boldsymbol{\chi \rightarrow \mathcal{H}}$；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 $\boldsymbol{x_i, x_j \in \chi}$，存在核函数$\boldsymbol{K(x_i, x_j) = \emptyset(x_i) \cdot \emptyset(x_j)}$ 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：

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

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

$\boldsymbol{C \geq \alpha_i \geq 0}$ ； $\mathbf{i = 1, 2, \dots, N}$

得到最优解$\boldsymbol{\alpha^* = (\alpha_1^*, \alpha_2^*, \dots, \alpha_m^*)^T}$ ；选择其中一个正分量$\boldsymbol{C \geq \alpha_j \geq 0}$ 对应的样本点$\boldsymbol{(x_j, y_j)}$，求偏置量$\boldsymbol{b^* = y_j - \sum_{i=1}^{N}\alpha_i^* y_i K(x_i, x_j)}$ ；

最终得到的决策函数为：

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

当核函数$\boldsymbol{K(x_i, x_j)}$**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

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

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

- 若$\boldsymbol{loss}[\boldsymbol{f}(\boldsymbol{x_i}),\boldsymbol{y_i}]=[\boldsymbol{f}(\boldsymbol{x_i})-\boldsymbol{y_i}]^2$ 平方误差，模型为最小二乘；
- 若$\boldsymbol{loss}[\boldsymbol{f}(\boldsymbol{x_i}),\boldsymbol{y_i}]=\{\boldsymbol{0},\boldsymbol{1}-\boldsymbol{y_i}\boldsymbol{f}(\boldsymbol{x_i})\}$ ***hinge*** 损失，模型为支持向量机；
- 若$\boldsymbol{loss}[\boldsymbol{f}(\boldsymbol{x_i}),\boldsymbol{y_i}]=\boldsymbol{exp}(-\boldsymbol{y_i}\boldsymbol{f}(\boldsymbol{x_i}))$ 指数损失，模型为 ***Boosting***，基分类器的指数损失函数为：$\boldsymbol{\ell_{exp}}(\boldsymbol{\alpha_t}\boldsymbol{h_t}|\boldsymbol{D_t})=\boldsymbol{E_{x\sim D_t}}\left[\boldsymbol{exp}\left(-\boldsymbol{y}\boldsymbol{\alpha_t}\boldsymbol{h_t}(\boldsymbol{x})\right)\right]$

  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：

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

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

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

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

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

$\boldsymbol{\lambda\Omega}(\boldsymbol{\omega})$为正则化项，$\boldsymbol{L_p}$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$\boldsymbol{\omega}=(\boldsymbol{\omega^1},\ \boldsymbol{\omega^2},\dots,\boldsymbol{\omega^m})$，则$\boldsymbol{L_0}$**范数**$||\boldsymbol{\omega}||_0$ **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$\boldsymbol{L_0}$范数非凸、不连续，数学性质不好，难于求解；

$\boldsymbol{L_1}$**范数**$||\boldsymbol{\omega}||_1=\sum_{i=1}^{m}|\boldsymbol{\omega^i}|$**，称为稀疏规则算子，对**$\boldsymbol{L_1}$**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；**

$\boldsymbol{L_2}$**范数**$||\boldsymbol{\omega}||_2=\sqrt{\sum_{i=1}^{m}(\boldsymbol{\omega^i})^2}$ **，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**$\boldsymbol{L_2}$**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：

$\boldsymbol{\kappa}(\boldsymbol{W})=||\boldsymbol{W}||\cdot||\boldsymbol{W^{-1}}||$ ；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$\boldsymbol{\kappa}(\boldsymbol{W})$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：

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

$\boldsymbol{\omega^* = (X^T X + \lambda I)^{-1} X^T y}$，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**$\boldsymbol{\lambda}$强凸性质**的函数：

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

而一般的凸函数定义为：

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

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$\boldsymbol{\omega^*}$，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$\boldsymbol{L_2}$ 范数作为正则化项 $\frac{\lambda}{2}||\boldsymbol{\omega}||^2$，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$\boldsymbol{L_1}$和$\boldsymbol{L_2}$规则化的优化问题：

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

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

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***：
$\boldsymbol{min_\omega\ f(\omega) + \lambda||\omega||_1}$ ；
若$\boldsymbol{f(\omega)}$可导，且$\boldsymbol{\nabla f(\omega) = \frac{\partial f(\omega)}{\partial \omega}}$满足 ***L-Lipschitz*** 条件：

$$\boldsymbol{\exists L > 0, \left|\left|\nabla f(\omega_1) - \nabla f(\omega_0)\right|\right|_2^2 \leq L\left|\left|\omega_1 - \omega_0\right|\right|_2^2\ (\forall \omega_1, \omega_0)}\ ；$$

将$\boldsymbol{f(\omega)}$二阶泰勒展开：

$$\boldsymbol{\hat{f}(\omega) \cong f(\omega_k) + [\nabla f(\omega_k)]^T \cdot (\omega - \omega_k) + \frac{L}{2}\left|\left|\omega - \omega_k\right|\right|_2^2}$$

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

即可得到最优化问题：

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

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

$$\boldsymbol{\omega_{k+1} = argmin_\omega \frac{L}{2}\left|\left|\omega - z\right|\right|_2^2 + \lambda||\omega||_1}；$$

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$\boldsymbol{P(X|\theta)}$，其中$\boldsymbol{X}$**是已观测属性，**$\boldsymbol{\theta}$**是模型参数，此外还有未观测属性**$\boldsymbol{R}$**作为隐含变量；**$\boldsymbol{P(X|\theta)}$**是不完全数据**$\boldsymbol{X}$**的边缘概率：**
$\boldsymbol{P(X|\theta) = \sum_R P(X,R|\theta) = \sum_R P(R|\theta)P(X|R,\theta)}$**；**
由不完全数据**边缘概率的对数似然函数**$\boldsymbol{L(\theta) = log[P(X|\theta)]}$**极大化的迭代求解**可以导出**完全数据的对数似然函数**$\boldsymbol{log[P(X,R|\theta)]}$关于在给定**观测数据**$\boldsymbol{X}$**和当前参数估计**$\boldsymbol{\theta^{(t)}}$**下对未观测属性(隐含变量)**$\boldsymbol{R}$**依条件概率**$\boldsymbol{P(R|X,\theta^{(t)})}$**的数学期望：**

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

称此期望为 Q 函数，即有：

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

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

EM 算法分为两步进行：

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

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

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 $\boldsymbol{x_i}, \boldsymbol{i=1,2,\dots,N}$ 赋一个权重$\boldsymbol{w_i}, \boldsymbol{i=1,2,\dots,N}$，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$\boldsymbol{h_t(x)}$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$\boldsymbol{\alpha_t}$，$\boldsymbol{t=1,2,\dots,T}$；最终得到一个线性加性模型$\boldsymbol{H(x)=\sum_{t=1}^{T} h_t(x)}$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$\boldsymbol{w_{1i}=\frac{1}{N}}$，并训练第一个基分类器 $\boldsymbol{h_1(x)}$；

(2) 迭代式：对第 ***t (t=1,2,…,T)***轮迭代，得到基分类器$\boldsymbol{h_t(x)}$，此时有组合分类器 $\boldsymbol{G_t(x)=\sum_{\tau=1}^{t} h_\tau(x)}$，对训练数据集中样本的分类误差为$\boldsymbol{\varepsilon_t=\sum_{i=1}^{N} w_{t,i} I(G_t(x_i)\neq y_i)}$，而**基分类器** $\boldsymbol{h_t(x)}$**的权重为** $\boldsymbol{\alpha_t=\frac{1}{2}ln(\frac{1-\varepsilon_t}{\varepsilon_t})}$，由**最小化指数损失函数**$\boldsymbol{\ell_{exp}(\alpha_t h_t|D_t)=\exp(-\alpha_t)(1-\varepsilon_t)+\exp(\alpha_t)\varepsilon_t}$ 求得；

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

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

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$\boldsymbol{r_{t,i}=-\left[\frac{\partial \ell(y_i,G(x_i))}{\partial G(x_i)}\right]|_{G(x)=G_{t-1}(x)}}$又称为**伪残差，是回归问题中的残差** $\boldsymbol{r_{t,i}=y_i-G_{t-1}(x_i)}$**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：$\boldsymbol{G_0(x)=\gamma_0 h_0(x)}$；$\boldsymbol{h_0(x)=1}$；

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

(2) 迭代式：对于第 ***t (t=1,2,…,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 ***deepth<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：

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

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

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,…,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：

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

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：

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

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：

$$\mathcal{L}_t\big(y, G(x)\big)$$

$$= \sum_{i=1}^{N} [\ell\big(y_i, G_{t-1}(x_i)\big) + \frac{\partial \ell\big(y_i,\ G(x_i)\big)}{\partial G(x_i)}|_{G(x)=G_{t-1}(x)} f_t(x_i)$$

$$+ \frac{1}{2}\frac{\partial^2 \ell\big(y_i,\ G(x_i)\big)}{\partial G^2(x_i)}|_{G(x)=G_{t-1}(x)} {f_t}^2(x_i)] + \Omega(f_t) + const$$

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

$$L_t\big(y, G(x)\big)$$

$$= \sum_{i=1}^{N} [\frac{\partial \ell\big(y_i,\ G(x_i)\big)}{\partial G(x_i)}|_{G(x)=G_{t-1}(x)} f_t(x_i)$$

$$+ \frac{1}{2}\frac{\partial^2 \ell\big(y_i,\ G(x_i)\big)}{\partial G^2(x_i)}|_{G(x)=G_{t-1}(x)} {f_t}^2(x_i)] + \Omega(f_t)$$

将每一次迭代得到的决策树$f_t(x)$表示为：

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

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：

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

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

则有：

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

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：

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

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**γ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：$\boldsymbol{P(Y|X;\theta)}$；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：$\boldsymbol{P(Y^1,Y^2,\dots,Y^T|X^1,X^2,\dots,X^T;\Theta)}$；作为分类问题的推广，标注问题的输入$\boldsymbol{\mathcal{X}=(X^1,X^2,\dots,X^T)}$**是一个观测序列，输出**$\boldsymbol{\mathcal{Y}=(Y^1,Y^2,\dots,Y^T)}$**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：$\boldsymbol{\hat{y}=f(x;\omega)}$；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

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

  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，选择数据集中的一个核心对象作为种子，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临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$o \in \Omega, Q = < o >$, $\Gamma = \Gamma \backslash \{o\}$；

  $while\ Q \neq \emptyset$：
  $q = Q.head$；取出队列的 ***head*** 元素；
  若 q 是一个核心对象，即$|N_\epsilon(q)| > MinPts$，则将其临域中未被访问过得样本点均加入队列，并将其从 Γ中剔除：

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

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，这些样本点中有的是核心对象，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$\{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1}^{N}$，**其中$m$维行向量**$\boldsymbol{x_i} = (\boldsymbol{x_i^{(1)}}, \boldsymbol{x_i^{(2)}}, \dots, \boldsymbol{x_i^{(m)}})$定义了样本$\boldsymbol{i}$的属性信息，包含$\boldsymbol{m}$个特征，$\boldsymbol{x_i^{(l)}} \in \mathcal{R}_l$，$\mathcal{R}_l$**为第$l$个特征的取值空间**$(\boldsymbol{l} = \boldsymbol{1}, \boldsymbol{2}, \dots, \boldsymbol{m})$，则分类器要完成所有特征的学习并找出其与样本标签$\boldsymbol{y_i}$的关系，至少需要$\prod_{l=1}^{m} |\mathcal{R}_l|$个样本，当$\boldsymbol{m}$**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$\boldsymbol{A} = \{\boldsymbol{a_l}\}_{l=1}^{m}$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$\boldsymbol{A_s} = \{\boldsymbol{a_1}\}$作为初始特征子集，逐步选择候选特征$\boldsymbol{a_l}\ \boldsymbol{for\ l\ in\ range(m)}$，在已划分子集$\boldsymbol{D_s}$ $(\boldsymbol{D} = \cup_s \boldsymbol{D_s})$上计算以新增特征$\boldsymbol{a_l}$**进行划分**的信息增益$\boldsymbol{g(D_s\,|a_l) = H(D_s) - \sum_v^{|\mathcal{R}_l|} \frac{|D_s^{(l),v}|}{|D_s|} H(D_s^{(l),v})}$，**若$\boldsymbol{g(D_s|a_l) > 0}$则在当前节点上以候选特征$\boldsymbol{a_l}$进行划分得到子集$\{\boldsymbol{D_s^{(l),v}}\}_v^{|\mathcal{R}_l|}$，并更新特征子集$\boldsymbol{A_s = A_s \cup \{a_l\}}$和划分子集$\boldsymbol{D_s = \{D_s^{(l),v}\}_v^{|\mathcal{R}_l|}}$，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$\boldsymbol{L_1}$正则化，易于得到稀疏解，仅选择权值向量$\boldsymbol{\omega}$**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：

$$\boldsymbol{Av = \lambda v;}$$

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：

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

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

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

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

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

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

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

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

特征值分解可以将矩阵写成$\boldsymbol{A = Q\Sigma Q^{-1}}$，其中$\boldsymbol{\Sigma = diag(\lambda_1, \lambda_2, \dots, \lambda_d)}$是由矩阵 A 的特征值组成的对角阵，$\boldsymbol{Q = [v_1, v_2, \dots, v_d]}$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针对方阵提取其特征，奇异值分解***(Singular Value Decomposition)SVD*** 适用于任意矩阵：

$\boldsymbol{A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T{}_{n\times n}}$ ；

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

$\boldsymbol{(A^TA)v_i = \lambda_i v_i}$

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

$\boldsymbol{u_i = \frac{1}{\sigma_i}Av_i}$ ；

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$\boldsymbol{\Sigma_{m\times n}}$**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：**

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

要存储一个较大的矩阵$\boldsymbol{A_{m\times n}}$，通过 ***SVD*** 运算只需要存储得到的三个较小的矩阵$\boldsymbol{U_{m\times r}}$，$\boldsymbol{\Sigma_{r\times r}}$，$\boldsymbol{V^T{}_{r\times n}}$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***

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

通过线性变换$\boldsymbol{P_{m\times m}}$**转换成**$\boldsymbol{\tilde{D}}$：$\boldsymbol{\tilde{D}_{N\times m} = D_{N\times m}P_{m\times m}}$，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**$\boldsymbol{P_{m\times m}}$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：$\boldsymbol{\tilde{D}_{N\times r} = D_{N\times m}P_{m\times r}}$ ，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$\boldsymbol{P_{m\times r}}$后新得到的$\boldsymbol{\tilde{D}_{N\times r}}$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：

SVD: $\boldsymbol{D_{N\times m} \cong U_{N\times r}\Sigma_{r\times r}V^T{}_{r\times m}}$ ；

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

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

其中 $\boldsymbol{V^T{}_{r\times m}V_{m\times r} = I_{r\times r}}$ ；

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

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$\boldsymbol{\sigma_1 > \sigma_2 > \sigma_3 > \cdots > \sigma_d}$，从中选择前面 ***r*** 个奇异值，使得 $\boldsymbol{\frac{\sum_{j=1}^{r}\sigma_j}{\sum_{j=1}^{d}\sigma_j} > 99\%}$，以$\boldsymbol{\{\sigma_1, \sigma_2, \dots, \sigma_r\}}$对应的左奇异向量$\boldsymbol{\{u_j = \frac{1}{\sigma_j}Dv_j\}_{j=1}^{r}}$构成变换矩阵$\boldsymbol{U_{N\times r} = (u_1, u_2, \dots, u_r)}$，并生成对角矩阵：

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

进而得到对原数据集的主成分抽取：

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

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$\boldsymbol{D_{N\times m}}$进行行压缩：
$\boldsymbol{Q_{r\times N}D_{N\times m}=\widetilde{D}_{N\times r}}$ ；
观察 ***SVD***：$\boldsymbol{D_{N\times m}\cong U_{N\times r}\Sigma_{r\times r}V^{T}{}_{r\times m}}$ ；
令 $\boldsymbol{Q_{r\times N}=U^{T}_{r\times N}}$ 得到：
$\boldsymbol{Q_{r\times N}D_{N\times m}\cong(U^{T}_{r\times N}U_{N\times r})\Sigma_{r\times r}V^{T}{}_{r\times m}}$ ，其中：$\boldsymbol{U^{T}_{r\times N}U_{N\times r}=I_{r\times r}}$ ；
则有$\boldsymbol{Q_{r\times N}D_{N\times m}\cong\Sigma_{r\times r}V^{T}{}_{r\times m}}$ ；

从矩阵$\boldsymbol{D_{N\times m}}$的奇异值中选取前 $\boldsymbol{r}$ 个后，以其对应的右奇异向量$\boldsymbol{\{v_j\}_{j=1}^{r}}$构成矩阵并转置得到变换矩阵$\boldsymbol{V^{T}{}_{r\times m}=(v_1,v_2,\dots,v_r)^{T}}$；
对角矩阵$\boldsymbol{\Sigma_{r\times r}=diag(\sigma_1,\sigma_2,\dots,\sigma_r)}$，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$A \to B$的支持度：$support(A \to B) = P(AB)$；

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

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：

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

7.22.2 Apriori 算法

Apriori 算法用来生成频繁项集，供规则挖掘使用，仅使用最小支持度阈值α作为参数对原始数据集进行搜索：

- 初始扫描数据集，构建候选项集$C_1 = \{\{Item_i\}\}_{i=1}^{m}$，其中 m 表示数据集中包含的所有条目种类数，C_1是单条目集合，$\{Item_i\} \cap \{Item_j\} = \emptyset, (i \neq j)$；
- 扫描候选项集C_1，将支持度低于阈值的项集剔除，得到支持项集(频繁-1 项集)：

  $L_1 = \{\{Item_j\}\}_{support(\{Item_j\})>\alpha}^{1 \le j \le m}$ ；
- 利用 *Apriori* 性质：频繁项集的子集一定是频繁的，而非频繁项集的超集必是非频繁的，在频繁-(k-1)项集L_{k-1}中寻找频繁 k 项集L_k：

  (1) $L_{k-1} = \{l[k-1]_j\}_{j>0}$，连接$L_{k-1}$中的子集：

  若$|l[k-1]_j \cap l[k-1]_i| = k-2, i \neq j$ 则将子集$l[k-1]_i$与$l[k-1]_j$合并生成$c[k]_t$，t 的数值由外循环决定，连接步生成候选项集

  $C_k = \{c[k]_j\}_{j>0}$ ；

  (2) 对候选项集$C_k = \{c[k]_j\}_{j>0}$进行剪枝：若存在$c[k]_j$的 k-1 项子集$c[k-1]_j$在频繁-(k-1)项集L_{k-1}中不存在，则直接将$c[k]_j$从候选项集中剔除：

  $C_k = C_k \backslash c[k]_j \ if \ exists \ c[k-1]_j \ not \ in \ L_{k-1}$ ；

  (3) 对所有候选子集$c[k]_j$计算其支持度，若$support(c[k]_j) < \alpha$，则将其从候选项集C_k中剔除，否则加入频繁-k 项集L_k中；

Apriori 算法的问题：产生候选项集时，循环产生的组合过多，没有直接排除不应加入候选项集的组合；计算支持度的时候，需要遍历整个数据集，增大了 *I/O* 开

销；
优化策略：
在逐层搜索生成频繁项集的第 ***k-1*** 步，统计L_{k-1}中的各个条目出现的次数，将出现次数少于$k-1$次的条目所在项集删除，从而排除由此条目引起的一定数量的组合，使用了频繁项集性质：L_k频繁项集中的所有条目在其L_{k-1}**频繁项集中至少出现 *k-1* 次(因为**L_k**频繁项集的 *k-1* 阶子集一定是频繁的)；**
同时对原始数据库 ***D*** 进行更新：在第 ***k*** 步生成候选项集C_k后，扫描数据库，查看是否所有事务均包含至少一个候选项集C_k中的子集，若不满足，则将此事务移至数据库末尾并加做标记，扫描完成后，从 ***D*** 中剔除做标记的事务得到新的数据集D'，随着 ***k*** 逐渐增大，D'的规模越来越小，减小在扫描事务时的 ***I/O*** 开销；

