

Table of Contents

Introduction	1.1
第一章 工具	1.2
第一节 Gitbook的安装与使用	1.2.1
第二节 编辑数学公式	1.2.2
库函数的使用与说明	1.2.3
库函数地址	1.2.4
代码自动测试Gtest	1.2.5
第二章 C++	1.3
基本语法	1.3.1
模板类与模板函数	1.3.2
COM	1.3.3
驱动	1.3.4
第五章 算法	1.4
第一节 插值	1.4.1
第二节 最长递增子序列	1.4.2
LU分解	1.4.3
SVD	1.4.4
霍夫曼编码	1.4.5
最小生成树Prim算法	1.4.6
最短路径Dijkstra算法	1.4.7
快速排序与二叉树	1.4.8
医学图像重建算法	1.4.9
第六章 最优化	1.5
Levenberg-Marquardt算法	1.5.1
Qusi-Netwon 拟牛顿法	1.5.2
信赖域方法	1.5.3
参数调节	1.5.4
ResiNN	1.5.5
linear search	1.5.6
最速下降法, 牛顿法, LBFGS	1.5.7
线性与非线性方程组解的稳定性分析	1.5.8

第七章 机器学习	1.6
机器学习面试问题集	1.6.1
机器学习学习任务	1.6.2
机器学习原理	1.6.3
SVM, KKT条件与核函数方法	1.6.4
SVM	1.6.4.1
KKT条件与对偶	1.6.4.2
核函数方法	1.6.4.3
决策树与集成学习	1.6.5
决策树	1.6.5.1
集成学习	1.6.5.2
贝叶斯方法	1.6.6
逻辑回归与最大熵模型	1.6.7
降维与无监督学习	1.6.8
分类问题与方法	1.6.9
回归问题与方法	1.6.10
K均值EM等聚类算法	1.6.11
正则化方法原理与实践	1.6.12
机器学习项目	1.6.13
多元回归分析	1.6.14
矩阵微分	1.6.15
正则化	1.6.16
训练神经网络的经验	1.6.17
Notes	1.6.18
特征工程	1.6.19
第八章 推荐系统	1.7
推荐系统概述	1.7.1
FM(Factorization Machines),FFM	1.7.2
Learning To Rank LTR	1.7.3
第九章 Kaggle	1.8
Titanic	1.8.1
第十章 问题	1.9
基本ML问题	1.9.1
常规算法问题	1.9.2
常见问题	1.9.3

STL

1.9.4

结束

1.10

序

日知录是为效法顾炎武的《日知录》而做，里面会记录自己在学习与工作中学到的东西，该书的目的是为了打造一本属于自己的百科全书，也是自己思想体系的体现。最终里面的每一章可能代表一个方向与学科，比如C++，C#最终会成为一章，经济，中国史会成为一章。如果经济学里面记录的内容不多，就编成一章，如果内容多了，则会编成多章，比如宏观经济学，微观经济学，计量经济学。这样，最终一门大的学科，比如，经济学，就编成了一卷。

第一章：工具

第一节 Gitbook的安装与使用

Gitbook是Github旗下的产品，它提供书籍的编写与管理的功能，一个核心特征就是，它管理书也像管理代码一样，可以对数据进行fork,建立新的branch,使得书也可以进行版本迭代。也可以与多人合作，来编辑，更新书籍，适合单人创作或者多人共同创作。怎么利用gitbook生成自己的书，可以参考如下链接 <https://www.jianshu.com/p/cf4989c20bd8>

<https://www.cnblogs.com/Lam7/p/6109872.html>

在安装好Node.js与gitbook之后，我们可以利用gitbook的GUI创建一本书，写入一些章节，然后保存，然后在cmd到该书summary.md所在的路径，然后再利用gitbook init进行初始化，然后再利用gitbook serve在浏览器上对书内容进行访问。当然，个gitbook serve这一步会出错，会出现fontsettings.js找不到的情况，我们需要修改

C:\Users\pili.gitbook\versions\3.2.3\lib\output\website\copyPluginAssets.js，设置confirm: false，再重启个gitbook serve就可以了。详见<https://github.com/GitbookIO/gitbook/issues/1309>然后就可以在<http://localhost:4000>访问书籍了

The screenshot shows a browser window with the URL localhost:4000/chapter1. A search bar at the top contains the query "fontsettings.js". Below the search bar, there is a message "No results". The main content area displays the first chapter of a book. The chapter title is "第一章" (Chapter 1). The content includes a link to a Jianshu article and a detailed description of the setup process for using gitbook.

Type to search

Find on page fontsettings.js No results

Introduction

第一章

第一节

第二节

第二章

第一节

第二节

结束

第一章

怎么利用gitbook生成自己的书，可以参考如下链接 <https://www.jianshu.com/p/cf4989c20bd8>

在安装好Node.js与gitbook之后，我们可以利用gitbook的GUI创建一本书，写入一些章节，然后保存，然后在cmd到该书summary.md所在的路径，然后再利用gitbook init进行初始化，然后再利用gitbook serve在浏览器上对书内容进行访问。当然，个gitbook serve这一步会出错，会出现fontsettings.js找不到的情况，我们需要修改C:\Users\pili.gitbook\versions\3.2.3\lib\output\website\copyPluginAssets.js，设置confirm: false，再重启个gitbook serve就可以了。详见<https://github.com/GitbookIO/gitbook/issues/1309>然后就可以在<http://localhost:4000>访问书籍了

Published with GitBook

如果想让章节有层次感的显示，则可以在summary.md中设置

The screenshot shows the GitBook Editor interface. The left sidebar lists files: assets, chapter1, chapter2, chapter3, chapter4, end, .gitignore, chapter1.md, README.md, and SUMMARY.md (which is selected). The main area displays the content of SUMMARY.md, which is a hierarchical list of files:

```
* [Introduction](README.md)
* [第一章 工具](chapter1/README.md)
  * [第一节 Gitbook的安装与使用](chapter1/section1.md)
  * [第二节](chapter1/section2.md)
* [第二章 代码](chapter2/README.md)
  * [第一节 C++](chapter2/section1.md)
  * [第二节 C#](chapter2/section2.md)
* [第三章 日志](chapter3/README.md)
  * [第一节 任务](chapter3/section1.md)
  * [第二节 问题](chapter3/section2.md)
  * [第三节 需要做的事情](chapter3/section3.md)
* [第四章 XXX](chapter4/README.md)
  * [第一节 比特币代码分析](chapter4/section1.md)
    * [第一段 VS2017下调试Bitcoincode](chapter4/section1/part1.md)
    * [第二节 密码学笔记](chapter4/section2.md)
    * [第三节 比特币下的区块链](chapter4/section3.md)
* [结束](end/README.md)
```

To the right, the 'Summary' page is shown with the same hierarchical structure, represented by nested bullet points.

Error: Error with command "svgexport"

在cmd中安装:npm install svgexport -g

生成pdf

gitbook pdf ./ ./ML2.pdf

换行

一行之后加两个以上的空格。

第二节 编辑数学公式

mathjax

可以参考https://598753468.gitbooks.io/tex/content/fei_xian_xing_fu_hao.html
数学公式的编辑类似于Latex.

需要安装mathjax

```
npm install mathjax  
安装特定版本的npm install mathjax@2.6.1
```

首先在书籍project的最顶成新建一个book.json,内容如下

```
{
  "gitbook": "3.2.3",
  "plugins": ["mathjax"],
  "links": {
    "sidebar": {
      "Contact us / Support": "https://www.gitbook.com/contact"
    }
  },
  "pluginsConfig": {
    "mathjax": {
      "forceSVG": true
    }
  }
}
```

然后再用gitbook install命令安装mathjax

安装之后gitbook就出现编译错误了，也不能编译生成pdf文件。不论mathjax是哪个版本，从2.5开始都出错。下面选择用katex

gitbook pdf ./ mybook.pdf (./ mybook.pdf 之间有空格)

<http://ldehai.com/blog/2016/11/30/write-with-gitbook/>

no such file fontsettings.js,在上一节有讲怎么处理。

安装Katex

<https://github.com/GitbookIO/plugin-katex>

1. 在你的书籍文件夹里创建book.json
2. 里面写入如下内容

```
{  
  "plugins": ["katex"]  
}
```

4. 运行安装 gitbook install就安装好了(安装很慢, 一个小时), 可以换个地方安装, 只要把 book.json换个地方, 再在这个folder安装gitbook install, 然后把node_modules 拷到你书籍所在的folder就可以。Katex支持的公式<https://khan.github.io/KaTeX/docs/supported.html> <https://utensil-site.github.io/available-in-katex/>

中文在cmd中乱码的问题：

1. 打开cmd, 输入chcp 65001chcp 65001
2. 右击cmd上方, 选择属性-->字体-->SimSun-ExtB就可以显示了。

语法高亮

```
Supported languages  
This is the list of all 120 languages currently supported by Prism, with their corresponding  
alias, to use in place of xxxx in the language-xxxx class:
```

```
Markup - markup  
CSS - css  
C-like - clike  
JavaScript - javascript  
ABAP - abap  
ActionScript - actionscript  
Ada - ada  
Apache Configuration - apacheconf  
APL - apl  
AppleScript - applescript  
AsciiDoc - asciidoc  
ASP.NET (C#) - aspnet  
AutoIt - autoit  
AutoHotkey - autohotkey  
Bash - bash  
BASIC - basic  
Batch - batch  
Bison - bison
```

```
Brainfuck - brainfuck
Bro - bro
C - c
C# - csharp
C++ - cpp
CoffeeScript - coffeescript
Crystal - crystal
CSS Extras - css-extras
D - d
Dart - dart
Diff - diff
Docker - docker
Eiffel - eiffel
Elixir - elixir
Erlang - erlang
F# - fsharp
Fortran - fortran
Gherkin - gherkin
Git - git
GLSL - glsl
Go - go
GraphQL - graphql
Groovy - groovy
Haml - haml
Handlebars - handlebars
Haskell - haskell
Haxe - haxe
HTTP - http
Icon - icon
Inform 7 - inform7
Ini - ini
J - j
Jade - jade
Java - java
Jolie - jolie
JSON - json
Julia - julia
Keyman - keyman
Kotlin - kotlin
LaTeX - latex
Less - less
LiveScript - livescript
LOLCODE - lolcode
Lua - lua
Makefile - makefile
Markdown - markdown
MATLAB - matlab
MEL - mel
Mizar - mizar
Monkey - monkey
NASM - nasm
nginx - nginx
```

```
Nim - nim
Nix - nix
NSIS - nsis
Objective-C - objectivec
OCaml - ocaml
Oz - oz
PARI/GP - parigp
Parser - parser
Pascal - pascal
Perl - perl
PHP - php
PHP Extras - php-extras
PowerShell - powershell
Processing - processing
Prolog - prolog
.properties - properties
Protocol Buffers - protobuf
Puppet - puppet
Pure - pure
Python - python
Q - q
Qore - qore
R - r
React JSX - jsx
Reason - reason
reST (reStructuredText) - rest
Rip - rip
Roboconf - roboconf
Ruby - ruby
Rust - rust
SAS - sas
Sass (Sass) - sass
Sass (Scss) - scss
Scala - scala
Scheme - scheme
Smalltalk - smalltalk
Smarty - smarty
SQL - sql
Stylus - stylus
Swift - swift
Tcl - tcl
Textile - textile
Twig - twig
TypeScript - typescript
Verilog - verilog
VHDL - vhdl
vim - vim
Wiki markup - wiki
Xojo (REALbasic) - xojo
YAML - yaml
```

$$\hat{f} \frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m} \sum_{i=1}^m (y^i - h_\theta(x^i)) x_j^i$$

http://www.aleacubase.com/cudalab/cudalab_usage-math_formatting_on_markdown.html

<https://www.zybuluo.com/codeep/note/163962#3%E5%A6%82%E4%BD%95%E8%BE%93%E5%85%A5%E6%8B%AC%E5%8F%B7%E5%92%8C%E5%88%86%E9%9A%94%E7%AC%A6>

库函数的使用与说明

开源库的使用说明：

如何选择开源许可证？



Github Fork:



机器学习算法工程师速查表大全 <https://github.com/kailashahirwar/cheatsheets-ai>

库函数地址

- [] C:\Data\Group\ShareFolder 是windows, linux的共享文件夹, 有关Linux开发的文件都在这个文件夹, 比如DeepLearning C:\Data\Group\ShareFolder\ToolsLibrary 存放的是Eigen, OpenCV库
- [] <https://github.com/tensorflow/models> 里面有很多models, 可以用来实际使用的。

Eigen

[API document](#) 整理一篇文章, 阐述Eigen有哪些功能, 能用来看做啥。OPT++

数据集

<https://archive.ics.uci.edu/ml/datasets.html>

Google Test

[Google C++ 单元测试框架](#) 核心是添加include与lib路径, 以及把运行时库设置成MTd 对工程名
右键->属性->配置属性->C/C++->代码生成->运行时库:与前面gtest配置一样, 选择MTd; 代码位
于:C:\Data\ShareFolder\Works\Miura\googletest-master

第六章 C++

第一节 基本语法

接口：即抽象类，就是包含至少一个纯虚函数的类

```
一个类里面实现多种接口Iinterface, IinterfaceB, IinterfaceC
IE84TPTimeOutUIAck : public IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE ResponseTpTimeOutUIAck(
        /* [in] */ short nPortNo,
        /* [in] */ short nTpNo) = 0;

};
```

New ATL object 时，选择simple object，然后属性设置如下：则可以实现一个类中有多组接口函数。



C++通过ATL来实现接口的继承。

```
// Cr3halObj
class ATL_NO_VTABLE Cr3halObj :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<Cr3halObj, &CLSID_r3halObj>,
    public IConnectionPointContainerImpl<Cr3halObj>,
    public Ir3halObj,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IHALEvents>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IE84Events>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IMMIEEvents>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IE90Events>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IE116Events>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IFFUEvents>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IWaferProtrusionEvent>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_ISignalTowerDevice>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IE84HOAVBLEEvents>,
    public IEmuConnectionPointImpl<Cr3halObj, &IID_IE84TPTimeOutUIAck>, //TP3 connection point
public IR3HALDiag
{  
    DECLARE_EMUCLASSFACTORY_SINGLETON(Cr3halObj)
```

COM组件接口继承的实现 <https://blog.csdn.net/dingbaosheng/article/details/624504>

TL学习笔记03

继承

C++:

C#:

<http://www.cnblogs.com/flyinthesky/archive/2008/06/18/1224774.html>

不能初始化的类被叫做抽象类，它们只提供部分实现，但是另一个类可以继承它并且能创建它们的实例。“一个包含一个或多个纯虚函数的类叫抽象类，抽象类不能被实例化，进一步一个抽象类只能通过接口和作为其它类的基类使用。

“抽象类能够被用于类，方法，属性，索引器和事件，使用abstract在一个类声明中表示该类倾向要作为其它类的基类

成员被标示成abstract，或被包含进一个抽象类，必须被其派生类实现。

一个抽象类可以包含抽象和非抽象方法，当一个类继承于抽象类，那么这个派生类必须实现所有的的基类抽象方法。

一个抽象方法是一个没有方法体的方法。

1. virtual修饰的方法必须有实现(哪怕是仅仅添加一对大括号)，而abstract修饰的方法一定不能实现。
2. virtual可以被子类重写，而abstract必须被子类重写。
3. 如果类成员被abstract修饰，则该类前必须添加abstract，因为只有抽象类才可以有抽象方法。
4. 无法创建abstract类的实例，只能被继承无法实例化。
5. C#中如果要在子类中重写方法，必须在父类方法前加virtual，在子类方法前添加override，这样就避免了程序员在子类中不小心重写了父类方法。
6. abstract方法必须重写，virtual方法必须有实现(即便它是在abstract类中定义的方法)。

mRet = TestHalfWafer(token, ref nSeq, true); //C# ref

模板类与模板函数

Why can templates only be implemented in the header file?

Clarification: header files are not the _only _portable solution. But they are the most convenient portable solution Error LNK2019 unresolved external symbol "public: int __thiscall Algo<int>::LongestIncreaseSubsequence(int * const,int)" (?LongestIncreaseSubsequence@? \$Algo@H@@QAEHQAH@Z) referenced in function _main Algo C:\Data\ShareFolder\Group\Tim\Code\Algo\Algo\Main.obj

当模板声明在头文件，实现在cpp中，一般会出现上面的问题，原因是
Algo<int> Algo实例化的时候，编译器会去创建一个int型的Algo的新类，以及
LongestIncreaseSubsequence(T arr[], int arrSize)方法，因此编译器会去找这个方法的实现，如果没有找到，自然会报错。因此有如下两个方法

1. 在头文件中实现
2. 在头文件中声明，再在cpp中显示实例化(explicit instantiations) template class Algo<int>;并实现模板函数

方式1: 实现在头文件里

```
#pragma once
#include <algorithm>
using namespace std;

template<class T>
class Algo
{
public:
    Algo() {};
    ~Algo() {};
public:
    int LongestIncreaseSubsequence(T arr[], int arrSize)
    {
        int *maxLen = new int[arrSize]();
        for (int i = 0; i < arrSize; i++)
        {
            maxLen[i] = 1;
        }
        for (int j = 1; j < arrSize; j++)
        {
            int maxLenJ = 1;
            for (int i = 0; i < j; i++)
            {
                if (arr[i] < arr[j])
                {

```

```

        maxLenJ = maxLen[i] + 1;
        maxLen[j] = std::max({ maxLen[j] ,maxLenJ });
    }
}
return maxLen[arrSize - 1];
}
};

```

```

//调用函数
#include "stdafx.h"
#include "Algo.h"
#include <iostream>
using namespace std;

int main()
{
    int param[6] = {5,3,4,8,6,7};
    int arrSize = sizeof(param) / sizeof(int);
    Algo<int> Alg;
    int maxLength = Alg.LongestIncreaseSubsequence(param, arrSize);
    cout << "Max length: " << maxLength << endl;
    return 0;
}

```

方式二:在头文件中声明, CPP中定义并对每种实例进行声明

```

Algo.h
#pragma once
#include <algorithm>
using namespace std;

template<class T>
class Algo
{
public:
    Algo() {};
    ~Algo() {};
public:
    int LongestIncreaseSubsequence(T arr[], int arrSize);
}

```

```

// Algo.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include "Algo.h"
#include <algorithm>
using namespace std;

```

```

template class Algo<int>; //必须先声明, 才可以用
template class Algo<double>;
template class Algo<char>;
template<class T>
int Algo<T>::LongestIncreaseSubsequence(T arr[], int arrSize)
{
    int *maxLen = new int[arrSize]();
    for (int i = 0; i < arrSize; i++)
    {
        maxLen[i] = 1;
    }
    for (int j = 1; j < arrSize; j++)
    {
        int maxLenJ = 1;
        for (int i = 0; i < j; i++)
        {
            if (arr[i] < arr[j])
            {
                maxLenJ = maxLen[i] + 1;
                maxLen[j] = std::max({ maxLen[j] ,maxLenJ });
            }
        }
    }
    return maxLen[arrSize - 1];
}

```

If all you want to know is how to fix this situation, read the next two FAQs.

But in order to understand why things are the way they are, first accept these facts:

A template is not a class or a function. A template is a “pattern” that the compiler uses to generate a family of classes or functions.

In order for the compiler to generate the code, it must see both the template definition (not just declaration) and the specific types/whatever used to “fill in” the template. For example, if you’re trying to use a `Foo<int>`, the compiler must see both the `Foo` template and the fact that you’re trying to make a specific `Foo<int>`.

Your compiler probably doesn’t remember the details of one .cpp file while it is compiling an other .cpp file. It could, but most do not and if you are reading this FAQ, it almost definitely does not. BTW this is called the “separate compilation model.”

Now based on those facts, here’s an example that shows why things are the way they are. Suppose you have a template `Foo` defined like this:

```

template<typename T>
class Foo {
public:
    Foo();
    void someMethod(T x);
private:
    T x;
};

Along with similar definitions for the member functions:

```

```

template<typename T>
Foo<T>::Foo()
{
    // ...
}

template<typename T>
void Foo<T>::someMethod(T x)
{
    // ...
}

Now suppose you have some code in file Bar.cpp that uses Foo<int>:

// Bar.cpp
void blah_blah_blah()
{
    // ...
    Foo<int> f;
    f.someMethod(5);
    // ...
}

```

Clearly somebody somewhere is going to have to use the “pattern” for the constructor definition and for the someMethod() definition and instantiate those when T is actually int.

But if you had put the definition of the constructor and someMethod() into file Foo.cpp, the compiler would see the template code when it compiled Foo.cpp and it would see Foo<int> when it compiled Bar.cpp,

but there would never be a time when it saw both the template code and Foo<int>. So by rule #2 above, it co

uld never generate the code for Foo<int>::someMethod().

A note to the experts: I have obviously made several simplifications above. This was intentional so please don't complain too loudly. If you know the difference between a .cpp file and a compilation unit, the difference between a class template and a template class, and the fact that templates really aren't just glorified macros, then don't complain: this particular question/answer wasn't aimed at you to begin with. I simplified things so newbies would “get it,” even if doing so offends some experts.

const对象只能访问**const**成员函数。因为**const**对象表示其不可改变，而非**const**成员函数可能在内部改变了对象，所以不能调用。

而非**const**对象既能访问**const**成员函数，也能访问非**const**成员函数，因为非**const**对象表示其可以改变。

```

#ifndef _MATRIX_H
#define _MATRIX_H
#include <iostream>
#include <algorithm>
using namespace std;

```

```

template<class T>
class CMatrix
{
public:
    CMatrix() {};
    CMatrix(int rows, int columns);
    CMatrix(const CMatrix&);
    ~CMatrix() {};
public:
    void Set(int row, int column, T val);
    T Get(int row, int column) const;
    CMatrix operator +(const CMatrix &mat2);
    CMatrix operator -(const CMatrix &mat2);
    CMatrix operator *(const CMatrix &mat2);
    CMatrix operator *(const T div);
    CMatrix operator /(const T div);
    CMatrix I(int n);
    int Rows() const{ return mRows; }
    int Columns() const{ return mColumns; }
    int Length() { return mRows*mColumns; }
    friend ostream &operator<<(ostream &os, const CMatrix &mat)
    {
        int row = mat.Rows();
        int col = mat.Columns();
        //mat是const, 因此只能访问const成员函数

        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < col; j++)
            {
                os << fixed << mat.Get(i, j) << '\t';
            }
            if (i < row)
            {
                os << "\n";
            }
        }
        os << "\n";
        return os;
    };

private:
    int mRows;
    int mColumns;
    T* startElement;

};

#endif

```


COM连接点

COM连接点 - 最简单的例子

```
AtAdvise(spCar, sinkptr, __uuidof(_IMyCarEvents), &cookies);
sinkptr指向了一个CComObject<CSink>
CComObject<CSink>* sinkptr = nullptr;
CComObject<CSink>::CreateInstance(&sinkptr);
```

换句话说，一旦将sink对象成功挂载到了COM对象，那么sink对象的生命周期就由对应的COM(spCar)对象来管理。

一旦挂载成功，sink对象就托管给对应的COM对象了，如果对应的COM对象析构了，那么所有它管理的sink对象也就释放了。

CLR是什么？

COM Interop(互操作)

引言

- 平台调用
- C++ Interop(互操作)
- COM Interop(互操作)

一、引言

这个系列是在C#基础知识中遗留下来的一个系列的，因为在C# 4.0中的一个新特性就是对COM互操作改进，然而COM互操作性却是.NET平台下其中一种互操作技术，为了帮助大家更好的了解.NET平台下的互操作技术，所以才有了这个系列。然而有些朋友们可能会有这样的疑问——“为什么我们需要掌握互操作技术的呢？”对于这个问题的解释就是——掌握了.NET平台下的互操作性技术可以帮助我们在.NET中调用非托管的dll和COM组件。.NET是建立在操作系统的之上的一个开发框架，其中.NET类库中的类也是对Windows API的抽象封装，然而.NET类库不可能对所有Windows API进行封装，当.NET中没有实现某个功能的类，然而该功能在Windows API被实现了，此时我们完全没必要去自己在.NET中自定义个类，这时候就可以调用Windows API中的函数来实现，此时就涉及到托管代码与非托管代码的交互，此时就需要使用到互操作性的技术来实现托管代码和非托管代码更好的交互。.NET平台下提供了3种互操作性的技术：

1. Platform Invoke(P/Invoke)，即平台调用，主要用于调用C库函数和Windows API
2. C++ Interop，主要用于Managed C++(托管C++)中调用C++类库

3. COM Interop, 主要用于在.NET中调用COM组件和在COM中使用.NET程序集。

下面就对这3种技术分别介绍下。

二、平台调用

使用平台调用的技术可以在托管代码中调用动态链接库(DLL)中实现的非托管函数, 如Win32 DLL和C/C++ 创建的dll。看到这里, 有些朋友们应该会有疑问——在怎样的场合我们可以使用平台调用技术来调用动态链接库中的非托管函数呢?

这个问题就如前面引言中说讲到的一样, 当在开发过程中, .NET类库中没有提供相关API然而Win32 API 中提供了相关的函数实现时, 此时就可以考虑使用平台调用的技术在.NET开发的应用程序中调用Win32 API中的函数;

然而还有一个使用场景就是——由于托管代码的效率不如非托管代码, 为了提高效率, 此时也可以考虑托管代码中调用C库函数。

2.1 在托管代码中通过平台调用来调用非托管代码的步骤

- (1). 获得非托管函数的信息, 即dll的名称, 需要调用的非托管函数名等信息
- (2). 在托管代码中对非托管函数进行声明, 并且附加平台调用所需要属性
- (3). 在托管代码中直接调用第二步中声明的托管函数

2.2 平台调用的调用过程

(1) 查找包含该函数的DLL, 当需要调用某个函数时, 当然第一步就需要知道包含该函数的DLL的位置, 所以平台调用的第一步也就是查找DLL, 其实在托管代码中调用非托管代码的调用过程可以想象成叫某个人做事情, 首先我们要找到那个人在哪里(即查找函数的**DLL**过程), 找到那个人之后需要把要做的事情告诉他(相当于加载**DLL**到内存中和传入参数), 最后让他去完成需要完成的事情(相当于让非托管函数去执行任务)。

(2) 将找到的DLL加载到内存中。

(3) 查找函数在内存中的地址并把其参数推入堆栈, 来封送所需的数据。CLR只会在第一次调用函数时, 才会去查找和加载DLL, 并查找函数在内存中的地址。当函数被调用过一次之后, CLR会将函数的地址缓存起来, CLR这种机制可以提高平台调用的效率。在应用程序域被卸载之前, 找到的DLL都一直存在于内存中。

(4) 执行非托管函数。

平台调用的过程可以通过下图更好地理解:



三、C++ Interop

第二部分主要向大家介绍了第一种互操作性技术，然后我们也可以使用C++ Interop技术来实现与非托管代码进行交互。然而C++ Interop 方式有一个与平台调用不一样的地方，就是C++ Interop 允许托管代码和非托管代码存在于一个程序集中，甚至同一个文件中。C++ Interop 是在源代码上直接链接和编译非托管代码来实现与非托管代码进行互操作的，而平台调用是加载编译后生成的非托管DLL并查找函数的入口地址来实现与非托管函数进行互操作的。C++ Interop 使用托管C++来包装非托管C++代码，然后编译生成程序集，然后再托管代码中引用该程序集，从而来实现与非托管代码的互操作。关于具体的使用和与平台调用的比较，这里就不多介绍，我将会在后面的专题中具体介绍。

COM(Component Object Model. 组件对象模型)是微软之前推荐的一种开发技术，由于微软过去十多年里开发了大量的COM组件，

然而不可能再使用.NET技术重写这些COM组件实现的功能，所以为了解决在.NET中的托管代码能够调用COM组件中间问题，.NET平台下提供了COM Interop, 即[COM互操作技术](#)，COM interop不仅支持在托管代码中使用COM组件，而且支持向COM组件中使用.NET程序集。

1. 在.NET中使用COM组件

在.NET中使用COM对象，主要有三种方法：

1. 使用TlbImpl工具为COM组件创建一个互操作程序集来绑定早期的COM对象，这样就可以在程序中添加互操作程序集来调用COM对象
2. 通过反射来后期绑定COM对象
3. 通过P/Invoke创建COM对象或使用C++ Interop为COM对象编写包装类

但是我们经常使用的都是方法一，下面介绍下使用方法一在.NET 中使用COM对象的步骤：

1. 找到要使用的COM 组件并注册它。使用 regsvr32.exe 注册或注销 COM DLL。
2. 在项目中添加对 COM 组件或类型库的引用。

添加引用时, **Visual Studio** 会用到 **Tlbimp.exe**(类型库导入程序), **Tlbimp.exe** 程序将生成一个 **.NET Framework** 互操作程序集。该程序集又称为运行时可调用包装 (RCW), 其中包含了包装 COM 组件中的类和接口。**Visual Studio** 将生成组件的引用添加至项目。

1. 创建 RCW 中类的实例, 这样就可以使用托管对象一样来使用 COM 对象。

下面通过一个图更好地说明在.NET 中使用 COM 组件的过程:



在 COM 中使用 .NET 程序集

.NET 公共语言运行时通过 COM 可调用包装 (COM Callable Wrapper, 即 CCW) 来完成与 COM 类型库的交互。CCW 可以使 COM 客户端认为是在与普通的 COM 类型交互, 同时使 .NET 组件认为它正在与托管应用程序交互。在这里 CCW 是非托管 COM 客户端与托管对象之间的一个代理。CCW 既可以维护托管对象的生命周期, 也负责数据类型在 COM 和 .NET 之间的相互转换。实现在 COM 使用 .NET 类型的基本步骤如:

1. 在 C# 项目中添加互操作特性

可以修改 C# 项目属性使程序集对 COM 可见。右键解决方案选择属性, 在“应用程序标签”中选择“程序集信息”按钮, 在弹出的对话框中选择“使程序集 COM 可见”选项, 如下图所示:



1. 生成 COM 类型库并对它进行注册以供 COM 客户端使用

在“生成”标签中，选中“为COM互操作注册”选项，如下图：



勾选“为COM互操作注册”选项后，Visual Studio会调用类型库导出工具(Tlbexp.exe)为.NET程序集生成COM类型库再使用程序集注册工具(Regasm.exe)来完成对.NET程序集和生成的COM类型库进行注册，这样COM客户端可以使用CCW服务来对.NET对象进行调用了。

总结

介绍到这里，本专题的内容就结束，本专题主要对.NET提供的互操作的技术做了一个总的概括，在后面的专题中将会有对具体的技术进行详细的介绍和给出一些简单的使用例子。

驱动

```
WINBASEAPI
BOOL
WINAPI
DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);

```

是WinAPI, 负责与硬件打交道, 收发数据, 然后在我们的驱动程序DispatchControl函数中, 去解析DeviceIoControl传递的数据

```
// Control.cpp -- IOCTL handlers for usb42 driver
// Copyright (C) 1999 by Walter Oney
// All rights reserved

#include "stdcls.h"
#include "driver.h"
#include "ioctls.h"

///////////////////////////////
#pragma PAGEDCODE

NTSTATUS DispatchControl(PDEVICE_OBJECT fdo, PIRP Irp)
{
    // DispatchControl
    PAGED_CODE();
    PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fdo->DeviceExtension;

    NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
    if (!NT_SUCCESS(status))
        return CompleteRequest(Irp, status, 0);
    ULONG info = 0;

    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
    ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;
    ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;
    ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;

    switch (code)
    {
        // process request

```

```

case IOCTL_USB42_READ:
{
    // IOCTL_USB42_READ
    if (cbout != 64)
    {
        status = STATUS_INVALID_PARAMETER;
        break;
    }

    URB urb;
    UsbBuildInterruptOrBulkTransferRequest(&urb, sizeof(_URB_BULK_OR_INTERRUPT_TRANSFER),
                                          pdx->hpipe, Irp->AssociatedIrp.SystemBuffer, NULL, cbout, USBD_TRANSFER_DIRECTION_IN | USBD_SHORT_TRANSFER_OK, NULL);
    status = SendAwaitUrb(fdo, &urb);
    if (!NT_SUCCESS(status))
        KdPrint(("USB42 - Error %X (USBD status code %X) trying to read endpoint\n", status, urb.UrbHeader.Status));
    else
        info = cbout;
    break;
} // IOCTL_USB42_READ

default:
    status = STATUS_INVALID_DEVICE_REQUEST;
    break;

} // process request

IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
return CompleteRequest(Irp, status, info);
} // DispatchControl

```

第五章：算法

第一节 Kriging插值

<https://xg1990.com/blog/archives/222>

空间插值问题，就是在已知空间上若干离散点 (x_i, y_i) 的某一属性(如气温, 海拔)的观测值

$z_i = z(x_i, y_i)$ 的条件下，估计空间上任意一点 (x, y) 的属性值的问题。

直观来讲，根据地理学第一定律，

大意就是，地理属性有空间相关性，相近的事物会更相似。由此人们发明了反距离插值，对于空间上任意一点 (x, y) 的属性 $z = z(x, y)$ ，

定义反距离插值公式估计量 $\hat{z} = \sum_{i=0}^n \frac{1}{d_i^\alpha} z_i$

其中 α 通常取 1 或者 2。

即，用空间上所有已知点的数据加权求和来估计未知点的值，权重取决于距离的倒数(或者倒数的平方)。

那么，距离近的点，权重就大；距离远的点，权重就小。反距离插值可以有效的基于地理学第一定律估计属性值空间分布，但仍然存在很多问题：

α 的值不确定 用倒数函数来描述空间关联程度不够准确

因此更加准确的克里金插值方法被提出来了

克里金插值公式 $\hat{z}_o = \sum_{i=0}^n \lambda_i z_i$

其中 \hat{z}_o 是点 (x_o, y_o) 处的估计值，即 $\hat{z}_o = z(x_o, y_o)$

假设条件：

1. 无偏约束条件 $E(\hat{z}_o - z_o) = 0$
2. 优化目标/代价函数 $J = Var((\hat{z}_o - z_o))$ 取极小值
3. 半方差函数 r_{ij} 与空间距离 d_{ij} 存在关联，并且这个关联可以通过这两组数据拟合出来，因此可以使用距离 d_{ij} 来求得 r_{ij}

半方差函数 $r_{ij} = \sigma^2 - C_{ij}$ ；等价于 $r_{ij} = \frac{1}{2} E[(z_i - z_j)^2]$

其中： $C_{ij} = Cov(z_i, z_j) = Cov(R_i, R_j)$

求得 r_{ij} 之后，我们就可以求得 λ_i

$$\begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} & 1 \\ r_{21} & r_{22} & \cdots & r_{2n} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ r_{n1} & r_{n2} & \cdots & r_{nn} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \cdots \\ \lambda_n \\ 0 \end{bmatrix} = \begin{bmatrix} r_{1o} \\ r_{2o} \\ \cdots \\ r_{no} \\ 1 \end{bmatrix}$$

最长递增子序列

对于序列5, 3, 4, 8, 6, 7, 其最长的递增子序列是3, 4, 6, 7, 这里, 不需要保证元素是连在一起的, 只需要序号上升即可。问题变成求解n元系列 $a[1], a[2], a[3], \dots, a[n]$,

对于 $i < j < n$, 假设我们已经求得前 i 个元素的最长递增子序列长度为 $\text{MaxLen}[i]$ 满足 $i < j$, 那么我们可以遍历 i 从1到 $j-1$ 来求得 $\text{MaxLen}[j]$ 。基本的一些小算法写在一个CPP里面, 最后封装成一个类, 一般要设计成模板类。

```
int Algo::LongestIncreaseSubsequence(int arr[], int arrSize)
{
    int *maxLen = new int[arrSize]();
    for (int i = 0; i < arrSize; i++)
    {
        maxLen[i] = 1;
    }
    for (int j = 1; j < arrSize; j++)
    {
        int maxLenJ = 1;
        for (int i=0; i<j; i++)
        {
            if (arr[i] < arr[j])
            {
                maxLenJ = maxLen[i] + 1;
                maxLen[j] = std::max({ maxLen[j] ,maxLenJ });
            }
        }
    }
    return maxLen[arrSize-1];
}
```

LU分解

任何非奇异方阵都可以分解成上三角阵与下三角阵之积

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} * \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

对比两边矩阵的，可以求得：

但是编程时，行列都是从0开始时，要注意转换。

第1行： $a_{1j} = u_{1j}, j = 1, 2, \dots, n.$ => $u_{1j} = a_{1j}.$

第1列： $a_{j1} = l_{j1}u_{11}, j = 1, 2, \dots, n.$ => $l_{j1} = a_{j1}/u_{11}.$

...