7.22.3 FP-Growth 算法

FP-Growth 算法***(Frequent Pattern)***首先通过对数据集的两次扫描构造 ***FP-Tree***：

- 第一遍扫描：遍历数据集，统计各个条目出现的次数，只保留出现次数大于支持度阈值α的条目(包含此类条目的项集肯定不是频繁的)，使用一个字典进行存储 **HeaderTable={key = name : value = count,…}**，并按照条目出现的频率进行排序；
- 第二遍扫描：读入每个项集并将其添加到一条已存在的路径，若路径不存在，则创建，在将事务添加到树中时，要按照 **HeaderTable** 中条目出现的频率先对事务中的条目进行排序，并且，相同的事务只在 **FP-Tree** 中出现一次；

根节点$\mathbf{root} = \emptyset$，遍历筛选和排序后的事务数据集，向 **FP-Tree** 中不断添加事务，若树中已存在被添加事务中的条目，则对此条目所在节点进行增 **1** 操作 **TreeNode.increment(1)，否则创建分支；**
基于 FP-Tree 挖掘频繁项集：

- 从 **FP-Tree** 中提取条件模式基(**Conditional Pattern Base**)：从待查找项所在节点到根节点的全部前缀路径组成的集合，遍历 **HeaderTable** 中的所有条目，找到其在 **FP-Tree** 中对应的节点回溯至根节点，记录此路径及其频率(**TreeNode.count**)；
- 创建条件树：以条件模式基为输入数据，首先对单条目项集构建条件树，删除支持度低于阈值α**的项集**，生成树；然后递归对 2 条目项集，多条目项集递归进行构建条件树，直至树中没有节点为止，即找到了最大频繁项集；

7.23 词频-逆文本词频

词频-逆文本词频*(Term Frequency – Inverse Document Frequency)TF-IDF*，是在文本向量化后的一种纠正措施：

文本向量化 *word2vec*：统计 ***N*** 篇文档**(语料库)**$\{\boldsymbol{X_i}\}_{\boldsymbol{i=1}}^{\boldsymbol{N}}$中各个单词在字典$\boldsymbol{\mathcal{W}} = (\boldsymbol{w_1}, \boldsymbol{w_2}, \dots, \boldsymbol{w_K})^{\boldsymbol{T}}$中出现的位置及次数，生成与字典相同维度的向量：

$\boldsymbol{X_i} = (\boldsymbol{x_i^{(1)}}, \boldsymbol{x_i^{(2)}}, \dots, \boldsymbol{x_i^{(K)}})^{\boldsymbol{T}}, \boldsymbol{i} = \boldsymbol{1}, \boldsymbol{2}, \dots, \boldsymbol{N}$；

其中$\boldsymbol{x_i^{(k)}}$表示在文档$\boldsymbol{X_i}$中对应字典$\boldsymbol{\mathcal{W}}$中的单词$\boldsymbol{w_k}$出现的次数；

文本向量化后得到的文本向量$\boldsymbol{X_i} = (\boldsymbol{x_i^{(1)}}, \boldsymbol{x_i^{(2)}}, \dots, \boldsymbol{x_i^{(K)}})^{\boldsymbol{T}}$中包含的单词出现次数并不能反应单词的重要程度(包含停用词以及其他副词介词等)，因此采用 ***TF-IDF*** 指标修正文本向量；

词频定义 $\boldsymbol{TF[x_i^{(k)}] = \frac{x_i^{(k)}}{\sum_{k=1}^{K} x_i^{(k)}}}$ **；**

逆文本词频定义 $\boldsymbol{IDF[w_k] = log\frac{N}{N(w_k)}}$ **；其中$\boldsymbol{N(w_k)}$表示出现单词$\boldsymbol{w_k}$的文档数，为避免出现分母为 0 的情况，使用拉普拉斯平滑：**

$\boldsymbol{IDF[w_k] = log\frac{N+1}{N(w_k)+1} + 1}$；

则修正文本向量得到：

$\boldsymbol{X_i} = (\boldsymbol{IDF[w_1] \cdot TF[x_i^{(1)}], IDF[w_2] \cdot TF[x_i^{(2)}], \dots, IDF[w_K] \cdot TF[x_i^{(K)}])^T, i = 1, 2, \dots, N}$；

7.24 Page-Rank

Page-Rank 算法基于网站接入链接的数量和质量假设：

- 在 ***Web*** 的图模型中，接入当前页面的链接数量越多，其重要性越高；
- 接入当前页面的源链接页面的重要性越高，其对当前页面重要程度的贡献越高；

7.24.1 爬虫与倒排索引

爬虫与倒排索引用于建立资料库***(key-value)***，其中 ***key*** 为页面的 ***url***，***value*** 为页面的内容，生成 ***Page-Rank*** 算法使用的数据集：

- 爬虫***(Crawler)***：从起始页面出发，通过 ***HTTP*** 协议获得页面的全部字段，将页面的 ***url*** 记录到 ***key*** 域中，将其内容记录到 ***value*** 域中，并分析内容中包含的链接，使用这些链接指向新的页面，重复上述过程可得到 ***Web*** 中的几乎所有的页面；
- 倒排索引***(Inversed Index)***：以关键字为 ***key***，以包含关键字的页面编号集合作为 ***value*** 建立 ***key-value*** 数据结构；

7.24.2 PR 迭代更新

假设整个 ***Web*** 是一个**强连通**的**图模型**，即从任一页面出发可转移到任意其他页面，转移概率矩阵：

$\boldsymbol{M}=\begin{bmatrix}\boldsymbol{m_{11}} & \cdots & \boldsymbol{m_{1N}}\\ \vdots & \ddots & \vdots\\ \boldsymbol{m_{N1}} & \cdots & \boldsymbol{m_{NN}}\end{bmatrix}$，**其中**$\boldsymbol{m_{ij}}$**表示页面** ***i*** **是从页面** ***j*** **转移而来的概率，且有：**

$\sum_{j=1}^{N}\boldsymbol{m_{i,j}}=\boldsymbol{1},\boldsymbol{i}=\boldsymbol{1},\boldsymbol{2},\ldots,\boldsymbol{N}$**；**

初始化每个页面的 ***PageRank*** 数值为$\boldsymbol{v_i}=\frac{\boldsymbol{1}}{\boldsymbol{N}}$,$\boldsymbol{i}=\boldsymbol{1},\boldsymbol{2},\ldots,\boldsymbol{N}$ ，$\boldsymbol{v}=\begin{bmatrix}\frac{\boldsymbol{1}}{\boldsymbol{N}}\\ \vdots\\ \frac{\boldsymbol{1}}{\boldsymbol{N}}\end{bmatrix}$，**转移概率矩阵**$\boldsymbol{M}$**中的第** ***i*** **行表示从各个页面转移到页面** ***i*** **的概率，而向量** ***v*** **表示各个页面的** ***Page-Rank*** **值，因此更新页面的** ***Page-Rank*** **为**$\boldsymbol{v_i}=\sum_{j=1}^{N}\boldsymbol{m_{ij}v_j}$,$\boldsymbol{i}=\boldsymbol{1},\boldsymbol{2},\ldots,\boldsymbol{N}$，**即：**$\boldsymbol{v}\leftarrow\boldsymbol{Mv}$ ；重复上述过程可以收敛于一个稳定的 ***Page-Rank*** 向量，即最终页面的排名，在每一步迭代中，对 ***Page-Rank*** 向量$\boldsymbol{v}=[\boldsymbol{v_i}]_{N\times1}$ 均有$\sum_{i=1}^{N}\boldsymbol{v_i}=\boldsymbol{1}$**保证了整个页面迭代更新过程是收敛的**；

而转移概率矩阵$\boldsymbol{M}=[\boldsymbol{m_{ij}}]_{N\times N}$**的计算遵循：**

$\boldsymbol{m_{ij}}$**表示页面** ***j*** **转移到页面** ***i*** **的概率，若页面** ***j*** **中存在** ***K*** **个外部链接，称其出度为**

K，这K个页面均分页面j的转移概率，则页面j到页面i的转移概率为 $m_{ij}=\frac{1}{N}$ ；

为了克服迭代过程中的 ***Spider Traps*** 和稀疏矩阵连续相乘带来的 ***Page-Rank*** 向量极度不平滑问题，对向量迭代公式进行修正：

$v'=(1-\beta)Mv+\frac{\beta}{N}e, e=[1]_{N\times 1}$ ；其中β设置为一个较小的数值，$\frac{\beta}{N}e$称为心灵转移概率向量；

7.24.3 话题倾向排序

话题倾向排序 *Topic-Sensitive*：先验地得到一个话题集合$\mathcal{T}$，对每一个页面进行话题归属划分(典型的分类问题)，可以使用聚类算法，也可以使用 ***TF-IDF*** 基于关键词分类，分类后得到多个 ***Topic*** 向量，将 ***Page-Rank*** 向量迭代公式$v'=(1-\beta)Mv+\frac{\beta}{N}e, e=[1]_{N\times 1}$变为：

$v'=(1-\beta)Mv+\frac{\beta}{||s_t||_0}s_t$ ，其中话题向量$s_t=[s_{tk}]_{N\times 1}, t\in\mathcal{T}$，$s_{tk}=1$表明页面 k 归属于此话题，$s_{tk}=0$表明页面 k 不属于此话题，$||s_t||_0$是向量s_t的L_0范数；而推荐给用户的话题可由用户使用浏览器的 ***cookie*** 收集追踪得到，也可以由用户的使用习惯进行偏好分析；

7.25 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$\boldsymbol{P(Y,R|X)}$，生成模型通过$\boldsymbol{P(Y,R,X)}$**得到条件概率**$\boldsymbol{P(Y,R|X)}$，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$\boldsymbol{P(Y|X)}$；
其中，判别模型直接考虑条件概率$\boldsymbol{P(Y|X)=\sum_R P(Y,R|X)}$；
生成模型无法直接获得条件概率，通过贝叶斯公式得到

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

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$\boldsymbol{P(X|\theta)}$，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$\boldsymbol{O(|\mathcal{X}|^{|X|}|\mathcal{R}|^{|R|})}$，**其中**$\boldsymbol{\mathcal{X}}$**为样本观测特征取值空间，**$\boldsymbol{\mathcal{R}}$**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**$\boldsymbol{P(Y|X)}$**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**$\boldsymbol{\lambda=(A,B,\pi)}$**后，生成状态序列**$\boldsymbol{R=(r_1,r_2,\dots,r_T)}$**，再由状态序列中的每一时刻状态生成对应的输出观测：**$\boldsymbol{\lambda\rightarrow r_t\rightarrow x_t,t=1,2,\dots,T}$**；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**$\boldsymbol{X=(X_1,X_2,\dots,X_n)}$**求其对应的输出标记序列**$\boldsymbol{Y=(Y_1,Y_2,\dots,Y_n)}$**的条件概率**$\boldsymbol{P(Y|X)}$**，而标记**$\boldsymbol{Y_i}$**不仅与整个输入观测序列**$\boldsymbol{X=(X_1,X_2,\dots,X_n)}$**有关，还与其它标记**$\boldsymbol{Y_{j\neq i}}$**有关，当 CRF 为线性链式结构时：**
  $\boldsymbol{P(Y_i|X,Y_1,\dots,Y_{i-1},Y_{i+1},\dots,Y_n)=P(Y_i|X,Y_{i-1},Y_{i+1}),i=1,2,\dots,n}$；

7.25.1 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$\boldsymbol{R=(r_1,r_2,\dots,r_T)}$，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$\boldsymbol{X=(x_1,x_2,\dots,x_T)}$；此时，模型的不可观测状态变量$\boldsymbol{r_t\in\mathbb{S}}$，$\boldsymbol{(t=1,2,\dots,T)}$与输出观测变量$\boldsymbol{x_t\in\mathbb{O}}$，$\boldsymbol{(t=1,2,\dots,T)}$**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**$\boldsymbol{\mathbb{S}=\{s_1,s_2,\dots,s_N\}}$**，观测空间**$\boldsymbol{\mathbb{O}=\{o_1,o_2,\dots,o_M\}}$**，*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,…,T)*，观测变量$\boldsymbol{x_t}$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：

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

HMM 由参数$\lambda = (A, B, \pi)$确定，其中π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$\mathbb{S} = \{s_1, s_2, \dots, s_N\}$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x_{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：$\alpha_1(i) = \pi_i b_1(x_1), i = 1, 2, \dots, N$; **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：$\alpha_t(i) = \left[\sum_{j=1}^{N} \alpha_{t-1}(j)a_{ji}\right] b_i(x_t), i = 1, 2, \dots, N$; **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**$[a_{.i}]_{N\times 1}$**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$[a_{i.}]_{1\times N}$，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 $P^{(p)}(X, r_1 = s_i|\bar{\lambda})$，$P^{(p)}(X, r_t = s_i, r_{t+1} = s_j|\bar{\lambda})$，$P^{(p)}(X, r_t = s_i|\bar{\lambda})$，$P^{(p)}(X, r_t = s_j, x_t = o_k|\bar{\lambda})$；

更新参数

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

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

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

直至达到最大迭代轮数为止；

- 预测(解码)问题：给定模型参数$\boldsymbol{\lambda} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{\pi})$和观测序列$\boldsymbol{X} = (\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_T})$求最有可能产生当前观测序列的状态序列$\boldsymbol{R} = (\boldsymbol{r_1}, \boldsymbol{r_2}, \dots, \boldsymbol{r_T})$，即使得条件概率$\boldsymbol{P(R|X)}$最大的状态序列$\boldsymbol{R} = (\boldsymbol{r_1}, \boldsymbol{r_2}, \dots, \boldsymbol{r_T})$；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$\boldsymbol{r_t = s_i}$的各条部分路径的最大概率：

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

  $$\boldsymbol{i = 1, 2, \dots, N;}$$

  由此得到最优路径的递推式：

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

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

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

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

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

7.25.2 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；

$\boldsymbol{P(x) = \frac{1}{Z^*}\prod_{Q \in C^*} \psi_Q(x_Q)}$；其中$\boldsymbol{Z^* = \sum_x \prod_{Q \in C^*} \psi_Q(x_Q)}$为概率归一化因子，表示对所有随机变量的$\boldsymbol{\prod_{Q \in C^*} \psi_Q(x_Q)}$求和；

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

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

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

7.25.3 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$\widehat{P}(Y|X)$；
- 给定输入观测序列x，求使得条件概率$\widehat{P}(y|x)$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，λ_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,…,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$\mathcal{Y}$，$\mathrm{m} = |\mathcal{Y}|$，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,…,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,…,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$\mathcal{Y}$，$\mathrm{m} = |\mathcal{Y}|$，所以$\alpha_i(y_i|x)$和$\beta_i(y_i|x)$都是 m 阶向量，***i=1,2,…,n***;
  则可得到$P(Y_i = y_i|x) = \frac{\alpha_i{}^T(y_i|x)\beta_i(y_i|x)}{Z(x)}$;
  $P(Y_{i-1} = y_{i-1}, Y_i = y_i|x) = \frac{\alpha_{i-1}{}^T(y_{i-1}|x)M_{i-1}(y_{i-1},y_i|x)\beta_i(y_i|x)}{Z(x)}$;
  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：

$E_{P(X,Y)}[f_k]$

$$= \sum_x \widetilde{P}(x) \sum_{i=1}^{n+1} \sum_{y_{i-1}y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}{}^T(y_{i-1}|x) M_{i-1}(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)},$$

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

$Z(x) = \alpha_n{}^T(y_n|x) \cdot [1]_{m\times 1} = [1^T]_{1\times m} \cdot \beta_1(y_1|x)$;

- 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 *IIS*，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$(t_1, t_2, \ldots, t_{K_1})$，状态特征函数$(s_1, s_2, \ldots, s_{K_2})$，以及输入观测序列与输出标记序列的经验分布$\widetilde{P}(x, y)$；

输出对权值向量的估计$\hat{w} = (\hat{w}_1, \hat{w}_2, \ldots, \hat{w}_K)^T$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：

$L(w) = log \prod_{x,y}[P_w(y|x)]^{\widetilde{P}(x,y)} = \sum_{x,y} \widetilde{P}(x, y)\, log\, P_w(y|x)$;

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：

$L(w) = \sum_{x,y}[\widetilde{P}(x, y) \sum_{k=1}^K w_k f_k(y, x) - \widetilde{P}(x, y) log\, Z_w(x)]$;

其中$\sum_{x,y} \widetilde{P}(x, y) = 1$;

$L(w) = \sum_{j=1}^n \sum_{k=1}^K w_k f_k(y_j, x_j) - \sum_{j=1}^n log\, Z_w(x_j), k = 1, 2, \ldots, K$；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：

$E_{\widetilde{P}}[t_k] = \sum_{x,y}[\widetilde{P}(x, y) \sum_{i=1}^{n+1} t_k(y_{i-1}, y_i, x, i) exp(\delta_k T(x, y))], k = 1, 2, \ldots, K_1$;

$E_{\widetilde{P}}[s_l] = \sum_{x,y}[\widetilde{P}(x, y) \sum_{i=1}^{n} s_l(y_i, x, i) exp(\delta_{K_1+l} T(x, y))], l = 1, 2, \ldots, K_2$;

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

其中，状态特征与转移特征函数依经验分布$\widetilde{P}(x, y)$的期望由下式得到：

$E_{\widetilde{P}}[f_k]$

$$= \sum_x \widetilde{P}(x) \sum_{i=1}^{n+1} \sum_{y_{i-1}y_i} f_k(y_{i-1}, y_i, x, i) \frac{\alpha_{i-1}{}^T(y_{i-1}|x) M_{i-1}(y_{i-1}, y_i|x) \beta_i(y_i|x)}{Z(x)},$$

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

$Z(x) = \alpha_n{}^T(y_n|x) \cdot [1]_{m\times 1} = [1^T]_{1\times m} \cdot \beta_1(y_1|x)$;

设置权值更新增量$\delta = (\delta_1, \delta_2, \ldots, \delta_K)^T$，解特征方程，求解相应的$\delta_k$，并更新参数：

$w_k \leftarrow w_k + \delta_k, k = 1, 2, \ldots, K$;

直至得到收敛的更新增量序列$\delta = (\delta_1, \delta_2, \ldots, \delta_K)^T$;

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：

$\boldsymbol{P_w(y|x) = \frac{exp[\sum_{i=1}^{n} w_i f_i(x,y)]}{\sum_y exp[\sum_{i=1}^{n} w_i f_i(x,y)]}}$；

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

构造$\boldsymbol{P_w(y|x)}$的对数似然函数：$\boldsymbol{L(w) = log \prod_{x,y}[P_w(y|x)]^{\tilde{P}(x,y)}}$，整理得到：

$\boldsymbol{L(w) = \sum_{x,y} \tilde{P}(x,y)\, log[P_w(y|x)] = \sum_{x,y} \tilde{P}(x,y) \sum_{i=1}^{n} w_i f_i(x,y) -}$

$\boldsymbol{\sum_{x,y} \tilde{P}(x,y)\, log \sum_y exp[\sum_{i=1}^{n} w_i f_i(x,y)]}$；

将对数似然函数极大化转换成目标函数最优化问题即：

$\boldsymbol{min_{w \in R^n} f(w) =}$

$\boldsymbol{\sum_x \tilde{P}(x)\, log \sum_y exp[\sum_{i=1}^{n} w_i f_i(x,y)] - \sum_{x,y} \tilde{P}(x,y) \sum_{i=1}^{n} w_i f_i(x,y)}$；

求目标函数$\boldsymbol{f(w)}$对$\boldsymbol{w}$的梯度：

$\boldsymbol{g(w) = \sum_{x,y} \tilde{P}(x) P_w(y|x) f(x,y) - E_{\tilde{P}}[f(x,y)]}$；

CRF 模型的 ***BFGS*** 算法输入为各个位置上的特征函数(对 k 求和)$\boldsymbol{f_1, f_2, \dots, f_n}$，以及训练数据的经验分布$\boldsymbol{\tilde{P}(x,y)}$；输出为估计参数值$\boldsymbol{\hat{w} = (\hat{w}_1, \hat{w}_2, \dots, \hat{w}_K)^T}$以及对应的模型$\boldsymbol{P_{\hat{w}}(y|x)}$；

选定初始点$\boldsymbol{w^{(0)} = (w_1^{(0)}, w_2^{(0)}, \dots, w_K^{(0)})^T}$，取$\boldsymbol{B_0}$为正定对称矩阵，**t=0**；

$\boldsymbol{B_t p_t = -g_t}$ ；求解向量$\boldsymbol{p_t}$，其中$\boldsymbol{g_t = g(w^{(t)})}$；

求$\boldsymbol{\lambda_t}$，满足条件$\boldsymbol{f(w^{(t)} + \lambda_t p_t) = min_{\lambda \geq 0} f(w^{(t)} + \lambda p_t)}$；

更新$\boldsymbol{w^{(t+1)} = w^{(t)} + \lambda_t p_t}$ ，计算$\boldsymbol{g_{t+1} = g(w^{(t+1)})}$，更新矩阵

$\boldsymbol{B_{t+1} = B_t + \frac{(\Delta g)_t (\Delta g)_t^T}{(\Delta g)_t^T (\Delta w)_t} - \frac{B_t (\Delta w)_t (\Delta w)_t^T B_t}{(\Delta w)_t^T B_t (\Delta w)_t}}$；

$\boldsymbol{(\Delta g)_t = g_{t+1} - g_t}$，$\boldsymbol{(\Delta w)_t = w^{(t+1)} - w^{(t)}}$；

$\boldsymbol{t = t + 1}$，重复上述过程迭代更新参数值，直至到达最大迭代轮数 $\boldsymbol{T}$ 或者在计算过程中使得梯度$\boldsymbol{g_\tau = g(w^{(\tau)}) = 0}$；

- 预测问题，***CRF*** 的预测问题是指给定模型$\boldsymbol{P(Y|X)}$以及输入观测序列$\boldsymbol{x}$，求解使得条件概率最大的输出观测序列$\boldsymbol{y^*}$，**是典型的标注问题**；

  $\boldsymbol{y^* = arg\, max_y\, P_w(y|x) = arg\, max_y \frac{exp(w \cdot F(y,x))}{Z_w(x)}}$；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：

  $\boldsymbol{y^* = arg\, max_y (w \cdot F(y,x))}$；*即求解非规范化概率最大的输出序列；*

  $\boldsymbol{w = (w_1, w_2, \dots, w_K)^T}$；

$F(y,x)=(f_1(y,x),f_2(y,x),\dots,f_K(y,x))^T$;
$f_k(y,x)=\sum_{i=1}^{n}f_k(y_{i-1},y_i,x_i,i),k=1,2,\dots,K$;
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：

$\delta_1(v_j)=\ w\cdot F_1(y_0=start,y_1=v_j,x),j=1,2,\dots,m$;

递推式得到最大非规范化概率及其对应的最佳路径为：

$\delta_i(v_l)=max_{1\le j\le m}\left[\delta_{i-1}(v_j)+w\cdot F_i(y_{i-1}=v_j,y_i=v_l,x)\right],l=1,2,\dots,m$;

$\psi_i(v_l)=arg\ max_{1\le j\le m}\left[\delta_{i-1}(v_j)+w\cdot F_i(y_{i-1}=v_j,y_i=v_l,x)\right]$,

$l=1,2,\dots,m$;
直至$i=n$处终止，此时得到全路径最大非规范化概率为：

$max_y(w\cdot F(y,x))=\ max_{1\le j\le m}\delta_n(v_j)$;

$y_n^*=\ arg\ max_{1\le j\le m}\delta_n(v_j)$;

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 $y^*=(y_1^*,y_2^*,\dots,y_n^*)^T$;

7.25.4 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，$\sum_{j=1}^{N}\beta_{k,j}=1$;

β_k由以 η 为参数的 *Dirichlet* 分布得到：

$Dir(\boldsymbol{\beta}_{k,1}, \boldsymbol{\beta}_{k,2}, \dots, \boldsymbol{\beta}_{k,N}|\boldsymbol{\eta}) = \frac{\Gamma(\sum_{j=1}^{N}\eta_j)}{\Gamma(\eta_1)\Gamma(\eta_2)\cdots\Gamma(\eta_N)}\prod_{j=1}^{N}\boldsymbol{\beta}_{k,j}^{\eta_j-1}$；

LDA 认为一篇文档中包含多个话题，以向量$\boldsymbol{\Theta}_t \in \mathcal{R}^K$表示在文档 $\boldsymbol{t}$ 中包含的话题分布，其中$\boldsymbol{\Theta}_{t,k}$表示话题 $\boldsymbol{k}$ 在文档 $\boldsymbol{t}$ 中所占比例，$\sum_{k=1}^{K}\boldsymbol{\Theta}_{t,k} = 1$；

- 根据参数为$\boldsymbol{\alpha}$的 ***Dirichlet*** 分布随机采样一个话题分布$\boldsymbol{\Theta}_t$：

$Dir(\boldsymbol{\Theta}_{t,1}, \boldsymbol{\Theta}_{t,2}, \dots, \boldsymbol{\Theta}_{t,K}|\boldsymbol{\alpha}) = \frac{\Gamma(\sum_{k=1}^{K}\alpha_k)}{\Gamma(\alpha_1)\Gamma(\alpha_2)\cdots\Gamma(\alpha_K)}\prod_{k=1}^{K}\boldsymbol{\Theta}_{t,k}^{\alpha_k-1}$ ；

- 生成文档中的 ***N*** 个词语：

  根据向量$\boldsymbol{\Theta}_t$进行话题指派，得到文档 $\boldsymbol{t}$ 中各个词语的所属的话题$\boldsymbol{k} = \boldsymbol{z}_{t,i}, \boldsymbol{k} = 1, 2, \dots, \boldsymbol{K}$ ；

  再根据指派话题对应的分布向量$\boldsymbol{\beta}_k$依 ***Dirichlet*** 分布随机采样生成词语；

文档 $\boldsymbol{t}$ 以不同的比例包含 ***K*** 个话题，话题$\boldsymbol{\Theta}_t$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$\boldsymbol{\omega}_{t,i}$可由文档词频统计得到，是唯一的可观测变量；

$\boldsymbol{\omega}_{t,i}$依赖于对词语的话题指派$\boldsymbol{z}_{t,i}$以及话题词频向量$\boldsymbol{\beta}_{k,j}, (\boldsymbol{k} = \boldsymbol{z}_{t,i}, \boldsymbol{j} = \boldsymbol{i})$；

话题指派$\boldsymbol{z}_{t,i}$依赖于话题分布$\boldsymbol{\Theta}_t$；

话题分布$\boldsymbol{\Theta}_t$依赖于 ***Dirichlet*** 分布及其参数$\boldsymbol{\alpha}$；

词频统计$\boldsymbol{\omega}_{t,i}$依赖于参数$\boldsymbol{\eta}$；

则得到 ***LDA*** 的概率分布为：

$\boldsymbol{p}(\boldsymbol{W}, \boldsymbol{z}, \boldsymbol{\beta}, \boldsymbol{\Theta}|\boldsymbol{\alpha}, \boldsymbol{\eta}) = \prod_{t=1}^{T}\boldsymbol{p}(\boldsymbol{\Theta}_t|\boldsymbol{\alpha})\prod_{k=1}^{K}\boldsymbol{p}(\boldsymbol{\beta}_k|\boldsymbol{\eta})\left[\prod_{i=1}^{N}\boldsymbol{P}(\boldsymbol{\omega}_{t,i}|\boldsymbol{z}_{t,i})\boldsymbol{P}(\boldsymbol{z}_{t,i}|\boldsymbol{\Theta}_t)\right]$ ；

其中，文档 $\boldsymbol{t}$ 中的话题先验分布：

$\boldsymbol{p}(\boldsymbol{\Theta}_t|\boldsymbol{\alpha}) = \frac{\Gamma(\sum_{k=1}^{K}\alpha_k)}{\Gamma(\alpha_1)\Gamma(\alpha_2)\cdots\Gamma(\alpha_K)}\prod_{k=1}^{K}\boldsymbol{\Theta}_{t,k}^{\alpha_k-1}$； *由此产生每个词语的话题指派* $\boldsymbol{k} = \boldsymbol{z}_{t,i}$ ；

其中话题 $\boldsymbol{k}$ 中的词频分布：

$\boldsymbol{p}(\boldsymbol{\beta}_k|\boldsymbol{\eta}) = \frac{\Gamma(\sum_{j=1}^{N}\eta_j)}{\Gamma(\eta_1)\Gamma(\eta_2)\cdots\Gamma(\eta_N)}\prod_{j=1}^{N}\boldsymbol{\beta}_{k,j}^{\eta_j-1}$ ；

由话题词频先验分布和话题先验分布产生的话题指派$\boldsymbol{k} = \boldsymbol{z}_{t,i}$共同产生文档词频 $\boldsymbol{\omega}_{t,i}$ ；

给定数据集$\boldsymbol{W} = \{\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \dots, \boldsymbol{\omega}_T\}, \boldsymbol{\omega}_t \in \mathcal{R}^N$， ***LDA*** 模型参数$\boldsymbol{\alpha}$，$\boldsymbol{\eta}$使用极大似然估计法求解：

$\boldsymbol{L}(\boldsymbol{\alpha}, \boldsymbol{\eta}) = \boldsymbol{ln}\prod_{t=1}^{T}\boldsymbol{p}(\boldsymbol{\omega}_t|\boldsymbol{\alpha}, \boldsymbol{\eta})$；

7.26 从伯努利到狄利克雷分布

7.26.1 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：
$\boldsymbol{Bern(x|\mu) = \mu^x(1-\mu)^{1-x}}$**；其中随机变量**$\boldsymbol{x \in \{0,1\}}$**，**$\boldsymbol{0 \le \mu \le 1}$**；**
伯努利分布描述了随机变量的分布$\boldsymbol{P(x=1|\mu) = \mu; P(x=0|\mu) = 1-\mu}$；
伯努利分布的均值为：
$\boldsymbol{E[x] = E_{Bern}[x] = 1 \cdot P(x=1|\mu) + 0 \cdot P(x=0|\mu) = \mu}$；
方差为：
$\boldsymbol{var[x] = (1-E[x])^2 \cdot P(x=1|\mu) + (0-E[x])^2 \cdot P(x=0|\mu) = \mu(1-\mu)}$；
伯努利分布的参数估计方法：
基于给定样本数据集$\boldsymbol{D = \{x_1, x_2, \dots, x_N\}}$**，估计模型参数**$\boldsymbol{\mu}$**，使用极大似然估计，**
$\boldsymbol{P(x_i|\mu) = \mu^{x_i}(1-\mu)^{1-x_i}, x_i \in D}$**；**

构造对数似然函数$\boldsymbol{L(\mu|D) = ln\prod_{i=1}^{N} P(x_i|\mu) = ln\prod_{i=1}^{N} \mu^{x_i}(1-\mu)^{1-x_i}}$**；**

对变量$\boldsymbol{\mu}$**求导，得到**$\boldsymbol{\frac{\partial L(\mu|D)}{\partial \mu} = \frac{1}{\mu}\sum_{i=1}^{N} x_i - \frac{1}{1-\mu}\sum_{i=1}^{N}(1-x_i)}$**；**

令$\boldsymbol{\frac{\partial L(\mu|D)}{\partial \mu} = 0}$**，得到**$\boldsymbol{\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i}$ **作为对参数**$\boldsymbol{\mu}$**的估计；**

7.26.2 二项分布

伯努利分布考虑的是**随机变量**$\boldsymbol{x \in \{0,1\}}$**，**$\boldsymbol{0 \le \mu \le 1}$**在伯努利试验中的分布律，**
即概率$\boldsymbol{Bern(x|\mu) = \mu^x(1-\mu)^{1-x}}$**，而二项分布(*Binomial Distribution*)考虑在数据集**$\boldsymbol{D = \{x_1, x_2, \dots, x_N\}}$**中，随机变量**$\boldsymbol{x_i}$**取值为 1 的个数 m 的概率分布，其中**
$\boldsymbol{m = \sum_{i=1}^{N} I(x_i = 1)}$**：**

$\boldsymbol{Bin(m|D;\mu) = C_N^m \mu^m (1-\mu)^{N-m}}$**；**
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$\boldsymbol{x=1}$**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；

$\boldsymbol{E[m] = \sum_{m=0}^{N} m \cdot Bin(m|D;\mu) = N \cdot \mu}$***；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：

$\boldsymbol{var[m] = \sum_{m=0}^{N}(m-E[m])^2 \cdot Bin(m|D;\mu) = N \cdot \mu(1-\mu)}$**；**

对参数$\boldsymbol{\mu}$**的估计是数据集中**$\boldsymbol{x_i = 1, i = 1, 2, \dots, N}$**所占的比例**$\boldsymbol{\hat{\mu} = \frac{m}{N}}$ **；**

7.26.3 *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$\boldsymbol{P(x|\mu)P(\mu)=P(\mu|x)P(x)}$得到$\boldsymbol{P(\mu|x)=\frac{P(x|\mu)P(\mu)}{P(x)}}$，即后验概率$\boldsymbol{P(\mu|x)}$正比于似然$\boldsymbol{P(x|\mu)}$与先验概率$\boldsymbol{P(\mu)}$的乘积，先验概率$\boldsymbol{P(x)}$在数据集给定的情况下可认为是常数：$\boldsymbol{f(\mu;D)\propto p(x|\mu)p(\mu)}$；

由于$\boldsymbol{x\sim Bern}$: $\boldsymbol{P(x|\mu)=\mu^x(1-\mu)^{1-x}}$，选择 ***Beta*** 分布：

$\boldsymbol{Beta(\mu|a,b)=\frac{\Gamma(a+b)}{\Gamma(a)\,\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1}}$；其中 *Gamma* 函数：

$\boldsymbol{\Gamma(a)=\int_0^{+\infty}t^{a-1}e^{-t}dt}$；

Beta 分布的期望为：$\boldsymbol{E[\mu]=\frac{a}{a+b}}$ ；方差为：$\boldsymbol{var[\mu]=\frac{ab}{(a+b)^2(a+b+1)}}$ ；

可以得到后验概率：

$\boldsymbol{P(\mu|x)=P(x|\mu)\cdot Beta(\mu|a,b)=C_N^m\mu^m(1-\mu)^{N-m}\frac{\Gamma(a+b)}{\Gamma(a)\,\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1}}$；

即有$\boldsymbol{P(\mu|x)\propto\mu^{m+a-1}(1-\mu)^{N-m+b-1}}$，令$\boldsymbol{l=N-m}$，再将其进行概率归一化得到拥有新参数的 ***Beta*** 分布：

$\boldsymbol{Beta(\mu|m+a,l+b)=\frac{\Gamma(a+b+m+l)}{\Gamma(a+m)\,\Gamma(b+l)}\mu^{a+m-1}(1-\mu)^{b+l-1}}$；*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$\boldsymbol{\mu}$的后验均值比其先验分布$\boldsymbol{Beta(\mu|a,b)}$给出的均值大，比其极大似然估计均值$\hat{\boldsymbol{\mu}}$小，当数据集规模趋近于无穷大时，二者得到相同的结果；

7.26.4 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$\boldsymbol{X}$的取值空间$\boldsymbol{|\mathcal{X}|>2}$ 即$\boldsymbol{x}$的属性拥有多种取值的情况，使用 ***one-hot*** 表示方法：$\boldsymbol{x=[x_k]_{K\times1}}$，$\boldsymbol{x_k\in\{0,1\}}$，且有$\boldsymbol{\sum_{k=1}^K x_k=1}$，使用$\boldsymbol{\mu_k}$表示$\boldsymbol{x_k=1}$的概率，则有$\boldsymbol{p(x|D)=\prod_{k=1}^K\mu_k{}^{x_k}}$ 表示随机变量的概率分布；

$\boldsymbol{\sum_x p(x|\mu)=\sum_{k=1}^K\mu_k=1}$；

$\boldsymbol{E_\mu[x]=\sum_x p(x|\mu)\cdot x=\mu=(\mu_1,\mu_2,\dots,\mu_K)^T}$；

给定数据集$\boldsymbol{D=\{x_1,x_2,\dots,x_N\}}$，似然函数为：

$\boldsymbol{p(D|\mu)=\prod_{i=1}^N\prod_{k=1}^K\mu_k{}^{x_{ik}}=\prod_{k=1}^K\mu_k{}^{m_k},m_k=\sum_{i=1}^N x_{ik}}$ ；

构造最优化问题：

$min_{\mu_k} - ln\, p(D|\mu)$；

$s.t. \sum_{k=1}^{K} \mu_k = 1$；

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：

$Mult(m_1, m_2, \dots, m_K|\mu, N) = C_{m_1 m_2 \dots m_K}^{N} \prod_{k=1}^{K} {\mu_k}^{m_k}$ ；

$C_{m_1 m_2 \dots m_K}^{N} = \frac{N!}{m_1! m_2! \cdots m_K!}$ ；且满足限制条件 $\sum_{k=1}^{K} \mu_k = 1$；

7.26.5 *Dirichlet* 分布

Beta 分布作为二项分布的参数μ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：

$p(\mu|\alpha) \propto \prod_{k=1}^{K} {\mu_k}^{\alpha_k}$；概率归一化后得到：

$Dir(\mu|\alpha) = \frac{\Gamma(\sum_{k=1}^{K} \alpha_k)}{\Gamma(\alpha_1)\, \Gamma(\alpha_2) \cdots \Gamma(\alpha_K)} \prod_{k=1}^{K} {\mu_k}^{\alpha_k - 1}$；

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：

$p(\mu|D; \alpha) = \frac{\Gamma(\sum_{k=1}^{K} \alpha_k + \sum_{k=1}^{K} m_k)}{\Gamma(\alpha_1 + m_1)\, \Gamma(\alpha_2 + m_2) \cdots \Gamma(\alpha_K + m_K)} \prod_{k=1}^{K} {\mu_k}^{\alpha_k + m_k - 1} = Dir(\mu|\alpha + m)$；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；

7.27 常用的不等式

- 在确定最大熵模型参数更新对数似然函数增量的下界时用到：
  $-log\alpha \geq 1-\alpha, \alpha > 0$；
- *Jensen* 不等式：
  $log\sum_i \lambda_i x_i \geq \sum_i \lambda_i log x_i$ ，$\lambda_i \geq 0, \sum_i \lambda_i = 1$；
  Jensen 不等式是凸函数性质从二维到多维的推广：
  $f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta)f(x_2), \theta \geq 0$；
  若有参数序列 $\theta = (\theta_1, \theta_2, \dots, \theta_K)^T$，满足$\theta_k \geq 0, \sum_k \theta_k = 1$，则有
  $f(\theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_K x_K) \leq \theta_1 f(x_1) + \theta_2 f(x_2) + \cdots + \theta_K f(x_K)$；