第k行： $a_{kj} = \sum_{i=1}^{i=k} l_{ki}u_{ij}, => u_{kj} = a_{kj} - \sum_{i=1}^{i=k-1} l_{ki}u_{ij}.$

第k列： $a_{jk} = \sum_{i=1}^{i=k} l_{ji}u_{ik}, => u_{jk} = [a_{jk} - \sum_{i=1}^{i=k-1} l_{ji}u_{ik}]/u_{kk}.$

因为前k-1行的 u_{ij} 都已知，前k-1列的 l_{ij} 都已知，因此可以求得第k行 u_{ij} ，第k列的 $l_{ij}.$

问题：得保证 a_{11} 非0，以及矩阵非奇异。

利用LU分解求线性方程组的解

求解线性方程组 $Ax=b$ 相当于求解 $LUX=b;$

设 $Y = UX$; 因此 $LY = b$; 首先求解 $LY = b,$

$$\begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{bmatrix}$$

求解上面的方程：

第1行： $y_1 = b_1.$

对于第k行： $b_k = \sum_{i=1}^{i=k} l_{ki}y_i => y_k = b_k - \sum_{i=1}^{i=k-1} l_{ki}y_i.$

求得Y之后，代入Y=UX求得X：

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix}$$

对于上三角矩阵，我们从第n行开始求解

对第n行： $y_n = u_{nn}x_n => x_n = y_n/u_{nn}.$

...。

对第k行: $y_k = \sum_{i=k}^{i=n} u_{ki}x_i \Rightarrow x_k = [y_n - \sum_{i=k+1}^{i=n} u_{ki}x_i]/u_{kk}$
这样通过LU分解矩阵就求得了线性方程组的解X.

矩阵求逆

我们可以利用LU分解来求非奇异方阵的逆矩阵。

AB=I

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 1 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

可以分解成求:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} * \begin{bmatrix} b_{1k} \\ b_{2k} \\ \cdots \\ b_{nk} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 1 \\ \cdots \\ 0 \end{bmatrix}$$

右边的列, 就只是第k行值非0;

$A * b_k = e_k$, 对所有的 e_k 求出 b_k 就可以得到A的逆矩阵 $A^{-1} = B$

实际求解中把A换成LU来减少计算量。总的计算开销还是 n^3 。但是这样并不比直接的高斯消元法来的快。

算法实现如下。

```
//LU
template<class T>
void CMATRIX<T>::LU(CMATRIX &mat, int N, CMATRIX* L, CMATRIX* U)
{
    for (int k = 0; k < N; k++)
    {
        for (int j = k; j < N; j++)
        {
            T U_k_j = mat.Get(k, j);
            T L_j_k = mat.Get(j, k);
            for (int i = 0; i < k; i++)
            {
                U_k_j -= L->Get(k, i)*U->Get(i, j);
            }
            U->Set(k, j, U_k_j);

            for (int i = 0; i < k; i++)
            {
                L_j_k -= L->Get(j, i)*U->Get(i, k);
            }
        }
    }
}
```

```

        }
        L_j_k = L_j_k / U->Get(k, k);
        L->Set(j, k, L_j_k);

    }
}

//solve linear algebra equations
CVector Solve(CMatrix<double>& mat, CVector& vec)
{
    CVector X(vec.Size());
    CVector Y(vec.Size());
    if (mat.Columns() != mat.Rows() && mat.Rows() != vec.Size())
    {
        printf("Dimension not match!");
        return X;
    }
    CMatrix<double> L(vec.Size(), vec.Size());
    CMatrix<double> U(vec.Size(), vec.Size());
    mat.LU(mat, vec.Size(), &L, &U);
    Y = SolveLow(L, vec);
    cout << "Y\n" << Y;
    X = SolveUpper(U, Y);
    cout << "X\n" << X;
    return X;
}

CVector SolveLow(CMatrix<double>& mat, CVector& vec)
{
    CVector vecRes(vec.Size());
    if (mat.Columns() != mat.Rows() && mat.Rows() != vec.Size())
    {
        printf("Dimension not match!");
        return vecRes;
    }

    for (int k = 0; k < vec.Size(); k++)
    {
        double mRes = 0;
        mRes = vec.Get(k);
        for (int i = 0; i < k; i++)
        {
            mRes -= mat.Get(k, i)*vecRes.Get(i);
        }
        vecRes.Set(k, mRes);
    }
    return vecRes;
}

CVector SolveUpper(CMatrix<double>& mat, CVector& vec)
{

```

```

CVector vecRes(vec.Size());
if (mat.Columns() != mat.Rows() && mat.Rows() != vec.Size())
{
    printf("Dimension not match!");
    return vecRes;
}

for (int k = vec.Size() -1; k >=0 ; k--)
{
    double mRes = 0;
    mRes = vec.Get(k);
    for (int i = k+1; i <vec.Size(); i++)
    {
        mRes -= mat.Get(k, i)*vecRes.Get(i);
    }
    mRes = mRes / mat.Get(k, k);
    vecRes.Set(k, mRes);
}
return vecRes;
}

//inverse matrix
template<class T>
CMATRIX<T> CMATRIX<T>::Inv()
{
    CMATRIX result = *this;
    CMATRIX<double> L(mRows, mColumns);
    CMATRIX<double> U(mRows, mColumns);
    CMATRIX<double> InvMat(mRows, mColumns);
    LU(result,mRows, &L, &U);
    for (int col = 0; col < mColumns; col++)
    {
        CVector vec(mRows, col);
        CVector mSolution(mRows);
        mSolution = SolveLow(L, vec);
        mSolution = SolveUpper(U, mSolution);
        for (int row = 0; row<mRows; row++)
        {
            InvMat.Set(row, col, mSolution.Get(row));
        }
    }
    return InvMat;
}

```

SVD

利用LU分解，我们可以求满秩的线性方程组的解。对于 $m \times n$ ($m > n$) 阶矩阵，对于超定方程（方程数目大于未知数的个数），因为一般没有解满足 $Ax = b$ ，这就是最小二乘法发挥作用的地方。

这个问题的解就是使得： $\|Ax - b\|_2^2$ 值极小的解，通过求导（把 $x \rightarrow x + e$, 求梯度等于0的 x ），可以得到解为： $x = (X^T A)^{-1} A^T b$

QR分解

定理：设 A 是 $m \times n$ 阶矩阵， $m \geq n$ ，假设 A 为满秩的，则存在一个唯一的正交矩阵 Q ($Q^T Q = I$) 和唯一的具有正对角元 $r_{ij} > 0$ 的 $n \times n$ 阶上三角阵 R 使得 $A = QR$.

Gram-Schmidt 正交化

Gram-Schmidt 正交化的基本想法，是利用投影原理在已有基的基础上构造一个新的正交基。

$((\beta))_1$

<http://elsenaju.eu/Calculator/QR-decomposition.htm>

https://rosettacode.org/wiki/QR_decomposition

https://www.wikiwand.com/en/QR_decomposition#/Example_2

https://www.wikiwand.com/en/Householder_transformation

C# 有开源的免费的代数库 `mathnet.numerics`，功能也比较多，推荐通过 Nuget 来安装这个库

Nuget 安装dll

Tools-->Nuget Package Manager-->Manager Nuget packages for solution..-->Browse

<https://numerics.mathdotnet.com/api/MathNet.Numerics.LinearAlgebra/Matrix`1.htm#QR>

<https://numerics.mathdotnet.com/matrix.html>



研究这个



医学图像重建算法

平行光束算法

2.6.4 FBP (先滤波后反投影) 算法的推导

我们首先给出二维傅里叶变换在极坐标系下的表达式

$$f(x, y) = \int_0^{2\pi} \int_{-\infty}^{\infty} F_{polar}(\omega, \theta) e^{2\pi i \omega(x \cos \theta + y \sin \theta)} d\omega d\theta.$$

因为 $F_{polar}(\omega, \theta) = F_{polar}(-\omega, \theta + \pi)$, 所以

$$f(x, y) = \int_0^{\pi} \int_{-\infty}^{\infty} F_{polar}(\omega, \theta) |\omega| e^{2\pi i \omega(x \cos \theta + y \sin \theta)} d\omega d\theta.$$

根据中心切片定理, 我们可以用 P 来代替 F :

$$f(x, y) = \int_0^{\pi} \int_{-\infty}^{\infty} P(\omega, \theta) |\omega| e^{2\pi i \omega(x \cos \theta + y \sin \theta)} d\omega d\theta.$$

我们意识到 $|\omega|$ 是斜坡率滤波器的传递函数, 令 $Q(\omega, \theta) = |\omega|P(\omega, \theta)$, 则

$$f(x, y) = \int_0^{\pi} \int_{-\infty}^{\infty} Q(\omega, \theta) e^{2\pi i \omega(x \cos \theta + y \sin \theta)} d\omega d\theta.$$

利用一维傅里叶反变换并记 Q 的反变换为 q , 我们最后得到

$$f(x, y) = \int_0^{\pi} q(x \cos \theta + y \sin \theta, \theta) d\theta,$$

或

$$f(x, y) = \int_0^{\pi} q(s, \theta) |s=x \cos \theta + y \sin \theta| d\theta.$$

这正是 $q(s, \theta)$ 的反投影 (见第1.5节)。

中心切片定理

Special example prove



1. 平行于y轴投影

$$p(x,0) = \int_{-\infty}^{+\infty} f(x,y) dy \quad (1)$$

2. 投影函数一维傅里叶变换

$$P(u) = \int_{-\infty}^{+\infty} p(x,0) e^{-j2\pi ux} dx = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x,y) e^{-j2\pi ux} dx dy \quad (2)$$

3. 密度函数二维傅里叶变换

$$F(u,v) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x,y) e^{-j2\pi(ux+vy)} dx dy \quad (3)$$

4. 在 $v=0$ 时, 傅里叶变换表示

$$F(u,v)|_{v=0} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x,y) e^{-j2\pi ux} dx dy \quad (4)$$

5. 式 (2) 与 (4) 相等, 证明完毕

3. 对投影函数中的变量t进行傅里叶变换

$$P(\omega, \theta) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f'(t,s) ds e^{-j2\pi\omega t} dt \quad (3)$$

4. 对 (3) 式右面进行坐标变换x, y

$$ds dt = J dx dy = \begin{vmatrix} \partial t / \partial x & \partial s / \partial x \\ \partial t / \partial y & \partial s / \partial y \end{vmatrix} dx dy = dx dy$$

雅克比转换

$$P(\omega, \theta) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x,y) e^{-j2\pi\omega(x\cos\theta+y\sin\theta)} dx dy \quad (4)$$

5. $f(x,y)$ 二维傅里叶变换

$$F(u,v) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x,y) e^{-j2\pi(ux+vy)} dx dy \quad (5)$$

6. 比较公式 (4) 和 (5), 令 $u = \omega\cos\theta, v = \omega\sin\theta$

$$F(\omega\cos\theta, \omega\sin\theta) = P(\omega, \theta) \quad (6)$$

这表明,

一个物体 0° 投影的傅里叶变换与该物体二维傅里叶变换中 $v=0$ 的直线相同。



傅里叶切片定理:

物体 $f(x,y)$ 平行投影的傅里叶变换, 是物体二维傅里叶变换的一个切片 (直线), 切片是在与投影相同的角度获得的。

平行光束算法到扇形光束算法

其本质就是从直角坐标系变换到极坐标系

在完成了从平行光束数据 $p(s, \theta)$ 到扇形束数据 $g(\gamma, \beta)$ 的替换，旧变量 s 和 θ 到新变量 γ 和 β 的替换，及加入一个雅可比因子 $J(\gamma, \beta)$ ，一个崭新的扇形束图像重建算法就诞生了(图 3.5)!



图 3.5 从平行光束图像重建算法到扇形束图像重建算法的推导过程。

投影

1.5.1 投影

设 $f(x, y)$ 为 x - y 平面上定义的密度函数，其投影函数(即射线和，线积分，及拉东变换) $p(s, \theta)$ 有下面不同的等价表达式：

$$p(s, \theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \delta(x \cos \theta + y \sin \theta - s) dx dy ,$$

$$p(s, \theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \delta(\bar{x} \cdot \bar{\theta} - s) dx dy ,$$

$$p(s, \theta) = \int_{-\infty}^{\infty} f(s \cos \theta - t \sin \theta, s \sin \theta + t \cos \theta) dt ,$$

$$p(s, \theta) = \int_{-\infty}^{\infty} f(s \bar{\theta} + t \bar{\theta}^\perp) dt ,$$

$$p(s, \theta) = \int_{-\infty}^{\infty} f_\theta(s, t) dt ,$$

其中 $\bar{x} = (x, y)$ ， $\bar{\theta} = (\cos \theta, \sin \theta)$ ， $\bar{\theta}^\perp = (-\sin \theta, \cos \theta)$ ， δ 是狄拉克 δ 函数，图像 f 旋转角度 θ 后记为 f_θ 。我们假设探测器按逆时针方向绕物体旋转。这等价于探测器不动，而物体按顺时针做旋转。有关的坐标系如图1.12所示。

狄拉克函数

$$\int_{-\infty}^{\infty} \delta(ax) f(x) dx = \frac{1}{|a|} f(0),$$

$$\int_{-\infty}^{\infty} \delta^{(n)}(x) f(x) dx = (-1)^n f^{(n)}(0) \text{ [第 } n \text{ 阶导数]},$$

$$\delta(g(x))f(x) = \sum_n \frac{1}{|g'(\lambda_n)|} \delta(x - \lambda_n), \text{ 其中 } \lambda_n \text{ 为 } g(x) \text{ 的零点。}$$

狄拉克 δ 函数在二维和三维的情形的定义分别是， $\delta(\bar{x}) = \delta(x)\delta(y)$ 和 $\delta(\bar{x}) = \delta(x)\delta(y)\delta(z)$ 。这时，在上面的最后一个性质中， $|g'|$ 要分别被

$$|grad(g)| = \sqrt{\left(\frac{\partial g}{\partial x}\right)^2 + \left(\frac{\partial g}{\partial y}\right)^2} \text{ 和 } |grad(g)| = \sqrt{\left(\frac{\partial g}{\partial x}\right)^2 + \left(\frac{\partial g}{\partial y}\right)^2 + \left(\frac{\partial g}{\partial z}\right)^2} \text{ 取代。}$$

在二维成像中，我们通常用 δ 函数 $\delta(\bar{x} - \bar{x}_0)$ 来表示位于 $\bar{x} = \bar{x}_0$ 的点源。函数 $f(\bar{x}) = \delta(\bar{x} - \bar{x}_0) = \delta(x - x_0)\delta(y - y_0)$ 的拉东变换就是

$$p(s, \theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\bar{x}) \delta(\bar{x} \cdot \bar{\theta} - s) d\bar{x},$$

$$p(s, \theta) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \delta(x - x_0) \delta(y - y_0) \delta(x \cos \theta + y \sin \theta - s) dx dy,$$

解析法应用的就是中心切片定理，迭代法一般应用共轭梯度法或者拟牛顿法。

My Awesome Book

This file serves as your book's preface, a great place to describe your book's content and ideas.

Levenberg-Marquardt算法

Levenberg-Marquardt

方法用于求解非线性最小二乘问题，结合了梯度下降法和高斯-牛顿法。

其主要改变是在Hessian阵中加入对角线项，加这一样可以保证Hessian阵是正定的。当然，这是一种L2正则化方式。

$$\begin{aligned} \mathbf{x}_{t+1} &= \mathbf{x}_t - (\mathbf{H} + \lambda \mathbf{I}_n)^{-1} \mathbf{g}, \\ \mathbf{x}_{t+1} &= \mathbf{x}_t - (\mathbf{J}^T \mathbf{J} + \lambda \mathbf{I}_n)^{-1} \mathbf{J}^T \mathbf{r} \end{aligned}$$

L-M算法的不足点。

当 λ 很大时， $\mathbf{J}^T \mathbf{J} + \lambda \mathbf{I}_n$ 根本没用，Marquardt为了让梯度小的方向移动的快一些，来防止小梯度方向的收敛，把中间的单位矩阵换成了 $\text{diag}(\mathbf{J}^T \mathbf{J})$ ，因此迭代变成：

$$\mathbf{x}_{t+1} = \mathbf{x}_t - (\mathbf{J}^T \mathbf{J} + \lambda \text{diag}(\mathbf{J}^T \mathbf{J}))^{-1} \mathbf{J}^T \mathbf{r}$$

阻尼项(damping parameter) λ 的选择，Marquardt 推荐一个初始值 λ_0 与因子 $v > 1$ ，开始时， $\lambda = \lambda_0$ ，然后计算 cost functions，第二次计算 $\lambda = \lambda_0/v$ ，如果两者 cost function 都比初始点高，然后我们就增大阻尼项，通过乘以 v ，直到我们发现当 $\lambda = \lambda_0 v^k$ 时，cost function 下降。

如果使用 $\lambda = \lambda_0/v$ 使得 cost function 下降，然后我们就把 λ_0/v 作为新的 λ

当 $\lambda = 0$ 时，就是高斯-牛顿法，当 λ 趋于无穷时，就是梯度下降法。如果使用 λ/v 没有使损失函数下降，使用 λ 导致损失函数下降，那么我们就继续使用 λ 做为阻尼项。

```
k := 0; x := x0; v := 2; A := J(x)^T J(x)g := J(x)^T f(x)
u := τ * max(aii)
found := (||g||∞ ≤ ε1)
while(not found) and (k < kmax)
    k := k + 1; solve -> (A + uI)hlm = -g
    if ||hlm|| ≤ ε(||x|| + ε2)
        found := true
    else :
        xnew := x + hlm
        ρ := (F(x) - F(x + hdl)) / (L(0) - L(hdl)))
        if ρ ≥ 0
            x := xnew; g := J(x)Tf(x)
            A := J(x)TJ(x)g := J(x)Tf(x)
            found := (||g||∞ ≤ ε1)
            u := u * max(1/3, 1 - (2ρ - 1)3); v := 2
        else
```

```
def LM_Solver(self):
    init_p = 0.5 * np.ones((2, 1))
```

```

init_p[0, 0] = 2.0
init_p[1, 0] = 1.0
epsilon_1 = 1e-15
epsilon_2 = 1e-15
epsilon_3 = 1e-15
alpha = 1
cost_history = np.zeros((self.training_steps, 3))
Jaco = self.Jacobian(init_p)
Hessian = np.dot(Jaco.transpose(), Jaco)
resi = self.residual(init_p)
m_g = np.dot(Jaco.transpose(), resi)
cost_old = np.dot(resi.transpose(), resi)
epoch = -1
train_stop = -1
miu = Hessian.max()
lamda_v = 2

matrix_I = np.identity(Hessian.shape[0])
while epoch < self.training_steps - 1 and train_stop < 0:
    epoch += 1
    print "Epoch: ", epoch
    go_next_epoch = -1

    while go_next_epoch < 0:
        sigma_p = np.dot(np.linalg.inv(Hessian + miu*matrix_I), m_g)
        square_g = np.linalg.norm(sigma_p)
        if square_g <= epsilon_2*np.linalg.norm(init_p):
            train_stop = 1
            break
        else:
            new_p = init_p + sigma_p
            resi_new = self.residual(new_p)
            cost_new = np.dot(resi_new.transpose(), resi_new)
            frac = np.dot(sigma_p.transpose(), miu* sigma_p + m_g)
            rou = (cost_old - cost_new) / frac
            if rou > 0:
                go_next_epoch = 1
                init_p = new_p
                Jaco = self.Jacobian(init_p)
                Hessian = np.dot(Jaco.transpose(), Jaco)
                resi = self.residual(init_p)
                m_g = np.dot(Jaco.transpose(), resi)
                if (np.abs(m_g)).max < epsilon_1:
                    train_stop = 1
                    break
                miu *= max(1.0 / 3.0, 1 - pow(2 * rou - 1, 3))
                lamda_v = 2
                print init_p.transpose(), miu, cost_new
                cost_history[epoch, 0] = cost_new
                cost_history[epoch, 1] = miu
                cost_history[epoch, 2] = np.linalg.norm(m_g)
                cost_old = cost_new
            go_next_epoch = 0
    epoch += 1

```

```
        else:  
            miu *= lamda_v  
            # lamda_miu *= 0  
            lamda_v *= 2  
    return cost_history, init_p
```

线性搜索与Armijo准则

符号约定：

■ $g_k : \nabla f(x_k)$, 即目标函数关于k次迭代值 x_k 的导数

■ $G_k : G(x_k) = \nabla^2 f(x_k)$, 即Hessian矩阵

■ d_k : 第k次迭代的步长因子, 在最速下降算法中, 有 $d_k = -g_k$

■ α_k : 第k次迭代的步长因子, 有 $x_{k+1} = x_k + \alpha_k d_k$

在精确线性搜索中, 步长因子 α_k 由下面的因子确定：

■ $\alpha_k = \operatorname{argmin}_\alpha f(x_k + \alpha d_k)$

而对于非精确线性搜索, 选取的 α_k 只要使得目标函数f得到可接受的下降量, 即:

■ $\Delta f(x_k) = f(x_k) - f(x_k + \alpha_k d_k)$

Armijo 准则用于非精确线性搜索中步长因子 α 的确定, 内容如下:

Armijo 准则:

已知当前位置 x_k 和优化方向 d_k , 参数 $\beta \in (0, 1), \delta \in (0, 0.5)$. 令步长因子

■ $\alpha_k = \beta^{m_k}$, 其中

m_k 为满足下列不等式的最小非负整数m:

■ $f(x_k + \beta^m d_k) \leq f(x_k) + \delta \beta^m g_k^T d_k$

由此确定下一个位置 $x_{k+1} = x_k + \alpha_k d_k$

对于梯度上升, 上面的方程变成:

■ $f(x_k - \beta^m d_k) \geq f(x_k) - \delta \beta^m g_k^T d_k$

由此确定下一个位置 $x_{k+1} = x_k - \alpha_k d_k$

Quasi-Newton methods

BFGS

我们知道, 在牛顿法中, 我们需要求解二阶导数矩阵--Hessian阵, 当变量很多时, 求解Hessian阵势比较费时间的, Quasi-Newton法主要是在构造Hessian阵上下功夫, 它是通过构造一个近似的Hessian阵, 或者Hessian阵的逆, 而不是解析求解或者利用差分法来求解这个Hessian阵。构造的Hessian阵通过迭代而改变。

比较出名的Quasi-Netwon方法有BFGS(以Charles George Broyden, Roger Fletcher, Donald Goldfarb and David Shanno命名)

在牛顿法中, k步搜寻步长与方向是 p_k , 满足下面方程

$$\blacksquare B_k p_k = -\nabla f(x_k)$$

$\blacksquare B_k$ 就是近似的Hessian阵。下面我们讨论 B_k 如何变化,

我们要求 B_k 的更新满足quasi-Netwon条件

$$\blacksquare B_{k+1}(x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k)$$

这个条件就是简单的求 $f(x)$ 的二阶导数。

令:

$$\blacksquare y_k = \nabla f(x_{k+1}) - \nabla f(x_k), \quad s_k = x_{k+1} - x_k, \text{ 因此} \quad \blacksquare B_{k+1} \text{满足 } B_{k+1}s_k = y_k$$

这就是割线方程(the secant equation), The curvature condition $s_k^T y_k > 0$ 需要满足。

k步的Hessian阵以如下方式更新,

$$\blacksquare B_{k+1} = B_k + U_k + V_k$$

为了保持 B_{k+1} 的正定性以及对称性。 B_{k+1} 可以取如下形式:

$$\blacksquare B_{k+1} = B_k + \alpha u u^T + \beta v v^T$$

选择 $u = y_k, v = B_k s_k$, 为了满足割线方程(the secant condition), 我们得到:

$$\blacksquare \alpha = \frac{1}{y_k^T s_k}$$

$$\blacksquare \beta = \frac{1}{s_k^T B_k s_k}$$

最终我们得到Hessian阵的更新方程:

$$\blacksquare B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k^T}{s_k^T B_k s_k}$$

利用 Sherman–Morrison formula,

$$\blacksquare (A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$$

其中A是可逆方阵, $1 + v^T A^{-1}u \neq 0$

可以方便的得到 B_{k+1} 的逆。

$$\blacksquare B_{k+1}^{-1} = (I - \frac{s_k y_k^T}{y_k^T s_k}) B_k^{-1} (1 - \frac{y_k s_k^T}{y_k^T s_k}) + \frac{s_k s_k^T}{y_k^T s_k}$$

$$\blacksquare B_{k+1}^{-1} = B_k^{-1} + \frac{(s_k^T y_k + y_k^T B_k^{-1} y_k)(s_k s_k^T)}{(s_k^T y_k)^2} - \frac{B_k^{-1} y_k s_k^T + s_k y_k^T B_k^{-1}}{s_k^T y_k}$$

DFP

参考 DFP的矫正公式如下：

$$H_{k+1} = H_k + \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{s_k^T y_k}$$

当采用精确线搜索时，矩阵序列 H_k 的正定新条件 $s_k^T y_k > 0$ 可以被满足。但对于 Armijo 搜索准则来说，不能满足这一条件，需要做如下修正：

$$H_{k+1} = H_k \quad s_k^T y_k \leq 0$$

$$H_{k+1} = H_k + \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} \quad s_k^T y_k > 0$$

Broyden族算法

之前BFGS和DFP校正都是由 y_k 和 $B_k s_k$ (或者 s_k 和 $H_k y_k$) 组成的秩2矩阵。而Droyden族算法采用了 BFGS 和 DFP 校正公式的凸组合，如下：

$$H_{k+1}^\phi = \phi_k H_{k+1}^{BFGS} + (1 - \phi_k) H_{k+1}^{DFP} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{s_k^T y_k} + \phi_k v_k v_k^T \text{ 其中}$$

$$\phi_k \in [0, 1], v_k \text{ 由下式定义: } v_k = \sqrt{\frac{s_k}{y_k^T H_k y_k}} \left(\frac{s_k}{y_k^T s_k} - \frac{H_k y_k}{y_k^T H_k y_k} \right)$$

Gauss-Netwon方法

对于函数 $f(x)$ 的小邻域展开

$$\blacksquare f(x+h) \simeq l(h) \equiv f(x) + J(x)h$$

$$\blacksquare F(x+h) \simeq L(h) \equiv \frac{1}{2}l(h)^T l(h) = \frac{1}{2}f^T f + h^T J^T f + \frac{1}{2}h^T J^T J h$$

我们需要选择步长 h_{gn} 来最小化 $L(h)$

$$\blacksquare h_{gn} = \operatorname{argmin}_h L(h)$$

$$\blacksquare (J^T J)h_{gn} = -J^T f$$

信赖域方法

Powell's Dog Leg Method是一种信赖域方法

在Gauss-Netwon迭代中

$$\blacksquare J(x)h \simeq -f(x)$$

最陡的方向由下面公式给出：

$$\blacksquare h_{sd} = -g = -J(x)^T f(x)$$

但是这只是给出了方向，而没有给出步长。

考虑线性模型

$$\blacksquare f(x + \alpha h_{sd}) \simeq f(x) + \alpha J(x)h_{sd}$$

$$\blacksquare F(x + \alpha h_{sd}) \simeq \frac{1}{2}||f(x) + \alpha J(x)h_{sd}||^2 = F(x) + \alpha h_{sd}^T J(x)^T f(x) + \frac{1}{2}\alpha^2 ||J(x)h_{sd}||^2$$

当 α 取如下值得时候，以上函数取最小值

$$\blacksquare \alpha = -\frac{h_{sd}^T J(x)^T f(x)}{||J(x)h_{sd}||^2} = \frac{||g||^2}{||J(x)h_{sd}||^2}$$

现在有两个步长的选择 $a = \alpha h_{sd}$ 以及 $b = h_{gn}$, Powell建议在信赖域半价是 Δ 的时候，步长可以如下选择

If $||h_{gn}|| \leq \Delta$, $h_{dl} = h_{gn}$

$$\blacksquare \text{Elseif: } ||\alpha h_{sd}|| \geq \Delta, h_{dl} = (\Delta / ||h_{sd}||)h_{sd}$$

else:

$$\blacksquare h_{dl} = \alpha h_{sd} + \beta(h_{gn} - \alpha h_{sd})$$

选择 β 使得 $||h_{dl}|| = \Delta$

在L-M算法中我们定义了增益因子，

$$\blacksquare \rho = (F(x) - F(x + h_{dl}) / (L(0) - L(h_{dl})))$$

其中L是线性模型

$$L(h) = \frac{1}{2} \|f(x) + J(x)h\|^2$$

在L-M我们通过 ρ 来控制阻尼因子, 在dog-leg算法中, 我们通过它来控制步长

Dog Leg Method

```
k := 0; x := x_0; Δ := Δ_0; g := J(x)^T f(x)
found := (||f(x)||_∞ ≤ ε_3) or (||g||_∞ ≤ ε_1)
while(not found) and (k < k_max)
    k := k + 1; computer α by (2.4)
    h_sd := -α g; solve J(x)h_gn ≈ -f(x)
    computer h_dl by (4.5)
    if ||h_dl|| ≤ ε(||x|| + ε_2)
        found := true
    else :
        x_new := x + h_dl
        ρ := (F(x) - F(x + h_dl)) / (L(0) - L(h_dl))
        if ρ ≥ 0
            x := x_new; g := J(x)^T f(x)
            found := (||f(x)||_∞ ≤ ε_3) or (||g||_∞ ≤ ε_1)
        if ρ ≥ 0.75
            Δ := max(Δ, 3 * ||h_dl||)
        elseif ρ < 0.25
            Δ := Δ/2; found := (Δ ≤ ε_2(||x|| + ε)
```

```
def Dogleg_Solver(self):
    init_para = np.ones((2, 1))
    epsilon_1 = 1e-15
    epsilon_2 = 1e-15
    epsilon_3 = 1e-15
    alpha = 1
    cost_history = np.zeros((self.training_steps, 3))
    Jaco = self.Jacobian(init_para) //get Jacobian
    Hessian = np.dot(Jaco.transpose(), Jaco)
    resi = self.residual(init_para)
    m_g = np.dot(Jaco.transpose(), resi)
    cost_old = np.dot(resi.transpose(), resi)

    epoch = -1
    train_stop = -1
    delta = 0.01
    while epoch < self.training_steps - 1 and train_stop < 0:
        epoch += 1
        print "Epoch: ", epoch
        go_next_epoch = -1

        while go_next_epoch < 0:
            square_g = np.linalg.norm(m_g)
            square_Jg = np.linalg.norm(np.dot(Jaco, m_g))
            alpha = pow(square_g, 2) / pow(square_Jg, 2)
```

```

    h_sd = -alpha * m_g
    h_gn = np.dot(np.linalg.inv(Hessian), -m_g)
    norm_para = np.linalg.norm(init_para)

    ##calculate h_dl
    if np.linalg.norm(h_gn) <= delta:
        h_dl = h_gn
        hdl_type = 0
    elif np.linalg.norm(h_sd) >= delta:
        h_dl = (delta / np.linalg.norm(h_sd)) * h_sd
        hdl_type = 1
    else:
        beta = self._get_h_dl(h_sd, h_gn, delta)
        h_dl = h_sd + beta * (h_gn - h_sd)
        hdl_type = 2

    # #epsilon_2*norm_para:
    if np.linalg.norm(h_dl) < epsilon_2 * (norm_para + epsilon_2):
        train_stop = 1
        break
    else:
        para_0 = init_para + h_dl
        #cost_old = pow(np.linalg.norm(resi), 2)
        resi_new = self.residual(para_0)
        cost_new = np.dot(resi_new.transpose(), resi_new)
        #pow(np.linalg.norm(resi_new), 2)
        #h_sub = h_sd - 0.5 * np.dot(Hessian, h_dl)
        #frac = np.dot(h_dl.transpose(), h_sub)
        if hdl_type == 0:
            frac = cost_new
        elif hdl_type == 1:
            frac = delta/(2*alpha)*(2*np.linalg.norm(alpha*m_g) - delta)
        else:
            frac = 0.5*alpha*pow(beta, 2)*pow(np.linalg.norm(alpha*m_g), 2)
            frac += beta*(2 - beta)*cost_new
        rou = (cost_old - cost_new) /frac
        if rou > 0:
            go_next_epoch = 1
            init_para = para_0
            Jaco = self.Jacobian(init_para)
            Hessian = np.dot(Jaco.transpose(), Jaco)
            resi = self.residual(init_para)
            m_g = np.dot(Jaco.transpose(), resi)

            cost_history[epoch, 0] = cost_new
            cost_history[epoch, 1] = delta
            cost_history[epoch, 2] = np.linalg.norm(m_g)
            print para_0.transpose(), cost_new, delta

            if (np.abs(m_g)).max() < epsilon_1 or (np.abs(resi_new)).max() < epsilon_1:
                train_stop = 1

```

```
        break
    if rou > 0.75:
        delta = max(delta, 3 * np.linalg.norm(h_dl))
    elif rou < 0.25:
        delta /= 2.0
    if delta < epsilon_2:
        break
    cost_old = cost_new
return cost_history, init_para
```

优化收敛位置

参考知乎上面

对于N个参数的系统(神经网络), 对应的Hessian阵是N阶的, 我们假定其本征值的正负概率都是0.5. 因此, 要保证是局部最小值, 则Hessian正定, 意味着所有的本征值都为正。其概率是 $\frac{1}{2^N}$ 。同样, 是局部最大值, 则Hessian阵负定, 概率也为 $\frac{1}{2^N}$ 。因此, 最有可能的是本征值有正有负, 也就是鞍点的情形。但是即使是普通的鞍点, 函数不会震荡, 而是马上逃离。

实际的情形是, 函数收敛到了一个大的平坦区域, 其表现就是一阶导数很小, 以至于在训练结束的时候还没有逃离出这个平坦区域。那么, 我们得从数学上讨论这种平坦区域(平坦马鞍面)具有什么样的性质。可以看下面一个方程:

$$f(x, y) = \frac{\alpha}{2}x^2 - \frac{\beta}{2}y^2$$

$$f_x = \alpha x; f_y = -\beta y.$$
 假设 $\alpha > 0, \beta > 0$

(0, 0)点是唯一的鞍点。在x方向, 该点是局部最小值, 但在y方向, 是局部极大值, 如果 β 很小, 很小是它与学习率r相乘结果与1来说的。假设学习率是r一开始 $y = \Delta_0$, 则一次迭代后

$$\Delta_1 = \Delta_0 + r * \beta * \Delta_0 = (1 + r\beta)\Delta_0$$

$$\Delta_2 = \Delta_1 + r * \beta * \Delta_1 = (1 + r\beta)\Delta_1$$

n次迭代后:

如果训练总的次数为N, $r\beta \leq \frac{1}{N}$, 那N次迭代之后

$(1 + r\beta)^N \Delta_0 \leq (1 + \frac{1}{N})^N \approx e\Delta_0 = 2.71828\Delta_0$, 这定量的给出了在训练结束的时候, 函数能多大程度的逃离(0, 0)点。

这个模型可以推广到N维情形。假设Hessian阵有N-K个本征值非负, K个小于零。

考虑通过平移后, 鞍点处于(0, 0...0)点附近的二阶展开:

$$L(X) = L(0) + \sum_{i=K+1}^{i=N} \frac{\alpha_i}{2} x_i^2 - \sum_{i=1}^{i=K} \frac{\alpha_i}{2} x_i^2$$

如果满足所有的 $\alpha_i \ll 1, i \in [1, K]$, 这时候, 函数会在很长时间内, 局域在鞍点附件。

可以考虑的时, 怎么通过设计Cost Function来避免产生这些超级马鞍面。

这些超级马鞍面的产生要考虑激活函数吗, 时候Relu比Sigmoid不容易产生这些马鞍面?