  即$f(E[x]) \leq E[f(x)]$，其中$f(x)$为凸函数；

8.深度学习

8.1 神经网络基本框架

8.1.1 神经网络训练

无论神经网络的结构如何，训练过程总可以分为三个步骤进行：

- **定义神经网络结构和前向传播的输出结果；**
  (1) 定义连接权重 weight = tf.Variable(tf.random_normal(shape, stddev, seed))
  和偏置量 bias = tf.Variable(tf.constant(const, shape))；
  (2) 定义数据集输入特征和样本标记 x = tf.placeholder(tf.float32, shape, name = 'x-input')
  y_ = tf.placeholder(tf.float32, shape, name = 'y-input')
  (3) 定义前向传导过程，设计网络结构；

网络结构和前向传播通常在函数 inference(…)中定义；

- **定义损失函数，选择反向传播优化算法：**
  (1) 定义损失函数 cross_entropy = … ;
  交叉熵定义以及损失函数的数学解释见常用的损失函数；
  通常由 **cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=y, labels=y_)**或者 **cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y, labels=argmax(y_, 1))** 得到的是整个 **batch** 的交叉熵，还需要对其取平均：**cross_entropy_mean = tf.reduce_mean(cross_entropy)**并计入正则化项得到最终的损失：**loss = cross_entropy_mean + regularization**
  (2) 定义反向传播算法：
  train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
  损失函数的选择见 Tensorflow 中的损失函数；
- **生成会话(tf.Session())，在训练数据上反复运行反向传播优化算法：**

```
with tf.Session() as sess:
    init_op = [tf.global_variables_initilaizer(), tf.local_varibales_initializer()]
    sess.run(init_op)
    #…
    for i in range(STEPS):
        sess.run(train_op, feed_dict={x: x_batch, y_: y_batch})
        …
        if i % ITERS == 0:
            total_cross_entropy = sess.run(cross_entropy, feed_dict={x: X, y_: Y})
```

损失函数定义、优化算法选择以及训练步骤的执行通常放置在 *train(…)*中实

现；
训练过程中数据的输入：

```
xs, ys = mnist.train.next_batch(BATCH_SIZE)
sess.run(train_op, feed_dict={x:xs, y_: ys})
…
mnist = input_data.read_data_sets(path, one_hot=True)
train(mnist)
```

验证数据输入：

```
validate_feed = {x: mnist.validation.images, y_: minist.validation.labels}
…
validate_accuracy = sess.run(accuracy, feed_dict=validate_feed)
```

通常神经网络的训练都是一次输入一个 Batch 的样本，因此，定义输入时：

```
x = tf.placeholder(tf.float32, shape=[BATCH_SIZE, input_nodes], name='x-input')
y_ = tf.placeholder(tf.float32, shape=[BATCH_SIZE, output_nodes], name='y-input')
```

8.1.2 神经网络优化

神经网络的优化过程分成两个阶段：

- 训练数据集按 Batch 输入 ***x*** 及其样本标记 ***y_***，前向传播得到 ***y***，比较 ***logits y*** *和* ***labels y_*** 的差距，即损失函数(Cross Entropy，MSE 等)；
- 使用 Mini-Batch Gradient Descent 方法计算输入样本 Batch 的损失函数对每个参数的梯度，并沿着负梯度方向按学习速率 η 更新参数，局部最小与全局最小；

除去选择损失函数和优化算法之外，在神经网络的训练中还采取以下措施：

- 学习速率衰减：

  decayed_learning_rate =
  learning_rate * decay_rate ^ (global_step/decay_steps)

  学习速率指数衰减策略：$\eta \leftarrow \eta\epsilon^{\tau}$，其中 η 为基础学习速率 ***(LEARNING_RATE_BASE)***，ϵ为学习率衰减率***(LEARNING_RATE_DECAY)***，$\tau = \frac{t_g}{t_d}$，

  t_g为当前训练的迭代次数***(global steps)***，t_d为预设进行衰减的迭代次数***(decay steps)***，通常设定每训练一个 ***Batch*** 数据就进行一次衰减，因此使用 tf.train.exponential_decay()使得学习率以阶梯状下降：

  learning_rate = tf.train.exponential_decay(LEARING_RATE_BASE, global_step, num_examples/BATCH_SIZE, LEARNING_RATE_DECAY, staircase=True)

  同时，在使用优化算法调整权值时，要将全局变量 global_step 传入，以使用更新的学习率：

  train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss_fcuntion, global_step = global_step)

  Adagrad/Adadelta/Adam 在解决梯度下降法易陷入局部最小问题时，均对学习速率进行了正则化约束，实现学习速率的衰减，见局部最小与全局最小；
- 连接权值 ***Dropout***：训练时随机让一些隐含节点不工作，即丢弃其连接权值，

使得参数更新不再依赖于固有的节点间的连接关系，**在一定程度上避免了特征$a_i, a_i \in \mathcal{A}$只有在其他某个特征$a_j, a_j \in \mathcal{A}, i \neq j$才起作用的情况；**

而对每次输入的样本 ***Batch***，相当于使用具有不同结构的模型进行训练，类似于 ***Bagging***，**利于提高模型的泛化性能；**
每个 ***Batch*** 只针对样本的一部分隐含特征进行训练，加入样本的属性扰动，提高了数据集方差，以 ***Naïve Bayes*** 的角度来看，等于加入属性间的独立性假设：

```
lstm_cell =
tf.nn.rnn_cell.DropoutWrapper(lstm_cell, output_keep_prob=KEEP_PROB)
```

- 正则化 ***Regularizer***：为避免过拟合，神经网络训练优化除了要极小化经验损失***(Cross Entropy，MSE 等)***，还要考虑结构损失，既可以使用 ***L1*** 正则化(倾向于得到稀疏解)，也可以使用 ***L2*** 正则化(倾向于得到平滑解)：

```
loss_function =
tf.reduce_mean(tf.square(y-y_)
+ tf.contrib.layers.l2_regularizer(lambda)(weights))
```

正则化项定义通常与网络结构定义部分放在一起，用以**直接计算网络权值的正则化损失**，而损失函数通常在 ***train_step*** 之前定义，二者可能不在同一个源文件中，当网络结构较复杂时，直接将正则化项加入损失函数会导致损失函数定义过长，代码可读性较差，因此，将正则化项加入集合中：

```
tf.add_to_collection('losses',
tf.contrib.layers.l2_regularizer(REGULARIZATION_RATE)(layer_weights))
...
loss = cross_entropy + tf.add_n(tf.get_collection('losses'))
```

计算最终的损失需要考虑选择的经验损失以及神经网络各个层的正则化损失，正则化损失的数学解释见正则化；

- 滑动平均模型是指在最后的模型选择时，倾向于选择所有待选模型的参数均值组成的模型，此模型往往比最后一次迭代得到的模型以及在验证数据集上表现最好的模型的性能更优；

$\tilde{w} \leftarrow \tilde{w} \cdot \rho + w_p \cdot (1 - \rho)$；其中$\tilde{w}$**为影子变量*(shadow variable)*，全程跟随变量值w_p更新而变化，ρ为衰减率，ρ越大，影子变量$\tilde{w}$越倾向于接近更新前的数值，ρ越小，则越接近更新后的数值w_p，此时参数波动较大，模型不够健壮，通常选择一个较大的数值(0.999/0.9999)，在 Tensorflow 中通过传入参数 num_updates 可以得到一个随训练过程调整的衰减率：$\rho = min\{\rho, \frac{1+p}{10+p}\}$，其中$p = num_updates$为当前迭代的轮数；**

对所有的变量应用滑动平均模型的实现需要指定衰减速率并应用于所有变量：

```
variable_averages =
tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY, global_step)
variable_averages_op = variable_averages.apply(tf.trainable_variables())
```

```
with tf.control_dependencies([train_step]):
    train_op = tf.group(variable_averages_op)
```

在使用优化算法更新完参数后，应用滑动平均模型得到参数的影子变量，可以随模型参数和优化器参数保存到 ***checkpoint(ckpt)***文件中，实现持久化，也可以从中恢复使用：

```
saver = tf.train.Saver()                                    #  保存模型
…
moving_average_variables_to_restore =                       #  指定要恢复的参数为
variable_averages.variables_to_restore()                    # ExponentialMovingAverage
saver = tf.train.Saver(moving_average_variables_to_restore)
saver.restore(sess, save_path)                              #  从 ckpt 中恢复参数
```

8.2 全连接神经网络

8.2.1 激活函数

为感知机***(Perceptron)***加入多层和非线性特性使得神经网络表达能力更加丰富，将权值向量与单元输入的乘积与偏置量求和并输入非线性单元完成非线性化激活，常用的非线性激活函数有：

- **限制线性单元*(Restricted Linear Unit)ReLU*：$f(x)=max(x,0)$，tf.nn.relu()；**
- **Sigmoid 函数：$f(x)=\frac{1}{1+exp(-x)}$ ，tf.nn.sigmoid()；**
- **正切函数：$f(x)=\frac{1-exp(-2x)}{1+exp(-2x)}$ ，tf.nn.tanh()；**

在多分类问题中，对输出节点做 **tf.argmax(y, 1)**处理即可得到待分类样本的标签信息；

8.2.2 前向传播

全连接神经网络的前向传播过程即将当前层输入 ***tensor*** 与连接权重相乘再加偏置后进行非线性化作为输出给下一层的过程：

layer1 = tf.nn.relu(tf.matmul**(**input_tensor, fc1_weights**)**+fc1_biases)

layer2 = tf.matmul(layer1, fc2_weights)+fc2_biases

预测时使用的是输出层不同的节点之间的相对大小，因此最终的输出层可以不加入非线性激活函数和 ***softmax***；

若使用了滑动平均模型，则需要使用 ***MovingAverage*** 后的参数值：

layer1 =

tf.nn.relu(tf.matmul**(**input_tensor,

average_class.average(fc1_weights)**)**+average_class.average(fc1_biases))

layer2 =

tf.matmul(average_class.average(layer1),

average_class.average(fc2_weights))+fc2_biases

8.3 卷积神经网络

8.3.1 常用的图像数据集

图像分类问题常用的数据集：

- ***Cifar***
- ***MNIST***
- ***ImageNet***

分类问题性能度量标准：TOP-N 正确率，即预测给出的前 N 个答案中有一个是正确答案的概率；

8.3.2 CNN

CNN 的标准结构：***Input Layer1 → Convolutional1 Layer2 → Pool1 Layer3 → Convolutional2 Layer4 → Pool2 Layer5 → Full Connected1 Layer6 → Full Connected2 Layer7 → Softmax Layer8***

- 输入层：图片的像素矩阵$\boldsymbol{A_{H\times W\times D}=[a_{x,y,z}]_{L\times W\times D}}$，其中 ***L*** 为图片的高度即纵向上像素的数量，***W*** 为宽度即横向上像素的数量，***D*** 为深度即图片的色彩通道，黑白图片深度为 ***D=1***，RGB 色彩模式下，图片深度为 ***D=3***；
- 卷积层：通过卷积核**(过滤器)*filter*** 将输入矩阵(可能是输入层的原始像素矩阵，也可能是池化层输出)的子节点矩阵$\boldsymbol{\overline{A}_{l\times w\times d}}$转换为一个**单位节点矩阵** $\boldsymbol{C_{1\times 1\times p}=[g_i]_{1\times 1\times p}}$：

  $\boldsymbol{g_i=f(\sum_{x=1}^{l}\sum_{y=1}^{w}\sum_{z=1}^{d}w_{x,y,z}^{(i)}\cdot a_{x,y,z}+b^{(i)})}$；即对输入层的子节点矩阵$\boldsymbol{\overline{A}_{l\times w\times d}}$中的元素按照 ***p*** 组权重$\boldsymbol{w_{x,y,z}^{(i)}}$和偏置$\boldsymbol{b^{(i)}}$求加和并进行非线性激活，即可得到卷积层单位节点矩阵的 ***p*** 个节点数值，其中，$\boldsymbol{l\times w\times p}$为卷积核的尺寸，而卷积的过程就是将卷积核从输入矩阵的左上角-右上角-左下角-右下角逐次移动，而在深度方向上不能移动，每次移动的步长决定了卷积输出矩阵的尺寸；

  当使用全 ***0*** 填充***(zero-padding)***时：

  $\boldsymbol{output_{length}=\lceil input_{length}/stride_{length}\rceil}$；

  $\boldsymbol{output_{width}=\lceil input_{width}/stride_{width}\rceil}$；

  不使用填充时：

  $\boldsymbol{output_{length}=\lceil input_{length}-filter_{length}+1/stride_{length}\rceil}$；

  $\boldsymbol{output_{width}=\lceil input_{width}-filter_{width}+1/stride_{width}\rceil}$；

  代码实现如下：

  #卷积核尺寸为 5*5，当前层深度为 3，卷积核深度为 16

filter_weight = tf.get_variable('weight',
[5,5,3,16], initializer=tf.truncated_normal_initializer(stddev=0.1))
biases = tf.get_variable('biases', [16], initializer=tf.constant_initializer(0.1))
#输入 input 为 4 维矩阵**[index, length, width, deepth]**，**第 0 维代表图片在 #*Batch* 中的索引；**
#卷积核作用不能跨越图片样本也不能跨越深度，因此 strides 的第 0 维与第 #3 维只能为 1
#padding='SAME' 表示全 0 填充，**padding='VALID'**不填充
conv = tf.nn.conv2d(input, filter_weight, strides=[1,1,1,1], padding='SAME')
#为所有节点加上相同的偏置项
bias = tf.nn.bias_add(conv, biases)
activated_conv = tf.nn.relu(bias)

- **池化层：**池化层作用与卷积层的输出之上，进一步缩小矩阵的尺寸，减少连接的个数，常使用的池化层有最大池化层***(max pooling)***和平均池化层***(average pooling)***，池化核(过滤器)filter 可以在深度方向上起作用：
  #ksize=[1,3,3,1]第 0 维与第 3 维必须为 1，不可跨样本和深度作用；
  #strides=[1,2,2,1]第 0 维与第 3 为必须为 1，不可减少样本个数和深度；
  pool = tf.nn.max_pool(activated_conv,
  ksize=[1,3,3,1], strides=[1,2,2,1], padding='SAME')
- **全连接层：**卷积神经网络最后通常要加入两层或以上全连接网络，用于完成分类任务；
- **Softmax 层：Softmax 层最终将分类结果输出节点上的数值归一化处理为概率分布；**

8.3.3 LeNet-5

LeNet-5 **的七层结构：*Input Layer → Convolutional1 Layer → Pool1 Layer → Convolutional2 Layer → Pool2 Layer → Full Connected1 Layer → Full Connected2 Layer → Full Connected3 Layer***
数据集：***MNIST***

- ***Input Layer：MNIST*** 图片维度 **32*32*1**
- ***Convolutional1 Layer：***
  filter_size=[5, 5, 1, 6]
  strides=[1,1,1,1]
  PADDING='VALID'
  输出尺寸=28*28*6
- ***Pool1 Layer：***
  filter_size=[1, 2, 2, 1]
  strides=[1,2,2,1]
  PADDING='VALID'
  输出尺寸=14*14*6
- ***Convolutional2 Layer：***
  filter_size=[5, 5, 6, 16]
  strides=[1,1,1,1]

PADDING='VALID'
输出尺寸=10*10*16

- ***Pool2 Layer***
  filter_size=[1, 2, 2, 1]
  strides=[1,2,2,1]
  PADDING='VALID'
  输出尺寸=5*5*16
- ***Full Connected1 Layer：***输入矩阵尺寸=**[batch_size, 5, 5, 16]，将第 1~3 维拼接称一个向量，即代表一个样本数据，即输入节点个数为 5*5*16=400，输出节点数 120；**
- ***Full Connected2 Layer：***输入节点数 120，输出节点数 84；
- ***Full Connected3 Layer：***输入节点数 84，输出节点数 10，完成分类任务；
- **分类正确率：测试数据集上可达 99.4%**

调整输入矩阵以及输入训练数据：

```
x = tf.placeholder(tf.float32, [BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE,
NUM_CHANNELS], name='x-input')
reshaped_xs = np.reshape(xs, (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE,
NUM_CHANNELS))
```

网络结构设计时，使用上下文管理器隔离命名空间，以便获取变量：

```
with tf.variable_scope('layer1-cov1'):
    conv1_weights = tf.get_variable("weight", [CONV1_SIZE, CONV1_SIZE,
    CONV1_CHANNELS, CONV1_DEEPTH],
    initializer=tf.truncated_normal_initializer(stddev=0.1))
    conv1_biases = tf.get_variable("bias", [CONV1_DEEPTH],
    initializer=tf.constant_initializer(0.0))
    conv1 = tf.nn.conv2d(input_tensor, conv1_weight, strides=[1,1,1,1],
    padding='SAME')
    relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_biases))

with tf.name_scope('layer2-pool1'):
    pool1 = tf.nn.max_pool(relu1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
```

在 ***Pool2 Layer*** 输出给全连接层时，要进行矩阵形状变换：

```
pool_shape = pool2.get_shape().as_list()
nodes = pool_shape[1]*pool_shape[2]*pool_shape[3]
reshaped = tf.reshape(pool2, [pool_shape[0], nodes])
```

卷积神经网络模型中正则化损失仅考虑全连接层的权重，且 dropout 仅考虑除最终输出层之外的全连接层的权重：

```
with tf.variable_scope('layer5-fc1'):
    fc1_weights = tf.get_variable("weight", [nodes, FC1_SIZE],
    initializer=tf.truncated_normal_initializer(stddev=0.1))
    if regularizer != None:
        tf.add_to_collection('losses', regularizer(fc1_weights))
    fc1_biases = tf.get_variable("bias", [FC1_SIZE],
    initializer=tf.constant_initializer(0.1))
```

```
    fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_weights) + fc1_biases)
    if train:
        fc1 = tf.nn.dropout(fc1, DROPOUT_KEEP_PROB)
…
#全连接层不经过 Softmax 处理直接给出结果，由
#tf.nn.sparse_softmax_cross_entropy_with_logits()处理
logit = tf.matmul(fc1, fc2_weights) + fc2_biases
return logit
```

通常选择卷积核尺寸不超过 5，一般为 3 或 1，深度通常采用逐层递增形式，卷积层步长通常设为 1；
池化层使用最普遍的是 max_pool，尺寸通常为 2 或 3，步长选择 2 或 3；
完整源码见“./src/LeNet-5”；

8.3.4 迁移学习

将已保留的 ***Inception-v3*** 模型中的所有卷积层看做对图片特征的提取，输出一个表达能力更强的特征向量，替换最后的全连接层，用于完成新的分类任务，将卷积层之前的网络层称为瓶颈层***(bottleneck)***；

- ***Inception-v3*** 模型由 ***Inception*** 模块组成，与 ***LeNet-5*** 模型不同，***Inception*** 模块采用并联多卷积核做卷积运算然后拼接成输出矩阵的方式，并联多个卷积核尺寸不同但都使用全 ***0*** 填充，因此卷积输出的矩阵尺寸***(length*width)***相同，模块输出仅需要在**深度方向(维度 3)**上进行拼接：

```
branch_1 = tf.concat(3, [slim.conv2d(branch_1, 384, [1,3], scope='Conv2d_0b_1x3'),
                         slim.conv2d (branch_1, 384, [3,1], scope='Conv2d_0b_3x1')])
```

实现较复杂的卷积层，可使用 tensorflow-slim 工具减少代码数量：

```
net = slim.conv2d(input, 32, [3,3]) #必填参数 input 输入矩阵，卷积核深度，卷积
                                    #核尺寸
```

其他可选参数可以使用 **slim.arg_scope()**函数设置：

```
with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d], stride=1,
                    padding='SAME')
```