激活函数的作用

增加非线性, 增强学习能力

为啥使用收敛慢的(随机)梯度下降法

因为只需要计算一阶导数，而不需要计算二阶，因为几十万以上的参数，计算二阶导数太费时间，内存也是个问题(除非用L-BFGS)

SVD PCA

SVD 将矩阵分解成累加求和的形式，其中每一项的系数即是原矩阵的奇异值。这些奇异值，按之前的几何解释，实际上就是空间超椭球各短轴的长度。现在想象二维平面中一个非常扁的椭圆（离心率非常高），它的长轴远远长于短轴，以至于整个椭圆看起来和一条线段没有什么区别。这时候，如果将椭圆的短轴强行置为零，从直观上看，椭圆退化为线段的过程并不突兀。回到 SVD 分解当中，较大的奇异值反映了矩阵本身的主要特征和信息；较小的奇异值则如例中椭圆非常短的短轴，几乎没有体现矩阵的特征和携带的信息。因此，若我们将 SVD 分解中较小的奇异值强行置为零，则相当于丢弃了矩阵中不重要的一部分信息。

因此，SVD 分解至少有两方面作用：

- 分析了解原矩阵的主要特征和携带的信息(取若干最大的奇异值)，这引出了主成分分析(PCA)；
- 丢弃忽略原矩阵的次要特征和携带的次要信息(丢弃若干较小的奇异值)，这引出了信息有损压缩、矩阵低秩近似等话题。

项目背景

随着半导体工艺中，芯片中的尺寸越来越小，光学衍射效应越来越明显，单纯的几何光学(初中物理老师告诉你，光是直线传播的，这其实说的是，光是粒子，但光通过狭窄的缝隙会发生衍射效应，这时，光不是直线传播的了)以及不适用，要考虑光的衍射效应。光通过Musk就相当于一次傅里叶变化，从时域变到了频率域。再通过棱镜，又相当于一次傅里叶变换，从频率域变到时域。但棱镜的尺寸有限，因此只接收了低频波，因此，用图像处理的话来说，棱镜的作用就是一个低通滤波器。去掉了高频部分，因此投影到wafer(晶圆)上的图案不再菱角分明，而是在边界处会比较圆滑。

如果没有一些傅里叶光学的背景，你们听起来会比较费劲，但是如果你们有图像处理的背景的话，就记住把Musk上的图案刻蚀到晶圆上就是经历了两次傅里叶变换再加一个低通滤波器。

光通过单缝后，其频谱是连续的，但是经过周期性(结构)缝，其频谱就是离散的，如果为了节约棱镜的开支，就只能取很低的频谱。

光学建模是个逆问题(逆问题的例子，就是CT，通过收集到的信号来反推身体的内部结构)。(怎么把逆问题与深度学习，机器学习联系起来？)，再通俗点，它实际上和机器学习，深度学习要处理的问题是一样的，就是求解一个模型，就是train一个model。正向过程就是你已经训练好了一个模型(取啥例子？)，然后输入一张图，看它的分类是啥，这很容易，难得是，我有一堆分类好的图片，让你训练一个model。逆问题就是一个优化问题，因此就设计到一些优化算法，这时候考点就来了？有哪些优化算法，总结下，有两种，线性搜索方法(linear search)以及 trust region。但是对于这个求逆问题而言，是很复杂的，直接离散的话，参数基本在10的15次方以上，总之，这个求逆问题是很难的，即使求出来，也不会很准。因此，我们有两个研究的方向。直接通过光学图像(也就是将要刻蚀在wafer上的电路板)来反求印刷这张图像的模板(Musk)，其实这个光刻就像拍照片，拍的物体(比如风景)就是Musk，wafer就相当于胶卷，wafer上的图像就相当于风景在胶卷上的投缘。

第一个方向是，直接通过光学图像(也就是将要刻蚀在wafer上的电路板)来反求印刷这张图像的模板(Musk)，我们的模型就是卷积网络，因为多层网络可以用来近似任何一个函数(通用近似定理 (Universal approximation theorem, 一译万能逼近定理)」指的是：如果一个前馈神经网络具有线性输出层和至少一层隐藏层，只要给予网络足够数量的神经元，便可以实现以足够高精度来逼近任意一个在 R^n 的紧子集 (Compact subset) 上的连续函数。只要神经元足够多，再加一个激活函数)，当然可以用来近似这个求逆过程。但是这条路风险很大，因此，我们有了第二条研究的路径，就是修正原有的物理模型，使得它更powerful，也就是说，我们仍然需要建立物理model，来进行求逆过程，得到一个物理的model，只是，这个model不是足够强大，会有一些缺陷，因此，我们想通过一个CNN来修正这个model，来让我们的模型更有效。这就是大致的背景。背景讲清楚了，我们就该讲我们是怎么做的了。

搭建GPU环境，装显卡驱动（显卡驱动，蓝屏），cuda，cudnn，但是发现不能使用apt-get install东西，重装了很多次系统，装了不同的ubuntu系统，从12.04到16.04的五个不同版本，折腾了一周。（因此因为公司的IP保护，无线网卡解决问题）我们一开始做了些尝试，用了CNN，VGG，遇到的问题（图像预处理，亚像素的偏移），数据集不够，因为真实数据都是机密（IP），不可以轻易拿到。自己去切割图像，通过有经验的人工来做分类。再通过翻转这些操作来增加数据集。此外，数据集也是很讲究的，你得保证同一数据集中的物理参数，焦距，偏振方向都是一致的，焦距不同，得到的图像模糊程度不同。很长一段时间，我们都在用记忆卷积，反卷积的深度神经网络，都没发现很好的效果。后来使用了pre-trained model，因为我们的数据量少（30000张图），但是经过CNN处理过的图像，并不比没有处理的图像好，这就尴尬了，因此，我们一直加深网络。

Linear Search Methods

the steepest descent algorithm proceeds as follows: at each step, starting from the point $x^{(k)}$, we conduct a line search in the direction $-\nabla f(x^{(k)})$, until a minimizer, $x^{(k+1)}$, is found.

$$\alpha_k = \operatorname{argmin}_{\alpha \geq 0} f(x^{(k)} - \alpha \nabla f(x^{(k)}))$$

Proposition 8.1: if $x^{(k)}$ is the steepest descent sequence for $f: R^n \rightarrow R$, then for each k the vector $x^{(k+1)} - x^{(k)}$ is orthogonal to the vector $x^{(k+2)} - x^{(k+1)}$

Proposition 8.2: if $x^{(k)}$ is the steepest descent sequence for $f: R^n \rightarrow R$ and if $\nabla f(x^{(k)}) \neq 0$, then

$$f(x^{(k+1)}) < f(x^{(k)})$$

Stopping criterion:

$$\frac{|f(x^{(k+1)}) - f(x^{(k)})|}{\|f(x^{(k)})\|} < \epsilon$$

Example: Quadratic function of the form:

$$f(x) = \frac{1}{2}x^T Q x - b^T x$$

Gradient: $g^{(k)} = \nabla f(x^{(k)}) = Qx - b$

$$\text{so } x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

where:

$$\alpha_k = \operatorname{argmin}_{\alpha \geq 0} f(x^{(k)} - \alpha g^{(k)}) = \operatorname{argmin}_{\alpha \geq 0} \left(\frac{1}{2}(x^{(k)} - \alpha g^{(k)})^T Q (x^{(k)} - \alpha g^{(k)}) - (x^{(k)} - \alpha g^{(k)})^T b \right)$$

Hence:

$$\alpha_k = \frac{g^{(k)T} g^{(k)}}{g^{(k)T} Q g^{(k)}}$$

Covergence properties:

Define:

$$V(x) = f(x) + \frac{1}{2}x^{*T} Q x^* = \frac{1}{2}(x - x^*)^T Q (x - x^*)$$

With: $x^* = Q^{-1}b$

Lemma 8.1 The iterative algorithm

$$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$$

with $g^{(k)} = Qx^{(k)} - b$ satisfies

$$V(x^{(k+1)}) = (1 - \gamma_k)V(x^{(k)}),$$

where, if $g^{(k)} = 0$ then $\gamma_k = 1$, and if $g^{(k)} \neq 0$ then:

$$\blacksquare \gamma_k = \alpha_k \frac{g^{(k)T} Q g^{(k)}}{g^{(k)T} Q^{-1} g^{(k)}} (2 \frac{g^{(k)T} g^{(k)}}{g^{(k)T} Q g^{(k)}} - \alpha_k)$$

Submit α_k into γ_k , then

$$\blacksquare \gamma_k = \frac{(g^{(k)T} g^{(k)})^2}{(g^{(k)T} Q g^{(k)})(g^{(k)T} Q^{-1} g^{(k)})}$$

Theorem 8.1 Let $x^{(k)}$ be the sequence resulting from a gradient algorithm

$x^{(k+1)} = x^{(k)} - \alpha_k g^{(k)}$. Let γ_k be as defined in Lemma 8.1, and suppose that $\gamma_k > 0$ for all k , then $x^{(k)}$ converges to x^* for any initial condition $x^{(0)}$ if and only if:

$$\blacksquare \sum_{x=0}^{\infty} \gamma_k = \infty$$

Theorem 8.2 In the steepest descent algorithm, we have

$$\blacksquare x^{(k)} \rightarrow x^* \text{ for any } x^{(0)}$$

Theorem 8.3 For the fixed step size gradient algorithm, $x^{(k)} \rightarrow x^*$ for any $x^{(0)}$

if and only if $0 < \alpha < \frac{2}{\lambda_{\max}(Q)}$

Convergence Rate:

Theorem 8.4 In the method of steepest descent applied to the quadratic function, at every step k , we have :

$$\blacksquare V(x^{(k+1)}) \leq \left(\frac{\lambda_{\max}(Q) - \lambda_{\min}(Q)}{\lambda_{\max}(Q)}\right) V(x^{(k)}).$$

Let:

$$\blacksquare r = \frac{\lambda_{\max}(Q)}{\lambda_{\min}(Q)} = \|Q\| \|Q^{-1}\| \text{ the so-called condition number of } Q.$$

Then, it follows from Theorem 8.4 that $V(x^{(k+1)}) \leq (1 - \frac{1}{r})V(x^{(k)})$. We refer to $1 - \frac{1}{r}$ as the convergence ratio. If $r = 1$, then $\lambda_{\max}(Q) = \lambda_{\min}(Q)$, corresponding to circular contours of f . You can use the convergence ratio r to judge the speed of convergence.

Definition 8.1 Given a sequence $x^{(k)}$ that converges to x^* , that is, $\lim_{k \rightarrow \infty} \|x^{(k)} - x^*\| = 0$, we say that the order of convergence is o , where $p \in R$, if $0 < \lim_{k \rightarrow \infty} \frac{x^{(k+1)} - x^*}{\|x^{(k)} - x^*\|^p} < \infty$, if for all $p > 0$, $\lim_{k \rightarrow \infty} \frac{x^{(k+1)} - x^*}{\|x^{(k)} - x^*\|^p} = 0$,

then we say that the order of convergence is ∞ .

Lemma 8.3. In the steepest descent algorithm, if $g^{(k)} \neq 0$ for all k , then $\gamma_k = 1$ if and only if $g^{(k)}$ is an eigenvector of Q .

Theorem 8.6 Let $x^{(k)}$ be the sequence resulting from a gradient algorithm applied to a function f . Then, the order of convergence of $x^{(k)}$ is 1 in the worst case, that is, there exist a function f and an initial $x^{(0)}$ such that the order of convergence of $x^{(k)}$ is equal 1.

Newton's Method

$$\blacksquare f(x) = f(x^{(k)}) + (x - x^{(k)})^T g^{(k)} + \frac{1}{2}(x - x^{(k)})^T F(x^{(k)})(x - x^{(k)})$$

Theorem 9.1 Suppose that $f \in C^3$, and $x^* \in R^n$ is a point such that $\nabla f(x^*) = 0$ and $F(x^*)$ is invertible. Then, for all $x^{(0)}$ sufficiently close to x^* , Newton's method is well defined for all k , and converges to x^* with order of convergence at least 2.

As stated in the above theorem, Newton's methods has superlinear convergence properties if the starting point is near the solution. However, the method is not guaranteed to converge to the solution if we start away from it (in fact, it may not even be well defined because the Hessian may be singular). In particular, the method may not be a descent method; that is, it is possible that $f(x^{(k+1)}) > f(x^{(k)})$. Fortunately, it is possible to modify the algorithm such that descent property holds. To see this, we need the following result.

Theorem 9.2 Let $x^{(k)}$ be the sequence generated by Newton's method for minimizing a given objective function $f(x)$. If the Hessian $F(x^{(k)}) > 0$ and $g^{(k)} = \nabla f(x^{(k)}) \neq 0$, then the direction $d^{(k)} = -F(x^{(k)})^{-1}g^{(k)} = x^{(k+1)} - x^{(k)}$ from $x^{(k)}$ to $x^{(k+1)}$ is a descent direction for f in the sense that there exists an $\alpha > 0$ such that for all $a \in (0, \alpha)$,

$$f(x^{(k)} + ad^{(k)}) < f(x^{(k)}).$$

The above theorem motivates the following modification of Newton's method:

$$\blacksquare x^{(k+1)} = x^{(k)} - \alpha_k F(x^{(k)})^{-1} g^{(k)} \text{ where}$$

$$\blacksquare \alpha_k = \operatorname{argmin}_{\alpha > 0} f(x^{(k)} - \alpha_k F(x^{(k)})^{-1} g^{(k)})$$

A drawback of Newton's method is that evaluation of $F(x^{(k)})$ for large n can be computationally expensive. Furthermore, we have to solve the set of n linear equations $F(x^{(k)})d^{(k)} = -g^{(k)}$. In the Chapters 10 and 11, we discuss methods that alleviate this difficulty.

Levenberg-Marquardt Modification:

If the Hessian matrix $F(x^{(k)})$ is not positive definite, then the search direction

$d^{(k)} = -F(x^{(k)})^{-1}g^{(k)}$ may not point in a descent direction. A simple technique to ensure that the search direction is a descent direction is to introduce the so-called Levenberg-Marquardt Modification to Newton's algorithm:

$$x^{(k+1)} = x^{(k)} - \alpha_k F(x^{(k)} + u_k I)^{-1} g^{(k)}$$

where $u_k \geq 0$

Newton's methods for nonlinear least-squares

Consider the following problem:

$$\text{minimize } \sum_{i=1}^m (r_i(x))^2$$

where $r_i : R^n \rightarrow R, i = 1, \dots, m$, are given functions. This particular problem is called a nonlinear least-squares problem

$$\boxed{F_{jk} = 2 \sum_i \left(\frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} + r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right)},$$

$$\text{Let } s(x) = r_i(x) \frac{\partial^2 r_i}{\partial x_j \partial x_k}(x)$$

So the Hessian matrix as

$$\boxed{F(x) = 2(J(x)^T J(x) + S(x))}.$$

Therefore, Newton's method applied to the nonlinear least-squares problems is given by

$\boxed{x^{(k+1)} = x^{(k)} - (J(x)^T J(x) + S(x))^{-1} J(x)^T r(x)}$. In some case, $s(x)$ can be ignored because its components are negligibly small. In this case, the above Newton's algorithm reduces to what is commonly called the Gauss-Newton method:

$$x^{(k+1)} = x^{(k)} - (J(x)^T J(x))^{-1} J(x)^T r(x).$$

A potential problem with the Gauss-Newton method is that the matrix $J(x)^T J(x)$ may not be positive definite. As described before, this problem can be overcome using a Levenberg-Marquardt modification:

$$\boxed{x^{(k+1)} = x^{(k)} - (J(x)^T J(x) + u_k I)^{-1} J(x)^T r(x)}.$$

Conjugate Direction Methods

The conjugate direction methods typically perform better than the method of steepest descent, but not as well as Newton's method. As we saw from the method of steepest descent and Newton's method, the crucial factor in the efficiency of an iterative search method is the direction of the search at each iteration.

Definition 10.1 Let Q be a real symmetric $n \times n$ matrix. The directions $d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(m)}$ are Q -conjugate if, for all $i \neq j$, we have $d(i)^T Q d(j) = 0$.

Lemma 10.1 Let Q be a symmetric positive definite $n \times n$ matrix. If the directions

$d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(m)} \in R^n, k \leq n - 1$, are nonzero and Q -conjugate, then they are linearly independent.

Basic conjugate Direction Algorithm. Given a start $x^{(0)}$, and Q -conjugate direction

$$\blacksquare d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(m)}, \text{ for } k \geq 0,$$

$$\blacksquare g^{(k)} = \nabla f(x^{(k)}) = Qx^{(k)} - b,$$

$$\blacksquare \alpha_k = -\frac{g^{(k)T} d^{(k)}}{d^{(k)T} Q d^{(k)}}$$

$$\blacksquare x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$$

Theorem 10.1 For any starting point $x^{(0)}$, the basic conjugate direction algorithm converges to the unique x^* (that solves $Qx = b$) in n steps; that is, $x^{(n)} = x^*$

□

Lemma 10.2 In the conjugate direction algorithm,

$$\blacksquare g^{(k+1)T} d^{(i)} = 0 \text{ for all } k,$$

$0 \leq k \leq n - 1$, and $0 \leq i \leq k$

The conjugate gradient algorithm is summarized below.

1. Set $k := 0$; select the initial point $x^{(0)}$

2. $\blacksquare g^{(0)} = \nabla f(x^{(0)})$. If $g^{(0)} = 0$, stop, else set $d^{(0)} = -g^{(0)}$. 3.

$$\blacksquare \alpha_k = -\frac{g^{(k)T} d^{(k)}}{d^{(k)T} Q d^{(k)}}$$

3. $\blacksquare x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$

4. $\blacksquare g^{(k+1)} = \nabla f(x^{k+1})$. If $g^{(k+1)} = 0$, stop.

5. $\blacksquare \beta_k = -\frac{g^{(k+1)T} Q d^{(k)}}{d^{(k)T} Q d^{(k)}}$ 7. $\blacksquare d^{(k+1)} = -g^{(k+1)} + \beta_k d^{(k)}$

6. Set $k := k+1$; go to step 3.

Proposition 10.1 In the conjugate gradient algorithm, the directions $d^{(0)}, d^{(1)}, d^{(2)}, \dots, d^{(n-1)}$ are Q-conjugate.

最优化

最速下降法, 牛顿法, LBFGS

1. 梯度下降法(Gradient Descent)

梯度下降法是最早最简单, 也是最为常用的最优化方法。梯度下降法实现简单, 当目标函数是凸函数时, 梯度下降法的解是全局解。一般情况下, 其解不保证是全局最优解, 梯度下降法的速度也未必是最快的。

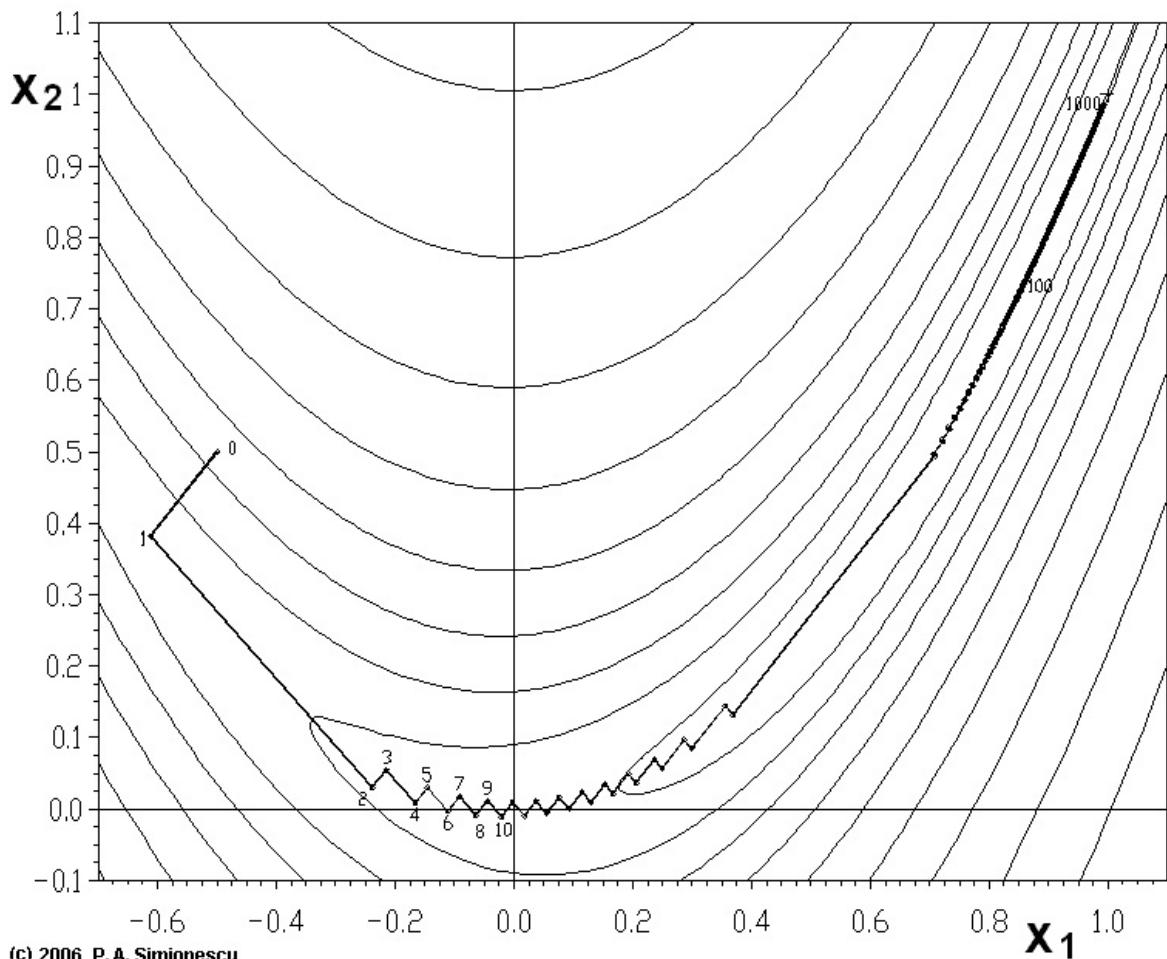
梯度下降法的优化思想是用当前位置负梯度方向作为搜索方向, 因为该方向为当前位置的最快下降方向, 所以也被称为是“最速下降法”。最速下降法越接近目标值, 梯度趋于0, 所以步长越小, 前进越慢。

梯度下降法的搜索迭代示意图如下图所示:



梯度下降法的缺点:

(1) 越靠近极小值的地方收敛速度越慢, 如下图所示



(c) 2006 P. A. Simionescu

从上图可以看出，梯度下降法在接近最优解的区域收敛速度明显变慢，利用梯度下降法求解需要很多次的迭代。

在机器学习中，基于基本的梯度下降法发展了两种梯度下降方法，分别为随机梯度下降法和批量梯度下降法。

比如对一个线性回归(Linear Logistics)模型，假设下面的 $h(x)$ 是要拟合的函数， $J(\theta)$ 为损失函数， θ 是参数，要迭代求解的值， θ 求解出来了那最终要拟合的函数 $h(\theta)$ 就出来了。其中 m 是训练集的样本个数， n 是特征的个数。

$$h(\theta) = \sum_{j=0}^n \theta_j x_j \quad J(\theta) = \frac{1}{2m} \sum_{i=0}^m (y^i - h_\theta(x^i))^2$$

//n是特征的个数，m的训练样本的数目

1) 批量梯度下降法 (Batch Gradient Descent, BDG)

(1) 将 $J(\theta)$ 对 θ 求偏导，得到每个 θ 对应的梯度：

$$\frac{\partial J(\theta)}{\partial \theta_j} = -\frac{1}{m} \sum_{i=1}^m (y^i - h_\theta(x^i)) x_j^i$$

(2) 由于要最小化风险函数，所以按照每个参数 θ 的负梯度方向来更新 θ

$$\theta'_j = \theta_j - \frac{\partial J(\theta)}{\partial \theta_j} = \theta_j + \frac{1}{m} \sum_{i=1}^m (y^i - h_\theta(x^i)) x_j^i$$

(3) 从上面公式可以注意到，它得到的是一个全局最优解，但是每迭代一步，都要用到训练集所有的数据，如果 m 很大，那么可想而知这种方法的迭代速度会相当的慢。所以，这就引入了另外一种方法——随机梯度下降。

一个实验结果，说明随机梯度下降法能收敛到最终结果：

对于批量梯度下降法，样本个数 m , x 为 n 维向量，一次迭代需要把 m 个样本全部带入计算，迭代一次计算量为 $m * n^2$ 。

2) 随机梯度下降 (Stochastic Gradient Descent, SGD)

(1) 上面的风险函数可以写成如下这种形式，损失函数对应的是训练集中每个样本的粒度，而上面批量梯度下降对应的是所有的训练样本

$$J(\theta) = \frac{1}{2m} \sum_{i=0}^m (y^i - h_\theta(x^i))^2 = \frac{1}{m} \sum_{i=1}^m cost(\theta, (x^i, y^i))$$

$$cost(\theta, (x^i, y^i)) = \frac{1}{2} (y^i - h_\theta(x^i))^2$$

(2) 每个样本的损失函数，对 θ 求偏导得到对应的梯度，来更新 θ

$$\theta'_j = \theta_j + (y^i - h_\theta(x^i)) x_j^i$$

(3) 随机梯度下降是通过每个样本来迭代更新一次，如果样本量很大的情况（例如几十万），那么可能只用其中几万条或者几千条的样本，就已经将 θ 迭代到最优解了，对比上面的批量梯度下降，迭代一次需要用到十几万训练样本，一次迭代不可能最优，如果迭代 10 次的话就需要遍历训练样本 10 次。但是，SGD 伴随的一个问题是噪音较 BGD 要多，使得 SGD 并不是每次迭代都向着整体最优化方向。

随机梯度下降每次迭代只使用一个样本，迭代一次计算量为 n^2 ，当样本个数 m 很大的时候，随机梯度下降迭代一次的速度要远高于批量梯度下降方法。两者的关系可以这样理解：随机梯度下降方法以损失很小的一部分精确度和增加一定数量的迭代次数为代价，换取了总体的优化效率的提升。增加的迭代次数远远小于样本的数量。

比较批量梯度下降法与随机梯度下降法：

用 $y = 2 * x1 + x2 - 1 + rand(0, 1)$ 产生1000组数, 用这一组数据来反求产生函数中的系数

$$(2, 1, -1)^T.$$

迭代停止条件就是, 训练得到的相邻两次参数的差的范数小于0.000001(严格来说, 停止条件是损失函数一介导数等于0, 也就是 $L'(\theta) = -\frac{1}{m}X^T(X\theta - Y) = 0$).

1)批量梯度下降法: 1000个样本, 设置的learn rate = 0.0001, 一次迭代所有1000个样本, 最终经过38000次迭代收敛到可以接受的标准。如果设置learn rate 是较大的值, 比如为1, 则发现结果马上发散了, 因此批量梯度下降法对learn rate比较敏感, 而下面的随机梯度下降法就不会出现这个问题, 因为每次只对一个样本进行处理, 即使这个样本对参数的梯度很大, 但是下一个样本又可以马上把参数拉回来。

```
C:\Windows\system32\cmd.exe
Grad:4.34235e-05 1.78119e-05 -0.00100021
Paras: 1.99971 0.999882 -0.993389 para diff norm: 1.00131e-06 iter num: 37888
Grad: 4.3417e-05 1.78092e-05 -0.00100005
Paras: 1.99971 0.999882 -0.99339 para diff norm: 1.00115e-06 iter num: 37889
Grad: 4.34104e-05 1.78065e-05 -0.000999903
Paras: 1.99971 0.999882 -0.993391 para diff norm: 1.001e-06 iter num: 37890
Grad: 4.34038e-05 1.78038e-05 -0.000999751
Paras: 1.99971 0.999882 -0.993392 para diff norm: 1.00085e-06 iter num: 37891
Grad:4.33973e-05 1.78011e-05 -0.0009996
Paras: 1.99971 0.999882 -0.993393 para diff norm: 1.0007e-06 iter num: 37892
Grad: 4.33907e-05 1.77984e-05 -0.000999449
Paras: 1.99971 0.999882 -0.993394 para diff norm: 1.00055e-06 iter num: 37893
Grad: 4.33841e-05 1.77957e-05 -0.000999298
Paras: 1.99971 0.999882 -0.993395 para diff norm: 1.0004e-06 iter num: 37894
Grad: 4.33776e-05 1.7793e-05 -0.000999147
Paras: 1.99971 0.999882 -0.993396 para diff norm: 1.00025e-06 iter num: 37895
Grad: 4.3371e-05 1.77903e-05 -0.000998996
Paras: 1.99971 0.999882 -0.993397 para diff norm: 1.00009e-06 iter num: 37896
Grad: 4.33644e-05 1.77876e-05 -0.000998844
Paras: 1.99971 0.999882 -0.993398 para diff norm: 9.99944e-07 iter num: 37897
pass
solution:
 1.99971
 0.999882
 -0.993398
Golden Solution:
 2
 1
 -1
Press any key to continue . . .
```

2)随机梯度下降法:

1000个样本, 设置的learn rate = 1, 一次迭代处理一个样本, 最终经过25000次迭代收敛到可以接受的标准。

```
cmd C:\Windows\system32\cmd.exe
Paras: 1.99795 0.9998981 -0.951616 para diff norm: 0.000446962 iter num: 24714
Paras: 1.99818 0.999232 -0.951604 para diff norm: 0.000338804 iter num: 24715
Paras: 1.99816 0.99922 -0.951605 para diff norm: 1.82845e-05 iter num: 24716
Paras: 1.99805 0.998911 -0.951617 para diff norm: 0.000328846 iter num: 24717
Paras: 1.99804 0.998721 -0.951623 para diff norm: 0.000190122 iter num: 24718
Paras: 1.9981 0.999161 -0.951611 para diff norm: 0.000444339 iter num: 24719
Paras: 1.99816 0.999501 -0.951604 para diff norm: 0.000344163 iter num: 24720
Paras: 1.99813 0.998416 -0.951627 para diff norm: 0.00108582 iter num: 24721
Paras: 1.99815 0.998517 -0.951624 para diff norm: 0.000102945 iter num: 24722
Paras: 1.99834 0.999645 -0.951597 para diff norm: 0.00114423 iter num: 24723
Paras: 1.99829 0.999464 -0.951642 para diff norm: 0.000192287 iter num: 24724
Paras: 1.99833 0.99955 -0.95164 para diff norm: 9.18398e-05 iter num: 24725
Paras: 1.99831 0.99951 -0.951641 para diff norm: 4.20315e-05 iter num: 24726
Paras: 1.99805 0.999223 -0.951665 para diff norm: 0.000389985 iter num: 24727
Paras: 1.99804 0.999007 -0.951669 para diff norm: 0.000216928 iter num: 24728
Paras: 1.99784 0.998829 -0.951689 para diff norm: 0.000266407 iter num: 24729
Paras: 1.99765 0.998724 -0.951707 para diff norm: 0.000220029 iter num: 24730
Paras: 1.99779 0.998928 -0.951696 para diff norm: 0.000248953 iter num: 24731
Paras: 1.99774 0.998884 -0.9517 para diff norm: 6.50401e-05 iter num: 24732
Paras: 1.99774 0.998884 -0.9517 para diff norm: 5.54896e-07 iter num: 24733
pass
solution:
1.99774
0.998884
-0.9517
Golden Solution:
2
1
-1
Press any key to continue . . .
```

3)牛顿法

```
cmd C:\Windows\system32\cmd.exe
Paras: 1.99954 1 -0.99909 para diff norm: 1.0188e-06 iter num: 7691
Paras: 1.99955 1 -0.999091 para diff norm: 1.01778e-06 iter num: 7692
Paras: 1.99955 1 -0.999091 para diff norm: 1.01676e-06 iter num: 7693
Paras: 1.99955 1 -0.999092 para diff norm: 1.01575e-06 iter num: 7694
Paras: 1.99955 1 -0.999093 para diff norm: 1.01473e-06 iter num: 7695
Paras: 1.99955 1 -0.999094 para diff norm: 1.01372e-06 iter num: 7696
Paras: 1.99955 1 -0.999095 para diff norm: 1.0127e-06 iter num: 7697
Paras: 1.99955 1 -0.999096 para diff norm: 1.01169e-06 iter num: 7698
Paras: 1.99955 1 -0.999097 para diff norm: 1.01068e-06 iter num: 7699
Paras: 1.99955 1 -0.999098 para diff norm: 1.00967e-06 iter num: 7700
Paras: 1.99955 1 -0.999099 para diff norm: 1.00866e-06 iter num: 7701
Paras: 1.99955 1 -0.9991 para diff norm: 1.00765e-06 iter num: 7702
Paras: 1.99955 1 -0.999101 para diff norm: 1.00664e-06 iter num: 7703
Paras: 1.99955 1 -0.999101 para diff norm: 1.00564e-06 iter num: 7704
Paras: 1.99955 1 -0.999102 para diff norm: 1.00463e-06 iter num: 7705
Paras: 1.99955 1 -0.999103 para diff norm: 1.00363e-06 iter num: 7706
Paras: 1.99955 1 -0.999104 para diff norm: 1.00262e-06 iter num: 7707
Paras: 1.99955 1 -0.999105 para diff norm: 1.00162e-06 iter num: 7708
Paras: 1.99955 1 -0.999106 para diff norm: 1.00062e-06 iter num: 7709
Paras: 1.99955 1 -0.999107 para diff norm: 9.99617e-07 iter num: 7710
pass
solution:
1.99955
1
-0.999107
Golden Solution:
2
1
-1
Press any key to continue . . .
```

对批量梯度下降法和随机梯度下降法的总结：

批量梯度下降---最小化所有训练样本的损失函数，使得最终求解的是全局的最优解，即求解的参数是使得风险函数最小，但是对于大规模样本问题效率低下。

随机梯度下降---最小化每条样本的损失函数，虽然不是每次迭代得到的损失函数都向着全局最优方向，但是大的整体的方向是向全局最优解的，最终的结果往往是在全局最优解附近，适用于大规模训练样本情况。

从实验数据来看，批量梯度下降法收敛到的是全局最优值，而随机梯度下降法收敛到最优值附件的地方。随机梯度下降法的好处是收敛远快于批量梯度下降法。

问题可以转化成寻找一组 λ 使得: $L(\lambda) = (X * \lambda - Y)^T (X * \lambda - Y)$ 极小

利用 $L(\lambda)$ 对向量 λ 求导可以得到¹:

$$\frac{\partial L(\lambda)}{\partial \lambda} = 2X^T(X\lambda - Y) = 0$$

我们也可以得到:

$$\frac{\partial^2 L(\lambda)}{\partial \lambda^2} = 2X^T X$$

$$X(n+1) = X(n) - f'(X(n))/f''(X(n))$$

```
MatrixXd Iteration::BathGradDescent(MatrixXd paras, int iterator_num)
{
    int rows = m_input_data.rows();
    int cols = m_input_data.cols();
    MatrixXd init_paras = paras;
    if (m_input_data.cols() == paras.rows())
    {
        MatrixXd grad = (1.0 / rows)*m_input_data.transpose()*(m_output_data - m_input_data*init_paras);
        paras += m_learn_rate*grad;
    }
    else
    {
        //error
    }
    return paras;
}

MatrixXd Iteration::StocGradDescent(MatrixXd paras, int index)
{
    int rows = m_input_data.rows();
    int cols = m_input_data.cols();
    MatrixXd input_data_i(1,cols);
    MatrixXd output_data_i(1, 1);
    output_data_i(0, 0) = m_output_data(index,0);

    for (int i = 0; i < cols; i++)
    {
        input_data_i(0, i) = m_input_data(index, i);
    }

    if (m_input_data.cols() == paras.rows())
    {
        MatrixXd grad = (1.0 / rows) *input_data_i.transpose()*(input_data_i*paras - output_data_i);
        paras = paras - grad;
    }
    else
    {
```

```

    //error
}
return paras;
}

```

由于牛顿法是基于当前位置的切线来确定下一次的位置，所以牛顿法又被很形象地称为是"切线法"。牛顿法的搜索路径(二维情况)如下图所示：

牛顿法搜索动态示例图：

牛顿法和拟牛顿法 (Newton's method &Quasi-Newton methods)

1)牛顿法 (Newton's method)

牛顿法是一种在实数域和复数域上近似求解方程的方法。方法使用函数 $f(x)$ 的泰勒级数的前面几项来寻找方程 $f(x) = 0$ 的根。¹牛顿法最大的特点就在于它的收敛速度很快。

步骤：

首先，选择一个接近函数 $f(x)$ 零点的 x_0 ，计算相应的计算相应的 $f(x_0)$ 和切线斜率 $f'(x_0)$ 。然后计算穿过点 $(x_0, f(x_0))$ 并且斜率为 $f'(x_0)$ 的直线和x轴的交点的x轴坐标，也就是如下方程：

$$x * f'(x_0) + f(x_0) - x_0 * f'(x_0) = 0$$

我们将新求得的点的 x 坐标命名为 x_1 ，通常 x_1 会比 x_0 更接近方程 $f(x) = 0$ 的解。因此我们现在可以利用 x_1 开始下一轮迭代。迭代公式可化简为如下所示：

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{1}{1}$$

如果X是向量，则可以写成向量形式：

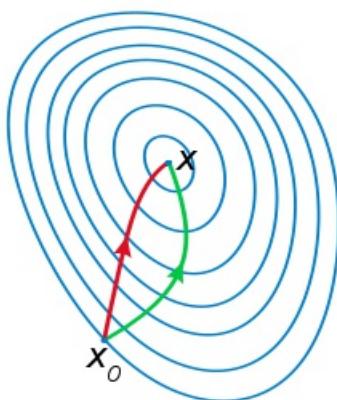
已经证明，如果 $f'(x)$ 是连续的，并且待求的零点 x 是孤立的，那么在零点 x 周围存在一个区域，只要初始值 x_0 位于这个邻近区域内，那么牛顿法必定收敛。并且，如果 $f'(x)$ 不为0，那么牛顿法将具有平方收敛的性能。粗略的说，这意味着每迭代一次，牛顿法结果的有效数字将增加一倍。下图为一个牛顿法执行过程的例子。



关于牛顿法和梯度下降法的效率对比：

从本质上去看，牛顿法是二阶收敛，梯度下降是一阶收敛，所以牛顿法就更快。如果更通俗地说的话，比如你想找一条最短的路径走到一个盆地的最底部，梯度下降法每次只从你当前所处位置选一个坡度最大的方向走一步，牛顿法在选择方向时，不仅会考虑坡度是否够大，还会考虑你走了一步之后，坡度是否会变得更大。所以，可以说牛顿法比梯度下降法看得更远一点，能更快地走到最底部。（牛顿法目光更加长远，所以少走弯路；相对而言，梯度下降法只考虑了局部的最优，没有全局思想。）

根据wiki上的解释，从几何上说，牛顿法就是用一个二次曲面去拟合你当前所处位置的局部曲面，而梯度下降法是用一个平面去拟合当前的局部曲面，通常情况下，二次曲面的拟合会比平面更好，所以牛顿法选择的下降路径会更符合真实的最优下降路径。



注：红色的牛顿法的迭代路径，绿色的是梯度下降法的迭代路径。

牛顿法的优缺点总结：

优点：二阶收敛，收敛速度快；

缺点：牛顿法是一种迭代算法，每一步都要求解目标函数的Hessian矩阵的逆矩阵，计算比较复杂。

2) 拟牛顿法 (Quasi-Newton Methods)

拟牛顿法是求解非线性优化问题最有效的方法之一，于20世纪50年代由美国Argonne国家实验室的物理学家W.C.Davidon所提出来。Davidon设计的这种算法在当时看来是非线性优化领域最具创造性的发明之一。不久R. Fletcher和M. J. D. Powell证实了这种新的算法远比其他方法快速和可靠，使得非线性优化这门学科在一夜之间突飞猛进。

拟牛顿法的本质思想是改善牛顿法每次都要求解复杂的Hessian矩阵的逆矩阵的缺陷，它使用正定矩阵来近似Hessian矩阵的逆，从而简化了运算的复杂度。拟牛顿法和最速下降法一样只要求每一步迭代时知道目标函数的梯度。通过测量梯度的变化，构造一个目标函数的模型使之足以产生超线性收敛性。这类方法大大优于最速下降法，尤其对于困难的问题。另外，因为拟牛顿法不需要二阶导数的信息，所以有时比牛顿法更为有效。如今，优化软件中包含了大量的拟牛顿算法用来解决无约束、约束和大规模的优化问题。

具体步骤：

拟牛顿法的基本思想如下。首先构造目标函数在当前迭代 x_k 的二次模型：

$$m_k(p) = f(x_k) + \nabla f(x_k)^T p + \frac{p^T B_k p}{2}$$

$$p_k = -B_k^{-1} \nabla f(x_k)$$

这里 B_k 是一个对称正定矩阵，于是我们取这个二次模型的最优解作为搜索方向，并且得到新的迭代点：

$$x_{k+1} = x_k + \alpha_k p_k$$

其中我们要求步长 α_k 满足Wolfe条件。这样的迭代与牛顿法类似，区别就在于用近似的Hesse矩阵 B_k 代替真实的Hesse矩阵。所以拟牛顿法最关键的地方就是每一步迭代中矩阵 B_k 的更新。现在假设得到一个新的迭代 x_{k+1} ，并得到一个新的二次模型：

这个公式被称为割线方程。常用的拟牛顿法有DFP算法和BFGS算法。

最优化方法：牛顿迭代法和拟牛顿迭代法

共轭梯度法

共轭梯度法 (Conjugate Gradient) 是介于最速下降法与牛顿法之间的一个方法，它仅需要一阶导数信息，但克服了最速下降法收敛慢的缺点，同时又避免了牛顿法需要储存和计算Hesse矩阵并求逆的缺点，共轭梯度法不仅是解决大型线性方程组最有用的方法之一，也是解大型非线性最优

化问题最有效的算法之一。向量共轭的定义：若 \mathbf{A} 是正定对称阵，若非0矢量 \mathbf{u}, \mathbf{v} 满足，

$\mathbf{u}^T \mathbf{A} \mathbf{v} = 0$, 则称矢量 \mathbf{u}, \mathbf{v} 是共轭的。假设：
 $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n]$ 是一组基于 \mathbf{A} 的共轭矢量，

则 $\mathbf{Ax} = \mathbf{b}$ 的解 \mathbf{x}^* 可以表示为：
 $\mathbf{x}^* = \sum_{i=1}^n \alpha_i \mathbf{p}_i$,

因此基于基矢量展开，我们有：

$$\mathbf{Ax}_* = \sum_{i=1}^n \alpha_i \mathbf{Ap}_i \text{ 左乘 } \mathbf{P}_k^T : \quad \mathbf{P}_k^T \mathbf{Ax}_* = \sum_{i=1}^n \alpha_i P_k^T \mathbf{Ap}_i \text{ 代入} :$$

$$\mathbf{P}_k^T \mathbf{Ax}_* = \mathbf{b}, \mathbf{u}^T \mathbf{Av} = \langle \mathbf{u}, \mathbf{v} \rangle_A,$$

利用 $i! = k$, 有 $\langle \mathbf{p}_k, \mathbf{p}_i \rangle_A = 0$ 就得到：

如果我们的小心的选择 \mathbf{p}_k , 为了获得解 \mathbf{x}_* 的一个好的近似，我们并不需要所有的基向量。因此，我们把共轭梯度算法当成一种迭代算法，它也允许我们近似 n 很大，以至于直接求解需要花费太多时间的系统。我们给 \mathbf{x}_* 一个初始猜测值 \mathbf{x}_0 , 假设 $\mathbf{x}_0 = \mathbf{0}$, 在求解的过程中，我们需要一种标准来告诉我们是否我们的解更靠近真实的解 \mathbf{x}_* , 实际上，求解 $\mathbf{Ax} = \mathbf{b}$ 等价于求解如下二次函数的极小

值：
 $\mathbf{f}(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{Ax} - \mathbf{x}^T \mathbf{b}$, 他存在唯一的最小值，因为他的二阶导是正定的：

$$\mathbf{D}^2 \mathbf{f}(\mathbf{x}) = \mathbf{A}, \text{ 解就是一阶导数等于0的地方} \quad \mathbf{D}\mathbf{f}(\mathbf{x}) = \mathbf{Ax} - \mathbf{b}, \text{ 假设第一个基矢 } \mathbf{p}_0 \text{ 就是 } f(x) \text{ 在 } \mathbf{x} = \mathbf{x}_0 \text{ 处的负梯度，我们可以假设}$$

$$\mathbf{x}_0 = \mathbf{0}. \text{ 因此:} \quad \mathbf{p}_0 = \mathbf{b} - \mathbf{Ax}_0. \text{ 令 } \mathbf{r}_0 \text{ 表示第k步的残差:}$$

$\mathbf{r}_k = \mathbf{b} - \mathbf{Ax}_k, \mathbf{r}_k$ 是 $f(x)$ 在 $x = x_k$ 处的负梯度。为了让 \mathbf{p}_k 与前面的所有 \mathbf{p}_i 相互共轭，
 \mathbf{p}_k 可以如下构造：
 $\mathbf{p}_k = \mathbf{r}_k - \sum_{i < k} \frac{\mathbf{p}_i^T \mathbf{Ar}_k}{\mathbf{p}_i^T \mathbf{Ap}_i} \mathbf{p}_i$ 沿着这个方向，因此下一步的优化方向就是：

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \text{ 满足:} \quad g'(\alpha_k = 0) \text{ 因此:}$$

$$\mathbf{f}(\mathbf{x}_{k+1}) = \mathbf{f}(\mathbf{x}_k + \alpha_k \mathbf{p}_k) =: g(\alpha_k), \quad \alpha_k = \frac{\mathbf{p}_k^T \mathbf{b}}{\mathbf{p}_k^T \mathbf{Ap}_k} = \frac{\mathbf{p}_k^T (\mathbf{r}_k + \mathbf{Ax}_k)}{\mathbf{p}_k^T \mathbf{Ap}_k} = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{Ap}_k}$$

算法：

$$\mathbf{r}_0 := \mathbf{b} - \mathbf{Ax}_0$$

$$\mathbf{p}_0 := \mathbf{r}_0$$

$$\mathbf{k} := 0$$

repeat:

$$\alpha_k := r_0$$

$$\alpha_k := \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{Ap}_k}$$

$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

$$\mathbf{r}_{k+1} := \mathbf{r}_k + \alpha_k \mathbf{Ap}_k$$

if r_{k+1} is sufficiently small, then exit loop.

$$\beta_k := \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$$

■ $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$

■ $k := k + 1$

protected Vector FillDataAbsolute(Vector f, Matrix J, Variable variable)

线性与非线性方程组解的稳定性分析

线性方程组解的稳定性分析

系统解的稳定性定义成系统小的扰动对系统解的影响。

Example 1:

$$\begin{bmatrix} 400 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

Solution:

$$x_1 = -100, x_2 = -200$$

If we give small change to matrix A, change A_{11} from 400 to 401 then:

$$\begin{bmatrix} 401 & -201 \\ -800 & 401 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 200 \\ -200 \end{bmatrix}$$

This time the solution is:

$$x_1 = 40000, x_2 = 79800$$

Ill-conditioned:

When the solution is highly sensitive to the values of the coefficient matrix A or the righthand side constant vector b, the equations are called to be ill-conditioned.

Condition Number

Let's linear equations:

$$\mathbf{Ax} = \mathbf{b},$$

Let us investigate first, how a small change in the \mathbf{b} vector changes the solution vector. \mathbf{x} is the solution of the original system and let $\mathbf{x} + \Delta\mathbf{x}$ is the solution when \mathbf{b} changes from \mathbf{b} to $\mathbf{b} + \Delta\mathbf{b}$.

Then we can write:

$$\mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b} + \Delta\mathbf{b}$$

$$\text{or } \mathbf{Ax} + \mathbf{A}\Delta\mathbf{x} = \mathbf{b} + \Delta\mathbf{b}$$

But because $\mathbf{Ax} = \mathbf{b}$, it follows that

$$\mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{b}$$

$$\text{So: } \mathbf{\delta x} = \mathbf{A}^{-1}\Delta\mathbf{b}$$

Using the matrix norm properties:

$$\|\mathbf{Ax}\| \leq \|\mathbf{A}\| \|\mathbf{x}\|$$

We can get:

$$\|\mathbf{A}^{-1}\Delta\mathbf{b}\| \leq \|\mathbf{A}^{-1}\| \|\Delta\mathbf{b}\|,$$

Also, we can get:

$$\|\mathbf{b}\| = \|\mathbf{Ax}\| \leq \|\mathbf{A}\| \|\mathbf{x}\|$$

Using equations 2.5 and 2.6 we can get:

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{A}\| \cdot \|\mathbf{x}\|} \leq \frac{\|\mathbf{A}^{-1}\| \cdot \|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

Let's define condition number as: $\mathbf{K}(\mathbf{A}) = \|\mathbf{A}\| \cdot \|\mathbf{A}^{-1}\|$

we can rewrite equation 2.7 as:

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \mathbf{K}(\mathbf{A}) \cdot \frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|}$$

Now, let us investigate what happens if a small change is made in the coefficient matrix

- A. Consider \mathbf{A} is changed to $\mathbf{A} + \Delta\mathbf{A}$ and the solution changes from \mathbf{x} to $\mathbf{x} + \Delta\mathbf{x}$
- $(\mathbf{A} + \Delta\mathbf{A})(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{b}$, we can obtain:

$$\frac{\|\Delta\mathbf{x}\|}{\|\mathbf{x} + \Delta\mathbf{x}\|} \leq \mathbf{K}(\mathbf{A}) \cdot \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}$$

$\mathbf{K}(\mathbf{A})$ is a measure of the relative sensitivity of the solution to changes in the right-hand side vector \mathbf{b} . When the condition number $\mathbf{K}(\mathbf{A})$ becomes large, the system is regarded as being illconditioned.

Matrices with condition numbers near 1 are said to be well-conditioned.

非线性方程组解的稳定性分析

For non-linear system,

最小二乘法求解线性, 非线性方程组

所有的线性方程组或者非线性方程组可以转化成一个最小二乘法问题, 使得拟合的方程与观察值之间的平方和最小。

$$\hat{\beta} = \operatorname{argmin}_{\beta} S(\beta) = \operatorname{argmin}_{\beta} \sum_{i=1}^m [y_i - f(x_i, \beta)]^2$$

y_i 是观测值, x_i 是已知变量, 一共m组观测值。

当 $f(x, \beta)$ 是线性方程组时

即 $\mathbf{f}(\mathbf{X}, \beta) = \mathbf{A}\beta$

cost function 可以表示如下：

$$\blacksquare L(\beta) = ||\mathbf{Y} - \mathbf{A}\beta||^2$$

其中 $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2 \dots \mathbf{y}_m]^T$

cost function 对 β

$$\blacksquare L(\beta) = \mathbf{Y}^T \mathbf{Y} - 2\mathbf{Y}^T \mathbf{A}\beta + \beta^T \mathbf{A}^T \mathbf{A}\beta$$

求导,

$$\blacksquare -2\mathbf{A}^T \mathbf{Y} + 2(\mathbf{A}^T \mathbf{A})\beta = 0$$

再让导数等于0, 就可以求得解为: β

$$\blacksquare \beta = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{Y}$$

我们接下来主要讨论非线性的情况:

也就是残差 $\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i, \beta)$ 不能表示成线性形式。

当然我们有很多方法来求解方程(4.1), 比如 梯度下降法, 牛顿法, 高斯-牛顿法, 以及 Levenberg-Marquardt 梯度下降法

把损失函数展开到一阶。

$$\blacksquare f(x + \alpha \mathbf{d}) = f(x_0) + \alpha f'(x_0) \mathbf{d} + O(\alpha^2)$$

$$\blacksquare x_{t+1} = x_t - \alpha f'(x_t),$$

牛顿法

把cost function进行展开到二阶:

$$\blacksquare f(x_{t+1}) = f(x_t) + g(x_{t+1} - x_t) + \frac{1}{2}(x_{t+1} - x_t)^T H(x_{t+1} - x_t)$$

求导, $\frac{\partial f}{\partial x_t} = g + H(x_{t+1} - x_t)$, 让导数为0就有

$$\blacksquare x_{t+1} = x_t - H^{-1}g$$

要是 H 是正定的, 上面的就是凸函数, 也就一定有了最小值。可惜 H 不一定是正定的, 这就引导出了下面的方法

高斯-牛顿法

cost function 可以表示成残差的形式:

$$\blacksquare L(x) = \sum_{i=1}^m r_i(x)^2$$

根据牛顿法:

$$v x_{t+1} = x_t - H^{-1}g,$$

梯度表示为:

$$g_j = 2 \sum_i r_i \frac{\partial r_i}{\partial x_j}$$

对方程(4.9)求二阶导, 我们可以得到Hessian矩阵:

$$H_{jk} = 2 \sum_i \left(\frac{\partial r_i}{\partial x_j} \frac{\partial r_i}{\partial x_k} + r_i \frac{\partial^2 r_i}{\partial x_j \partial x_k} \right),$$

当残差 r_i 很小的时候, 我们就可以去掉最后一项, 因此

$$H_{jk} \approx 2 \sum_i J_{ij} J_{ik} \quad \text{With} \quad J_{ij} = \frac{\partial r_i}{\partial x_j}$$

这样Hessian就是半正定了。

因此参数迭代公式(4.10)可以写成:

$$x_{t+1} = x_t - (J^T J)^{-1} J^T r,$$

Levenberg-Marquardt

方法用于求解非线性最小二乘问题, 结合了梯度下降法和高斯-牛顿法。

其主要改变是在Hessian阵中加入对角线项。当然, 这是一种L2正则化方式。

$$x_{t+1} = x_t - (H + \lambda I_n)^{-1} g,$$

$$x_{t+1} = x_t - (J^T J + \lambda I_n)^{-1} J^T r$$

L-M算法的不足点。

当 λ 很大时, $J^T J + \lambda I_n$ 根本没用, Marquardt为了让梯度小的方向移动的快一些, 来防止小梯度

方向的收敛, 把中间的单位矩阵换成了 $\text{diag}(J^T J)$, 因此迭代变成:

$$x_{t+1} = x_t - (J^T J + \lambda \text{diag}(J^T J))^{-1} J^T r$$

阻尼项(damping parameter) λ 的选择, Marquardt 推荐一个初始值 λ_0 与因子 $v > 1$, 开始时,

$\lambda = \lambda_0$, 然后计算cost functions, 第二次计算 $\lambda = \lambda_0/v$, 如果两者cost function都比初始点高, 然后

我们就增大阻尼项, 通过乘以 v , 直到我们发现当 $\lambda = \lambda_0 v^k$ 时, cost function下降。

如果使用 $\lambda = \lambda_0/v$ 使得cost function下降, 然后我们就把 λ_0/v 作为新的 λ

当 $\lambda = 0$ 时, 就是高斯-牛顿法, 当 λ 趋于无穷时, 就是梯度下降法。如果使用 λ/v 没有使损失函数下降, 使用 λ 导致损失函数下降, 那么我们就继续使用 λ 做为阻尼项。

归一化的残差

■ $J_S = \Sigma J, \Sigma_{ij} = \frac{1}{y_i} \delta_{ij}$

L2正则化与Levenberg-Marquardt算法

我们在cost function加上L2正则项

■ $L(\beta) = \sum_{i=1}^m [y_i - f(x_i, \beta)]^2 + \lambda \|\beta\|^2$

可以表示成：

■ $L(\beta) = L(\beta_0) + (\beta - \beta_0)^T \nabla_{\beta} L(\beta_0) + \frac{1}{2} (\beta - \beta_0)^T H(\beta - \beta_0) + \lambda \|\beta\|^2 + O(\beta^3)$

求一阶导数：

■ $L(\beta) = L(\beta_0) + (\beta - \beta_0)^T g + \frac{1}{2} (\beta - \beta_0)^T H(\beta - \beta_0) + \frac{1}{2} \lambda \|\beta\|^2 + O(\beta^3)$

令其为0：

■ $\frac{\partial L(\beta)}{\partial \beta} = g + H(\beta - \beta_0) + \lambda \beta = 0$

就得到：

■ $\beta = (H + \lambda I_n)^{-1} H \beta_0 - (H + \lambda I_n)^{-1} g,$

第八章 机器学习

机器学习

1. 过拟合与欠拟合, 交差验证的目的, 超参数搜索方法, EarlyStopping
2. L1正则和L2正则的做法, 正则化背后的思想, BatchNorm, Covariance Shift, L1正则产生稀疏解的原理
3. 逻辑回归为何是线性模型, LR如何解决低维不可分, 从图模型角度看LR,
4. 和朴素贝叶斯和无监督
5. 几种参数估计方法MLE, MAP, 贝叶斯的联系与区别
6. 简单说下SVM的支持向量, KKT, 何为对偶, 核的通俗理解
7. GBDT, 随机森林能否并行, 问下bagging, boosting
8. 生成模型, 判别模型举个例子
9. 聚类方法的掌握, 问下kmeans的EM推导思路, 谱聚类和Graph-cut的理解
10. 梯度下降类方法和牛顿类方法的区别, 随便问问Adam, L-BFGS的思路
11. 半监督的思想, 问一些特定半监督算法是如何利用无标签数据的, 从MAP角度看半监督
12. 常见的分类模型的评价指标(顺便问问交叉熵, ROC如何绘制, AUC的物理含义, 类别不均匀样本)

神经网络

1. CNN中卷积操作和卷积核作用, maxpooling作用
2. 卷积层与全连接层的联系
3. 梯度爆炸与消失的概念(顺便问问神经网络权重初始化的方法, 为何能减缓梯度爆炸与消失, CNN中有哪些解决方法, LSTM如何解决的, 如何梯度裁剪, dropout如何用在RNN系列网络中, dropout如何防止过拟合)
4. 为何卷积可以用在图像, 语音, 语句上, 顺便问问channel在不同类型数据源中的含义

自然语言处理, 推荐系统

1. CRF跟逻辑回归, 最大熵模型的关系
2. CRF的优化方法, CRF和MRF的联系, HMM与CRF的关系(顺便问问朴素贝叶斯和HMM的联系, LSTM+CRF用于序列标注的原理, CRF的点函数和边函数, CRF的经验分布)
3. wordEmbedding的几种常用方法和原理(language model, perplexity评价指标, word2vec跟Glove的异同)
4. Topic model说一说, 为何CNN能用在文本分类, syntactic 和semantic问题举例,
5. 常见的sentence embedding方法, 注意力机制(注意力机制的几种不同情形, 为何引入, seq2seq原理)
6. 序列标注的评价指标, 语义消歧的做法, 常见的跟word有关的特征
7. factorization machine, 常见矩阵分解模型
8. 如何把分类模型用于商品推荐(包括数据集划分, 模型验证等)
9. 序列学习, wide & deep model(顺便问问为何wide和deep)

第一步(一个月), 公式推导必须会, 核心思想会, 并且要融会贯通。

1. SVM
2. LR, LTR
3. 决策树, RF, GBDT, xgboost
4. 贝叶斯
5. 降维:PCA, SVD
6. BP的公式推导, 以及写一个简单的神经网络, 并跑个小的数据。用python, numpy。
7. LeetCode上面刷Easy与Medium的题
8. 看面经, 找面试常遇到的问题

第二步是最优化总结(半个月), 总结常用的优化算法, 比如梯度下降, 牛顿, 拟牛顿, trust region, 二次规划。

机器学习score函数设置:

1. 回归, 分类, 决策树, 深度神经网络, 图模型, 概率统计, 最优化方法
2. 分类, 聚类, 特征选择, 降维等数据挖掘技术
3. 机器学习, 概率统计, 最优化
4. GBDT, LR, LTR, 特征提取
5. 推荐系统基本算法: LR, GBDT, SVD/SVD++, FM/FFM具体实用场景与变形
6. 对分类, 回归, 聚类, 标注等统计机器学习问题有深入研究, 熟悉常用模型:LR, KNN, Naive bayes, rf, GBDT, SVM, PCA, SVD, kmeans, kmodes等
7. 精通主流机器学习算法, 对贝叶斯, 随机森林, SVM, 神经网络, 聚类, PCA等有深入研究
8. 熟悉数据分析思路, 熟悉经典的数据挖掘, 机器学习算法, 如:LR, 决策树, BP神经网络, SVM等浅层学习, 熟悉集成算法, 诸如bagging, boosting, 了解深度学习CNN, RNN, 熟悉使用python, 了解pandas, sklearn, xgboost
9. 熟悉常用的最优化算法设计与实现, 对于非凸优化问题有深入的理解(并行模拟退火, 蒙特卡洛等优化方法)
10. 机器学习算法: 贝叶斯, 聚类, 逻辑回归, SVM, GBDT, RF

总结:

1. 分类:
 - i. SVM,
 - ii. LR
2. 回归:
3. 集成学习:
 - i. bagging
 - ii. Boosting
4. 工具
 - i. sklearn, xgboost, tensorflow

分类总结：

1. Regression:
 2.
 - i. Ordinary Least Squares Regression(OLSR)
 - ii. Linear Regression
 - iii. Logistic Regression
 - iv. Stepwise Regression
 - v. Multivariate Adaptive Regression Splines
 - vi. Locally Estimated Scatterplot Smoothing
 3. Regularization Algorithms
 4.
 - i. Ridge Regression
 - ii. Least Absolute Shrinkage and Selection Operator(LASSO)
 - iii. Least-Angle Regression(LARS)
 5. Ensemble Algorithms
 6.
 - i. Boosting
 - ii. Booststrapped Aggregation(Bagging)
 - iii. Adaboost
 - iv. Stacked Generalization(Blending)
 - v. Gradient Boosting Machines(GBM)
 - vi. Gradient Boosted Regression Trees(GBRT)
 - vii. Random Forest
 7. Decision Tree Algorithms
 8.
 - i. Classification and Regression Tree(CART)
 - ii. Iterative Dichotomiser3(ID3)
 - iii. C4.5 and C5.0
 - iv. Chi-Squared Automatic Interaction Detection(CHAID)
 - v. Decision Stump
 - vi. M5
 - vii. Conditional Decision Trees
 9. Dimensionality Reduction Algorithms
 10.
 - i. Principle Component Analysis(PCA)
 - ii. Principle Component Regression(PCR)
 - iii. Sammon Mapping
 - iv. Multidimensional Scaling(MDS)
 - v. Projection Pursuit
 - vi. Linear Discriminant Analysis(LDA)
 - vii. Mixture Discriminant Analysis(MDA)

- viii. Quadratic Discriminant Analysis(QDA)
- ix. Flexible Discriminant Analysis(FDA)
- 11. Bayesian Algorithms
- 12.
 - i. Naive Bayes
 - ii. Gaussian Naive Bayes
 - iii. Multinomial Naive Bayes
 - iv. Averaged One-Dependence Estimators(AODE)
 - v. Bayesian Belief Network(BBN)
 - vi. Bayesian Network(BN)
- 13. Clustering Algorithms
- 14.
 - i. K-Means
 - ii. K-Medians
 - iii. Expectation Maximisation(EM)
 - iv. Hierarchical
- 15. Instance-Based Algorithms
- 16.
 - i. K-Nearest Neighbor(KNN)
 - ii. Learning Vector Quantization(LVQ)
 - iii. Self-Organizing Map(SOM)
 - iv. Locally Weighted Learning(LWL)
- 17. Graphical Models
- 18.
 - i. Bayesian Network
 - ii. Markov random field
 - iii. Chain Graphs
 - iv. Ancestral Graph
- 19. Association Rule Learning Algorithms
- 20.
 - i. Apriori Algorithm
 - ii. Eclat Algorithm
 - iii. FP-growth
- 1. Deep Learning
- 2.
 - i. Deep Boltzmann Machine(DBM)
 - ii. Deep Belief Networks(DBN)
 - iii. Convolutional Neural Network(CNN)
 - iv. Stacked Auto-Encoders
- 3. Artificial Neural Network

4.