- 数据集：***flower_photos***
- 模型：***classify_image_graph_def.pb***
- **tensor.name()获得所需的迁移模型的图片样本输入张量 'DecodeJpeg/contents:0'和瓶颈层输出结果张量'pool_3/_reshape:0'，将从输入层到瓶颈层之间的模型视作黑盒，在新模型训练时直接使用；**
- 实现：
  (1) 划分数据集：留出法，遍历**'./data/flower_photos'**下所有子路径，得到所有图片样本的不带路径的文件名称 ***filename.extension***，并以子路径名称作为样本的 **label**，调用 np.random.randint(100)生成随机数，若小于 **VALIDATION_PERCENTAGE**，则放入验证集，若大于 **VALIDATION_PERCENTAGE** 而小于 **VALIDATION_PERCENTAGE+TESTING_PERCENTAGE** 则放入测试集，否则放入训练集，最终使用一个字典保存当前分类的数据集：

image_lists[label_name]={'dir': dir_name, **#子路径**
'training': training_images, **#训练集**
'validation': validation_images, **#验证集**
'testing': testing_images} **#测试集**

(2) 在训练时，从训练集中随机获取图片 get_random_cached_bottlenecks(**sess**, n_classes, image_lists, how_many, **category**, jpeg_data_tensor, bottleneck_tensor)，得到从各个 label 中随机选取的图片构成的训练集，**how_many=BATCH_SIZE**，**category='training'**，

jpeg_data_tensor, bottleneck_tensor = tf.import_graph_def(graph_def, return_elements=[JPEG_DATA_TENSOR_NAME, BOTTLENECK_TENSOR_NAME])

…

#***classify_image_graph_def.pb***

```
with gfile.FastGFile(os.path.join(MODEL_PATH, MODEL_FILE)) as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())
```

若**./tmp/bottleneck/label_name** 目录下没有待选择图片的特征向量文件，则为其创建：

bottleneck_values = **sess**.run(bottleneck_tensor, {**image_data_tensor**: image_data})#image_data_tensor 即已获得的 jpeg_data_tensor

为输入后面的全连接层，应当将其拉伸成一维向量：

bottleneck_values = np.squeeze(bottleneck_values)

转换成字符串格式，以逗号间隔：

```
bottleneck_string = ','.join(str(x) for x in bottleneck_values)
with open(bottleneck_path, 'w') as bottleneck_file:
    bottleneck_file.write(bottleneck_string)
```

同时生成一个 ***n_classes*** **维 0 向量 ground_truth，置对应的 label_index 为 1，以 ground_truth 作为其类别标签；**

bottlenecks.append(bottleneck_values) → train_bottlenecks

ground_truths.append(ground_truth) → train_ground_truths

得到训练数据的一个 ***Batch***；

(3) 将图片的 **bottleneck** 输入全连接层实现最终分类：

```
bottleneck_input = tf.placeholder(tf.float32, [None,
BOTTLENECK_TENSOR_SIZE], name='BottleneckInput')
ground_truth_input = tf.placeholder(tf.float32, [None, n_classes],
name='GroundtruthInput')
with tf.name_scope('full_connected_layer'):
    weights = tf.get_variable("weight", [BOTTLENECK_TENSOR_SIZE,
    n_classes], initializer=tf.truncated_normal_initializer(stddev=0.001))
    biases = tf.get_variable("biases", [n_classes],
    initializer=tf.zeros_initializer())
    logits = tf.matmul(bottleneck_input, weights) + biases
    #经过 Softmax 层处理
    final_tensor = tf.nn.softmax(logits)
```

```
#损失函数为交叉熵
cross_entropy = tf.sparse_softmax_entropy_with_logits(logits=logits,
labels=ground_truth_input)
cross_entropy_mean = tf.reduce_mean(cross_entropy)
#模型训练 op
train_step =
tf.GradientDescentOptimizer(LEARNING_RATE).minimize(cross_entroy_mean)
#模型验证 op
with tf.name_scope('evaluation'):
    correct_predication = tf.equal(tf.argmax(final_tensor, 1),
    tf.argmax(ground_truth_input, 1))
    validation_step = tf.reduce_mean(correct_predication, tf.float32)
#模型训练
sess.run(train_op, feed_dict={bottleneck_input: train_bottlenecks,
ground_truth_input: train_ground_truths})
#模型验证
for i % 100 == 0 or i + 1 == TOTAL_STEPS:
    validation_bottlenecks, validation_ground_truths =
    get_random_cached_bottlenecks(sess, n_classes, image_lists, BATCH,
    'validation', jpeg_data_tensor, bottleneck_tensor)
    validation_accuracy = sess.run(validation_step,
    feed_dict={ bottleneck_input: validation_bottlenecks,
                ground_truth_input: validation_ground_truths })
```

完整源码见“./src/imgration_learning”

8.4 循环神经网络

8.4.1 RNN

循环神经网络***(Recurrent Neural Network)CNN*** 用于解决序列化数据的预测问题，处于同一层的节点之间也存在连接，网络的当前状态由当前时刻的输入和前一时刻的状态共同决定：$\boldsymbol{C_t = R(x_t, C_{t-1}), t = 1, 2, \dots, T}$，基本构成单元叫做循环体；

输入序列：$\mathbb{X} = \{x_t\}_{t=0}^{T}$ ，$x_t = (x_t^{(1)}, x_t^{(2)}, \dots, x_t^{(N)})^T$为 ***t*** 时刻的输入向量，输入权值矩阵为：$W_i = [w_{jk}]_{M \times N}$ ，对于不同时刻的输入，其权值是共享的，即权值矩阵保持不变；

t-1 时刻的网络状态提供给当前时刻的网络，作为输入的一部分：

$C_{t-1} = (c_{t-1}^{(1)}, c_{t-1}^{(2)}, \dots, c_{t-1}^{(M)})^T$，状态权值矩阵为$W_c = [w_{ij}]_{M \times M}$ ；

t 时刻的网络状态既要输出给 ***t+1*** 时刻，也要计算生成当前时刻的输出：

$C_t = f(W_C \cdot C_{t-1} + W_i \cdot x_t + b_c), b_c = (b_c^{(1)}, b_c^{(2)}, \dots, b_c^{(M)})^T$；$f(x)$为激活函数；

$o_t = f(W_o \cdot C_t + b_o), b_o$为输出偏置；

对于每一个输入的时序序列$\mathbb{X} = \{x_t\}_{t=0}^{T}$即输入样本，时间 ***t*** 从 ***0*** 到 ***T*** 的过程(一个样本序列)，权值矩阵W_i，W_c，W_o以及偏置向量b_c和b_o均保持不变，将x_t看做序列$\mathbb{X}$的一个属性，因此 ***RNN*** 计算损失函数时要考量完整样本的损失，即整个时序上的输出的损失函数$\ell(o) = \sum_{t=0}^{T} \ell(o_t, o_{t+1})$，在实际工程应用过程中，超长的序列会带来梯度消散(Gradient Vanishing)或梯度爆炸(Gradient Explosion)问题，因此需要将序列进行截断处理，详见梯度消散/爆炸与序列截断；

RNN 只能解决在序列中预测位置距离相关信息较近的问题，若二者间的间隔增大时，***RNN*** 会丧失学习能力；

将 ***RNN*** 循环体在同一时刻重复多次即可得到深层循环神经网络***(deep RNN)***，每层循环体中的参数相同，而多层之间参数可以不同，同时在多层循环体之间的连接可以使用 ***dropout***，而在时序方向上不能使用 ***dropout***；

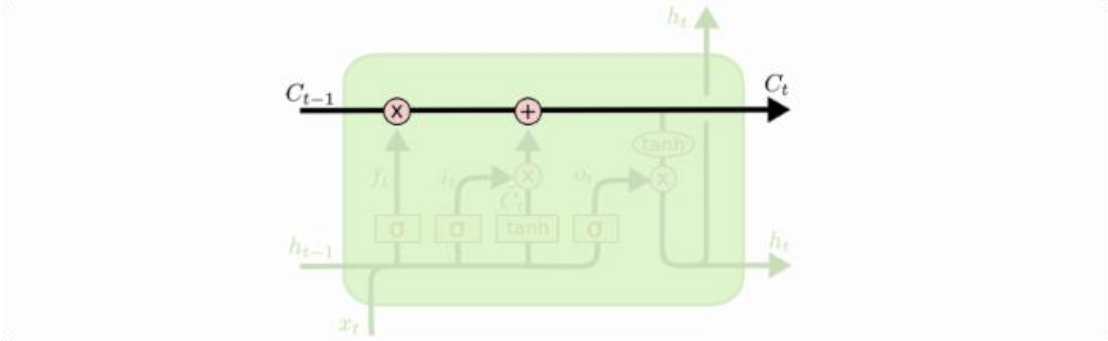
#实现基本的 ***LSTM Cell***

```
lstm = tf.nn.rnn_cell.BasicLSTMCell(LSTM_SIZE)
lstm = tf.nn.rnn_cell.DropoutWrapper(lstm, output_keep_prob=KEEP_PROB)
deep_lstm = tf.nn.rnn_cell.MultiRNNCell([lstm]*LAYERS)
state = deep_lstm.get_state(BATCH_SIZE, tf.float32)
#前向传播
for step in range(TRAINING_STEPS):
    if step > 0:
        tf.get_variables_scope().reuse_variables()
    deep_lstm_output, state = deep_lstm(input_tensor, state)
    final_output = fully_connected_layer(deep_lstm_output)
```

loss += calculate_loss(final_output, expected_output)

8.4.2 LSTM

长短期记忆网络***(Long Short Term Memory)LSTM***，可以解决预测问题对先前状态的长期依赖问题，在 ***RNN*** 中，$\boldsymbol{A_t = f(W_A \cdot A_{t-1} + W_X \cdot X_t + \alpha)}$，若激活函数$\boldsymbol{f(z) = tanh(z)}$，则循环体中仅有一个简单的$\boldsymbol{tanh}$结构，而 ***LSTM*** 在循环体内部增加了三种门控单元：



- **遗忘门：$\boldsymbol{f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)}$，遗忘门层输出一个与状态向量$\boldsymbol{C_{t-1}}$相同维度的向量$\boldsymbol{f_t = (f_t^{(1)}, f_t^{(2)}, \dots, f_t^{(|C_{t-1}|)})^T}$，其中：**

  $\boldsymbol{0 \le f_t^{(i)} \le 1, i = 1, 2 \dots, |C_{t-1}|}$，对前一时刻状态向量各个元素的保留概率，0 表示不保留，1 表示完全保留；
- **输入门：$\boldsymbol{i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)}$，需要更新的状态信息的概率；**

  $\boldsymbol{\widetilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)}$，生成新的候选状态信息；

  $\boldsymbol{C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t}$，输入门层通过对应向量的点乘运算并求和完成对旧的状态信息的遗忘和新的状态信息的添加，得到当前时刻的状态向量；
- **输出门：$\boldsymbol{o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)}$，当前时刻的输出门控；**
  $\boldsymbol{h_t = o_t * tanh(C_t)}$，输出门层对当前时刻的状态信息进行控制输出，得到预测值$\boldsymbol{h_t}$作为当前时刻的输出；

8.4.3 GRU

门控重置单元***(Gate Reset Unit)GRU***，在 ***LSTM*** 的基础上将输入门和遗忘门合并成一个更新门：**$\boldsymbol{r_t = \sigma(W_r \cdot [h_{t-1}, x_t])}$，作为上一时刻输出信号的门控；**
$\boldsymbol{z_t = \sigma(W_z \cdot [h_{t-1}, x_t])}$，作为当前时刻输出的更新门控；

输出候选信号$\boldsymbol{\widetilde{h}_t = tanh(W \cdot [r_t * h_{t-1} + x_t])}$；

当前时刻输出信号$\boldsymbol{h_t = (1 - z_t) * h_{t-1} + z_t * \widetilde{h}_t}$ ；

8.4.4 梯度消散/爆炸与序列截断

设有三个隐含层的神经网络：
$\boldsymbol{a_0 \to a_1 = \sigma(z_1) \to a_2 = \sigma(z_2) \to a_3 = \sigma(z_3) \to a_4 = \sigma(z_4) \to C}$，其中：
$\boldsymbol{a_l, (l = 1, 2, \dots, 4)}$ 表示隐含层输出，$\boldsymbol{z_l = w_l \cdot a_{l-1} + b_l, (l = 1, 2, 3, 4)}$，无论是权值 $\boldsymbol{w}$ 还是偏置 $\boldsymbol{b}$，其数值改变均会沿着梯度一直传递到最终的输出$\boldsymbol{C}$，即梯度的链式反应：

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial a_4}\frac{\partial a_4}{\partial a_3}\frac{\partial a_3}{\partial a_2}\frac{\partial a_2}{\partial a_1}\frac{\partial a_1}{\partial z_1}\frac{\partial z_1}{\partial w_1} = \frac{\partial C}{\partial a_4}\sigma'(z_4)w_4\sigma'(z_3)w_3\sigma'(z_2)w_2\sigma'(z_1)a_1;$$

若激活函数$\boldsymbol{\sigma(z) = \frac{1}{1+exp(-z)}}$，则有$\boldsymbol{\sigma'(z) = \sigma(z)[1 - \sigma(z)]}$，在$\boldsymbol{z = 0}$处取得最大值$\boldsymbol{\sigma'(0) = \frac{1}{4}}$，若$\boldsymbol{|w_l| \le 1}$，则有$\boldsymbol{|\sigma'(z_l)w_l| \le \frac{1}{4}}$，随着层数加深，梯度会呈指数下降直至消失，此时出现梯度消散(Gradient Vanishing)；
若$\boldsymbol{|w_l| \ge 1}$，且有$\boldsymbol{|\sigma'(z_l)w_l| \ge 1}$，则随着层数加深，会导致梯度指数上升直至发生上溢，此时出现梯度爆炸(Gradient Explosion)；
通常使用 ReLU 做为激活函数，$\boldsymbol{r(z) = max(0, z)}$，不会带来梯度消散问题；
在 ***RNN*** 的训练中，通常会在数据预处理时，将输入的样本序列截断成较短的长度来避免梯度消散和梯度爆炸问题；
在 Tensorflow 中可使用 tf.clip_by_global_norm(g, MAX_GRAD_NORM)进行梯度修剪：
指定了 MAX_GRAD_NORM 限定梯度向量的 L2 范数，对梯度向量 g 进行修剪，返回一个修剪后的梯度列表 grads 和中间计算变量 global_norm，形式化表达：

$\boldsymbol{g = [g_1, g_2, \dots, g_{|w|}]}$为待修剪梯度向量；

$\boldsymbol{||g||_2 = \sqrt{\sum_{i=1}^{|w|} g_i^2}}$ 即 global_norm，是梯度向量的 L2 范数；
传入参数 MAX_GRAD_NORM 以$\boldsymbol{N_c}$表示；
则有 $\boldsymbol{\hat{g}_i = \frac{g_i}{max(||g||_2, N_c)}}$；
即当梯度向量的 L2 范数大于指定的最大范数值时，就将所有梯度进行等比例缩放；

8.4.5 BPTT

时序反向传播算法***(Back Propagation Through Time)BPTT***，是 ***RNN*** 参数更新使用的算法：
状态向量$\boldsymbol{s_t = \phi(Ux_t + Ws_{t-1})}$；输出向量$\boldsymbol{o_t = \varphi(Vs_t)}$；

损失函数$\boldsymbol{L_t = L(o_t, y_t)}$，其中$\boldsymbol{y_t}$是常数，且有总损失函数$\boldsymbol{L = \sum_{t=1}^{T} L_t}$；

设有 $\boldsymbol{\tilde{s}_t = Ux_t + Ws_{t-1}}$；
$\boldsymbol{\tilde{o}_t = Vs_t}$；

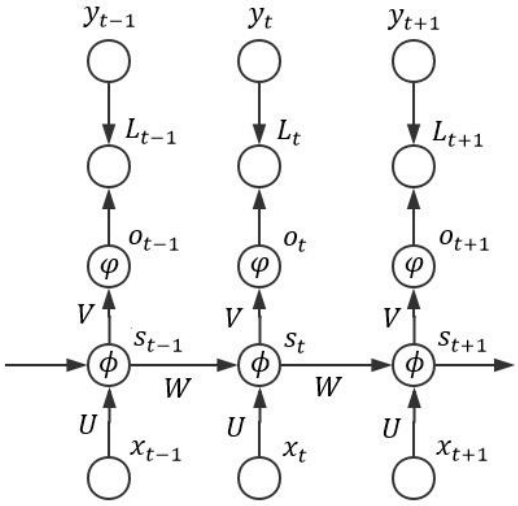
损失函数对权值矩阵$\boldsymbol{V}$求导：$\boldsymbol{\frac{\partial L}{\partial V} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial o_t} * \varphi'(\tilde{o}_t) \cdot s_t}$，$\boldsymbol{s_t = \phi(Ux_t + Ws_{t-1})}$与

矩阵 V 无关，因此对权值矩阵 V 的反向传播与时序无关；

$\frac{\partial L}{\partial U} = \sum_{t=1}^{T}\sum_{k=1}^{t}\frac{\partial L_t}{\partial \tilde{s}_k}\cdot\frac{\partial \tilde{s}_k}{\partial U} = \sum_{t=1}^{T}\sum_{k=1}^{t}\frac{\partial L_t}{\partial \tilde{s}_k}\cdot x_k^T$ ；

$\frac{\partial L}{\partial W} = \sum_{t=1}^{T}\sum_{k=1}^{t}\frac{\partial L_t}{\partial \tilde{s}_k}\cdot\frac{\partial \tilde{s}_k}{\partial W} = \frac{\partial L}{\partial U} = \sum_{t=1}^{T}\sum_{k=1}^{t}\frac{\partial L_t}{\partial \tilde{s}_k}\cdot s_{k-1}^T$；

即梯度是沿着时间通道进行传播的：



8.4.6 CNN 与 RNN 对比

将含有多个隐含层的神经网络统称为深度神经网络***(Deep Neural Network)DNN***，全连接神经网络***(Full Connected Nerual Network)FCNN***、卷积神经网络 ***CNN*** 和递归神经网络 ***RNN*** 的区别：

- **全连接神经网络又称前馈神经网络*(Feed-forward Neural Network)FNN*** 仅仅单纯地接收样本数据，认为样本间是相互独立的，其性能在一定程度上依赖于前期数据处理中的特征工程；
- **卷积神经网络 *CNN*** 注重挖掘单个样本的结构性数据相邻区域间的关联，生成抽象的极具有概括性的特征；
- **递归神经网络 *RNN* 可以使用同一套网络结构处理不同的序列化数据，注重于挖掘样本与样本之间时序上的关联，*RNN* 是时间循环神经网络*(Recurrent Neural Netwok)*和结构递归神经网络*(Recursive Neural Network)*的统称；**

8.4.7 NLP

自然语言处理***(Natural Language Processing)NLP*** 是循环神经网络的一个典型应用，称为语言模型，语言模型要解决两个问题：

- 学习问题：计算一个句子$S = (w_1, w_2, \dots, w_m)$出现的概率即：
  $P(S) = P(w_1, w_2, \dots, w_m) =$
  $P(w_1)P(w_2|w_1)P(w_3|w_2, w_1)\cdots P(w_m|w_{m-1}, \dots, w_1)$；
  传统方法常基于 ***n-gram*** 假设处理语言模型的学习问题，即假设句子中每个

词语的出现仅与之前的 n 个单词有关，从而简化模型的学习：
$P(S) = \prod_{i=1}^{m} P(w_i|w_{i-1}, \cdots, w_{i-n+1})$；

- 预测问题：语言模型的预测问题即根据已学习的模型和输入序列预测下一个单词出现的概率$P(w_i|w_{i-1}, \cdots, w_1)$；

语言模型效果评价指标：

$perplexity(S) = p(w_1, w_2, \ldots, w_m)^{-\frac{1}{m}}$ ；

$p(w_1, w_2, \ldots, w_m) = \prod_{i=1}^{m} p(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)$；

得到：$perplexity(S) = \sqrt[m]{\frac{1}{\prod_{i=1}^{m} p(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)}}$ ；

防止连乘造成的数值下溢，取对数，得到模型的对数复杂度：

$log[perplexity(S)] = -\frac{1}{m}\sum_{i=1}^{m} log[p(w_i|w_{i-1}, w_{i-2}, \ldots, w_1)]$；

使用 MultiRNN 实现语言模型：

- **数据集：PTB(Penn Treebank Dataset)文本数据集，词汇量 10000，句子结束标记为换行符，稀有单词标记为<unk>；**
- 需要借助 **Tensorflow** 官方提供的 **tf.models package**，在 **tensorflow1.0** 及其更高版本中，已经将 **models** 分离出来，要使用 **models** 中提供的 **API** 以及运行 **tutorials** 中的 **research demo**，需要从 https://github.com/tensorflow/models **clone** 仓库到本地，导入所需的包即可：

  import tensorflow_ptb_reader as reader

  train_data, validate_data, test_data = reader.ptb_raw_data(DATA_PATH)

  使用函数 reader.ptb_iterator(train_data, BATCH_SIZE, NUM_STEPS)将训练数据组织成包含 N = BATCH_SIZE 个序列，每个序列长度为 m = NUM_STEPS 的 batch；

  $x = [w_1, w_2, w_3, \ldots, w_{m-1}, w_m]$ **as feature**

  $y\_ = [w_2, w_3, \ldots, w_{m-1}, w_m, w_{m+1}]$ **as label**

  …

  共 N 个$[x, y_]$组成一个 **Batch** 作为 **RNN** 的输入；

  (1) PTBModel 构建模型，定义各类操作：

  ```
  class PTBModel(object):
      def __init__(self, is_training, batch_size, num_steps):
      …
      #以下代码都在初始化方法中
  ```

  读入参数作为类属性并定义输入，输入是一个矩阵**[batch_size, num_steps]**：

  ```
  self.batch_size = batch_size
  self.num_steps = num_steps
  self.inputs = tf.placeholder(tf.int32, shape=[batch_size, num_steps],
      name='x-input')
  self.targets = tf.placeholder(tf.int32, shape=[batch_size, num_steps],
      name='y-input')
  ```

  定义单个 LSTMCell 结构：

  ```
  lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(HIDDEN_SIZE,
  ```

state_is_tuple=True)

其中，**LSTM** 隐含层规模为 **size=HIDDEN_SIZE**，由 **LSTM** 的更新方程

$$\boldsymbol{f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)};$$

$$\boldsymbol{i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)};$$

$$\boldsymbol{\widetilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)};$$

$$\boldsymbol{C_t = f_t * C_{t-1} + i_t * \widetilde{C}_t};$$

$$\boldsymbol{o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o};$$

$$\boldsymbol{h_t = o_t * tanh(C_t)};$$

可以看出，**LSTM** 内部本质上使用的也是全连接网络，将前向传导的结果经过门控处理，因此在定义 **LSTM** 结构时，要指定隐含层的规模；

此外，**state_is_tuple** 参数指定了输出的状态值 **c** 与输出值 **h** 是否以元组形式返回，若为 **False**，则拼接成一个向量；

训练时，加入层间 **dropout**：

```
if is_train==True:
    lstm_cell = tf.nn.rnn_cell.DropoutWrapper(lstm_cell,
        output_keep_prob=KEEP_PROB)
```

构造两层 **LSTM** 网络，第一层输出作为第二层的输入

```
cell = tf.nn.rnn_cell.MultiRNNCell([lstm_cell]*NUM_LAYERS,
        state_is_tuple=True)
```

初始化状态为 0 向量：

```
self.initial_state = cell.zero_state(batch_size, tf.float32)
```

embedding layer **将字典中所有的单词转换成单词向量：**

```
embedding = tf.get_variable("embedding", [VOCAB_SIZE, HIDDEN_SIZE])
```

将一个 ***Batch*** 的训练样本转换成对应的单词向量，self.input_data 给出的是单词在字典中的 id，embedding 建立了一个与字典容量相同的空间，将单词 id 映射成 HIDDEN_SIZE 维的向量，此映射表示为$\boldsymbol{f: w \in \mathcal{W}^{|\mathcal{D}|} \rightarrow v \in \mathcal{V}^{|\mathcal{H}|}}$，通过查表 **tf.embedding_lookup(embedding_matrix, id)**可以得到指定单词的向量表示，详见 Word2vec，此时输入 **inputs 为[batch_size, num_steps, HIDDEN_SIZE]** 矩阵；

```
inputs = tf.nn.embedding_lookup(embedding, self.input_data)
outputs = []
states = self.initial_state
```

模型的评价要对整个时序上的损失之和进行考量，因此，要保存在整个时序上的输出结果；

```
with tf.variable_scope("RNN", reuse=tf.AUTO_REUSE):
    for step in range(num_steps):
        cell_output, state = lstm_cell([:, step, :], state)
        outputs.append(cell_output)
```

每一个时刻输出的 cell_output 是一个**[batch_size, 1, HIDDEN_SIZE]**的矩阵，经过 **num_steps** 步后，将整个时序上的样本输入，得到 **batch** 中的全部样本的预测输出，此输出与输入有相同的维度；

得到当前输入 ***Batch*** 前向传播的结果后还要经过一个全连接层进行处理，最终输出的是对下一时刻序列中可能会出现的单词的概率，即一个长度为 **VOCAB_SIZE** 的向量，全连接层输出维度为 **VOCAB_SIZE**，输入应当是对应的单词向量，其维度为 **HIDDEN_SIZE**：

```
outputs = tf.reshape(tf.concat(outputs, 1), [-1, HIDDEN_SIZE])
weight = tf.get_variable("weight", shape=[HIDDEN_SIZE, VOCAB_SIZE])
bias = tf.get_variable("bias", shape=[VOCAB_SIZE])
logits = tf.matmul(outputs, weight) + bias
```

此处变量 **weight** 和 **bias** 的初始化值统一由主函数调用 ***PTBModel*** 时传入；

计算序列的交叉熵，设置序列中各个时刻的预测值占有相同的权重：

```
loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example([logits],
        [tf.reshape(self.targets, [-1])],
        [tf.ones([batch_size*num_steps], dtype=tf.float32)])
```

计算 Mini-Batch Loss 并保存最终的状态向量：

```
self.cost = tf.reduce_sum(loss)/batch_size
self.final_state = state
```

模型训练时，还要对梯度进行修剪，防止出现梯度膨胀和梯度消散：

```
grads, _ = tf.clip_by_global_norm(tf.gradients(self.cost,
        tf.trainable_variables()), MAX_GRAD_NORM)
optimizer = tf.train.GradientDescentOptimizer(LEARNING_RATE)
self.train_op = optimizer.apply_gradients(zip(grads,
        tf.trainable_variables()))
```

tf.gradients(self_cost, trainable_variables)返回一个总损失函数对所有变量的梯度列表：$\boldsymbol{g} = [\boldsymbol{g_1}, \boldsymbol{g_2}, \dots, \boldsymbol{g_{|w|}}]$，$\boldsymbol{g_i} = \frac{\partial \boldsymbol{L}(\boldsymbol{w})}{\partial \boldsymbol{w_i}}, \boldsymbol{i} = \boldsymbol{1}, \boldsymbol{2}, \dots, |\boldsymbol{w}|$；

tf.clip_by_global_norm(g, MAX_GRAD_NORM)是对梯度进行修剪，见[梯度消散/爆炸与序列截断]；

函数 optimizer.apply_gradients(zip(grads, tf.trainable_variables()))使用已修剪过的梯度向量对所有参数进行更新，替代 optimizer.minimize(self.cost)；

(2) 输入数据，运行模型 run_epoch：

调用 ptb_reader.iterator(data, model.batch_size, model.num_steps)从数据集中生成 batch 供模型训练使用：

```
for step, (xs, ys) in enumerate(reader.ptb_reader.iterator(data,
        model.batch_size, model.num_steps)):
    cost, state, _ = sess.run([model.cost, model.final_state, train_op],
        feed_dict={self.input_data: xs, self.target: ys, self.initial_state=state})
    total_cost += cost
    iters += model.num_steps
    count += 1
return np.exp(total_cost/iters)
```

total_cost 返回的数值是$\boldsymbol{\ell_T(p)} = -\sum_{i=1}^{m} \boldsymbol{log}[\boldsymbol{p}(\boldsymbol{w_i}|\boldsymbol{w_{i-1}}, \boldsymbol{w_{i-2}}, \dots, \boldsymbol{w_1})]$，**要求模型的复杂度，则有**$\boldsymbol{perplexity(p)} = \boldsymbol{exp}[\frac{\boldsymbol{\ell_T(p)}}{\boldsymbol{m}}]$，**作为模型的评价指标；**

(3) 主函数，运行测试模型代码：

获得训练数据集 train_data、验证数据集 validate_data 和测试数据集 test_data；
定义 **initializer** 初始化模型中所有的参数：

```
initializer = tf.random_uniform_initializer(-0.05, 0.05)
```

生成训练模型与验证模型，两者区别在于是否需要反向传播优化参数：

```
with tf.variable_scope("language_model", reuse=tf.AUTO_REUSE,
                       initializer=initializer):
    train_model = PTBModel(True, TRAIN_BATCH_SIZE, TRAIN_NUM_STEP)
with tf.variable_scope("language_model", reuse=tf.AUTO_REUSE,
                       initializer=initializer):
    eval_model = PTBModel(False, EVAL_BATCH_SIZE, EVAL_NUM_STEP)
```

启动会话，训练并评测模型：

```
with tf.Session() as sess:
    sess.run([tf.local_variables_initializer, tf.global_variables_initializer])
    for epoch in range(NUM_EPOCHS):
        run_epoch(session, train_model, train_data, train_model.train_op,
                  True) # outlog 输出日志控制
        validate_perplexity = run_epoch(session, eval_model, validate_data,
                  tf.no_op(), False)
    test_perplexity = run_epoch(session, eval_model, eval_data, tf.no_op(),
                  False)
```

运行源码，测试入口：

```
if __name__ == 'main':
    tf.app.run()
```

完整源码见“./src/language_model”；

8.4.8 Word2vec

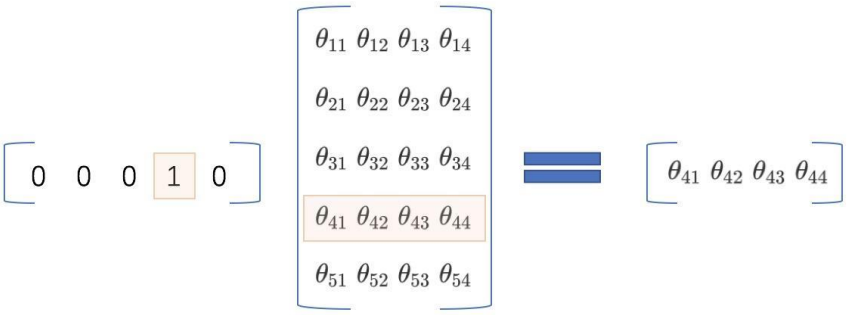
在语言模型中，常用的单词表达方式有两种：

- **One-hot Representation：**
  以每个单词在词汇表(字典)中出现的位置作为标识，生成此单词的向量，若字典容量为$|\mathcal{D}|$，则生成的单词向量维度为$|\mathcal{D}|$，即有：

  字典 $\mathcal{D} = [\boldsymbol{d_1}, \boldsymbol{d_2}, \dots, \boldsymbol{d_{|\mathcal{D}|}}]^{\boldsymbol{T}}$，

  映射单词向量为 $\boldsymbol{v} = [\boldsymbol{I}(\boldsymbol{w} = \boldsymbol{d_1}), \boldsymbol{I}(\boldsymbol{w} = \boldsymbol{d_2}), \dots, \boldsymbol{I}(\boldsymbol{w} = \boldsymbol{d_{|\mathcal{D}|}})]^{\boldsymbol{T}}$，$\boldsymbol{w}$ **为待表示的单词，**$\boldsymbol{v}$ **为表示后的向量，1 元素在** $\boldsymbol{v}$ **中的位置即单词** $\boldsymbol{w}$ **在字典中的位置；**
  One-hot 编码得到的向量维度较大，而且假设各个维度之间彼此独立，此时网络中参数较多；
- **Distributed Representation：**
  其数学含义为：
  $\boldsymbol{f}: \boldsymbol{w} \in \boldsymbol{\mathcal{W}}^{|\mathcal{D}|} \rightarrow \boldsymbol{v} \in \boldsymbol{\mathcal{V}}^{|\mathcal{H}|}$，即将**文本空间**$\mathcal{D}$ **(字典)中的单词**映射成一个高维空间$\mathcal{H}$中的向量；
  使用向量的形式表示词语，可以借助距离度量较为方便的寻找词语间的

相似性，如余弦距离$\boldsymbol{d_{ij} = dist(v_i,\ v_j) = \frac{<v_i, v_j>}{||v_i||_2 \cdot ||v_j||_2}}$，另外 google 提供的 word2vec tools 还提供了一种 analogy 性质，类似于：
$\boldsymbol{dist(Paris, Franch) = dist(Beijing, China)}$；
通常 embedding vector 由 one-hot 编码转换得到，寻找单词之间的联系：使用无监督聚类方法或者寻找一个映射，将 one-hot 向量映射到另外一个高维空间中，tensorflow.embedding_lookup(embedding_matrix, id)提供了一种直接根据单词 id 查找对应 embedding vector 的方法：
使用 embedding_lookup()函数将字典中所有的单词转换成单词向量：
embedding = tf.get_variable("embedding", **[VOCAB_SIZE, HIDDEN_SIZE]**)
将一个 Batch 的训练样本转换称对应的单词向量，此时输入 **inputs** 为 **[batch_size, num_steps, HIDDEN_SIZE]**矩阵；
inputs = tf.nn.embedding_lookup(**embedding**, self.input_data)



Embedding 在 ***LSTM Language Model*** 中是以一个全连接层存在的，模型的输入 **self.input_data** 只有经过 ***embedding*** 层处理后得到的输出 **inputs** 才能提供给 ***LSTM Cell*** 进行预测，模型的训练过程中包含了 **Embedding Layer** 参数的优化；

8.4.9 TF-Learn

Tensorflow 高层封装工具 **TFLearn** 可以将模型实现模板化：
import tensorflow.contrib.learn as learn

- 自定义模型 **lstm_model**：
  (1) 定义网络结构和前向传播过程：

```
lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(HIDDEN_SIZE, state_is_tuple=True)
lstm_cell = tf.nn.rnn_cell.DropoutWrapper(lstm_cell,
                output_keep_prob=KEEP_PROB)
lstm = tf.nn.rnn_cell.MultiRNNCell([lstm_cell]*NUM_LAYERS)
x = tf.placeholder(dtype, [BATCH_SIZE, NUM_FEATURES], name='x-input')
y_ = tf.placeholder(dtype, [BATCH_SIZE, NUM_FEATURES], name='y-input')
y = tf.nn.dynamic_rnn(lstm, x, dtype=dtype)
```

  (2) 设置全连接层，用于完成最后的预测：

```
with tf.variable_scope("fcc", reuse=tf.AUTO_REUSE):
    weight = tf.get_variable("weight", [HIDDEN_SIZE, LABEL_LENGTH])
    bias = tf.get_variable("bias", [LABEL_LENGTH])
```

logits = tf.matmul(y, weight) + bias

(3) 定义损失函数及优化算法选择:

loss = tf.contrib.legacy_seq2seq.sequence_loss_by_example(logits, y_, example_loss_weights)

grads, _ = tf.clip_by_global_norm(tf.gradients(loss, tf.trainbale_variables()))

train_op = tf.GradientDescentOptimizer(LEARNING_RATE).apply_gradients(zip(grads, tf.trainable_variables()))

(4) 固定返回值：

return logits, loss, train_op

- 对模型进行封装：regressor = **learn**.Estimator(model_fn=lstm_model)
- 训练模型，执行迭代：

regressor.fit(train_data, train_label, batch_size=BATCH_SIZE, steps=TRAIN_STEPS)

- 结果预测：**predicts** = **[predict] for predict in regressor.predict(test_data)**
- 计算精度：**rmse = np.sqrt(((predicts – test_label)**2).mean(axis=0))**

8.5 RBM 与吉布斯采样

8.5.1 受限玻尔兹曼机

受限玻尔兹曼机***(Restricted Boltzman Machine)RBM***，是一种生成式随机网络***(Generative Stochastic Neural Network)***，由观测变量节点***(Visible Unit)***与隐藏变量***(Hidden Uint)***节点构成，观测变量与隐藏变量都是布尔型数据，整个网络是一张二部图，仅观测变量与隐藏变量异类变量之间存在连接，同类变量之间不存在连接，设观测变量节点集合为$\boldsymbol{X}=\{\boldsymbol{x}^{(k)}\}_{k=1}^{|X|}$，隐含变量节点集合$\boldsymbol{R}=\{\boldsymbol{r}^{(l)}\}_{l=1}^{|R|}$，连接权值矩阵为：

$\boldsymbol{W}=[\boldsymbol{w_{kl}}]_{|X|\times|R|}$；偏置向量$\boldsymbol{a}=(\boldsymbol{a_1},\boldsymbol{a_2},\dots,\boldsymbol{a}_{|X|})^{\boldsymbol{T}}$，$\boldsymbol{b}=(\boldsymbol{b_1},\boldsymbol{b_2},\dots,\boldsymbol{b}_{|R|})^{\boldsymbol{T}}$；基于观测变量和隐含变量的联合配置***(Joint Configuration)***能量建模：

$\boldsymbol{E(X,R;\theta)}=-\sum_{k=1}^{|X|}\sum_{l=1}^{|R|}\boldsymbol{w_{kl}}\,\boldsymbol{x}^{(k)}\boldsymbol{r}^{(l)}-\sum_{k=1}^{|X|}\boldsymbol{x}^{(k)}\boldsymbol{a_k}-\sum_{l=1}^{|R|}\boldsymbol{r}^{(l)}\boldsymbol{b_l}$；

$\boldsymbol{\theta}=(\boldsymbol{W},\boldsymbol{a},\boldsymbol{b})$**是模型的参数组；**

依据联合配置能量得到其联合概率分布：

$\boldsymbol{P_\theta(X,R)}=\frac{\boldsymbol{1}}{\boldsymbol{Z(\theta)}}\boldsymbol{exp}[-\boldsymbol{E(X,R;\theta)}]$；

即有$\boldsymbol{P_\theta(X,R)}=\frac{\boldsymbol{1}}{\boldsymbol{Z(\theta)}}\boldsymbol{exp}[\boldsymbol{x^TWr}+\boldsymbol{a^Tx}+\boldsymbol{b^Tr}]$；**其中**$\boldsymbol{x}=(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)},\dots,\boldsymbol{x}^{(|X|)})^{\boldsymbol{T}}$，

$\boldsymbol{r}=(\boldsymbol{r}^{(1)},\boldsymbol{r}^{(2)},\dots,\boldsymbol{r}^{(|R|)})^{\boldsymbol{T}}$**为变量节点取值张成的向量；**

控制变量，求观测变量的边缘分布：$\boldsymbol{P_\theta(x)}=\frac{\boldsymbol{1}}{\boldsymbol{Z(\theta)}}\sum_r\boldsymbol{exp}[\boldsymbol{x^TWr}+\boldsymbol{a^Tx}+\boldsymbol{b^Tr}]$；

依据观测数据估计参数，构造似然函数：

$\boldsymbol{L(\theta)}=\sum_{n=1}^{N}\boldsymbol{log}[\boldsymbol{P_\theta(x_n)}]$；

模型学习的任务是求参数使得似然函数极大化：

$\boldsymbol{\theta^*}=\boldsymbol{argmax_\theta}\,\boldsymbol{L(\theta)}=\boldsymbol{argmax_\theta}\sum_{n=1}^{N}\boldsymbol{log}[\boldsymbol{P_\theta(x_n)}]$；