- i. Perception
- ii. Back-Propagation
- iii. Radial Basis Function Network(RBFN)

机器学习原理

偏差, 方差, 噪声

偏差-方差分解可以用来对学习算法的期望泛化错误率进行拆分。同时, 偏差-方差分解为我们设计模型与分析提供了一个比较清晰的方向。

假设测试样本是 \mathbf{x} , 令 y_D 为 \mathbf{x} 在数据集上的标记(可能存在标记错误的情况), y 为 \mathbf{x} 的真实标记, $f(\mathbf{x}; D)$ 为训练集 D 上学得模型 f 在 \mathbf{x} 上的预测输出, 以回归模型为例, 学习算法的期望预测为:

$$\hat{f}(\mathbf{x}) = E_D[f(\mathbf{x}, D)]$$

方差是:

$$var(\mathbf{x}) = E_D[f(\mathbf{x}, D) - \hat{f}(\mathbf{x})]^2$$

噪声是:

$$\epsilon^2 = E_D[y_D - y]^2$$

输出期望与真实标记之间的差别成为偏差(Bias), 即:

$$bias^2(\mathbf{x}) = E_D[\hat{f}(\mathbf{x}) - y]^2$$

为方便讨论, 假设噪声的期望为0; 即: $E_D[y_D - y] = 0$. 下面来对模型的期望泛化误差进行分解:

$$\begin{aligned} E(f; D) &= E_D[(f(\mathbf{x}, D) - y_D)^2] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}) + \hat{f}(\mathbf{x}) - y_D)^2] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y_D)^2] + E_D[2(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))(\hat{f}(\mathbf{x}) - y_D)] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y_D)^2] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y + y - y_D)^2] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y)^2] + E_D[(y - y_D)^2] + E_D[2(\hat{f}(\mathbf{x}) - y)(y - y_D)] \\ &= E_D[(f(\mathbf{x}, D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y)^2] + E_D[(y - y_D)^2] \end{aligned}$$

因此就得到:

$E(f; D) = bias^2(\mathbf{x}) + var(\mathbf{x}) + \epsilon^2$ 也就是泛化误差分成偏差, 方差与噪声之和。我们设计模型分析时, 可以从这三方面去考虑。

噪声

为了消除噪声的影响, 我们需要对数据进行清洗, 进行预处理。在很大程度上就是为了得到更干净的数据。

Bias

偏差小，说明的是模型很准，可能有过拟合的倾向，增加模型的复杂度可以使得bias减小。但是泛化能力变差，也就是variance会大，模型对数据很敏感。如果模型bias大，通过对简单的模型进行boost，来提升模型的准确性，也就是Boost系列算法做的事情，如：AdaBoost, GBDT, XGBoost。

方差

方差小，说明模型很稳定，也就是说模型的泛化能力好，可能测试集上的数据相对于训练集来说有一些变化，但是也不影响模型的输出结果，此时，可能模型欠拟合，因此泛化能力好。如果模型variance大，可以通过训练多个模型，并让各个模型之间的关联性很小，通过求平均来减小Variance，这就是集成模型中bagging系列算法做的事情，如Bagging, Random Forest.

最大似然函数

输入： $X \in R^n$

输出： $Y \in (1, 2, \dots, K)$

条件概率： $P(X = x|Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)}|Y = C_k)$, k=1,2,...,K

假设有N个样本点 $(X_i, Y_i), i = 1, 2, \dots, N$

条件概率： $P(Y = y|X = x, \theta)$, 其中 θ 是模型的参数。

似然函数定义为：每个样本点发生概率的乘积： $L(\theta) = \prod_{n=1}^N P(Y = Y_n|X = X_n, \theta)$

我们需要求得一个模型，使得在所有样本点在该模型发生的几率极大，几率是联合几率，是每个样本发生几率的乘积。也就是极大化似然函数。

分类与回归问题似乎都可以转化成求解似然函数。

用极大似然函数求解回归问题

输入： $\mathbf{X} \in R^n$

输出： $Y \in R$

我们的模型是 $f(\mathbf{X}, \Theta)$ ，其中 Θ 是模型参数， \mathbf{X} 是输入变量。假设我们的损失函数是平方误差，因此我们的cost function可以表示如下：

$$L(\mathbf{X}, \Theta) = \frac{1}{2} \sum_{n=1}^N (f(X_n, \Theta) - Y_n)^2$$

等价于：

$$e^{L(\mathbf{X}, \Theta)} = e^{\frac{1}{2} \sum_{n=1}^N (f(X_n, \Theta) - Y_n)^2} = \prod_{n=1}^N e^{\frac{1}{2} (f(X_n, \Theta) - Y_n)^2}$$

实际上，最小化 $L(\mathbf{X}, \Theta)$ 等价于最小化 $e^{L(\mathbf{X}, \Theta)}$ ，等价于最大化 $e^{-L(\mathbf{X}, \Theta)}$ ，也就是等价于最大化

$$\Gamma(\Theta) = \prod_{n=1}^N e^{-\frac{1}{2}(f(X_n, \Theta) - Y_n)^2} = \prod_{n=1}^N P(X_n, \Theta)$$

其中 $P(X_n, \Theta) = e^{-\frac{1}{2}(f(X_n, \Theta) - Y_n)^2}$

定义成回归模型中样本 (X_n, Y_n) 发生的几率，这样我们就把回归问题与几率联系起来了。我们把求解损失函数最小，转化成极大似然函数的求解。

对于不同的损失函数，我们都可以转化成对应的概率。

对于分类问题，怎么用一个统一的框架来解析？可以是基于概率的。

用极大似然函数求解多类分类问题

在这里我们以多项逻辑斯蒂回归为例，讨论怎么用最大化似然函数来求解这一类问题。

假设离散性随机变量Y的取值是 $(1, 2, \dots, K)$ ，输入变量 $\mathbf{X} \in R^n$

则，模型如下：

$$P(Y = k | \mathbf{x}) = \frac{\exp(\mathbf{w}_k \cdot \mathbf{x})}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}, k = 1, 2, \dots, K - 1$$

$$P(Y = K | \mathbf{x}) = \frac{1}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}$$

这里 $\mathbf{x} \in \mathbf{R}^{n+1}$ ； $\mathbf{w}_k \in \mathbf{R}^{n+1}$ 。

因此，总的似然函数可以表示为：

$$L(\mathbf{W}) = \prod_{n=1}^N P(Y = Y_n | X = X_n) \text{ 这样，我们需要求的参数是 } W = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_{n+1}).$$

用极大似然与SVM来做多类分类问题

核心，定义出到超平面的距离，每一个类，一个超平面，然后定义距离，然后转化成概率，最终转化成极大似然函数求解。

极大似然与最大熵原理的等价性

最大熵原理

奥卡姆剃刀

SVM, KKT条件与核函数方法

SVM

线性可分二分类问题

一组数据 $(x_i, y_i), i = 1, 2 \dots N, y_i \in +1, -1, x_i \in R^m$

SVM算法是为了求得一个分割超平面 $\mathbf{w}\mathbf{x} + b = 0$ 使得所有的样本点到超平面的几何距离都不小于 γ , 且这个 γ 是最大的。

首先我们定义几何距离和函数距离:

几何距离: $\gamma_i = y_i(\frac{\mathbf{w}\mathbf{x}_i}{\|\mathbf{w}\|} + \frac{b}{\|\mathbf{w}\|})$

除以 $\|\mathbf{w}\|$ 是因为 \mathbf{x}, b 同时乘以一个因子, 超平面不变。

函数间距: $\gamma_i = y_i(\mathbf{w}\mathbf{x}_i + b)$

我们要使: $\gamma_i \geq \gamma$. 对任何的样本点 i 都成立, 并求得 γ 的最大值。故上面的问题可以用如下数学来刻画:

$$\max_{\mathbf{w}, \mathbf{b}} \gamma$$

$$s.t. \quad y_i(\frac{\mathbf{w}\mathbf{x}_i}{\|\mathbf{w}\|} + \frac{b}{\|\mathbf{w}\|}) \geq \gamma$$

转化为函数间隔:

$$\max_{\mathbf{w}, \mathbf{b}} \frac{\gamma}{\|\mathbf{w}\|}$$

$$s.t. \quad y_i(\mathbf{w}\mathbf{x}_i + b) \geq \|\mathbf{w}\|\gamma = \hat{\gamma}$$

为了简化计算, 我们取 $\hat{\gamma} = 1$

因此问题转化为:

$$\max_{\mathbf{w}, \mathbf{b}} \frac{1}{\|\mathbf{w}\|}$$

$$s.t. \quad y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1, i = 1, 2, \dots, N$$

转化为求极小问题:

$$\max_{\mathbf{w}, \mathbf{b}} \frac{1}{2} \|\mathbf{w}\|^2 \quad s.t. \quad y_i(\mathbf{w}\mathbf{x}_i + b) \geq 1, i = 1, 2, \dots, N$$

问题转化成带约束的优化问题(在这里是凸优化问题)。带等式的约束问题可以通过引入拉格朗日乘子求解, 带不等式约束的问题可以引入KKT乘子求解。下面我们先来讨论下带约束的凸优化问题。

Karush-Kuhn-Tucker(KKT)条件

1. 先讨论一般带约束优化问题的数据描述
2. 然后引入其作用约束与不起作用约束的定义
3. 再引入正则点的定义
4. 接下来引入KKT条件 5. 针对局部极小问题，我们考虑优化函数的二阶导数问题。

一般形式的优化问题：

$$\begin{aligned} & \text{minimize } f(\mathbf{x}) \\ & \text{s.t. } \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ & \text{s.t. } \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \end{aligned}$$

其中： $f : R^n \rightarrow R$, $\mathbf{h} : R^n \rightarrow R^m$, $m \leq n$; $\mathbf{g} : R^n \rightarrow R^p$;

针对这一问题，引入如下定义：

定义21.1. 对于一个不等式约束 $g_i(x) \leq 0$, 如果在 x^* 处, $g_i(x^*) = 0$, 那么称该不等式约束是 x^* 处的起作用约束。如果 $g_i(x^*) < 0$, 那么称该不等式约束是 x^* 处的不起作用约束。

定义21.2. 设 \mathbf{x}^* 满足 $\mathbf{h}(\mathbf{x}^*) = \mathbf{0}$, $\mathbf{g}(\mathbf{x}^*) \leq \mathbf{0}$, 设 $J(\mathbf{x}^*)$ 为起作用不等式约束下标集,

$$J(\mathbf{x}^*) = \{j : g_j(\mathbf{x}^*) = 0\}$$

如果： $\nabla h_i(\mathbf{x}^*), \nabla g_j(\mathbf{x}^*), i \leq j \leq m, j \in J(\mathbf{x}^*)$

是线性无关的，则称 \mathbf{x}^* 是一个正则点。

KKT条件：某个点是局部极小点所满足的一阶必要条件。

定理：KKT条件，设 $f, \mathbf{g}, \mathbf{h} \in C^1$, 设 \mathbf{x}^* 是问题 $\mathbf{h} = \mathbf{0}, \mathbf{g} \leq \mathbf{0}$ 的一个正则点和局部极小点，那么必然存在 $\lambda^* \in R^m$ 和 $u^* \in R^p$ 使得以下条件成立：

$$\mathbf{u}^* \geq \mathbf{0}$$

$$Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$$

$$\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$$

λ^* 为拉格朗日乘子向量, \mathbf{u}^* 是KKT乘子向量, 其元素分别成为拉格朗日乘子, KKT乘子。

充分条件

上面讲了局部极小的必要条件, 这里我们讨论局部极小的充分条件。然后我们就利用KKT条件去求不等式约束问题。

二阶充分必要条件；

定义如下矩阵：

$$L(\mathbf{x}, \lambda, \mathbf{u}) = F(\mathbf{x}) + \lambda \mathbf{H}(\mathbf{x}) + \mathbf{u} \mathbf{G}(\mathbf{x})$$

$F(\mathbf{x})$ 是在 \mathbf{x} 处的 Hessian 矩阵；

$$\lambda \mathbf{H}(\mathbf{x}) = \lambda_1 H_1(\mathbf{x}) + \dots + \lambda_m H_m(\mathbf{x})$$

$$\mathbf{u} \mathbf{G}(\mathbf{x}) = u_1 G_1(\mathbf{x}) + \dots + u_p G_p(\mathbf{x})$$

其中 $G_k(\mathbf{x})$ 是 g_k 处的 Hessian 矩阵。

起作用约束所定义曲面的切线空间： $T(\mathbf{x}^*) = \{y \in R^n : Dh(x^*)y = 0, Dg_j(x^*)y = 0, j \in J(x^*)\}$.

定理：二阶必要条件：如果 \mathbf{x}^* 是上面讨论的优化问题的极小点，那么存在 $\lambda^* \in R^m, \mathbf{u}^* \in R^p$ 使得：

1. $\mathbf{u}^* \geq 0,$
2. $Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$
3. $\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$

4. 对所有 $\mathbf{y} \in T(\mathbf{x}^*, \mathbf{u}^*), \mathbf{y} \neq 0$, 都有 $\mathbf{y}^T L(\mathbf{x}, \lambda, \mathbf{u}) \mathbf{y} > 0$

定理：二阶充分条件：

假设 $f, \mathbf{g}, \mathbf{h} \in C^2, \mathbf{x}^* \in R^n$ 是一个可行点，存在向量 $\lambda^* \in R^m, \mathbf{u}^* \in R^p$ 使得：

4. $\mathbf{u}^* \geq 0,$
5. $Df(\mathbf{x}^*) + \lambda^* Dh(\mathbf{x}^*) + \mathbf{u}^* Dg(\mathbf{x}^*) = \mathbf{0}^T$
6. $\mathbf{u}^{*T} \mathbf{g}(\mathbf{x}^*) = 0$

4. 对所有 $\mathbf{y} \in T(\mathbf{x}^*, \mathbf{u}^*), \mathbf{y} \neq 0$, 都有 $\mathbf{y}^T L(\mathbf{x}, \lambda, \mathbf{u}) \mathbf{y} > 0$

那么 \mathbf{x}^* 是优化问题

$$s.t. \quad \mathbf{h}(\mathbf{x}) = \mathbf{0}$$

$$s.t. \quad \mathbf{g}(\mathbf{x}) \leq \mathbf{0}$$

的严格局部极小点。

$$T(\mathbf{x}^*) = \{y \in R^n : Dh(x^*)y = 0, Dg_j(x^*)y = 0, j \in J(x^*)\}$$

其中： $J(\mathbf{x}^*, \mathbf{u}^*) = \{j : g_j(\mathbf{x}^*) = 0\}, \mathbf{u}^* > 0$

核函数方法

决策树与集成学习

决策树与集成学习

决策树是一种基本的回归与分类的方法。决策树由节点与边构成，节点分类内部节点与叶子节点；内部节点表示属性或者特征，叶子节点表示一个类。

决策树学习

训练集 $D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N))$ 其中 $x_i = (x_i^{(1)}, x_i^{(2)} \dots x_i^{(n)})$ 是 n 维变量， n 表示特征的数目， $y_i \in (1, 2 \dots K)$ 类标签。训练的本质是基于一定标准得到一组分类规则。我们需要得到一组与训练数据矛盾较小并且泛化能力也好的分类规则，其中泛化能力说的是在测试集上其错误率也小，错误率有不同的定义方式，可以是均方误差，也可以是似然函数，一般回归问题选择均方误差，分类问题选择似然函数，想想这是为什么？。

分类规则就是特征选择的过程，特征选择是基于一些可以量化的函数，一般基于信息熵增益最大或者信息熵增益率最大或者 Gini 系数下降最大的规则。因此，接下来我们要定义如下概念。

1. 信息熵
2. 条件熵
3. 信息增益
4. 信息增益率
5. Gini 系数

1. 信息熵

熵的概念来自于统计物理，描述微观系统的混乱程度，由 Rudolf Clausius 提出，

$$S = k_B \ln \Omega = -k_B \sum_i p_i \ln p_i$$

其中 k_B 是 Boltzmann constant, Ω 表示系统的可能状态数。

它表示的就是 $\ln(p_i)$ 的期望值。香农把这个概念引入信息学，定义了信息熵。

考虑一个只能取有限 n 个值的系统，值对应的就是状态，比如投硬币，只能取两个值，就是说只能有 2 个状态，因此 $n=2$ ，在统计物理中，状态对应的是系统的能级，也就是一个能级对应一个状态，系统处于不同能级的概率就是 p_i

$$P(X = x_i) = p_i, i = 1, 2..n$$

熵的定义如下： $H(x) = -\sum_i^{i=n} p_i \ln p_i$

若系统是确定的，则有一个 $p_i = 1$ ，其他的 $p_i = 0$ 等于 0，因此代入熵的公式可知，熵为 0。可以证明，对于 n 个状态的系统，系统的熵满足如下不等式：

熵的概念来自于统计物理，描述微观系统的混乱程度，由Rudolf Clausius提出。

$$S = k_B \ln \Omega = -k_B \sum_i p_i \ln p_i$$

其中 k_B 是Boltzmann constant, Ω 表示系统的可能状态数。

它表示的就是 $\ln(p_i)$ 的期望值。香农把这个概念引入信息学，定义了信息熵。

考虑一个只能取有限n个值的系统，值对应的就是状态，比如投硬币，只能取两个值，就是说只能有2个状态，因此n=2，在统计物理中，状态对应的是系统的能级，也就是一个能级对应一个状态，系统处于不同能级的概率就是 p_i

$$P(X = x_i) = p_i, i = 1, 2..n$$

熵的定义如下： $H(x) = - \sum_i^{i=n} p_i \ln p_i$

若系统是确定的，则有一个 $p_i = 1$ ，其他的 $p_i = 0$ 等于0，因此代入熵的公式可知，熵为0.可以证明，对于n个状态的系统，系统的熵满足如下不等式：

$$0 \leq H(x) = - \sum_i^{i=n} p_i \ln p_i \leq \ln n.$$

2. 条件熵

联合概率说的是两个及两个系统随机变量共同发生的几率问题，条件熵 $H(Y|X)$ 说的是随机变量X给定的情况下，随机变量Y的条件熵。

计算如下：

$$H(Y|X) = \sum_i^{i=n} p_i H(Y|X = x_i)$$

这里， $p_i = P(X = x_i), i = 1, 2..n$ 。

$H(Y|X = x_i)$ 计算与上面的信息熵一样，只是对于 $H(Y|X = x_i)$ ，我们只计算 $X = x_i$ 的那些样本的熵，因为我们会对所有X取不同值得样本进行求和，因此会遍历整个样本。

3. 信息增益

信息增益是基于以上两个概念的，是针对于特征而言的，特征A对训练数据集D的信息增益 $g(D, A)$ 定义成经验熵 $H(D)$ 与特征A给定的条件下，D的经验条件熵 $H(D|A)$ 之差。即：

$$g(D, A) = H(D) - H(D|A).$$

如果我们知道数据集D的信息熵计算，以及条件熵的计算，则信息增益的计算不难。

4. 信息增益率

假设样本D中有K个类, C_k 是第k类的集合, $|C_k|$ 是集合 C_k 的大小, 因此样本集合的基尼系数定义成:

$$Gini(p) = 1 - \sum_{i=1}^{i=K} \left(\frac{|C_i|}{|D|} \right)^2$$

基于不同的指标, 比如信息增益, 信息增益率, 基尼系数, 我们会得到不同的决策树生成算法, 依次是ID3, C4.5, CART。

关于Gini系数的讨论(一家之言)

- 考察Gini系数的图像、熵、分类误差率三者之间的关系
- 将 $f(x)=-\ln x$ 在 $x=1$ 处一阶展开, 忽略高阶无穷小, 得到 $f(x) \approx 1-x$

$$\begin{aligned} H(X) &= -\sum_{k=1}^K p_k \ln p_k \\ &\approx \sum_{k=1}^K p_k (1-p_k) \end{aligned}$$



信息熵与基尼系数的函数大致一致, 实际上在iris数据集中, 生成决策树时, 两者没有差别。

ID3算法

ID3算法基于信息增益最大, 代码实现如下, 它的缺点在于, 它倾向于选择取值很多的特征, 因为当特征能取很多值得时候, 此时系统的不确定度降低, 极端情况是, 特征能取N个不同的值, N个训练集的数量, 此时特征A条件熵为0. 为了避免这种情况, 而引入了C4.5算法。

假设特征A是信息增益最大的特征, 当特征A取离散值时, 假设特征A可以取K个不同值, 我们选择A作为特征之后会生出K个节点, 对于每个节点, 我们要重复上面的步骤, 继续寻找信息增益最大的特征, 只是, 这时候特征数目减小了1, 减小的这个特征就是A特征本身。当一个节点, 它的熵小于一定阈值的时候, 我们就不再分割这个节点而是把他当成一个叶子节点。我们可以使用递归来

关于Gini系数的讨论(一家之言)

□ 考察Gini系数的图像、熵、分类误差率三者之间的关系

■ 将 $f(x) = -\ln x$ 在 $x=1$ 处一阶展开，忽略高阶无穷小，得到 $f(x) \approx 1-x$

$$\begin{aligned} H(X) &= -\sum_{k=1}^K p_k \ln p_k \\ &\approx \sum_{k=1}^K p_k (1-p_k) \end{aligned}$$



信息熵与基尼系数的函数大致一致，实际上在iris数据集中，生成决策树时，两者没有差别。

ID3算法

ID3算法基于信息增益最大，代码实现如下，它的缺点在于，它倾向于选择取值很多的特征，因为当特征能取很多值得时候，此时系统的不确定度降低，极端情况是，特征能取N个不同的值，N个训练集的数量，此时特征A条件熵为0。为了避免这种情况，而引入了C4.5算法。

假设特征A是信息增益最大的特征，当特征A取离散值时，假设特征A可以取K个不同值，我们选择A作为特征之后会生出K个节点，对于每个节点，我们要重复上面的步骤，继续寻找信息增益最大的特征，只是，这时候特征数目减小了1，减小的这个特征就是A特征本身。当一个节点，它的熵小于一定阈值的时候，我们就不再分割这个节点而是把他当成一个叶子节点。我们可以使用递归来实现ID3算法。

对于特征值取连续的情况，我们通过计算特征值大于阈值与小于阈值的两部分熵之和来计算熵。算法实现的时候要注意的一点就是，计算熵时，不能让概率等于0，否则会出错。

```
import scipy
import numpy as np
from sklearn import tree
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

class DecisionTree(object):
```

```

        for n in range(1, sorted_data.shape[0]):
            threshold = (sorted_data[n-1, feature_index] + sorted_data[n, feature_index])/
2
            featue_result_data = np.zeros((sorted_data.shape[0], 2))
            featue_result_data[:, 0] = sorted_data[:, feature_index]
            featue_result_data[:, -1] = sorted_data[:, -1]
            condition_entropy = self.distribution(featue_result_data, n)
            if condition_entropy < best_entropy:
                best_threshold = threshold
                best_entropy = condition_entropy
                best_feature = feature_index
            print best_entropy, best_feature, best_threshold
        return best_entropy, best_feature, best_threshold

if __name__ == '__main__':
    DT = DecisionTree()
    iris = load_iris()
    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(iris.data, iris.target)
    data = iris.data
    target = iris.target
    iris_data = np.zeros((data.shape[0], data.shape[1] + 1))
    iris_data[:, 0:data.shape[1]] = data
    iris_data[:, -1] = target
    best_entropy, best_feature, best_threshold = DT.best_feature_threshold(iris_data)

```



C4.5

C4.5算法弥补了ID3算法的不足，通过使用信息增益率来作为特征的选择标准。

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

其中 $H(A) = \sum_{i=1}^K p_i \ln p_i$, K是特征A能取不同值得数目, p_i 是取不同值得概率。

模型的剪枝

为什么需要剪枝：提高泛化能力，

我们可以一步步的细分节点而使得系统的分类误差很小，但是这只是训练数据集上的结果，当我们把模型运用到测试集时，误差率会很高，这是因为模型过拟合你，我们可以通过对决策树进行剪枝来提高模型的泛化能力。

为了进行剪枝，我们得定义剪枝的标准，也就是定义损失函数，损失函数至少要包括两项，一个是在训练集上的误差，一个是模型的复杂程度。模型的复杂程度可以定义成与叶子节点正相关。模型在训练集上的误差率可以如下定义。

$$C(T) = \sum_{t=1}^{t=|T|} N_t H_t$$

```

        return best_entropy, best_feature, best_threshold

if __name__ == '__main__':
    DT = DecisionTree()
    iris = load_iris()
    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(iris.data, iris.target)
    data = iris.data
    target = iris.target
    iris_data = np.zeros((data.shape[0], data.shape[1] + 1))
    iris_data[:, 0:data.shape[1]] = data
    iris_data[:, -1] = target
    best_entropy, best_feature, best_threshold = DT.best_feature_threshold(iris_data)

```

< >

C4.5

C4.5算法弥补了ID3算法的不足，通过使用信息增益率来作为特征的选择标准。

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

其中 $H(A) = \sum_{i=1}^{i=K} p_i \ln p_i$, K是特征A能取不同值得数目, p_i 是取不同值得概率。

模型的剪枝

为什么需要剪枝:提高泛化能力,

我们可以一步步的细分节点而使得系统的分类误差很小,但是这只是训练数据集上的结果,当我们把模型运用到测试集时,误差率会很高,这是因为模型过拟合你,我们可以通过对决策树进行剪枝来提高模型的泛化能力。

为了进行剪枝,我们得定义剪枝的标准,也就是定义损失函数,损失函数至少要包括两项,一个是指模型在训练集上的误差,一个是模型的复杂程度。模型的复杂程度可以定义成与叶子节点正相关。模型在训练集上的误差率可以如下定义。

$$C(T) = \sum_{t=1}^{t=|T|} N_t H_t$$

其中 N_t 是叶子节点t上的样本点数目, H_t 是叶子节点t的熵。

叶子节点的熵定义为:

$$H_t = - \sum_{k=1}^{k=K} \frac{N_{tk}}{N_t} \ln \frac{N_{tk}}{N_t},$$

其中 $k \in (1, 2 \dots K)$ 是样本结果可以取不同值的数目,也就是标签有K类 N_t 是叶子节点t的样本数

目, N_{tk} 是叶子节点t中标签属于k类的数目。总体而言, $C(T)$ 刻画的是模型在训练集上的误差。结合这两部分,我们可以定义剪枝的损失函数:

$$C_\alpha(T) = C(T) + \alpha |T|$$

假设X和Y分别为输入与输出变量, 并且Y是连续变量, 给定训练数据集

$$D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N)).$$

假设已将输入空间划分为M个单元 R_1, R_2, \dots, R_M , 并且每个单元 R_m 上有一个固定的输出值 c_m , 于是回归树模型可以表示为:

$$f(x) = \sum_{m=1}^{m=M} c_m I(x \in R_m)$$

当输入空间的划分确定时, 可以用平方误差

$$\sum_{x_i \in R_m} (y_i - f(x_i))^2$$

来表示回归树对于训练数据的预测误差, 用平方误差最小的准则来求解每个单元上的最优输出值。

可以求得

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m).$$

问题是, 怎么对输入的空间进行划分?

回归树可以处理离散特征与连续特征, 对于连续特征, 若这里按第j个特征的取值s进行划分, 切分后的两个区域分别为:

$$R_1(j, s) = (x_i | x_i^j \leq s)$$

$$R_2(j, s) = (x_i | x_i^j > s)$$

若为离散特征, 则找到第j个特征下的取值s:

$$R_1(j, s) = (x_i | x_i^j = s)$$

$$R_2(j, s) = (x_i | x_i^j \neq s)$$

分别计算 R_1, R_2 的类别估计 c_1, c_2 , 然后计算按照(j,s)切分后的损失:

$$\min_{j, s} [\sum_{x_i \in R_1} (y_i - c_1)^2 + \sum_{x_i \in R_2} (y_i - c_2)^2], \text{ 找到是的损失最小的(j,s)对}$$

即可, 也就是说找到最优特征 j^* , 并找到最优特征的分割值 s^* 。递归执行(j,s)的选择过程, 知道满足停止条件为止。递归的意思是, 对分割出的两个区域 R_1, R_2 分别进行如上的步骤, 再分割下去。

总结:

回归树算法:

输入: 训练集 $D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N)).$

输出: 回归树T

1. 求解选择的切分特征j与切分特征取值s, j将训练集D划分成两部分, R_1, R_2 ,

$$R_1(j, s) = (x_i | x_i^j \leq s)$$

$$R_2(j, s) = (x_i | x_i^j > s)$$

其中:

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m)。$$

问题是，怎么对输入的空间进行划分？

回归树可以处理离散特征与连续特征，对于连续特征，若这里按第j个特征的取值s进行划分，切分后的两个区域分别为：

$$R_1(j, s) = (x_i | x_i^j \leq s)$$

$$R_2(j, s) = (x_i | x_i^j > s)$$

若为离散特征，则找到第j个特征下的取值s：

$$R_1(j, s) = (x_i | x_i^j = s)$$

$$R_2(j, s) = (x_i | x_i^j \neq s)$$

分别计算 R_1, R_2 的类别估计 c_1, c_2 ，然后计算按照(j,s)切分后的损失：

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

即可，也就是说找到最优特征 j^* ，并找到最优特征的分割值 s^* 。递归执行(j,s)的选择过程，知道满足停止条件为止。递归的意思是，对分割出的两个区域 R_1, R_2 分别进行如上的步骤，再分割下去。

总结：

回归树算法：

输入：训练集 $D = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N))$ 。

输出：回归树T

1.求解选择的切分特征j与切分特征取值s，j将训练集D划分成两部分， R_1, R_2 ，

$$R_1(j, s) = (x_i | x_i^j \leq s)$$

$$R_2(j, s) = (x_i | x_i^j > s)$$

其中：

$$c_1 = \frac{1}{N_1} \sum_{x_i \in R_1} y_i$$

$$c_2 = \frac{1}{N_2} \sum_{x_i \in R_2} y_i$$

2.遍历所有的可能解(j,s)，找到最优的 (j^*, s^*) ，最优解使得如下损失函数最小，

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

按照最优特征 (j^*, s^*) 来切分即可

3.递归调用1., 2.步骤，直到满足停止条件

4.将输入空间划分为M个区域 R_1, R_2, \dots, R_M ，返回决策树T

$$f(x) = \sum_{m=1}^{m=M} \hat{c}_m I(x \in R_m)$$

对于固定的 α , 存在唯一的最有子树 $C_\alpha(T)$.

Breiman等人证明, 可以用递归的方法对树进行剪枝, 将 α 从0开始增大, $0 \leq \alpha_1 \leq \alpha_2 \leq \dots \leq \alpha_n$, 得到一系列最优子树系列 (T_0, T_1, T_n) , 序列中的子树是嵌套的(?)。

具体的, 从整棵树 T_0 开始剪枝, 对 T_0 的任意内部节点 t , 以为单节点书的损失函数是(也就是把 t 下面的所有节点减掉之后的损失函数):

$$\blacksquare C_\alpha(t) = C(t) + \alpha$$

以为根结点的字数 T_t 的损失函数是(不进行剪枝时的损失函数):

$$\blacksquare C_\alpha(T_t) = C(T_t) + \alpha|T_t|$$

当\alpha 小于某个值时, 不剪枝会使得整体的损失函数较小, 也就是:

$$\blacksquare C_\alpha(T_t) < C_\alpha(t)$$

当\alpha 增大时, 在某一\alpha有:

$$\blacksquare C_\alpha(T_t) = C_\alpha(t)$$

当\alpha再增大时, 不等式反向, 只要 $\alpha = \frac{C(t)-C(T_t)}{|T_t|-1}$, T_t 与t有相同的损失函数。

为此, 对 T_0 中每一个内部节点t, 计算:

$$\blacksquare g(t) = \frac{C(t)-C(T_t)}{|T_t|-1}$$

它表示剪枝之后整体损失函数减小的程度, 在 T_0 中减去 $g(t)$ 最小的 T_t , 将得到的子树作为 T_1 , 同时将最小的 $g(t)$ 设为 α_1 , T_1 为区间 $[\alpha_1, \alpha_2]$ 的最优子树。

如此剪枝下去, 在这一过程中, 不断的增加\alpha的取值, 产生新的区间, 最终会的到一组决策树 (T_0, T_1, T_n) , 对应于椅子确定的权衡参数 (a_0, a_1, a_n) , 通过验证集合中每颗树的总体误差, 也就得到了最终的最优决策树 T^* .

T是整颗决策树吗?

输入:CART 生成树 T_0

输出:剪枝后的最优树 T^*

1) 设 $k=0$, $T = T_0$, $a = +\infty$

3) 自下而上的对内部节点 t 计算 :

当\alpha 小于某个值时, 不剪枝会使得整体的损失函数较小, 也就是:

$$\blacksquare C_\alpha(T_t) < C_\alpha(t)$$

当\alpha增大时, 在某一\alpha有:

$$\blacksquare C_\alpha(T_t) = C_\alpha(t)$$

当\alpha再增大时, 不等式反向, 只要\alpha = \frac{C(t)-C(T_t)}{|T_t|-1}, T_t与t有相同的损失函数。

为此, 对T_0中每一个内部节点t,计算:

$$\blacksquare g(t) = \frac{C(t)-C(T_t)}{|T_t|-1}$$

它表示剪枝之后整体损失函数减小的程度, 在T_0中减去g(t)最小的T_t,将得到的子树作为T_1,同时

将最小的g(t)设为\alpha_1, T_1为区间[\alpha_1, \alpha_2)的最优子树。

如此剪枝下去, 在这一过程中, 不断的增加\alpha的取值, 产生新的区间, 最终会的到一组决策树

(T_0, T_1, T_n), 对应于椅子确定的权衡参数(a_0, a_1, a_n), 通过验证集合中每棵树的总体误差, 也就得到了最终的最优决策树T*.

T是整颗决策树吗?

输入:CART 生成树 T_0

输出:剪枝后的最优树 T*

1) 设 k=0 , T = T_0 , a = +\infty

3) 自下而上的对内部节点 t 计算 :

$$\blacksquare g(t) = \frac{C(t)-C(T_t)}{|T_t|-1}$$

a=\min(a, g(t))

4) 自上而下的访问内部节点 t , 对最小的 g(t)=a 进行剪枝, 并对叶节点 tt 以多数表决形式决定其类别, 得到树 TT

5) k=k+1, a_k = a, T_k = T

6) 如果 T 为非单节点树, 回到 4).

7) 对于产生的子树序列 (T_0, T_1, T_n) 分别计算损失, 得到最优子树T*并返回.

使得这两部分的误差和最小，这就是该特征的最佳切分点，再在该数据集中的所有特征中这个误差，找到最佳的切分特征，最后得到最佳特征以及最佳特征的切分点。基于这两个值，把数据集分成两部分，再对这两部分分别进行如上的操作（求最佳特征与该特征下的最佳切分点）

1. 决策树之 CART . ↵

额的# 集成学习

集成学习就是组合一系列的弱分类器生成强分类器。组合的弱分类器之间的关系有两种：

一种是彼此间相互依赖,一系列学习器之间需要串行生成,代表算法是Boosting系列算法,代表算法是AdaBoost与GBDT.

一种是学习器彼此间无关联,一些列算法可以并行的生成,代表算法是Bagging(Bootstrap Aggregating)和随机森林(Random Forest)系列。随机森林是时Bagging的进行版,随机表现在两个方面,一方面是取样本点是随机的,另一方面,选择特征也是随机的(比如总共有N个特征,随机选取小于N的K个特征)

还有一种是两者的结合物:stacking

PAC, 弱可学习, 强可学习

PAC(Probably approximately correct)

强可学习:如果存在一个多项式的学习算法学习呀,学习的正确率很高

弱可学习:如果存在一个多项式的学习算法学习呀,学习的正确率仅比随机猜测略好
在PAC学习框架下,一个概念是强可学习的充分必要条件是这个概念是弱可学习的。

集成学习的理论基础

我们考虑多个不相干分类器叠加处理二分类的问题, $y \in (-1, +1)$ 和真实的函数 f ,假设基分类器的错误率都是 ϵ ,即对每个分类器都有

$$\bullet P(h_i(x) \neq f(x)) = \epsilon$$

假设集成通过简单的投票发结合T个基分类器,若半数的基分类器正确,则集成分类正确。

$$\{\% \text{math}\%\}\text{kern}{10 \text{em}}H(x) = \text{sign}(\sum_{i=1}^T h_i(x))\{\% \text{endmath}\}$$

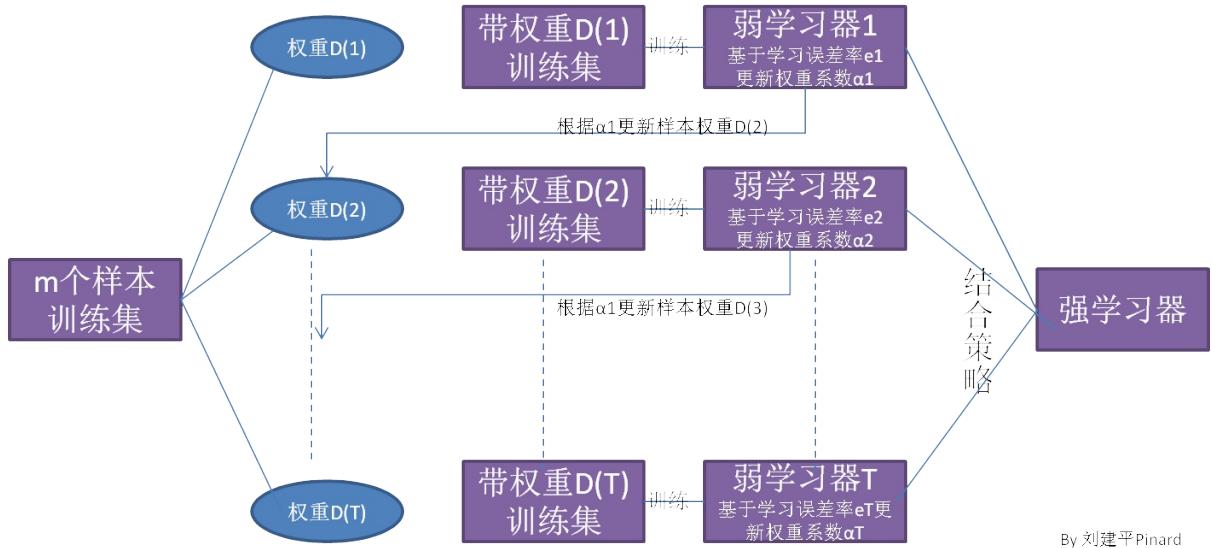
价格基分类器之间的错误率是相互独立的,则由Hoeffding不等式可知,集成的错误率为:

$$\bullet P(H(x) \neq f(x)) = \sum_{i=0}^{[T/2]} \binom{T}{k} (1 - \epsilon)^k \epsilon^{T-k} \leq \exp(-\frac{1}{2} T (1 - 2\epsilon)^2)$$

上式表明,随着集成中个体分类器数目T的增大,集成的错误率将指数下降,最终趋向于0.

Boosting:

Boosting系列方法的原理图如下:



By 刘建平 Pinard

AdaBoost

Boosting系列算法的代表就是AdaBoost。

算法如下：

输入：训练数据集 $T = ((x_1, y_1), (x_2, y_2) \dots (x_N, y_N))$, 其中 $x_i \in X \subseteq R^n, y \in (-1, +1)$

输出：最终的分类器 $G(x)$

(1) 初始化训练数据集的权重分布

$$D_1 = (w_{11}, \dots, w_{1i}, \dots, w_{1N}),$$

$$w_{1i} = \frac{1}{N}, i = 1, 2, \dots, N$$

(2) 对 $m=1, 2, \dots, M$

使用具有权值分布 D_m 的训练数据集进行训练，得到基分类器

$$G_m(x) : \chi \in (-1, +1)$$

(b) 计算 $G_m(x)$ 在训练集上的分类误差率：

$$e_m = P(G_m(x_i) \neq y_i) = \sum_{i=1}^N w_{mi} I(G_m(x_i) \neq y_i)$$

(c) 计算 $G_m(x)$ 的系数

$$\alpha_m = \frac{1}{2} \log \frac{1-e_m}{e_m}$$

这里用的是自然对数。

(d) 更新训练数据集的权值分布：

$$D_1 = (w_{m+1,1}, \dots, w_{m+1,i}, \dots, w_{m+1,N}),$$

$$w_{m+1,i} = \frac{w_{mi}}{Z_m} \exp(-\alpha_m y_i G_m(x_i)), i = 1, 2, \dots, N$$

这里， Z_m 是归一化因子：

$$\bullet Z_m = \sum_{i=1}^N w_{mi} \exp(-\alpha_m y_i G_m(x_i))$$

它使 D_{m+1} 成为一个概率分布。

(3) 构建基本分类器的线性组合：

$$\bullet f(x) = \sum_{i=1}^N \alpha_m G_m(x)$$

得到最终分类器

$$\bullet G(x) = \text{sign}(f(x)) = \text{sign}\left(\sum_{i=1}^N \alpha_m G_m(x)\right)$$

由 α_m 的定义可以知道，第 m 个分类器的误差率越小，则 α_m 的值越大，因此第 m 个分类器在总的分类器中占的比重越大。因此，AdaBoost 会加大那些准确率很高的分类器的权重。

同时，在第 $m+1$ 个分类器求解时，对于上一轮被分错的样本的权值会变大。

不改变所给的训练数据，而不断的改变训练数据权值的分布，使得训练数据在基本分类器的学习中起不同的作用，这是 AdaBoost 的一个特点。

可以参考《统计机器学习》8.1 节的例子来加深理解。

AdaBoost 算法的训练误差分析

定理：(AdaBoost 的训练误差界)

AdaBoost 算法最终分类器的训练误差界为：

$$\bullet \frac{1}{N} \sum_{i=1}^N I(G_m(x_i) \neq y_i) \leq \frac{1}{N} \sum_{i=1}^N \exp(-y_i f(x_i)) = \prod_m Z_m$$

定理(二分类问题 AdaBoost 的训练误差界)

$$\bullet \prod_m Z_m = \prod_{m=1}^M [2 \sqrt{e_m(1-e_m)}] = \prod_{m=1}^M \sqrt{1 - 4\gamma_m^2} \leq \exp(-2 \sum_{m=1}^M \gamma_m^2)$$

其中： $\gamma_m = \frac{1}{2} - e_m$

梯度提升

提升树是以分类树或者回归树为基本分类器的提升方法，提升树被认为是统计学习中性能最好的方法之一。

提升树实际采用加法模型(即基函数的线性组合)与前向分步算法。以决策树为基函数的提升方法称为提升树。对分类问题决策树是二叉分类树，对回归问题决策树是二叉回归树。

决策树桩(decision stump)：可以看成是一个根节点直接连接两个叶节点的简单决策树。提升树模型可以表示为决策树的加法模型。

$$\bullet f_M(x) = \sum_{m=1}^M T(x; \Theta_m)$$

其中： $T(x; \Theta_m)$ 表示决策树； Θ_m 为决策树的参数； M 为树的个数。

梯度提升算法：

提升树算法采用前向分步算法。首先缺点初始提升树 $f_0(x) = 0$, 第m步的模型是：

$$\blacksquare f_m(x) = f_{m-1}(x) + T(x; \Theta_m)$$

其中, $f_{m-1}(x)$ 为当前模型, 通过经验风险极小化来确定下一刻决策树的参数 Θ_m

$$\blacksquare \hat{\Theta}_m = \operatorname{argmin}_{\Theta_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)).$$

对于一个训练数据集 $T = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$, $x_i \in \chi \in R^N$, χ 为输入控件, $y_i \in R$, 为输出空间。如果将输入控件 χ 划分为J个互不相交的区域 R_1, R_2, \dots, R_J , 并且在每个区域上确定输出的常量 c_j , 那么树可以表示为：

$$\blacksquare T(x; \theta) = \sum_{j=1}^J c_j I(x \in R_j).$$

其中, 参数 $\Theta = ((R_1, c_1), (R_2, c_2), \dots, (R_J, c_J))$ 表示树的区域划分和各区域上的常数。J是回归树的发杂都, 即叶节点个数。

回归为题提升树使用以下前向分步算法：

$$\blacksquare f_0(x) = 0$$

$$\blacksquare f_m(x) = f_{m-1}(x) + T(x; \Theta_m), m = 1, 2, \dots, M$$

$$\blacksquare f_m(x) = \sum_{m=1}^M T(x; \Theta_m)$$

在前向分步算法的第m步, 给定当前模型 $f_{m-1}(x)$, 需求解：

$$\blacksquare \hat{\Theta}_m = \operatorname{argmin}_{\Theta_m} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + T(x_i; \Theta_m)).$$

得到 $\hat{\Theta}_m$, 即第m颗树的参数。

采用平方误差损失函数时：

$$\blacksquare L(y, f(x)) = (y - f(x))^2$$

其损失变成：

$$\blacksquare L(y, f_{m-1}(x) + T(x; \Theta_m)) = (y - f_{m-1}(x) - T(x; \Theta_m))^2 = [r - T(x; \Theta_m)]^2$$

这里,

$$\blacksquare r = y - f_{m-1}(x)$$

是当前模型拟合数据的残差。所以, 对回归问题的提升树算法来说, 只需要简单地你和当前模型的残差。

GBDT

当损失函数是平方损失和指数损失函数时，每一步优化是很简单的，但是对于一般损失函数而言，往往每一步优化并不容易。针对这一问题，Freidman提出了梯度提升(gradient boosting)算法，这是利用最速下降法的近似方法，其关键是利用损失函数的负梯度在当前模型的值。

$$\blacksquare - \left[\frac{\partial L(y, f(x_i))}{\partial f(x_i)} \right]_{f(x)=f_{m-1}(x)}$$

作为回归问题提升算法中的残差的近似值，拟合一个回归树。

梯度提升树算法

输入：训练数据集 $T = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$, $x_i \in \chi \in R^N$, χ 为输入, $y_i \in R$, 损失函数为 $L(y, f(x))$:

输出：回归树 $\hat{f}(x)$.

(1) 初始化

$$\blacksquare f_0(x) = \operatorname{argmin}_c \sum_{i=1}^N L(y_i, c).$$

(2) 对 $m=1, 2, \dots, M$

(a) 对 $i=1, 2, \dots, N$, 计算

$$\blacksquare r_{mi} = - \left[\frac{\partial L(y, f(x_i))}{\partial f(x_i)} \right]_{f(x)=f_{m-1}(x)}$$

(b) 对 r_{mi} 拟合一个回归树，得到第 m 课树叶节点区域 $R_{mj}, j = 1, 2, \dots, J$

(c) 对 $j=1, 2, \dots, J$, 计算

$$\blacksquare c_{mj} = \operatorname{argmin}_c \sum_{x_i \in R_{mj}} L(y_i, f_{m-1}(x_i) + c).$$

$$\blacksquare \text{(d) 更新 } f_m(x) = f_{m-1}(x) + \sum_{j=1}^J c_{mj} I(x \in R_{mj}), m = 1, 2, \dots, M$$

(3) 得到回归树

$$\blacksquare \hat{f}(x) = f_M(x) = \sum_{m=1}^M \sum_{j=1}^J c_{mj} I(x \in R_{mj})$$

函数空间的数值优化：

XGBoost¹

模型复杂度惩罚是XGBoost相对于MART的提升。

$$\blacksquare = \sum_{m=1}^M [\gamma T_m + \frac{1}{2} \lambda ||w_m||_2^2 + \alpha ||w_m||_1]$$

MART includes row subsampling, while XGBoost includes both row and column subsampling

Newton boosting

Input: Data set D, A loss function L, A base learner L_Φ , the number of iterations M. The learning rate η

1. Initialize $\hat{f}^{(0)}(x) = \hat{f}_{(0)}(x) = \hat{\theta}_0 = \operatorname{argmin}_{\theta} \sum_{i=1}^n L(y_i, \theta)$;
2. for m = 1,2,...,M do
3. $\hat{g}_m(x_i) = [\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f^{(m-1)}(x)}$
4. $\hat{h}_m(x_i) = [\frac{\partial^2 L(y, f(x_i))}{\partial f(x_i)^2}]_{f(x)=f^{(m-1)}(x)}$
5. $\hat{\phi}_m = \operatorname{argmin}_{\phi \in \Phi} \sum_{i=1}^n \frac{1}{2} \hat{h}_m(x_i) [(-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}) - \phi(x_i)]^2$
6. $\hat{f}_m(x) = \eta \hat{\phi}_m(x);$
7. $\hat{f}^m(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x);$
8. end

Output: $\hat{f}(x) = \hat{g}^{(M)}(x) = \sum_{m=1}^M \hat{f}_m(x)$

对于没有惩罚项的Netwon tree boosting(NTB),每一次迭代都最小化如下损失函数:

$$\blacksquare J_m(\phi_m) = \sum_{i=1}^n L(y_i, \hat{f}^{(m-1)}(x_i) + \phi_m(x_i))$$

对于梯度提升算法, 基函数是如下的树:

$$\blacksquare \phi_m(x) = \sum_{j=1}^T w_{jm} I(x \in R_{jm})$$

这里的T是第m颗树叶子节点的个数, w_{jm} 是第m颗树中第j个叶子节点的权重。

对上式进行二阶泰勒展开,并忽略掉常数项可以得到:

$$\blacksquare J_m(\phi_m) = \sum_{i=1}^n [\hat{g}_m(x_i) \phi_m(x_i) + \frac{1}{2} \hat{h}_m(x_i) \phi_m(x_i)^2]$$

代入:

$$\begin{aligned} \blacksquare \phi_m(x) &= \sum_{j=1}^T w_{jm} I(x \in R_{jm}) \\ \blacksquare J_m(\phi_m) &= \sum_{i=1}^n [\hat{g}_m(x_i) \sum_{j=1}^T w_{jm} I(x \in R_{jm}) + \frac{1}{2} \hat{h}_m(x_i) (\sum_{j=1}^T w_{jm} I(x \in R_{jm}))^2] \end{aligned}$$

由于每个样本点只能属于一个叶子节点, 因此:

$$\blacksquare I(x \in R_{jm}) I(x \in R_{im}) = \delta_{ij}$$

因此上面可以简化成:

$$\begin{aligned} \blacksquare J_m(\phi_m) &= \sum_{i=1}^n [\hat{g}_m(x_i) \sum_{j=1}^T w_{jm} I(x \in R_{jm}) + \frac{1}{2} \hat{h}_m(x_i) \sum_{j=1}^T w_{jm}^2 I(x \in R_{jm})] \\ \blacksquare &= \sum_{j=1}^T \sum_{i \in I_{jm}} [\hat{g}_m(x_i) w_{jm} + \frac{1}{2} \hat{h}_m(x_i) w_{jm}^2] \end{aligned}$$

定义:

$$\blacksquare G_{jm} = \sum_{i \in I_{jm}} \hat{g}_m(x_i)$$

$$\boxed{H_{jm} = \sum_{i \in I_{jm}} \hat{h}_m(x_i)}$$

因此, 可以把cost function写成:

$$\begin{aligned} J_m(\phi_m) &= \sum_{j=1}^T [G_{jm}w_{jm} + \frac{1}{2}H_{jm}w_{jm}^2] \\ &\geq -\sum_{j=1}^T \frac{1}{2} \frac{G_{jm}^2}{H_{jm}} \end{aligned}$$

成立条件是:

$$w_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, 2, \dots, T$$

为了寻找最佳分裂点j, 也就是最大化如下的Gain:

$$\boxed{\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}} \right]}$$

总结起来:Newton tree boosting算法如下:

Input: Data set D, A loss function L, A base learner L_Φ , the number of iterations M. The learning rate η

1. Initialize $\hat{f}^{(0)}(x) = \hat{f}_{(0)}(x) = \hat{\theta}_0 = \operatorname{argmin}_\theta \sum_{i=1}^n L(y_i, \theta);$
2. for m = 1,2,...,M do
3. $\hat{g}_m(x_i) = \left[\frac{\partial L(y, f(x_i))}{\partial f(x_i)} \right]_{f(x)=f^{(m-1)}(x)}$
4. $\hat{h}_m(x_i) = \left[\frac{\partial^2 L(y, f(x_i))}{\partial f(x_i)^2} \right]_{f(x)=f^{(m-1)}(x)}$
5. Determin the structure $\hat{R}_{jm}, j = 1, \dots, T$ by selecting splits which maximize

$$\boxed{\text{Gain} = \frac{1}{2} \left[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}} \right]}$$

6. Determine the leaf weights $w_{jm}, j = 1, \dots, T$ for the learnt structure by

$$\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, 2, \dots, T$$

$$7. \hat{f}_m(x) = \eta \sum_{j=1}^T \hat{w}_{jm} I(x \in \hat{R}_{jm});$$

$$8. \hat{f}^m(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x);$$

9. end

$$\text{Output: } \hat{f}(x) = \hat{g}^{(M)}(x) = \sum_{m=1}^M \hat{f}_m(x)$$

Bagging

Bagging系列方法的原理图如下：



Bagging的本质思想是平均一个个多noisy(variance大)但是近似无偏的模型, 因此可以减少variance。树是bagging理想的候选者, 因为他们能抓住数据中复杂的相互作用结构。在大多数问题中, boosting相对于bagging有压倒性优势, 成为更好的选择。

Boosting通过弱学习者的时间演化, 成员投一个带权重的票。

$$\text{Regression: } \hat{f}_{rf}^B(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

因为从bagging中生成的树是id(identically distributed), B颗树的期望与单棵树的期望一样, 如果输之间是IID(independent identically distributed), 每棵树的variance是 σ^2 , 则B颗树的variance是 $\frac{\sigma^2}{B}$, 假设树之间的关联系数是c, 则B颗树的variance是:

$$c\sigma^2 + \frac{1-c}{B}\sigma^2$$

如果c=1, 就回到iid情况, 如果c ≠ 0, 则上面的第一项不会随着B增大而减小, 只有第二项会随着B增大而减小, 因此我们要尽量使得树之间的关联系数c趋于0. 这可以通过随机化来实现, 对样本和对特征这两方面随机化。

问题: 怎么构造具有负关联系数的系统?

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}^*(x)$$

当B趋于无穷时, 上述表达式就是bagging 估计的一个Monte Carlo估计。

Random Forest

参数推荐:

对于分类问题, 默认m是 \sqrt{p} , 最小的节点数是1.

对于回归问题, 默认的m是 $p/3$, 最小的节点尺寸是5.

OOB:out of box, bagging中有 $(1 - \frac{1}{N})^N = 1/e$ 的样本没有被选中, 可以用来做验证, 因此不需要交叉验证。当OOB误差稳定后, 训练也就可以结束了。

贝叶斯方法

朴素贝叶斯

朴素贝叶斯是基于特征条件独立假设。输入变量中所有特征之间是独立的因此，联合条件概率是各个条件概率的乘积。

输入： $X \in R^n$

输出： $Y \in (1, 2, \dots, K)$

条件概率： $P(X = x|Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)}|Y = C_k), k=1,2,\dots,K$

条件独立假设：

$$P(X = x|Y = y) = P(X = x|Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, \dots, X^{(n)} = x^{(n)}|Y = C_k) \\ = \prod_{j=1}^n P(X^{(j)} = x^{(j)}|Y = C_k)$$

在上面的公式中，先验概率计算如下：

$$P(Y = C_k) = \frac{\sum_{i=1}^N I(y_i=c_k)}{N}, k = 1, 2, \dots, K$$

条件概率计算

假设第 j 个特征 $x^{(j)}$ 的可能取值集合是 $(a_{j1}, a_{j2}, \dots, a_{js_j})$ ，条件概率计算如下：

$$P(X^{(j)} = a_{jl}|Y = C_k) = \frac{\sum_{i=1}^N I(x_i^{(j)} = a_{jl}, y_i = c_k)}{\sum_{i=1}^N I(y_i = c_k)}$$

$j = 1, 2, \dots, n; l = 1, 2, \dots, S_j; k = 1, 2, \dots, K$

后验概率计算

基于贝叶斯定理计算后验概率：

$$P(Y = C_k | X = x) = \frac{P(X=x|Y=C_k)P(Y=C_k)}{\sum_k P(X=x|Y=C_k)P(Y=C_k)}$$

$$\text{把条件独立假设代入可以得到: } P(Y = C_k | X = x) = \frac{P(Y=C_k) \prod_{j=1}^n P(X^{(j)}=x^{(j)}|Y=C_k)}{\sum_k P(Y=C_k) \prod_{j=1}^n P(X^{(j)}=x^{(j)}|Y=C_k)}$$

因为分母对不同的类别k都是一样的，因此只需要求分子就可以了。计算不同的类别k对应的值，分类结果对应概率最大的。

因此朴素贝叶斯可以表示为：

$$y = f(x) = \operatorname{argmax}_{c_k} P(Y = C_k) \prod_{j=1}^n P(X^{(j)}=x^{(j)}|Y=C_k)$$

Laplace Smoothing

在实际计算条件概率的时候，或出现0的情况，这会影响后面的后验概率的计算。一般通过 Laplace Smoothing 来处理。

$$P(X^{(j)} = a_{jl} | Y = C_k) = \frac{\sum_{i=1}^N I(x_i^{(j)}=a_{jl}, y_i=c_k) + \lambda}{\sum_{i=1}^N I(y_i=c_k) + s_j \lambda}$$

$$\lambda > 0, j = 1, 2, \dots, n; l = 1, 2, \dots, S_j; k = 1, 2, \dots, K$$

基本上到此，朴素贝叶斯的介绍也就完结了，朴素贝叶斯不需要计算任何参数，计算简单，缺点是分类性能不一定高。

极大似然估计

极大似然估计是试图在所有的模型参数可能取值中，找到一个能使得数据出现的可能性最大的值。

贝叶斯网络

逻辑回归

在这里我们以多项逻辑斯蒂回归为例，讨论怎么用最大化似然函数来求解这一类问题。

假设离散性随机变量Y的取值是(1,2,...,K),输入变量 $\mathbf{X} \in R^n$

则，模型如下：

$$\begin{aligned}\blacksquare P(Y = k|x) &= \frac{\exp(\mathbf{w}_k \cdot \mathbf{x})}{\sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}, k = 1, 2, \dots, K - 1 \\ \blacksquare P(Y = K|x) &= \frac{1}{1 + \sum_{k=1}^{K-1} \exp(\mathbf{w}_k \cdot \mathbf{x})}\end{aligned}$$

这里 $\mathbf{x} \in \mathbf{R}^{n+1}; \mathbf{w}_k \in \mathbf{R}^{n+1}$.

因此，总的似然函数可以表示为：

$$\blacksquare L(\mathbf{W}) = \prod_{n=1}^N P(Y = Y_n | X = X_n)$$
 这样，我们需要求的参数是 $W = (\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_{n+1})$.

最简单的就是二项逻辑斯蒂回归，在这里 $K=2$: 似然函数可以写成如下简单的形式：

$$\blacksquare L(\mathbf{W}) = \prod_{n=1}^N [\pi(x_i)^{y_i}][1 - \pi(x_i)]^{1-y_i}$$

其中： $\blacksquare P(Y = 1|x) = \pi(x) = \frac{\exp(\mathbf{w} \cdot \mathbf{x})}{1 + \exp(\mathbf{w} \cdot \mathbf{x})}$
 $\blacksquare P(Y = 0|x) = 1 - \pi(x)$

当 $y_i = 0$ 时：

$$\blacksquare [\pi(x_i)^{y_i}][1 - \pi(x_i)]^{1-y_i} = 1 - \pi(x_i)$$
 当 $y_i = 1$ 时：

$$\blacksquare [\pi(x_i)^{y_i}][1 - \pi(x_i)]^{1-y_i} = \pi(x_i)$$

实际上分别是 $y_i = 0, y_i = 1$ 发生的概率。

一般连乘形式的似然函数容易发生值溢出，故而一般求对数似然函数：

$$\begin{aligned}\blacksquare L(\mathbf{W}) &= \sum_{n=1}^N [y_i \log \pi(x_i) + (1 - y_i) \log[1 - \pi(x_i)]] \\ &= \sum_{n=1}^N [y_i(w \cdot x_i) + \log(1 + \exp(w \cdot x_i))]\end{aligned}$$

求 $L(\mathbf{W})$ 的极大值，得到 \mathbf{w} 的估计值。

问题转化为以对数似然函数为目标函数的最优化问题，一般采用梯度下降法或者拟牛顿法求解。
(为啥不使用牛顿法？因为求Hessian矩阵比较麻烦，而拟牛顿法只需要构造Hessian阵就可以了，用迭代法求)

最大熵模型

最大熵原理

最大熵原理是概率模型学习的一个准则，最大熵原理认为，学习概率模型时，在所有可能的概率模型（分布）中，熵最大的模型是最好的模型。通常用约束条件来确定概率模型的集合。所以，最大熵原理也可以表述为在满足约束条件的模型集合中，选取熵最大的模型。

怎么建立起最大熵原理与最大似然原理之间的联系？

最大熵模型如下：

对于给定的训练数据集： $T = ((x_1, y_1), (x_2, y_2), \dots, (x_N, y_N))$

以及特征函数 $f_i(\mathbf{x}, \mathbf{y})$, $i = 1, 2, \dots, n$, 最大熵模型的学习等价于约束最优化问题：

$$\begin{aligned} & \max_{P \in C} H(P) = \sum_{x,y} \hat{P}(x) P(y|x) \log P(y|x) \\ & \text{s.t. } E_P(f_i) = E_{\hat{P}}(f_i), i = 1, 2, \dots, n \quad \sum_y P(y|x) = 1 \end{aligned}$$

可以转化为求极小问题：

$$\begin{aligned} & \min_{P \in C} H(P) = - \sum_{x,y} \hat{P}(x) P(y|x) \log P(y|x) \\ & \text{s.t. } E_P(f_i) - E_{\hat{P}}(f_i) = 0, i = 1, 2, \dots, n \quad \sum_y P(y|x) = 1 \end{aligned}$$

一般，带约束的优化问题可以通过引入拉格朗日乘子与KKT乘子来求解，也就是把原始问题转化为对偶问题来求解。在满足KKT条件时，对偶问题与原始问题等价：

首先引入拉格朗日乘子： w_0, w_1, \dots, w_n , 定义拉格朗日函数 $L(P, \mathbf{w})$ ；

$$\begin{aligned} L(P, \mathbf{w}) &= -H(P) + w_0(1 - \sum_y P(y|x)) + \sum_{i=1}^n w_i(E_P(f_i) - E_{\hat{P}}(f_i)) \\ &= - \sum_{x,y} \hat{P}(x) P(y|x) \log P(y|x) + w_0(1 - \sum_y P(y|x)) \\ &\quad + \sum_{i=1}^n w_i (\sum_{x,y} \hat{P}(x, y) f_i(x, y) - \sum_{x,y} \hat{P}(x) P(y|x) f_i(x, y)) \end{aligned}$$

最优化的原始问题是：

$$\min_{P \in C} \max_{\mathbf{w}} L(P, \mathbf{w})$$

最优化的对偶问题是：

$$\max_{\mathbf{w}} \min_{P \in C} L(P, \mathbf{w})$$

由于拉格朗日函数 $L(P, \mathbf{w})$ 是 P 的凸函数，因此原始问题与对偶问题的解释等价的。（为什么？，用KKT条件能解析吗？）

令： $\Psi(\mathbf{w}) = \min_{P \in C} L(P, \mathbf{w})$

$\Psi(\mathbf{w})$ 称为对偶函数。同时，将其解记为：

$$\blacksquare P_{\mathbf{w}} = \arg \min_{P \in C} L(P, \mathbf{w}).$$

具体计算就是 $L(P, \mathbf{w})$ 对 P 求导，并让其等于0.

$$\begin{aligned} \blacksquare \frac{\partial L(P, \mathbf{w})}{\partial P(y|x)} &= \sum_{x,y} \hat{P}(x) (\log P(y|x) + 1) - \sum_y w_0 - \sum_{x,y} (\hat{P}(x) \sum_{i=1}^n w_i f_i(x, y)) \\ &= \sum_{x,y} \hat{P}(x) (\log P(y|x) + 1 - w_0 - \sum_{i=1}^n w_i f_i(x, y)) \end{aligned}$$

令其偏导数等于0, 在 $\hat{P}(x) > 0$ 的情况下, 解得：

$$\blacksquare P(y|x) = \exp \left(\sum_{i=1}^n w_i f_i(x, y) \right) + w_0 - 1$$

由于： $\sum_y P(y|x) = 1$, 得：

$$\blacksquare P_w(y|x) = \frac{1}{Z_w(x)} \exp \left(\sum_{i=1}^n w_i f_i(x, y) \right)$$

其中：

$$\blacksquare Z_w(x) = \sum_{i=1}^n \exp \left(\sum_y w_i f_i(x, y) \right)$$

其中： $Z_w(x)$ 称为规范化因子, $f_i(x, y)$ 称为特征函数, w_i 是特征函的权重。

之后再求解对偶问题外部的极大化问题：

$$\blacksquare \max_w \Psi(\mathbf{w})$$

将其解记作 w^* , 即：

$$\blacksquare w^* = \arg \max_w \Psi(\mathbf{w})$$

降维与无监督学习

隐语义模型

LFM(Latent factor model):通过隐含特征(Latent Factor)联系用户兴趣和物品.

LFM通过如下公式计算用户u对物品i的兴趣:

$$Preference(u, i) = r_{ui} = p_u^T q_i = \sum_{k=1}^K p_{u,k} q_{i,k}$$

公式中 $p_{u,k}$ 和 $q_{i,k}$ 是模型的参数, 其中 $p_{u,k}$ 度量了用户u的兴趣和第k个隐类的关系。而 $q_{i,k}$ 都凉了第k个隐类和物品i之间的关系。

$p_{u,k}, q_{i,k}$ 通过如下方式求解:

$$C = \sum_{(u,i) \in K} (r_{ui} - \hat{r}_{ui})^2 = \sum_{(u,i) \in K} (r_{ui} - \sum_{k=1}^K p_{u,k} q_{i,k})^2 + \lambda ||p_u||^2 + \lambda ||q_i||^2$$

SVD++

指标

1. 准确率
2. 召回率
3. 覆盖率
4. 多样性

麦克斯韦妖与信息熵

不存在一个监测系统, 能区分粒子的速度而对粒子做出区分, 使得原本温度相同的系统, 自发的形成高温与低温子系统。因为在区分例子速度时, 就是一个信息处理的过程, 使得系统的信息熵减小。那么, 怎么根据熵增原理, 确定系统的熵减数量与检测系统需要付出的信息熵的数量, 或者能量。

非负矩阵分解

$N * p$ 数据矩阵X近似表示为:

$$\mathbf{X} \approx \mathbf{WH}$$

其中: $\mathbf{W} \in R^{N \times r}$, $\mathbf{H} \in R^{r \times p}$, $r \leq \max(N, p)$, 我们假定 $x_{ij}, w_{ik}, h_{kj} \geq 0$.