RBM 的学习算法是对比散度法***(Contrastive Divergence)CD*** 算法：

$\frac{\boldsymbol{\partial L(\theta)}}{\boldsymbol{\partial\theta}}=\sum_{n=1}^{N}\frac{\boldsymbol{\partial}}{\boldsymbol{\partial\theta}}\boldsymbol{log}(\sum_r\boldsymbol{exp}[\boldsymbol{x_n^TWr}+\boldsymbol{a^Tx_n}+\boldsymbol{b^Tr}])-\sum_{n=1}^{N}\frac{\boldsymbol{\partial log[Z(\theta)]}}{\boldsymbol{\partial\theta}}$；

由于概率归一化因子$\boldsymbol{Z(\theta)}=\sum_{x,r}\boldsymbol{exp}[-\boldsymbol{E(X,R;\theta)}]$，**考虑节点各种取值组合；**

得到：

$\frac{\boldsymbol{\partial L(\theta)}}{\boldsymbol{\partial\theta}}=$

$\sum_{n=1}^{N}\sum_r\frac{\boldsymbol{exp}[\boldsymbol{x_n^TWr}+\boldsymbol{a^Tx_n}+\boldsymbol{b^Tr}]}{\sum_r\boldsymbol{exp}[\boldsymbol{x_n^TWr}+\boldsymbol{a^Tx_n}+\boldsymbol{b^Tr}]}\cdot\frac{\boldsymbol{\partial}[\boldsymbol{x_n^TWr}+\boldsymbol{a^Tx_n}+\boldsymbol{b^Tr}]}{\boldsymbol{\partial\theta}}-\sum_{n=1}^{N}\sum_{x,r}\frac{\boldsymbol{exp}[\boldsymbol{x^TWr}+\boldsymbol{a^Tx}+\boldsymbol{b^Tr}]}{\sum_{x,r}\boldsymbol{exp}[\boldsymbol{x^TWr}+\boldsymbol{a^Tx}+\boldsymbol{b^Tr}]}\cdot$

$\frac{\partial[x^T W r + a^T x + b^T r]}{\partial \theta} =$

$\sum_{n=1}^{N} \mathbb{E}_{\tilde{P}(r|x_n;\theta)} \left[\frac{\partial[x_n^T W r + a^T x_n + b^T r]}{\partial \theta}\right] - \sum_{n=1}^{N} \mathbb{E}_{P(x,r|\theta)} \left[\frac{\partial[x^T W r + a^T x + b^T r]}{\partial \theta}\right]$;

第一部分中的$\tilde{P}(r|x_n;\theta)$表示在限定观测节点数据为样本x_n时，隐含节点的经验分布；

第二部分$P(x,r|\theta)$表示观测节点与隐含节点的联合分布，无法直接计算，只能通过 Gibbs Sampling 得到其近似值；

不考虑求和项，可得到各个参数的梯度：

$\frac{\partial \ell_n(\theta)}{\partial w_{kl}} = \mathbb{E}_{\tilde{P}(r|x_n;\theta)}[x_k r_l] - \mathbb{E}_{P(x,r|\theta)}[x_k r_l]$ ；

$\frac{\partial \ell_n(\theta)}{\partial a_k} = \mathbb{E}_{\tilde{P}(r|x_n;\theta)}[x_k] - \mathbb{E}_{P(x,r|\theta)}[x_k]$ ；

$\frac{\partial \ell_n(\theta)}{\partial b_l} = \mathbb{E}_{\tilde{P}(r|x_n;\theta)}[r_l] - \mathbb{E}_{P(x,r|\theta)}[r_l]$ ；

使用 Gibbs Sampling 可以得到参数梯度的近似

CD 算法：取样本数据赋予观测节点x_1，随机初始化连接权值矩阵$W = [w_{kl}]_{|X|\times|R|}$

以及偏置向量$a = (a_1, a_2, \dots, a_{|X|})^T$，$b = (b_1, b_2, \dots, b_{|R|})^T$；

外循环：**for step in range(*MAX_STEPS*):**

(1) 以给定观测数据生成隐含数据的条件分布，对所有$|R|$个隐含节点：

$P(r|x_1) = \prod_{l=1}^{|R|} P(r^{(l)}|x_1)$，其中$P(r^{(l)} = 1|x_1) = \frac{1}{1+exp[-\sum_{k=1}^{|X|} w_{kl} x_1^{(l)} - a_l]}$；

上式表示给观测节点取值为x_1条件下，$r^{(l)}$以$P(r^{(l)} = 1|x_1)$的概率为 1，否则为 0，按照节点取值的概率分布得到所有隐含节点的状态：

$r_1^{(l)} = I[P(r^{(l)} = 1|x_1) > 0.5], l = 1, 2, \dots, |R|$；

(2) 由**(1)**中得到的隐含节点的取值r_1，对$|X|$个可见节点层进行重构，

$P(x|r_1) = \prod_{k=1}^{|X|} P(x^{(k)}|r_1)$，其中$P(x^{(k)} = 1|r_1) = \frac{1}{1+exp[-\sum_{l=1}^{|R|} w_{kl} r_1^{(l)} - b_k]}$；

上式表示给定隐含节点取值r_1条件下，$x^{(k)}$以$P(x^{(k)} = 1|r_1)$的概率为 1，否则为 0，按照节点取值的概率分布重构所有观测节点的状态：

$x_2^{(k)} = I[P(x^{(k)} = 1|r_1) > 0.5], k = 1, 2, \dots, |X|$；

(3) 按照重构的观测节点取值，得到隐含节点的取值：

$P(r|x_2) = \prod_{l=1}^{|R|} P(r^{(l)}|x_2)$，其中$P(r^{(l)} = 1|x_2) = \frac{1}{1+exp[-\sum_{k=1}^{|X|} w_{kl} x_2^{(l)} - a_l]}$；

$r_2^{(l)} = I[P(r^{(l)} = 1|x_2) > 0.5], l = 1, 2, \dots, |R|$；

(4) 更新参数：

$W \leftarrow W + \eta \cdot [P(r_1^{(\cdot)} = 1|x_1) \cdot x_1^T - P(r_2^{(\cdot)} = 1|x_2) \cdot x_2^T]$；

$\boldsymbol{a} \leftarrow \boldsymbol{a} + \boldsymbol{\eta} \cdot (\boldsymbol{x_1} - \boldsymbol{x_2})$；

$\boldsymbol{b} \leftarrow \boldsymbol{b} + \boldsymbol{\eta} \cdot [\boldsymbol{P}\left(\boldsymbol{r_1^{(.)}} = \boldsymbol{1} \middle| \boldsymbol{x_1}\right) - \boldsymbol{P}\left(\boldsymbol{r_2^{(.)}} = \boldsymbol{1} \middle| \boldsymbol{x_2}\right)]$；

观测单元数量$|\boldsymbol{X}|$等于训练数据的特征维度，隐含节点$|\boldsymbol{R}|$个数需要事先指定；

8.5.2 吉布斯采样

吉布斯采样*(Gibbs Sampling)*：对 ***K*** 维随机向量$\boldsymbol{X} = (\boldsymbol{X^{(1)}}, \boldsymbol{X^{(2)}}, \dots, \boldsymbol{X^{(K)}})$，无法得知其联合概率分布，但已知给定其他 ***K-1*** 个分量的条件下，第 ***k*** 个分量的条件分布：

$\boldsymbol{P}\left(\boldsymbol{X^k} \middle| \boldsymbol{X^{k^-}}\right)$; $\boldsymbol{X^{k^-}} = (\boldsymbol{X^{(1)}}, \boldsymbol{X^{(2)}}, \dots, \boldsymbol{X^{(k-1)}}, \boldsymbol{X^{(k-2)}}, \dots, \boldsymbol{X^{(K)}})$；

从随机向量的任意状态出发$\boldsymbol{x_0} = (\boldsymbol{x_0^{(1)}}, \boldsymbol{x_0^{(2)}}, \dots, \boldsymbol{x_0^{(K)}})$，依据条件概率分布$\boldsymbol{P}\left(\boldsymbol{X^k} \middle| \boldsymbol{X^{k^-}}\right)$，迭代地对其各个分量进行采样，随着采样次数的增加，随机向量$\boldsymbol{x_n} = (\boldsymbol{x_n^{(1)}}, \boldsymbol{x_n^{(2)}}, \dots, \boldsymbol{x_n^{(K)}})$以 ***n*** 的几何级数速率逼近随机向量的联合概率分布；

应用于 ***RBM*** 模型中，可以使用吉布斯采样法得到联合概率分布$\boldsymbol{P_\theta(X, R)}$：

$\boldsymbol{r_0 \sim P_\theta(r|x_0)}$ → $\boldsymbol{x_1 \sim P_\theta(x|r_0)}$ → $\boldsymbol{r_1 \sim P_\theta(r|x_1)}$ → $\boldsymbol{x_2 \sim P_\theta(x|r_1)}$ → …

→$\boldsymbol{x_{k+1} \sim P_\theta(x|r_k)}$

在采样步数足够大的情况下，可以得到近似联合概率分布$\boldsymbol{P_\theta(X, R)}$；

8.6 GAN

生成式对抗网络***(Generative Adversarial Network)GAN***，设计一个生成器***G(Generator)***，一个判别器***D(Discriminator)***：
神经网络的训练过程中，对训练样本加入微小的扰动可能会带来错误的输出，此时称这种样本为**对抗样本**，对抗样本的误识别是由神经网络的线性性质带来的$(\boldsymbol{w^T x + b})$；
生成式对抗网络可以从样本中学习出新的样本，生成器学习样本的分布，用服从某一分布***(Gaussian/Evenly)***的噪声 ***z*** 生成一个类似于真实训练数据的**伪样本**，评价结果是，二者越接近越好；判别器则接受混杂了伪样本的训练数据集，若接收到真实样本则输出一个较大的概率值，若接收到一个伪样本则输出一个较小的概率值；
训练时，固定生成器 ***G***，更新判别器 ***D*** 的权重，进而固定判别器 ***D*** 的权重，更新生成器的权重，交替进行，直至达到纳什均衡状态，此时，生成器输出的样本分布接近于训练样本中的数据分布，而判别器 ***D*** 的分类结果接近 ***50%***，等同于随机猜测；

$$\boldsymbol{min_G\ max_D\ V(G,D)} = \mathbb{E}_{\boldsymbol{x\sim\tilde{P}(x)}}[\boldsymbol{logD(x)}] + \mathbb{E}_{\boldsymbol{z\sim P_z(z)}}[\boldsymbol{log(1-D(G(z)))}];$$

8.7 Tensorflow

8.7.1 三大组件

- 计算图 ***Graph***：计算模型，用于管理不同的集合，**节点**是其基本的运算单元，**边**是节点间的依赖关系：
  tf.get_default_graph()获取当前默认计算图；
  tf.Graph()生成新的计算图；
- 张量 ***Tensor***：数据模型，本身不存储数据，只是对运算结果的引用：
  tf.constant(shaped_array, name, dtype)
  tensor.get_shape()获取张量的维度信息；
  tf.Session().run(result)运行张量，输出运算结果；
- 会话 ***Session***：运算模型，管理 tensorflow 的系统资源和计算操作：
  with tf.Session() as sess:　　#上下文管理机制
  　　sess.run(…)

在 Tensorflow 中，要同时完成多个操作，有两种机制可以使用：

- **使用 tf.control_dependencies()绑定多个操作，定义一个操作占位符：**

```
with tf.control_dependencies([train_step, variables_average_op]):
    train_op = tf.no_op(name='train')
```

- **使用 tf.group()机制：**

train_op = tf.group(train_step, variables_average_op)，若要求 train_step 先执行，则可以通过以下代码实现

```
with tf.control_dependencies([train_step]):
    train_op = tf.group(variables_average_op)
```

Tensorflow 通过元图(MetaGraph)记录计算图中的节点信息以及运行计算图中节点所需要的数据，元图的数据格式由 MetaGraphDef Protocol Buffer 定义，以.meta 为后缀的文件保存：

```
Message MetaGraphDef{
    MetaInfoDef meta_info_def=1;    #元数据：计算图版本号，运算方法信息
    GraphDef graph_def = 2;         #计算节点间的连接信息
    SaverDef saver_def = 3;         #持久化所需的参数
    map<string, CollectionDef> collection_def = 4;      #集合信息
}
```

计算图中所有变量的取值保存在 **model.ckpt** 文件中，是一个 **key-value 列表，以 SSTable 格式存储；**

8.7.2 Collection

在计算图 ***Graph*** 中，使用集合***(collection)***管理不同类别的计算资源(张量、变量、队列等)：

```
Tf.add_to_collection()          #将资源加入集合
Tf.get_collection()             #从集合中读取资源
```

常用的集合：

Tf.GraphKeys.VARIABLES	#持久化模型时获得所有变量
Tf.GraphKeys.TRAINABLE_VARIABLES	#模型训练时获得所有可训练变量
Tf.GraphKeys.SUMMARIES	#日志生成的相关变量
Tf.GraphKeys.QUEUE_RUNNERS	#输入处理 QueueRunner
Tf.GraphKeys.MOVING_AVERAGE_VARIABLES	#计算变量的滑动平均值

8.7.3 Variable 和变量管理

Tensorflow 中的 **Variable** 是一种特殊的张量，所有变量会自动加入集合 **GraphKeys.VARIABLES** 中，变量的数据类型有：

- tf.int32 -- int/signed [int]/long [int]/signed long [int] ：32 位有符号整型
- tf.int64 -- ：64 位有符号整型，C/C++中无对应，仅有 long long 64 位无符号长整型
- tf.int16 -- short [int]/signed short [int]：16 位有符号整型
- tf.int8 -- char/signed char：8 位有符号整型
- tf.uint8 – unsigned char：8 位无符号整型
- tf.float32 -- float：32 位浮点型
- tf.float64 -- double：64 位浮点型
- tf.bool -- bool：单字节布尔型

Tensorflow 在神经网络的训练中，对数据集输入、连接权重、前向传播结果进行不同的处理：

- **数据输入 x 使用占位符：x = tf.placeholder(dtype, shape, name)，可以使得样本数据按 batch 输入，避免生成大量的常量占用资源；**
  sess.run(y, feed_dict={x: …})组织样本集数据输入；
- **连接权重使用 Variable：w = tf.Variable(tf.random_normal(shape, stddev))，在会话 tf.Session()中需要初始化定义的变量**
  sess.run([tf.global_varibales_initializer(), tf.local_variables_initializer()])
- **前向传播结果 y = network_output()，需要在会话中运行才能得到结果：**
  sess.run(y, feed_dict={x: …})

运行 sess.run(train_step) 后，会按照已设定的优化算法对集合 **tf.GraphKeys.TRAINABLE_VARIABLES** 中的变量进行优化，常用的优化算法有：

- 梯度下降法：tf.train.GradientDescentOptimizer()；
- 带动量的梯度下降法：tf.MomentumOptimizer()；
- 自适应动量项梯度下降法：tf.AdamOptimizer()；

以上优化算法数学解释见局部最小与全局最小；

在 ***Tensorflow*** 中可以使用两种方式创建变量：

tf.Variable("v", shape=[1], initializer=tf.constant_initializer(1.0))

tf.get_variable(tf.constant(1.0, shape=[1]), name="v")

tf.constant_initializer()与 tf.constant()是等效的，对于 tf.Variable()参数 name="v"是可选的，但是 tf.get_variable()中则是必填项，函数会根据变量名 name="v"创建变量；

在神经网络训练过程中，使用命名空间隔离不同层，使得 tf.get_variable()函数可以直接使用变量名称获得变量：

with tf.variable_scope(“layer1”, reuse=**tf.AUTO_REUSE**):
    weights = tf.get_variable(“weights”, [input_node, output_node], initializer=tf.random_normal_initializer(stddev=0.1))
第一次调用 tf.get_variable()时，变量 layer/weights 尚未创建，则调用 initializer=tf.random_normal_initializer(stddev=0.1)创建一个维度为 [input_node, output_node]的矩阵，一旦创建成功后，initializer 便不再起作用，后续调用 tf.get_variable()得到的是经过更新后的变量(反向传播优化)；
也会使用 with tf.name_scope(“layer”)管理上下文，在使用 tf.get_variable()创建变量时，不会为其加上前缀，但都会为 tf.Variable()创建的变量加前缀；
在应用 Tensorflow 训练神经网络时，通常将指示训练轮数的变量 ***global_step*** 指定为不可训练变量：**global_step = tf.Variable(0, trainable=False)，global_step 变量在优化器中进行更新：**
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss_function, global_step=global_step)

8.7.4 多线程

Tensorflow 可使用队列机制实现多线程输入数据，多线程操作队列使用三个工具：
Queue 是 **tensorflow** 实现队列和缓存机制的数据结构；
QueueRunner 是对同一队列多线程操作的封装，无论通过显式调用 **qr.create_threads(sess, coord=coord)** 创建线程，还是隐式调用 **tf.train.start_queue_runners(sess, coord=coord)** 都会返回创建的线程组 **queue_op_threads**，将其加入 **Coordinator** 实现多线程协同：**coord.join(queue_op_threads)**；
两种队列：

- **FIFOQueue**：先入先出队列
- **RandomShuffleQueue**：随机打乱元素次序队列

创建队列 q = tf.FIFOQueue(capacity, “dtype”)，”dtype” like “int32” “float”…
队列操作 ***API***：

- q.enqueue([element]) #入队列
- x = q.dequeue() #出队列
- q.enqueue_many([element0, element1, …]) #多元素入队列

tf.QueueRunner()启动多个线程操作同一个队列，线程间通信使用 **tf.Coordinator()**：
python 多线程操作导入包 **threading** 实现：

```
def Loop(coord, worker_id):
    #检查线程停止标志
    while not coord.should_stop():
        if np.random.rand() < 0.1:
            print(“Stoping from id : %d\n” % worker_id)
            #通知其他线程退出
            coord.request_stop()
        else:
            print(“Working on id : %d\n” % worker_id)
```

```
        time.sleep(1)
#创建 tf.train.Coordinator()协同多线程
coord = tf.train.Coordinator()
#创建多个线程
threads = [threading.Thread(target=Loop, args=(coord, i, )) for i in range(5)]
for t in threads:
    t.start()
#等待所有线程退出
coord.join(threads)
```

- 显式创建 **tf.train.QueueRunner()**后可通过调用 **create_threads()**启动多线程，启动的线程数也是显式指定的：

```
queue = tf.FIFOQueue(capacity=50, dtype=[tf.float32, tf.int32])
enqueue_op = queue.enqueue_many([feature, label])
x, y = queue.dequeue()
qr = tf.train.QueueRunner(queue, [enqueue_op]*5)
with tf.Session() as sess:
    coord = tf.Coordinator()
    #创建并启动多个线程
    enqueue_threads = qr.create_threads(sess, coord, start=True)
    for step in range(100):
        if coord.should_stop():
            break
        sample, label = sess.run([x, y])
    coord.request_stop()
    coord.join(enqueue_threads)
```

- 创建 **QueueRunner**，将其加入默认集合 **tf.GraphKeys.QUEUE_RUNNERS**，并显式调用 **tf.train.start_queue_runners()**启动集合中的线程：

```
…
tf.train.add_queue_runner(qr)
…
enqueue_threads = tf.train.start_queue_runners(sess=sess, coord=coord)
…
coord.join(enqueue_threads)
```

8.7.5 数据预处理

统一输入图片数据格式为 ***TFRecord***，***TFRecord*** 是一种文件类型，文件内容使用 **tf.train.Example Protocol Buffer** 存储，**Example** 实例由数据结构 **Features** 组织，**Features** 是一个由属性名称 **string** 到属性实体 **Feature** 的映射，而属性 **Feature** 可以是字符串 **BytesList**，实数列表 **FloatList** 或者整数列表 **Int64List**：

```
message Example{
    Features features = 1;
};
```

```
message Features{
    map<string, Feature> feature = 1;
};
message Feature{
    one of kind{
        BytesList bytes_list = 1;
        FloatList float_list =2;
        Int64List int64_list =3;
    }
};
```