矩阵W, H通过最大化下面似然函数确定:

$$L(\mathbf{W}, \mathbf{H}) = \sum_{i=1}^N \sum_{j=1}^p [x_{ij} \log(\mathbf{WH})_{ij} - (\mathbf{WH})_{ij}]$$

这个从一个 x_{ij} 满足均值为 $(\mathbf{WH})_{ij}$ 的泊松分布的模型的似然函数。
通过梯度下降法可以求解。

分类问题与方法

回归问题与方法

K均值EM等聚类算法

k-means对初始值的设置很敏感，所以有了k-means++、intelligent k-means、genetic k-means。k-means对噪声和离群值非常敏感，所以有了k-medoids和k-medians。k-means只用于numerical类型数据，不适用于categorical类型数据，所以k-modes。k-means不能解决非凸(non-convex)数据，所以有了kernel k-means

正则化方法原理

Ridge回归, Shrink与SVD

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

等价于:

$$\begin{aligned} \hat{\beta}^{ridge} &= \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 \\ &\text{subject to : } \sum_{j=1}^p \beta_j^2 \leq t. \end{aligned}$$

上面的问题也等价于假定, y_i, β_i 服从如下分布:

$$y_i \in N(\beta_0 + x_i^T \beta, \sigma^2)$$

$$\beta_i \in N(0, \tau^2)$$

其中: $\lambda = \sigma^2 / \tau^2$

因此RRS(Root sum square)可以写成如下形式:

$$RRS(\lambda) = (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda \beta^T \beta$$

通过对 β 求导并令其为0, 可以得到:

$$\hat{\beta}^{ridge} = (\mathbf{X}^T \mathbf{X} + \lambda I)^{-1} \mathbf{X}^T \mathbf{y}$$

为了建立起L2与SVD之间的联系, 我们对 \mathbf{X} 进行SVD分解:

$$\mathbf{X} = \mathbf{U} \mathbf{D} \mathbf{V}^T$$

我们重写没有正则化的最小二乘拟合:

$$\mathbf{X}\beta^{ls} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} = \mathbf{U} \mathbf{U}^T \mathbf{y}$$

对于L2正则化的最小二乘法:

$$\begin{aligned} \mathbf{X}\beta^{ls} &= (\mathbf{X}^T \mathbf{X} + \lambda I)^{-1} \mathbf{X}^T \mathbf{y} = \mathbf{U} \mathbf{D} (\mathbf{D}^2 + \lambda \mathbf{I})^{-1} \mathbf{D} \mathbf{U}^T \mathbf{y} \\ &= \sum_{j=1}^p \mathbf{u}_j \frac{d_j^2}{d_j^2 + \lambda} \mathbf{u}_j^T \mathbf{y} \end{aligned}$$

从上面的分式 $\frac{d_j^2}{d_j^2 + \lambda}$ 可以知道, 对于 $d_j \ll \lambda$, 则相应的方向会收缩到0。可以认为L2就是对数据进行了SVD分解后, 只保留了 $d_j > \lambda$ 的分量。

Lasso(L1)回归

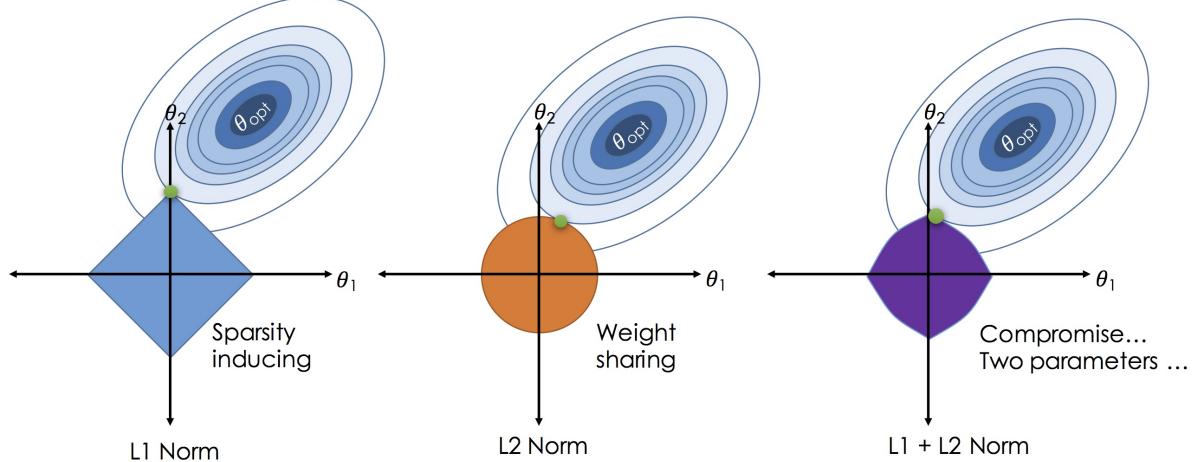
$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

等价于：

$$\hat{\beta}^{ridge} = \arg \min_{\beta} \sum_{i=1}^N (y_i - \beta_0 - \sum_{j=1}^p x_{ij}\beta_j)^2$$

$$subject to : \sum_{j=1}^p |\beta_j| \leq t.$$

L1更容易产生稀疏性,因为椭圆与菱形更易在菱形顶点相交,而与圆形不容易在坐标轴上相切。



L1稀疏性的数学解析

一个数学上更严格^{严格}的解析。。上面的问题也等价于假定, y_i, β_i 服从如下分布:

$$y_i \in N(\beta_0 + x_i^T \beta, \sigma^2)$$

$$\beta_i \in (1/2\tau) \exp(-|\beta|/\tau)$$

其中: $\tau = 1/\lambda$

因此 β_i 更容易分布在0的附件,也就是能级排斥(量子混沌里面的概念, L1是可积系统, L2是GOE)

最小二乘一般通过Cholesky分解($p^3 + Np^2/2$)或者QR分解(NP^2)来实现, 前者一般更快, 但是没有后者稳定。

机器学习项目

目录

1. **AlphaZero-Gomoku**
2. **OpenPose**
3. **Face Recognition**
4. **Magenta**
5. **YOLOv2**
6. **MUSE**
7. **Arnold**
8. **FoolNLTK**
9. **Gym**
10. **style2paints v2.0**

winequality 可以作为线性回归的练习

从回归到分类，到推进系统，数据集在下面 <http://archive.ics.uci.edu/ml/datasets.html> 对于回归，可以先用方程产生带噪声的数据，再对这些带噪声的数据进行回归分析。一元高次方程回归。

$$y(x) = ax^3 + bx^2 + cx + d + \text{random noise}$$

多元回归分析可以通过求伪逆来求解。

自己需要做的是写一个regression与classification的类，来实现不同的回归与分类算法，先设计成一个类，如果有必要再设计成一个工厂类。

多元回归分析

多元回归方程

$$\hat{y} = \lambda_0 + \lambda_1 x_1 + \lambda_2 x_2 + \dots + \lambda_m x_m$$

其中 b_j 是 x_j 的偏回归系数, \hat{y} 是样本的估计值。

根据最小值原理, 可以求得偏回归系数。

$$L(b) = \sum_{i=0}^{n-1} [y_i - (\lambda_0 + \lambda_1 x_{1i} + \lambda_2 x_{2i} + \dots + \lambda_m x_{mi})]^2$$

变量数为m, 样本数为n. 最小二乘法的目的就是找到一组偏回归系数使得损失函数L极小, 极端情况就是L=0, 则所有样本估计值就是样本的观测值。这在满秩的情况下存在。一般只是求得L的极小值点。对损失函数求偏导, 可以得到:

实际问题可以转化成矩阵分析

$$\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} & 1 \\ x_{21} & x_{22} & \cdots & x_{2m} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} y_{1o} \\ y_{2o} \\ \vdots \\ y_{no} \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

即:

$$X * \lambda = Y + \epsilon$$

寻找一组 λ 使得 $L(\lambda) = (X * \lambda - Y)^T (X * \lambda - Y)$ 极小

利用 $L(\lambda)$ 对向量 λ 求导可以得到¹:

$$\frac{\partial L(\lambda)}{\partial \lambda} = 2X^T(X\lambda - Y) = 0$$

我们也可以得到: $\frac{\partial^2 L(\lambda)}{\partial \lambda^2} = 2X^T X$

求得: $\lambda = (X^T X)^{-1} X^T Y$

$(X^T X)^{-1} X^T$ 也称为X的伪逆。

多项式拟合

函数的多项式表示方式: $y = \sum_{k=0}^m \lambda_k x^k$ 对于n组数据 (x_i, y_i) 若我们想用多项式去拟合这组数据, 就是寻找使下列函数值极小的一组系数 λ

$$L(\lambda) = \frac{1}{2m} \sum_{i=0}^{n-1} (y_i - \sum_{k=0}^m \lambda_k x_i^k)^2 \text{ 令: } \lambda = [\lambda_0, \lambda_1, \dots, \lambda_m]^T$$

$$X_i = [x_i^0, x_i^1, \dots, x_i^m] \quad Y = [Y_0, Y_1, \dots, Y_{n-1}]^T$$

则上面损失函数可以表示为: $L(\lambda) = (X * \lambda - Y)^T (X * \lambda - Y)$

对其求一阶导，结果为0； $\frac{\partial L(\lambda)}{\partial \lambda} = 0$ 最终方程可以化简成如下形式：

$$\boxed{\begin{bmatrix} \sum_{j=0}^{n-1} x_j^0 & \sum_{j=0}^{n-1} x_j^1 & \cdots & \sum_{j=0}^{n-1} x_j^m \\ \sum_{j=0}^{n-1} x_j^1 & \sum_{j=0}^{n-1} x_j^2 & \cdots & \sum_{j=0}^{n-1} x_j^{m+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{j=0}^{n-1} x_j^m & \sum_{j=0}^{n-1} x_j^{m+1} & \cdots & \sum_{j=0}^{n-1} x_j^{2m} \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \cdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n-1} y_j * x_j^0 \\ \sum_{j=0}^{n-1} y_j * x_j^1 \\ \cdots \\ \sum_{j=0}^{n-1} y_j * x_j^m \end{bmatrix}} \text{ 即: } X * \lambda = Y$$

得到 $\lambda = X^{-1}Y$ 若在方程上加上一个正则项，

$L(\lambda) = \frac{1}{2m} \sum_{i=0}^{n-1} (y_i - \sum_{k=0}^m \lambda_k x_i^k)^2 + \sum_{k=0}^m \lambda_k^2$ 得到如下的矩阵：

$$\boxed{\begin{bmatrix} \sum_{j=0}^{n-1} x_j^0 - 2n & \sum_{j=0}^{n-1} x_j^1 & \cdots & \sum_{j=0}^{n-1} x_j^m \\ \sum_{j=0}^{n-1} x_j^1 & \sum_{j=0}^{n-1} x_j^2 - 2n & \cdots & \sum_{j=0}^{n-1} x_j^{m+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{j=0}^{n-1} x_j^m & \sum_{j=0}^{n-1} x_j^{m+1} & \cdots & \sum_{j=0}^{n-1} x_j^{2m} - 2n \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \cdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n-1} y_j * x_j^0 \\ \sum_{j=0}^{n-1} y_j * x_j^1 \\ \cdots \\ \sum_{j=0}^{n-1} y_j * x_j^m \end{bmatrix}}$$

类似于Ridge回归：但是对于加噪声参数的多项式数据，发现不加正则项能给出正确的结果，加正则项反倒不能。

迭代法求解

求解上面的方程也可以使用迭代法求解：

1. 矩阵求导术(上), <https://zhuanlan.zhihu.com/p/24709748> ↵

矩阵微分

参考闲话矩阵求导

向量 \mathbf{y} 对标量 x 求导

我们假定所有的向量都是列向量

$$\frac{\partial \mathbf{y}}{\partial x} = \left[\frac{\partial y_1}{\partial x} \quad \frac{\partial y_2}{\partial x} \quad \cdots \quad \frac{\partial y_m}{\partial x} \right]$$

标量 y 对向量 \mathbf{x} 求导：

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_m} \end{bmatrix}$$

向量对向量求导

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

标量对矩阵求导，

$$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial x_{11}} & \frac{\partial y}{\partial x_{12}} & \cdots & \frac{\partial y}{\partial x_{1q}} \\ \frac{\partial y}{\partial x_{21}} & \frac{\partial y}{\partial x_{22}} & \cdots & \frac{\partial y}{\partial x_{2q}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial x_{p1}} & \frac{\partial y}{\partial x_{p2}} & \cdots & \frac{\partial y}{\partial x_{pq}} \end{bmatrix}$$

矩阵对标量求导，

注意有个类似于转置的操作，因为 \mathbf{Y} 是 $m \times n$ 矩阵

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{12}}{\partial x} & \cdots & \frac{\partial y_{1n}}{\partial x} \\ \frac{\partial y_{21}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \cdots & \frac{\partial y_{2n}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{m1}}{\partial x} & \frac{\partial y_{m2}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$$

用维度分析来解决求导的形式问题

$$\text{向量对向量的微分 } \frac{\partial(\mathbf{Ax})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial(\mathbf{Ax})_1}{\partial x_1} & \frac{\partial(\mathbf{Ax})_2}{\partial x_1} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_1} \\ \frac{\partial(\mathbf{Ax})_1}{\partial x_2} & \frac{\partial(\mathbf{Ax})_2}{\partial x_2} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial(\mathbf{Ax})_1}{\partial x_n} & \frac{\partial(\mathbf{Ax})_2}{\partial x_n} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix} = \mathbf{A}^T$$

考虑 $\frac{\partial \mathbf{Au}}{\partial \mathbf{x}}$, \mathbf{A} 与 \mathbf{x} 无关, 所以 \mathbf{A} 肯定可以提出来, 只是其形式不知。假如

$\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{u} \in \mathbb{R}^{n \times 1}$, $\mathbf{x} \in \mathbb{R}^{p \times 1}$ 我们知道最终结果肯定和 $\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$ 有关, 注意到 $\frac{\partial \mathbf{u}}{\partial \mathbf{x}} \in \mathbb{R}^{p \times n}$, 于是 \mathbf{A} 只能

转置以后添在后面, 因此: $\frac{\partial \mathbf{Au}}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \mathbf{A}^T$

$$dL = \frac{\partial \mathbf{L}^T}{\partial \mathbf{w}} d\mathbf{w}$$

再考虑如下问题:

$$\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}}, \mathbf{x} \in \mathbb{R}^{m \times 1}, \mathbf{y} \in \mathbb{R}^{n \times 1} \text{ 其中 } \mathbf{A} \text{ 与 } \mathbf{x} \text{ 无关, 我们知道 } \frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1} \text{ 因此}$$

$$\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}} = f(A, y) + g\left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \mathbf{x}^T A\right) \mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{y} \in \mathbb{R}^{n \times 1}, \mathbf{x}^T \mathbf{A} \in \mathbb{R}^{1 \times n}, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n} \text{ 因}$$

此, 为了满足 $\frac{\partial \mathbf{x}^T \mathbf{Ay}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1}$, 我们可以知道函数 f, g 的形式。最终有:

$$\frac{\partial(\mathbf{x}^T \mathbf{A}) \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \mathbf{A}^T \mathbf{x} + \mathbf{A} \mathbf{y} \text{ 当 } \mathbf{x} = \mathbf{y} \text{ 时,} \quad \frac{\partial(\mathbf{x}^T \mathbf{A}) \mathbf{y}}{\partial \mathbf{x}} = (\mathbf{A}^T + \mathbf{A}) \mathbf{x}$$

最后一个例子(还没解出来): $\frac{\partial \mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}}{\partial \mathbf{x}}, \mathbf{a}, \mathbf{b}, \mathbf{x} \in \mathbb{R}^{m \times 1}$ 知道 $\frac{\partial \mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times 1}$ 还需要对迹形式进行求解.

正则化

L0 is OMP-k

Ensemble learning(团体学习)

利用多个学习方法来获得更好的预测能力，机器学习中的系综学习的样本是有限的，但是统计力学中的系综方法其样本是无限的。机器学习中的系综方法包括Bagging,boosting

Bagging算法(Bootstrap aggregating)

参考 是一种团体学习方法,可以看成事一种圆桌会议,或者投票选举的形式,其思想是“群众的眼光是雪亮的”,可以训练多个模型,之后将这些模型进行加权组合,一般这类方法的效果,都会好于单个模型的效果。在实践中,在特征一定的情况下,大家总是使用Bagging的思想去提升效果。算法步骤 给定一个大小为n的训练集D, Bagging算法从中均匀、有放回地(即使用自助抽样法)选出m个大小为 n' 的子集 D_i , 作为新的训练集。在这m个训练集上使用分类、回归等算法,则可得到m个模型,同一个训练集中的成员可以有重重复,再通过取平均值、取多数票等方法,即可得到Bagging的结果。

Boosting

在Bagging方法中, 我们假设每个训练样本的权重都是一致的; 而Boosting算法则更加关注错分的样本, 越是容易错分的样本, 约要花更多精力去关注。对应到数据中, 就是该数据对模型的权重越大, 后续的模型就越要拼命将这些经常分错的样本分正确。最后训练出来的模型也有不同权重, 所以boosting更像是会整, 级别高, 权威的医师的话语权就重些。训练:先初始化每个训练样本的权重相等为 $1/d$, d为样本数量; 之后每次使用一部分训练样本去训练弱分类器, 且只保留错误率小于0.5的弱分类器, 对于分对的训练样本, 将其权重调整为 $\text{error}(M_i)/(1-\text{error}(M_i))$, 其中 $\text{error}(M_i)$ 为第*i*个弱分类器的错误率(降低正确分类的样本的权重, 相当于增加分错样本的权重);

测试:每个弱分类器均给出自己的预测结果, 且弱分类器的权重为 $\log(1-\text{error}(M_i))/\text{error}(M_i)$ 权重最高的类别, 即为最终预测结果。

在adaboost中, 弱分类器的个数的设计可以有多种方式, 例如最简单的就是使用一维特征的树作为弱分类器。

adaboost在一定弱分类器数量控制下, 速度较快, 且效果还不错。

我们在实际应用中使用adaboost对输入关键词和推荐候选关键词进行相关性判断。随着新的模型方法的出现，adaboost效果已经稍显逊色，我们在同一数据集下，实验了GBDT和adaboost，在保证召回基本不变的情况下，简单调参后的Random Forest准确率居然比adaboost高5个点以上，效果令人吃惊。。。

Bagging和Boosting都可以视为比较传统的集成学习思路。现在常用的Random Forest, GBDT, GBRank其实都是更加精细化，效果更好的方法。后续会有更加详细的内容专门介绍。

训练神经网络的经验

第一步(一个月), 公式推导必须会, 核心思想会, 并且要融会贯通。

1. SVM
2. LR, LTR
3. 决策树, RF, GBDT, xgboost
4. 贝叶斯
5. 降维:PCA, SVD
6. BP的公式推导, 以及写一个简单的神经网络, 并跑个小的数据。用python, numpy。
7. LeetCode上面刷Easy与Medium的题
8. 看面经, 找面试常遇到的问题

第二步是最优化总结(半个月), 总结常用的优化算法, 比如梯度下降, 牛顿, 拟牛顿, trust region, 二次规划。

特征工程

特征工程是什么

有这么一句话在业界广泛流传：数据和特征决定了机器学习的上限，而模型和算法只是逼近这个上限而已。那特征工程到底是什么呢？顾名思义，其本质是一项工程活动，目的是最大限度地从原始数据中提取特征以供算法和模型使用。通过总结和归纳，人们认为特征工程包括以下方面：

特征选择方式

参考知乎上[这篇文章](#)

特征选择与特征提取是特征工程中的两大核心问题。特征选择是指选择获得相应模型和算法最好性能的特征集，工程上常用的方法如下：

1. 计算每一个特征与相应变量的相关性。一般是算皮尔逊系数和互信息系数。皮尔逊系数只能衡量线性相关性，互信息系数能够很好地度量各种相关性，但是计算相对复杂，很多toolkit里面包含了这个工具（如sklearn的MI），得到相关性之后就可以排序选择特征了。
2. 构建单个特征的模型，通过模型的准确性为特征排序，借此来选择特征。
3. 通过L1正则项来选择特征，L1正则方法具有稀疏解的特性，因此天然具备特征选择的特性。
4. 训练能够对特征打分的预选模型：RF和LR等都对模型的特征打分，通过打分获得相关性后再训练最终模型。
5. 通过特征组合后回来选择特征：如对用户ID和用户特征的组合来获得较大的特征集再来选择特征，这种做法在推荐系统和广告系统中比较常见，这也是所谓亿级甚至十亿级特征的主要来源，原因是用户数据比较稀疏，特征组合能同时兼顾全局模型和个性化模型。
6. 通过深度学习来进行特征选择/目前这种手段

总体而言，有两种方法：

基于方差的方法（PCA），基于相关性方法。

1. 相关性：一般用皮尔逊相关系数来定义。
2. 方差法：去掉值变化不大的特征，因为可以把这些特征看成常数值，他们对分类或者回归结果影响不大。

特征选择

1. Filter方法：自变量与目标变量之间的关联
 - i. Pearson相关系数
 - ii. 卡方检测
 - iii. 信息增益，互信息(MIC)

2. Wrapper方法:通过目标函数来决定是否加入一个变量

i. 迭代:产生特征子集, 评价

ii. 完全搜索

iii. 启发式搜索

iv. 随机搜索(GA, SA)

3. Embedded方法:学习器自身自动选择特征

i. 正则化L1, L2

ii. 决策树(熵-信息增益, 基尼系数)

iii. 深度学习

Filter

Pearson相关系数

Pearson相关系数(取值在[-1,1]之间, 大于0是正相关, 小于0是负相关, 等于0就没有相关性。只对具有线性相关性的变量有效, 不能处理具有非线性关系的两组变量)

互信息(MIC)

$$MIC : I(X, Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log\left(\frac{p(x, y)}{p(x)p(y)}\right)$$

对于线性与非线性关系的数据都实用。

距离相关系数(Distance Correlation)

为了克服Pearson相关系数只对具有线性关系的变量起作用而引入。

样本: $(X_k, Y_k), k = 1, 2, \dots, n$

定义距离矩阵:

$$a_{j,k} = \|\mathbf{X}_j - \mathbf{X}_k\|, \quad j, k = 1, 2, \dots, n$$

$b_{j,k} = \|\mathbf{Y}_j - \mathbf{Y}_k\|, \quad j, k = 1, 2, \dots, n$ || * || 是欧氏距离。取所有的双中心距离。

$$\mathbf{A}_{j,k} := a_{j,k} - \hat{a}_{j\cdot} - \hat{a}_{\cdot k} + \hat{a}_{\cdot\cdot}$$

$$\mathbf{B}_{j,k} := b_{j,k} - \hat{b}_{j\cdot} - \hat{b}_{\cdot k} + \hat{b}_{\cdot\cdot}$$

$\hat{a}_{j\cdot}$ 是 j -th row 的平均, $\hat{a}_{\cdot k}$ 是 k -th column 的平均, $\hat{a}_{\cdot\cdot}$ 是全局平均。

定义 The squared sample distance covariance:

$$dCov_n^2(X, Y) := \frac{1}{n^2} \sum_{j=1}^n \sum_{k=1}^n A_{jk} B_{jk}$$

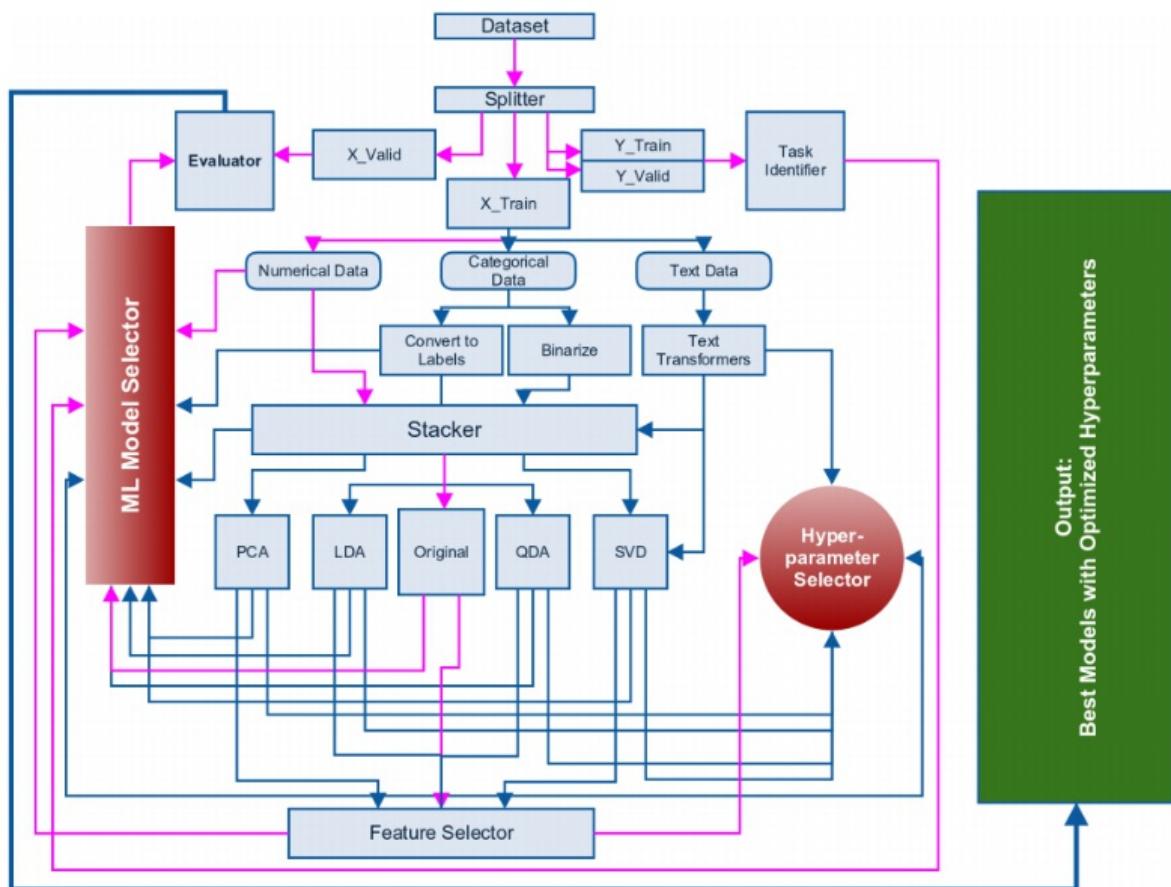
虽然MIC与距离相关系数能处理具有线性与非线性关系的变量之间的相关性, 但是Pearson还是不可替代的, 第一, Pearson系数计算速度快; 第二, 相对于其它两种取值在[0,1], Pearson取值[-1,1], 正负表关系的正负, 绝对值表示强度。前提就是两个变量是线性相关的。

特征工程小结

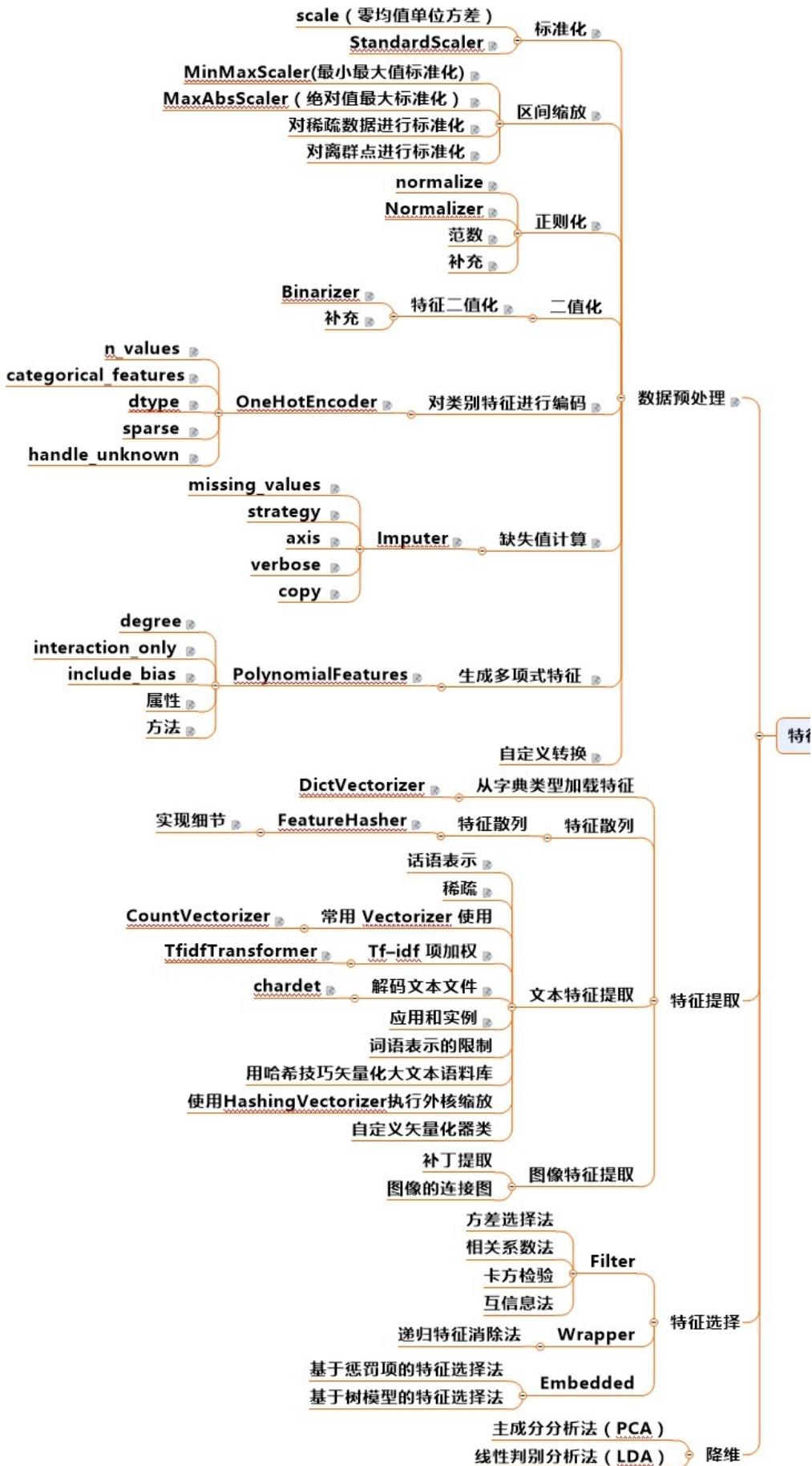
- 特征工程: 利用数据领域的相关知识来创建能够使机器学习算法达到最佳性能的特征的过程。
 - 特征构建: 是原始数据中人工的构建新的特征。
 - 特征提取: 自动地构建新的特征, 将原始特征转换为一组具有明显物理意义或者统计意义或核的特征。
 - 特征选择: 从特征集合中挑选一组最具统计意义的特征子集, 从而达到降维的效果。
- 重要性排名: 特征构建>特征提取>特征选择

ML框架¹

参考



特征工程



模型的超参数调节

Model	Parameters to optimize	Good range of values
Linear Regression	<ul style="list-style-type: none">• fit_intercept• normalize	<ul style="list-style-type: none">• True / False• True / False
Ridge	<ul style="list-style-type: none">• alpha• Fit_intercept• Normalize	<ul style="list-style-type: none">• 0.01, 0.1, 1.0, 10, 100• True/False• True/False
k-neighbors	<ul style="list-style-type: none">• N_neighbors• p	<ul style="list-style-type: none">• 2, 4, 8, 16• 2, 3
SVM	<ul style="list-style-type: none">• C• Gamma• class_weight	<ul style="list-style-type: none">• 0.001, 0.01.....10...100...1000• 'Auto', RS*• 'Balanced' , None
Logistic Regression	<ul style="list-style-type: none">• Penalty• C	<ul style="list-style-type: none">• L1 or L2• 0.001, 0.01.....10...100
Naive Bayes (all variations)	NONE	NONE
Lasso	<ul style="list-style-type: none">• Alpha• Normalize	<ul style="list-style-type: none">• 0.1, 1.0, 10• True/False
Random Forest	<ul style="list-style-type: none">• N_estimators• Max_depth• Min_samples_split• Min_samples_leaf• Max features	<ul style="list-style-type: none">• 120, 300, 500, 800, 1200• 5, 8, 15, 25, 30, None• 1, 2, 5, 10, 15, 100• 1, 2, 5, 10• Log2, sqrt, None
Xgboost	<ul style="list-style-type: none">• Eta• Gamma• Max_depth• Min_child_weight• Subsample• Colsample_bytree• Lambda• alpha	<ul style="list-style-type: none">• 0.01,0.015, 0.025, 0.05, 0.1• 0.05-0.1,0.3,0.5,0.7,0.9,1.0• 3, 5, 7, 9, 12, 15, 17, 25• 1, 3, 5, 7• 0.6, 0.7, 0.8, 0.9, 1.0• 0.6, 0.7, 0.8, 0.9, 1.0• 0.01-0.1, 1.0 , RS*• 0, 0.1, 0.5, 1.0 RS*

NLP 特征工程

Data Pre-processing

- Text Cleaning
- Spell Checking
- Stemming
- Lemmatization

Feature Engineering

- Categorical Features
- Counting Features

More Features

- Co-occurrence Features
- Semantic Features
- Statistical Features

Models

- XGBoost
- Ridge Regression
- GBM
- Extra Trees
- Random Forest

特征工程可以参考[这篇文章](#)

<http://blog.kaggle.com/2016/07/21/approaching-almost-any-machine-learning-problem-abhishek-thakur/>

第八章 推荐系统

推荐系统概述

FM(Factorization Machines), FFM

FM

- 一般的线性模型为:
$$y = w_0 + \sum_{i=1}^n w_i x_i$$
一般模型中，各个特征是独立考虑的，没

有考虑特征之间的相互关系。如果考虑特征 x_i, x_j 之间的相互关系，模型修改如下：

$$y = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} x_i x_j$$

■如果系统的特征比较多的话，计算复杂度会大大提升。为了降低时间复杂度，我们引入了辅助向量latent vector

$$\mathbf{V}_i = [v_{i1}, v_{i2}, \dots, v_{ik}]^T$$

,辅助变量是描述变量之间的相关性。模型修改如下：

$$y = w_0 + \sum_{i=1}^n w_i x_i + \sum_{i=1}^n \sum_{j=i+1}^n (V_i, V_j) x_i x_j$$

以上就是FM模型。k是超参数，一般娶30或者40。时间复杂度是 $O(kn^2)$ ，可通过如下方式化简。

$$\begin{aligned} & \sum_{i=1}^n \sum_{j=i+1}^n (V_i, V_j) x_i x_j \\ &= \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n (V_i, V_j) x_i x_j - \frac{1}{2} \sum_{i=1}^n (V_i, V_i) x_i x_i \\ &= \frac{1}{2} \sum_{f=1}^n \left(\sum_{i=1}^n v_{if} x_i \right)^2 - \sum_{i=1}^n v_{if}^2 x_i^2 \end{aligned}$$

通过对每个特征引入latent vector \mathbf{V}_i ，并对公式进行化简，可以把时间复杂度降为 $O(kn)$.