对于图片数据来说，**一张图片可以看做一个 Example，而图片的像素矩阵是 BytesList 属性实例，图片的分辨率作为 Int64List 属性实例，图片的类别是 Int64List 实例，还要创建一个从图片数据到三种属性的映射函数**，参见条目“图片数据格式”；

- 图片数据格式：一个 ***RGB*** 色彩模式的图像可以看做一个三维矩阵，图像存储时经过压缩编码处理，要还原图像矩阵，首先要对图像进行解码 **(tf.image.decode_jpeg(image_raw_data)/tf.image.decode_png(image_raw_data))**得到一个 **tensor**：

```
image_raw_data = tf.gfile.FastGFile(“path/picture.jpeg”, ‘rb’).read()
with tf.Session() as sess:
    image_data = tf.image.decode_jpeg(image_raw_data)
    …
    #image 矩阵数据格式转换
    image_data = tf.image.convert_image_dtype(image_data, dtype=tf.float32)
    #重新编码并写入文件
    encoded_image = tf.image.encode_jpeg(image_data)
    with tf.gfile.GFile(“path/picture_0.jpeg”, ‘wb’) as f:
        f.write(encoded_image.eval())
```

将训练数据集中的所有图片转换成 **tf.train.Example Protocol Buffer** 格式，并写入 ***TFRecord*** 文件，便于多线程读入组织 Batch 进行训练：

```
writer = tf.python_io.TFRecordWriter(tfrecord)
# Convert into Example Protocol Buffer .
example = tf.train.Example(features=tf.train.Feature(features={
        'length': _int64_feature(length),
        'width': _int64_feature(width),
        'channel': _int64_feature(channel),
        'image_raw': _bytes_feature(image_raw),
        'label': _int64_feature(label)
    }))

writer.write(example.SerializedToString(example))
writer.close()
```

多线程读取 TFRecord 文件见条目“多线程组织 Batch”；

对应的解析方法为：

```
features = tf.parse_single_example(
                example, features={
                    'length': tf.FixedLenFeature([], tf.int64),
                    'width': tf.FixedLenFeature([], tf.int64),
                    'channel': tf.FixedLenFeature([], tf.int64),
                    'image_raw': tf.FixedLenFeature([], tf.string),
                    'label': tf.FixedLenFeature([], tf.int64)})
image = tf.decode_raw(features['image_raw'], tf.float32)
label = tf.cast(features['label'], tf.int32)
....
```

若要解析多个 example，使用 tf.parse_example()；

- 图片预处理：图片预处理遵循以下流程：

(1) 读入图像文件： image_raw_data = tf.gfile.FastGFile("path/picture**.jpeg**", 'rb').**read()**；

(2) 解码读入的图像文件：

```
image_data = tf.image.decode_jpeg(image_raw_data)；
```

(3) 设置标注框：

```
boxes = tf.constant([[[box1_y0, box1_x0, box1_y1, box1_x1],
                      [box2_y0, box2_x0, box2_y1, box2_x1]]])；
```

(4) 调整图片至预定尺寸：

```
image = tf.image.resize_images(image, [height, width],
          method=tf.random.randint(4))
```

(5) 随机生成标注框用于标注样本，增大样本方差：

```
bbox_begin, bbox_size, draw_bbox =
     tf.image.sample_distorted_bounding_box(tf.shape(image),
     bounding_boxes=boxes, min_object_covered=0.1)
```

(6) 转换图片矩阵的数据格式：

```
image = tf.image.convert_image_dtype(image, dtype=tf.float32)
```

(7) 标注框绘制函数 tf.image.draw_bounding_boxes()要求输入图片格式为 tf.float32 的 4-D 矩阵[batch_index, height, width, channel]，因此要扩展 image 的第 0 维：

```
image = tf.expand_dims(image, 0)
```

(8) 绘制标注框：

```
distorted_image = tf.image.draw_bounding_boxes(image, draw_bbox)
```

(9) 其他图像处理函数的输入均要求参数为 3-D 矩阵，因此要重新调整图像尺寸，并沿着标注框切割：

```
distorted_image = tf.reshape(distorted_image, [height, width, channel])
distorted_iamge = tf.slice(distorted_image, bbox_begin, bbox_size)
```

(10) 翻转与调色：

```
distorted_image = tf.random_flip_left_right(distorted_image)
distorted_image = distort_color(distorted_image)
```

最后可以使用预处理后的 **distorted_image** 生成 **tf.train.Example** 并写入 ***TFRecord*** 文件中，按照**'training'/'validation'/'testing'**的 **category** 划分可以写入不同的 ***TFRecord*** 文件中；

- 多线程组织 **Batch**：

(1) 获得原始数据集列表：

files = tf.train.match_filenames_once('path/data.tfrecords-*')，使用正则表达式模式匹配所有 tfrecord 文件，得到一个文件列表；

(2) 创建文件列表队列：**file_queue = tf.train.string_input_producer(files, shuffle=True, num_epochs=REPEAT_ITERS)，依据获得的文件列表创建一个文件列表队列，系统会一直拥塞至显式调用 tf.train.start_queue_runners(sess=sess, coord=coord)，且定义了重复加载文件列表的轮数为 REPEAT_ITERS；**

(3) 从文件队列中读取数据：创建 **tf.TFRecordReader()**，读取文件，解析 **Example**；

```
reader = tf.TFRecordReader()
_, serialized_example = reader.read(filename_queue)
features = tf.parse_single_example(serialized_example, features={
        'image_raw': tf.FixedLenFeature([], tf.float32),
        'label': tf.FixedLenFeature([], tf.int32),…})
with tf.Session() as sess:
    sess.run([tf.global_variables_initializer(),
            tf.local_variables_initializer()])
    coord = tf.train.Coordinator()
    #显式调用，启动集合 tf.GraphKeys.QUEUE_RUNNERS 中所有的 QueueRunner，创建线程；
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    …
    coord.request_stop()
    coord.join(threads)
```

(4) 数据预处理：将(3)中得到的样本数据和标签转换成所需的数据类型：

```
image = tf.decode_raw(features['image_raw'], tf.float32)
label = tf.cast(features['label'], tf.int32)
```

依据条目“图像预处理”中流程对图片进行标注框、裁剪、随机翻转、调色等操作，得到调整后的图像 **distorted_image** 及其标签 **label** 将单个样例组织成 **Batch**；

(5) 整理成 **Batch** 输入神经网络：

```
example_batch, label_batch = tf.train.batch([image, label],
batch_size=batch_size, capacity=capacity, num_threads=NUM_THREADS)
```

调用函数 tf.train.batch()/tf.train.shuffle_batch()执行样例及其标签的入队操作，将样例组合成 batch，其中 capacity = 1000 + 3 * batch_size，是队列的最大长度：

实际工程应用时，通常要求样本被随机打乱：

```
example_batch, label_batch = tf.train.shuffle_batch([image, label],
    batch_size=batch_size, capacity=capacity,
    min_after_dequeue=LIMITED_EXAMPLE_IN_QUEUE,
    num_threads=NUM_THREADS)
```

其中参数 **min_after_dequeue** 限制了出队操作时队列中的最少元素数量；**Batch** 组织方法定义完成后，即可组织输出给网络训练使用，还可定义参数 **num_threads 指定执行入队操作的线程数**，此时多线程对同一个文件中的样例执行**入队操作（数据读取、数据预处理）**，使用 **tf.train.shuffle_batch_join()**可以实现对多个文件的入队，而文件队列是由 tf.train.string_input_producer(filenames)得到的：

```
curr_example_batch, curr_label_batch = sess.run([example_batch,
                                                 label_batch])
logits = inference(curr_example_batch)
loss = calc_loss(logits, label_batch)
train_step =
    tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)
with tf.Session() as sess:
    sess.run([tf.global_variables_initializer(),
              tf.local_variables_initializer()])
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    for step in range(TRAINING_STEPS):
        sess.run(train_step)
    coord.request_stop()
    coord.join(threads)
```

Tensorflow API 性能对比：

tf.shuffle_batch()：多个线程从同一个 **TFRecord** 文件中读取样例，并将读到的样例随机打乱入队列，然而当 **TFRecord** 文件中的样例相似度较高时，例如将相同 ***label*** 的样本写入同一个文件，极大程度上**会造成过拟合**；

tf.shuffle_batch_join()：多个线程从不同的 **TFRecord** 文件中读取样例，并随机打乱入队列组织 **Batch**，此时应当控制线程数量，若线程数远远多于文件数量，则效果等同于 **tf.shuffle_batch()**，而且多个线程读取多个文件会带来过多的硬盘寻址，**增大 IO 开销**；

8.7.6 图像识别通用框架

卷积神经网络 ***CNN*** 处理图像问题的通用框架：

- 依据给定路径，读入图片文件并转换成 **tf.train.Example Protocol Buffer** 格式写入 **TFRecord** 文件，依据图片的 **label**，可以将其写入不同的 ***TFRecord*** 文件：

```
#遍历 path/data 下所有子目录，将图片逐一转换成 TFRecord 文件
def create_image_lists(testing_percentage, validation_percentage):
    #创建原始图片文件列表
    …
def get_image_path(image_lists, image_dir, label_name, index, catagory):
    #获得图片路径
    …
def get_tfrecord_path(image_lists, label_name, index, category):
    #获得图片对应的 tfrecord 文件路径
```

```
    …
def convert_to_tfrecord(sess, image, n_classes, label_index, tfrecord):
    #将指定图片转换成 tfrecord 文件
    …
```

- 建立文件列表队列，得到单个样例：

```
filenames = tf.train.match_filenames_once("./path/pattern-*")
file_queue = tf.train.string_input_producer(filenames)
reader = tf.TFRecordReader()
serialized_example_key, serialized_example = reader.read(file_queue)
```

- 解析图片 **Example**，得到对应的 **Features**：

```
features = tf.parse_single_example(serialized_example, features={
'image_raw': tf.FixedLenFeature([], tf.string),
'length': tf.FixedLenFeature([], tf.int32),
'width': tf.FixedLenFeature([], tf.int32),
'channel': tf.FixedLenFeature([], tf.int32),
'label': tf.FixedLenFeature([], tf.int32)})
    image_raw, label = features['image_raw'], features['label']
    length, width = features['length'], features['width']
    channel = features['channel']
```

- 对得到的图片 **Features** 进行预处理(标注框、翻转、调色、裁剪等)：

```
image = tf.decode_raw(image_raw)
image.set_shape([height, width, channels])
image_size = IMAGE_SIZE
boxes = tf.constant([[[box1_y0, box1_x0, box1_y1, box1_x1],
[box2_y0, box2_x0, box2_y1, box2_x1]]])
distorted_image = preprocess_for_train(image, image_size, image_size, boxes)
```

- 组织样本 ***Batch***，为神经网络训练提供输入，此时没有指定启动线程，只是隐式启动集合 ***(collection)*** **tf.GraphKeys.QUEUERUNNERS** 中所有的 **QueueRunner，集合中的 QueueRunner 由系统自动维护**：

```
capacity = MIN_AFTER_DEQUEUE + BATCH_SIZE * 3
image_batch, label_batch = tf.train.shuffle_batch.join([distorted_image, label],
batch_size=BATCH_SIZE, capacity=capacity,
min_after_queue=MIN_AFTER_QUEUE)
```

- 定义前向传播：

```
logits = inference(image_batch)
loss = calculate_loss(logits=logits, labels=label_batch)
train_step = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(loss)
```

- 启动会话，完成训练：

```
with tf.Session() as sess:
    sess.run([tf.global_variables_initializer(),
```

```
            tf.local_variables.initializer()])
        coord = tf.train.Coordinator()
        threads = tf.train.start_queue_runners(sess=sess, coord=coord)
        for step in range(TRAINING_STEPS):
            sess.run(train_step)
        coord.should_stop()
        coord.join(threads)
```

使用 tf.train.batch()/tf.train.batch_join()/tf.train.shuffle_batch()/ tf.train.shuffle_batch_join()函数直接为网络训练提供输入，无需 feed_dict，可极大地节省内存开销；

8.7.7 模型持久化

模型持久化使用 **API tf.train.Saver()**实现，将计算图 Graph 的结构和参数取值保存到指定路径下的 ***checkpoint*** 文件中：

```
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.save(sess, “path/model.ckpt”)
…
```

从持久化模型中恢复，变量声明要与持久化模型中保持一致：

```
v1 = tf.Variable(tf.constant(1.0, shape=[1]), name=”v1”)
v2 = tf.Variable(tf.constant(2.0, shape=[1]), name=”v2”)
result = v1 + v2
saver = tf.train.Saver()
with tf.Session() as sess:
    saver.restore(sess, “path/model.ckpt”)
    sess.run(result)
```

若在新模型中声明变量别名，则可使用一个字典参数列表将模型变量名称与新定义变量联系起来：

```
v1 = tf.Variable(tf.constant(1.0, shape=[1]), name=”other_v1”)
v2 = tf.Variable(tf.constant(2.0, shape=[1]), name=”other_v2”)
saver = tf.train.Saver({“v1”: v1, “v2”: v2})
with tf.Session() as sess:
    saver.restore(sess, “path/model.ckpt”)
    sess.run(result)
```

迁移学习中仅需要从持久化模型中获得前向传播的参数取值即可，*Tensorflow* 提供了 graph_util.convert_variables_to_constants()将变量及其取值以常量形式保存，将整个计算图保存在单个文件中：

```
from tensorflow.python.framework import graph_util
v1 = tf.Variable(tf.constant(1.0, shape=[1]), name=”v1”)
v2 = tf.Variable(tf.constant(2.0, shape=[1]), name=”v2”)
result = v1 + v2
with tf.Session() as sess:
```

```
    sess.run(tf.global_variables_initializer())
    #获得当前计算图的 GraphDef 部分并保存
    graph_def = tf.get_default_graph().as_graph_def()
    #仅保存指定的节点
    output_graph_def =
    graph_util.convert_variables_to_constants(sess, graph_def, ['add'])
    #保存模型计算图
    with tf.gfile.GFile("path/combined_model.pb", "wb") as f:
        f.write(output_graph_def.SerializeToString())
    …
```

加载训练好的模型：

```
import tensorflow.python.platform import gfile
with tf.Session() as sess:
    model_filename = "path/combined_model.pb"
    with gfile.FastGFile(model_filename, 'rb') as f:
        graph_def = tf.GraphDef()
        graph_def.parseFromString(f.read())
    result = tf.import_graph_def(graph_def, return_elements=["add:0"])
    sess.run(result)
```

测试程序中，从持久化模型中读取参数的滑动平均值作为测试模型，通过参数重命名方式加载模型：

```
variable_averages = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY)
variables_to_restore = variable_averages.variables_to_restore()
saver = tf.train.Saver(variables_to_restore)
…
ckpt = tf.train.get_checkpoint_state(MODEL_SAVE_PATH)
saver.restore(sess, ckpt.model_checkpoint_path)
```

8.7.8 常用生成函数

```
tf.random_normal(shape, mean, stddev, dtype)          #正态分布
tf.random_normal_initializer(mean, stddev, dtype)

tf.truncated_normal(shape, mean, stddev, dtype)       #正态分布，偏离校正
tf.truncated_normal_initializer(mean, stddev, dtype)

tf.random_uniform(shape, min, max, dtype)             #平均分布
tf.random_uniform_initializer(min, max, dtype)

tf.random_gamma()                                     #Gamma 分布

tf.zeros(shape, dtype)                                #全 0 数组
tf.zeros_initializer(dtype)
```

tf.ones(shape, dtype) **#全 1 数组**
tf.ones_initializer(dtype)

tf.fill(shape, number) **#填充数组**
tf.constant([…]) **#指定数值数组**

tf.identify(input) **#返回一个与输入相同的 tensor**
np.identity(dim, dtype) **#生成对角 1 矩阵**
enumerate(list) **#返回元组(index, element)**

8.7.9 常用数据处理函数

tf.clip_by_value(value, min, max) **#限制数值范围**
tensor.eval() **#启动运算，以 array 形式得到结果**
v1 = tf.constant([[1., 2.], [3., 4.]])
v2 = tf.constant([[5., 6.], [7., 8.]])
(v1*v2).eval() **#元素直接相乘**
tf.matmul(v1, v2) **#矩阵乘法**

tf.where(tf.greater(v1, v2), (v1-v2)*a, (v1-v2)*b) **# tf.greater()逐项比较选择较大的元素，tf.where()根据 tf.greater()判断结果选择后面的元素，若为 True 选择第二个参数，否则选择第三个参数**

tf.cast(tensor, dtype) **#将输入 tensor 转换为指定数据类型**
tf.equal(tensor1, tensor2) **#比较并返回 bool 型 tensor**
tf.concat([tensor1, tensor2], axis) **#拼接张量 axis=0 扩展行，axis=1 扩展列**
tf.reduce_sum(input_tensor, axis)

```
x = tf.constant([[1, 1, 1], [1, 1, 1]])
tf.reduce_sum(x)  # 6
tf.reduce_sum(x, 0)  # [2, 2, 2]
tf.reduce_sum(x, 1)  # [3, 3]
tf.reduce_sum(x, 1, keep_dims=True)  # [[3], [3]]
tf.reduce_sum(x, [0, 1])  # 6
```

tf.reduce_mean(input_tensor, axis)
tf.stack(values, axis=0) **#将张量按照指定维度打包**

```
x = tf.constant([1, 4])
y = tf.constant([2, 5])
z = tf.constant([3, 6])
tf.stack([x, y, z])  # [[1, 4], [2, 5], [3, 6]] (Pack along first dim.)
tf.stack([x, y, z], axis=1)  # [[1, 2, 3], [4, 5, 6]]
```

8.7.10 常用随机数生成函数

```
np.random.randint(integer)          #生成 integer 以内的正整数
np.random.rand()                    #生成 0~1 之间的随机数
np.random.randn(1000, 4)            #生成 1000 个 4-D vector
```

8.7.11 常用文件处理函数

```
with open(file_full_path, 'w') as file:
    file.write(string_line)
with open(file_full_path, 'r') as file:
    string_line = file.read()

os.walk(DIR)                        #深度优先得到指定路径下所有子路径

with gfile.FastGFile(os.path.join(MODEL_DIR, MODEL_NAME), 'rb') as f:
    graph_def = tf.GraphDef()
    graph_def.ParseFromString(f.read())

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    #获得当前计算图的 GraphDef 部分并保存
    graph_def = tf.get_default_graph().as_graph_def()
    #仅保存指定的节点
    output_graph_def =
    graph_util.convert_variables_to_constants(sess, graph_def, ['add'])
    #保存模型计算图
    with tf.gfile.GFile("path/combined_model.pb", "wb") as f:
        f.write(output_graph_def.SerializeToString())
    …

filename = "path/output.tfrecords"
writer = tf.python_io.TFRecordWriter(filename)
image_raw = image.tostring()
example = tf.train.Example(features=tf.train.Features(feature={
    'pixels': _int64_feature(pixels),
    'labels': _int64_feature(np.argmax(labels)),
    'image_raw': _bytes_feature(image_raw)}))
writer.write(example.SerializeToString())
writer.close()
```

图片数据的解码见数据预处理；

```
image_raw_data = tf.gfile.FastGFile("path/picture.jpeg", 'rb').read()
```

```
with tf.Session() as sess:
    image_data = tf.image.decode_jpeg(image_raw_data)
    …
    #image 矩阵数据格式转换
    image_data = tf.image.convert_image_dtype(image_data, dtype=tf.float32)
    #重新编码并写入文件
    encoded_image = tf.image.encode_jpeg(image_data)
    with tf.gfile.GFile("path/picture_0.jpeg", 'wb') as f:
        f.write(encoded_image.eval())
```