LTR

LTR or machine-learned ranking(MLR)是运用机器学习,典型的监督,半监督或者强化学习来为信息检索系统构建排序模型

排序学习可以在信息检索(IR),NLP, DM等领域被广泛应用,典型的应用有文献检索,专家检索系统,定义查询系统,协同过滤,问答系统,关键词提取,文档摘要还有机器翻译等。

互联网搜索方面的一个新趋势是使用机器学习方法去自动的建立评价模型 $f(q, d)$ 。

对于标注训练集,选定LTR方法,确定损失函数,以最小化损失函数为目标进行优化即可得到排序模型的相关参数,这就是学习过程。预测过程就是将待预测结果输入到学习到的排序模型中,即可以得到结果的相关得,利用该得分进行排序即可得到待预测结果的最终顺序。

第九章 Kaggle

Titanic

Variable Description

Survived: Survived (1) or died (0)
Pclass: Passenger's class
Name: Passenger's name
Sex: Passenger's sex
Age: Passenger's age
SibSp: Number of siblings/spouses aboard
Parch: Number of parents/children aboard
Ticket: Ticket number
Fare: Fare
Cabin: Cabin
Embarked: Port of embarkation

常用的数据清洗技巧

导入数据

```
train = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\train.csv")
test = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\test.csv")
test_Y = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\gender_submission.csv")
```

导入库函数

```
import warnings
warnings.filterwarnings('ignore')
# Handle table-like data and matrices
import numpy as np
import pandas as pd

# Modelling Algorithms
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier

# Modelling Helpers
```

```

from sklearn.preprocessing import Imputer , Normalizer , scale
from sklearn.cross_validation import train_test_split , StratifiedKFold
from sklearn.feature_selection import RFECV

# Visualisation
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import seaborn as sns

# Configure visualisations
%matplotlib inline
mpl.style.use( 'ggplot' )
sns.set_style( 'white' )
pylab.rcParams[ 'figure.figsize' ] = 8 , 6

```

数据的统计属性

```

train.shape ##数据的维度
train.head ##显示所有数据

```

相关系数与统计属性

In [6]: `train.describe()`

Out[6]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [7]: `train.corr()`

Out[7]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
PassengerId	1.000000	-0.005007	-0.035144	0.036847	-0.057527	-0.001652	0.012658
Survived	-0.005007	1.000000	-0.338481	-0.077221	-0.035322	0.081629	0.257307
Pclass	-0.035144	-0.338481	1.000000	-0.369226	0.083081	0.018443	-0.549500
Age	0.036847	-0.077221	-0.369226	1.000000	-0.308247	-0.189119	0.096067
SibSp	-0.057527	-0.035322	0.083081	-0.308247	1.000000	0.414838	0.159651
Parch	-0.001652	0.081629	0.018443	-0.189119	0.414838	1.000000	0.216225
Fare	0.012658	0.257307	-0.549500	0.096067	0.159651	0.216225	1.000000

相关系数, 分布, 类别图

```

def plot_correlation_map( df ):
    corr = train.corr()
    _ , ax = plt.subplots( figsize =( 12 , 10 ) )
    cmap = sns.diverging_palette( 220 , 10 , as_cmap = True )
    _ = sns.heatmap(
        corr,
        cmap = cmap,
        square=True,
        cbar_kws={ 'shrink' : .9 },
        ax=ax,
        annot = True,
        annot_kws = { 'fontsize' : 12 }
    )

def plot_distribution( df , var , target , **kwargs ):
    row = kwargs.get( 'row' , None )
    col = kwargs.get( 'col' , None )
    facet = sns.FacetGrid( df , hue=target , aspect=4 , row = row , col = col )
    facet.map( sns.kdeplot , var , shade= True )
    facet.set( xlim=( 0 , df[ var ].max() ) )
    facet.add_legend()

def plot_categories(fd, cat, target, **kwargs):
    row = kwargs.get('row', None)
    col = kwargs.get('col', None)
    facet = sns.FacetGrid(fd, row = row, col = col)
    facet.map(sns.barplot, cat, target)
    facet.add_legend()

plot_correlation_map(train)
plot_distribution( train , var = 'Age' , target = 'Survived' , row = 'Sex' )
plot_categories(train, cat = 'Embarked', target = 'Survived')

```

plot_correlation_map(train)



类标签数字化

```
sex = pd.Series( np.where( train.Sex == 'male' , 1 , 0 ) , name = 'Sex' )
```

通过pandas产生一个新的数组，它是把原来sex中的male转化成1，其它的sex类别转化成0.

对类标签添加前缀

```
embarked = pd.get_dummies( train.Embarked , prefix='Embarked' )
```

原来Embarked的类标签是C, Q, S。现在添加了前缀Embarked_，并且把原来的一个类分成了三个类。

```
In [47]: embarked.head()
```

```
Out[47]:
```

	Embarked_C	Embarked_Q	Embarked_S
0	0	0	1
1	1	0	0
2	0	0	1
3	0	0	1
4	0	0	1

缺失值填充

通过均值来填充

```
imputed = pd.DataFrame()
imputed[ 'Age' ] = train.Age.fillna( train.Age.mean() )
imputed[ 'Fare' ] = train.Fare.fillna( train.Fare.mean() )
imputed[ 'Pclass' ] = train.Pclass.fillna( train.Pclass.mean() )
```

数据拼接

```
DataSet = pd.concat([imputed, embarked, sex, train.Pclass, train.SibSp, train.Parch],axis = 1)
```

DataSet.shape就是(891,10)

构造训练数据

```
train_X = DataSet[0:891]
train_Y = train.Survived
```

模型选择与训练

```
model = RandomForestClassifier(max_depth=3, max_features='auto',n_estimators=100)
model.fit(train_X, train_Y)
print model.score(train_X, train_Y),model.score(test_X, test_Y)

0.8305274971941639 0.8851674641148325
```

当你选定一个模型后，就根据这个模型，进行相应的参数输入，最基本的包括输入(X,Y)，其它的就是模型参数的设定，这个你可以根据sklearn的API进行设置。当然有目的性的调参考验的就是你的理论功底了。

总的效果

```
import warnings
warnings.filterwarnings('ignore')
# Handle table-like data and matrices
import numpy as np
import pandas as pd

from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC, LinearSVC
from sklearn.ensemble import RandomForestClassifier , GradientBoostingClassifier

# Modelling Helpers
from sklearn.preprocessing import Imputer , Normalizer , scale
from sklearn.cross_validation import train_test_split , StratifiedKFold
from sklearn.feature_selection import RFECV
import xgboost as xgb

train = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\train.csv")
test = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\test.csv")
test_Y = pd.read_csv("C:\Data\Group\ShareFolder\Kaggle\Titanic\\gender_submission.csv")

imputed_train = pd.DataFrame()
imputed_train[ 'Age' ] = train.Age.fillna( train.Age.mean() )
imputed_train[ 'Fare' ] = train.Fare.fillna( train.Fare.mean() )
imputed_train[ 'Pclass' ] = train.Pclass.fillna( train.Pclass.mean() )
embarked_train = pd.get_dummies(train.Embarked, prefix = 'Embarked')
sex_train= pd.Series( np.where( train.Sex == 'male' , 1 , 0 ) , name = 'Sex' )
DataSet_train = pd.concat([imputed_train, embarked_train, sex_train, train.Pclass, train.SibSp, train.Parch],axis = 1)

imputed_test = pd.DataFrame()
imputed_test[ 'Age' ] = test.Age.fillna( train.Age.mean() )
imputed_test[ 'Fare' ] = test.Fare.fillna( train.Fare.mean() )
imputed_test[ 'Pclass' ] = test.Pclass.fillna( train.Pclass.mean() )
embarked_test = pd.get_dummies(test.Embarked, prefix = 'Embarked')
sex_test = pd.Series( np.where( test.Sex == 'male' , 1 , 0 ) , name = 'Sex' )
DataSet_test = pd.concat([imputed_test, embarked_test, sex_test, test.Pclass, test.SibSp, test.Parch],axis = 1

test_X = DataSet_test[0:418]
train_X = DataSet_train[0:891]
train_Y = train.Survived
train_X.shape,train_Y.shape,test_X.shape,test_Y.shape
```

最终结果

使用SVM在Titanic数据上，训练集效果依次是线性SVM, GBDT,LR,RF,DT,KNN。最诡异的是，有事测试集上准确率比训练集上高。

对于随机森林RF，会发现，一开始随着树的深度增加，RF整体的准确率会上升，也就是说，RF需要准确性高的(或者说bias小的)分类器做基函数，对于GBDT，发现随深度增加，准确率下降，也就是GBDT需要深度浅，variance很小的树作为分类器，这符合他们的原理。即RF基于bias小的基分类器，通过增加树的数目来降低variance，boosting基于variance小的基分类器，通过boost来降低bias。

但是有个问题是，RF在100棵树时效果最好，选择1000, 10000都没有100的效果好。

```
In [78]: model = GaussianNB()
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score =  0.7710437710437711
test score =  0.7870813397129187

In [79]: model = DecisionTreeClassifier()
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score =  0.9820426487093153
test score =  0.8110047846889952

In [81]: model = RandomForestClassifier(max_depth=6, min_samples_leaf = 1, max_features='auto',n_estimators=100)
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score =  0.8664421997755332
test score =  0.8947368421052632

In [77]: model = LogisticRegression()
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score =  0.8024691358024691
test score =  0.9497607655502392

In [80]: model = GradientBoostingClassifier(max_depth=1,min_samples_leaf=3)
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score =  0.8148148148148148
test score =  0.9784688995215312

In [82]: model = SVC(kernel = 'linear')
model.fit(train_X, train_Y)
print "train score = ", model.score(train_X, train_Y)
print "test score = ", model.score(test_X, test_Y)

train score =  0.7867564534231201
test score =  1.0
```

第十章 面试

机器学习类面试问题集

常规问题

数值的整数次方

■ $a^n = a^{(n-1)/2}a^{(n-1)/2}$, a is even

■ $a^n = a^{(n-1)/2}a^{(n-1)/2}a$, a is odd

```
double Chapter3::Power(double base, int exponent)
{
    if(base <= 0)
    {
        this->bmInvideSetting = false;
        return 0;
    }
    if(exponent == 0)
    {
        return 1;
    }

    if(exponent < 0)
    {
        base = 1.0/base;
        exponent = -exponent;
    }
    double result = Power(base, exponent>>1);
    result *= result;
    if(exponent & 0x1 == 1)
        result *= base;
    return result;
}
```

数组奇偶分类

奇数在前，偶数在后排列，复杂度 $O(n)$

```
void Chapter3::RecordOddEven(int *pData, unsigned int length)
{
    if(pData == NULL || length == 0)
        return;
    int *pStart = pData;
    int *pEnd = pData + length - 1;
    int temp;
    while(pStart < pEnd)
    {
        if(*pStart & 1 == 1)
```

```

    {
        pStart++;
    }
    else
    {

        temp = *pEnd;
        *pEnd = *pStart;
        *pStart = temp;
        pEnd--;
    }
}
}

```

pStart < pEnd比大小，就是比较内存中位置先后。

一次查找链表倒数第k个节点

```

ListNode* Chapter3::FindKthToTail(ListNode* pHeadList, unsigned int k)
{
    if(pHeadList == NULL)
        return NULL;
    ListNode *KthNode = pHeadList;
    int num = 0;

    while(++num<k)
    {
        pHeadList = pHeadList->m_pNext;
        if(pHeadList ==NULL)
            return NULL;
    }
    while(pHeadList != NULL)
    {
        pHeadList = pHeadList->m_pNext;
        if(pHeadList ==NULL)
            return KthNode;
        KthNode = KthNode->m_pNext;

    }
}

```

```

//test FindKthToTail
int _tmain(int argc, _TCHAR* argv[])
{
    Chapter3* chp = new Chapter3();
    ListNode* p;
    ListNode* head = new ListNode();
    p = head;
    for(int n = 0; n < 10; n++)

```

```
{  
    ListNode* node = new ListNode();  
    node->m_value = n;  
    p->m_pNext = node;  
    p = node;  
}  
p->m_pNext = NULL;  
int k_th = 2;  
ListNode *kthNode = chp->FindKthToTail(head, k_th);  
cout << "Last "<< k_th << " is:"<<kthNode->m_value<<endl;  
return 0;  
}
```

C/C++ 面试知识总结

C/C++ 面试知识总结，只为复习、分享。部分知识点与图片来自网络，侵删。

勘误请 Issue、Pull，新增请 Issue，建议、讨论请 # issues/12

使用建议

- `Ctrl + F` : 快速查找定位知识点
- TOC 导航 : 使用 [jawil/GayHub](#) 插件快速目录跳转
- `T` : 按 `T` 激活文件查找器快速查找 / 跳转文件

目录

- [C/C++](#)
- [STL](#)
- [数据结构](#)
- [算法](#)
- [Problems](#)
- [操作系统](#)
- [计算机网络](#)
- [网络编程](#)
- [数据库](#)
- [设计模式](#)
- [链接装载库](#)
- [海量数据处理](#)
- [音视频](#)
- [其他](#)
- [书籍](#)
- [复习刷题网站](#)
- [招聘时间岗位](#)
- [面试题目经验](#)

C/C++

const

作用

1. 修饰变量，说明该变量不可以被改变；
2. 修饰指针，分为指向常量的指针和指针常量；
3. 常量引用，经常用于形参类型，即避免了拷贝，又避免了函数对值的修改；
4. 修饰成员函数，说明该成员函数内不能修改成员变量。

使用

▶ const 使用

static

作用

1. 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在 main 函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。
2. 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命令函数重名，可以将函数定位为 static。
3. 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
4. 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在 static 函数内不能访问非静态成员。

this 指针

1. `this` 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向正在被该成员函数操作的那个对象。
2. 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 `this` 指针，然后调用成员函数，每次成员函数存取数据成员时，由隐含使用 `this` 指针。
3. 当一个成员函数被调用时，自动向它传递一个隐含的参数，该参数是一个指向这个成员函数所在的对象的指针。
4. `this` 指针被隐含地声明为: `ClassName *const this`，这意味着不能给 `this` 指针赋值；在 `ClassName` 类的 `const` 成员函数中，`this` 指针的类型为: `const ClassName* const`，这说明不能对 `this` 指针所指向的这种对象是不可修改的(即不能对这种对象的数据成员进行赋值操作)；
5. `this` 并不是一个常规变量，而是个右值，所以不能取得 `this` 的地址(不能 `&this`)。
6. 在以下场景中，经常需要显式引用 `this` 指针：
 - i. 为实现对象的链式引用；
 - ii. 为避免对同一对象进行赋值操作；

iii. 在实现一些数据结构时，如 `list`。

inline 内联函数

特征

- 相当于把内联函数里面的内容写在调用内联函数处；
- 相当于不用执行进入函数的步骤，直接执行函数体；
- 相当于宏，却比宏多了类型检查，真正具有函数特性；
- 不能包含循环、递归、switch 等复杂操作；
- 类中除了虚函数的其他函数都会自动隐式地当成内联函数。

使用

► inline 使用

编译器对 inline 函数的处理步骤

1. 将 inline 函数体复制到 inline 函数调用点处；
2. 为所用 inline 函数中的局部变量分配内存空间；
3. 将 inline 函数的输入参数和返回值映射到调用方法的局部变量空间中；
4. 如果 inline 函数有多个返回点，将其转变为 inline 函数代码块末尾的分支（使用 GOTO）。

优缺点

优点

1. 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
2. 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
3. 在类中声明同时定义的成员函数，自动转化为内联函数，因此内联函数可以访问类的成员变量，宏定义则不能。
4. 内联函数在运行时可调试，而宏定义不可以。

缺点

1. 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
2. inline 函数无法随着函数库升级而升级。inline 函数的改变需要重新编译，不像 non-inline 可以直接链接。
3. 是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

器。

虚函数(**virtual**)可以是内联函数(**inline**)吗?

Are "inline virtual" member functions ever actually "inlined"?

- 虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。
- 内联是在编译器建议编译器内联，而虚函数的多态性在运行期，编译器无法知道运行期调用哪个代码，因此虚函数表现为多态性时(运行期)不可以内联。
- `inline virtual` 唯一可以内联的时候是：编译器知道所调用的对象是哪个类(如 `Base::who()`)，这只有在编译器具有实际对象而不是对象的指针或引用时才会发生。

► 虚函数内联使用

assert()

断言，是宏，而非函数。`assert` 宏的原型定义在 `<cassert.h>` (C)、`<cassert>` (C++) 中，其作用是如果它的条件返回错误，则终止程序执行。如：

```
assert( p != NULL );
```

sizeof()

- `sizeof` 对数组，得到整个数组所占空间大小。
- `sizeof` 对指针，得到指针本身所占空间大小。

#pragma pack(n)

设定结构体、联合以及类成员变量以 n 字节方式对齐

► #pragma pack(n) 使用

位域

```
Bit mode: 2;      // mode 占 2 位
```

类可以将其(非静态)数据成员定义为位域(bit-field)，在一个位域中含有一定数量的二进制位。当一个程序需要向其他程序或硬件设备传递二进制数据时，通常会用到位域。

- 位域在内存中的布局是与机器有关的
- 位域的类型必须是整型或枚举类型，带符号类型中的位域的行为将因具体实现而定

- 取地址运算符(&)不能作用于位域, 任何指针都无法指向类的位域

volatile

```
volatile int i = 10;
```

- volatile 关键字是一种类型修饰符, 用它声明的类型变量表示可以被某些编译器未知的因素(操作系统、硬件、其它线程等)更改。所以使用 volatile 告诉编译器不应对这样的对象进行优化。
- volatile 关键字声明的变量, 每次访问时都必须从内存中取出值(没有被 volatile 修饰的变量, 可能由于编译器的优化, 从 CPU 寄存器中取值)
- const 可以是 volatile (如只读的状态寄存器)
- 指针可以是 volatile

extern "C"

- 被 extern 限定的函数或变量是 extern 类型的
- 被 `extern "C"` 修饰的变量和函数是按照 C 语言方式编译和连接的

`extern "C"` 的作用是让 C++ 编译器将 `extern "C"` 声明的代码当作 C 语言代码处理, 可以避免 C++ 因符号修饰导致代码不能和 C 语言库中的符号进行链接的问题。

► `extern "C"` 使用

struct 和 `typedef struct`

C 中

```
// c
typedef struct Student {
    int age;
} S;
```

等价于

```
// c
struct Student {
    int age;
};

typedef struct Student S;
```

此时 `s` 等价于 `struct Student`，但两个标识符名称空间不相同。

另外还可以定义与 `struct Student` 不冲突的 `void Student() {}`。

C++ 中

由于编译器定位符号的规则(搜索规则)改变，导致不同于C语言。

一、如果在类标识符空间定义了 `struct Student {...};`，使用 `Student me;` 时，编译器将搜索全局标识符表，`Student` 未找到，则在类标识符内搜索。

即表现为可以使用 `Student` 也可以使用 `struct Student`，如下：

```
// cpp
struct Student {
    int age;
};

void f(Student me);
// 正确, "struct" 关键字可省略
```

二、若定义了与 `Student` 同名函数之后，则 `Student` 只代表函数，不代表结构体，如下：

```
typedef struct Student {
    int age;
} S;

void Student();
// 正确, 定义后 "Student" 只代表此函数

//void S();
// 错误, 符号 "S" 已经被定义为一个 "struct Student" 的别名

int main() {
    Student();
    struct Student me; // 或者 "S me";
    return 0;
}
```

C++ 中 `struct` 和 `class`

总的来说，`struct` 更适合看成是一个数据结构的实现体，`class` 更适合看成是一个对象的实现体。

区别

- 最本质的一个区别就是默认的访问控制
 - 默认的继承访问权限。`struct` 是 `public` 的，`class` 是 `private` 的。
 - `struct` 作为数据结构的实现体，它默认的数据访问控制是 `public` 的，而 `class` 作为对象的实现体，它默认的成员变量访问控制是 `private` 的。

union 联合

联合(union)是一种节省空间的特殊的类,一个union可以有多个数据成员,但是在任意时刻只有一个数据成员可以有值。当某个成员被赋值后其他成员变为未定义状态。联合有如下特点:

- 默认访问控制符为 public
- 可以含有构造函数、析构函数
- 不能含有引用类型的成员
- 不能继承自其他类,不能作为基类
- 不能含有虚函数
- 匿名 union 在定义所在作用域可直接访问 union 成员
- 匿名 union 不能包含 protected 成员或 private 成员
- 全局匿名联合必须是静态(static)的

► union 使用

C 实现 **C++** 类

[C 语言实现封装、继承和多态](#)

explicit(显式)构造函数

explicit 修饰的构造函数可用来防止隐式转换

► explicit 使用

friend 友元类和友元函数

- 能访问私有成员
- 破坏封装性
- 友元关系不可传递
- 友元关系的单向性
- 友元声明的形式及数量不受限制

using

using 声明

一条 using 声明语句一次只引入命名空间的一个成员。它使得我们可以清楚知道程序中所引用的到底是哪个名字。如:

```
using namespace_name::name;
```

构造函数的 using 声明【C++11】

在 C++11 中，派生类能够重用其直接基类定义的构造函数。

```
class Derived : Base {  
public:  
    using Base::Base;  
    /* ... */  
};
```

如上 using 声明，对于基类的每个构造函数，编译器都生成一个与之对应(形参列表完全相同)的派生类构造函数。生成如下类型构造函数：

```
derived parms) : base(args) {}
```

using 指示

using 指示 使得某个特定命名空间中所有名字都可见，这样我们就无需再为它们添加任何前缀限定符了。如：

```
using namespace_name name;
```

尽量少使用 using 指示 污染命名空间

一般说来，使用 using 命令比使用 using 编译命令更安全，这是由于它只导入了制定的名称。如果该名称与局部名称发生冲突，编译器将发出指示。using 编译命令导入所有的名称，包括可能并不需要的名称。如果与局部名称发生冲突，则局部名称将覆盖名称空间版本，而编译器并不会发出警告。另外，名称空间的开放性意味着名称空间的名称可能分散在多个地方，这使得难以准确知道添加了哪些名称。

► using 使用

:: 范围解析运算符

分类

1. 全局作用域符 (::name)：用于类型名称(类、类成员、成员函数、变量等)前，表示作用域为全局命名空间
2. 类作用域符 (class::name)：用于表示指定类型的作用域范围是具体某个类的
3. 命名空间作用域符 (namespace::name)：用于表示指定类型的作用域范围是具体某个命名空间的

► :: 使用

enum 枚举类型

限定作用域的枚举类型

```
enum class open_modes { input, output, append };
```

不限定作用域的枚举类型

```
enum color { red, yellow, green };
enum { floatPrec = 6, doublePrec = 10 };
```

decltype

decltype 关键字用于检查实体的声明类型或表达式的类型及值分类。语法：

```
decltype ( expression )
```

► decltype 使用

引用

左值引用

常规引用，一般表示对象的身份。

右值引用

右值引用就是必须绑定到右值(一个临时对象、将要销毁的对象)的引用，一般表示对象的值。

右值引用可实现转移语义(Move Semantics)和精确传递(Perfect Forwarding)，它的主要目的有两个方面：

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。
- 能够更简洁明确地定义泛型函数。

引用折叠

- `x& &`、`x& &&`、`x&& &` 可折叠成 `x&`

- `x&& &&` 可折叠成 `x&&`

宏

- 宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对“参数”进行的是一对一的替换。

成员初始化列表

好处

- 更高效：少了一次调用默认构造函数的过程。
- 有些场合必须用初始化列表：
 1. 常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面
 2. 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面
 3. 没有默认构造函数的类类型，因为使用初始化列表可以不必调用默认构造函数来初始化，而是直接调用拷贝构造函数初始化。

`initializer_list` 列表初始化【C++11】

用花括号初始化器列表列表初始化一个对象，其中对应构造函数接受一个 `std::initializer_list` 参数。

► `initializer_list` 使用

面向对象

面向对象程序设计(Object-oriented programming, OOP)是种具有对象概念的程序编程典范，同时也是一种程序开发的抽象方针。



面向对象三大特征 —— 封装、继承、多态

封装

- 把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。
- 关键字：public, protected, friendly, private。不写默认为 friendly。

关键字	当前类	包内	子孙类	包外
public	√	√	√	√
protected	√	√	√	✗
friendly	√	√	✗	✗
private	√	✗	✗	✗

继承

- 基类(父类)——> 派生类(子类)

多态

- 多态，即多种状态，在面向对象语言中，接口的多种不同的实现方式即为多态。
- C++ 多态有两种：静态多态(早绑定)、动态多态(晚绑定)。静态多态是通过函数重载实现的；动态多态是通过虚函数实现的。
- 多态是以封装和继承为基础的。

静态多态(早绑定)

函数重载

```
class A
{
public:
    void do(int a);
    void do(int a, int b);
};
```

动态多态(晚绑定)

- 虚函数：用 virtual 修饰成员函数，使其成为虚函数

注意：

- 普通函数(非类成员函数)不能是虚函数
- 静态函数(static)不能是虚函数
- 构造函数不能是虚函数(因为在调用构造函数时，虚表指针并没有在对象的内存空间中，必

须要构造函数调用完成后才会形成虚表指针)

- 内联函数不能是表现多态性时的虚函数, 解释见:[虚函数\(virtual\)可以是内联函数\(inline\)吗?](#)

► 动态多态使用

虚析构函数

虚析构函数是为了解决基类的指针指向派生类对象, 并用基类的指针删除派生类对象。

► 虚析构函数使用

纯虚函数

纯虚函数是一种特殊的虚函数, 在基类中不能对虚函数给出有意义的实现, 而把它声明为纯虚函数, 它的实现留给该基类的派生类去做。

```
virtual int A() = 0;
```

虚函数、纯虚函数

[CSDN . C++ 中的虚函数、纯虚函数区别和联系](#)

- 类里如果声明了虚函数, 这个函数是实现的, 哪怕是空实现, 它的作用就是为了让这个函数在它的子类里面可以被覆盖, 这样的话, 这样编译器就可以使用后期绑定来达到多态了。纯虚函数只是一个接口, 是个函数的声明而已, 它要留到子类里去实现。
- 虚函数在子类里面也可以不重载的; 但纯虚函数必须在子类去实现。
- 虚函数的类用于“实作继承”, 继承接口的同时也继承了父类的实现。当然大家也可以完成自己的实现。纯虚函数关注的是接口的统一性, 实现由子类完成。
- 带纯虚函数的类叫虚基类, 这种基类不能直接生成对象, 而只有被继承, 并重写其虚函数后, 才能使用。这样的类也叫抽象类。抽象类和大家口头常说的虚基类还是有区别的, 在 C# 中用 abstract 定义抽象类, 而在 C++ 中有抽象类的概念, 但是没有这个关键字。抽象类被继承后, 子类可以继续是抽象类, 也可以是普通类, 而虚基类, 是含有纯虚函数的类, 它如果被继承, 那么子类就必须实现虚基类里面的所有纯虚函数, 其子类不能是抽象类。

虚函数指针、虚函数表

- 虚函数指针: 在含有虚函数类的对象中, 指向虚函数表, 在运行时确定。
- 虚函数表: 在程序只读数据段(.rodata section, 见:[目标文件存储结构](#)), 存放虚函数指针, 如果派生类实现了基类的某个虚函数, 则在虚表中覆盖原本基类的那个虚函数指针, 在编译时根据类的声明创建。

虚继承

虚继承用于解决多继承条件下的菱形继承问题(浪费存储空间、存在二义性)。

底层实现原理与编译器相关,一般通过虚基类指针和虚基类表实现,每个虚继承的子类都有一个虚基类指针(占用一个指针的存储空间,4字节)和虚基类表(不占用类对象的存储空间)(需要强调的是,虚基类依旧会在子类里面存在拷贝,只是仅仅最多存在一份而已,并不是不在子类里面了);当虚继承的子类被当做父类继承时,虚基类指针也会被继承。

实际上,vbptr指的是虚基类表指针(virtual base table pointer),该指针指向了一个虚基类表(virtual table),虚表中记录了虚基类与本类的偏移地址;通过偏移地址,这样就找到了虚基类成员,而虚继承也不用像普通多继承那样维持着公共基类(虚基类)的两份同样的拷贝,节省了存储空间。

虚继承、虚函数

- 相同之处:都利用了虚指针(均占用类的存储空间)和虚表(均不占用类的存储空间)
- 不同之处:
 - 虚继承
 - 虚基类依旧存在继承类中,只占用存储空间
 - 虚基类表存储的是虚基类相对直接继承类的偏移
 - 虚函数
 - 虚函数不占用存储空间
 - 虚函数表存储的是虚函数地址

模板类、成员模板、虚函数

- 模板类中可以使用虚函数
- 一个类(无论是普通类还是类模板)的成员模板(本身是模板的成员函数)不能是虚函数

抽象类、接口类、聚合类

- 抽象类:含有纯虚函数的类
- 接口类:仅含有纯虚函数的抽象类
- 聚合类:用户可以直接访问其成员,并且具有特殊的初始化语法形式。满足如下特点:
 - 所有成员都是 public
 - 没有定于任何构造函数
 - 没有类内初始化
 - 没有基类,也没有 virtual 函数

内存分配和管理

malloc、calloc、realloc、alloca

1. malloc: 申请指定字节数的内存。申请到的内存中的初始值不确定。
2. calloc: 为指定长度的对象，分配能容纳其指定个数的内存。申请到的内存的每一位(bit)都初始化为 0。
3. realloc: 更改以前分配的内存长度(增加或减少)。当增加长度时，可能需将以前分配区的内容移到另一个足够大的区域，而新增区域内的初始值则不确定。
4. alloca: 在栈上申请内存。程序在出栈的时候，会自动释放内存。但是需要注意的是，alloca 不具可移植性，而且在没有传统堆栈的机器上很难实现。alloca 不宜使用在必须广泛移植的程序中。C99 中支持变长数组 (VLA)，可以用来替代 alloca。

malloc、free

用于分配、释放内存

► malloc、free 使用

new、delete

1. new / new[]: 完成两件事，先底层调用 malloc 分配内存，然后调用构造函数(创建对象)。
2. delete/delete[]: 也完成两件事，先调用析构函数(清理资源)，然后底层调用 free 释放空间。
3. new 在申请内存时会自动计算所需字节数，而 malloc 则需我们自己输入申请内存空间的字节数。

► new、delete 使用

定位 new

定位 new(placement new) 允许我们向 new 传递额外的参数。

```
new (palce_address) type
new (palce_address) type (initializers)
new (palce_address) type [size]
new (palce_address) type [size] { braced initializer list }
```

- `palce_address` 是个指针
- `initializers` 提供一个(可能为空的)以逗号分隔的初始值列表

delete this 合法吗？

[Is it legal \(and moral\) for a member function to say delete this?](#)

合法，但：

1. 必须保证 `this` 对象是通过 `new` (不是 `new[]`、不是 placement new、不是栈上、不是全局、不是其他对象成员) 分配的
2. 必须保证调用 `delete this` 的成员函数是最后一个调用 `this` 的成员函数
3. 必须保证成员函数的 `delete this` 后面没有调用 `this` 了
4. 必须保证 `delete this` 后没有人使用了

如何定义一个只能在堆上(栈上)生成对象的类?

[如何定义一个只能在堆上\(栈上\)生成对象的类?](#)

只能在堆上

方法: 将析构函数设置为私有

原因: C++ 是静态绑定语言, 编译器管理栈上对象的生命周期, 编译器在为类对象分配栈空间时, 会先检查类的析构函数的访问性。若析构函数不可访问, 则不能在栈上创建对象。

只能在栈上

方法: 将 `new` 和 `delete` 重载为私有

原因: 在堆上生成对象, 使用 `new` 关键词操作, 其过程分为两阶段: 第一阶段, 使用 `new` 在堆上寻找可用内存, 分配给对象; 第二阶段, 调用构造函数生成对象。将 `new` 操作设置为私有, 那么第一阶段就无法完成, 就不能够在堆上生成对象。

智能指针

C++ 标准库(STL)中

头文件: `#include <memory>`

C++ 98

```
std::auto_ptr<std::string> ps (new std::string(str));
```

C++ 11

1. `shared_ptr`
2. `unique_ptr`
3. `weak_ptr`
4. `auto_ptr`(被 C++11 弃用)

5. Class `shared_ptr` 实现共享式拥有(shared ownership)概念。多个智能指针指向相同对象，该对象和其相关资源会在“最后一个 reference 被销毁”时被释放。为了在结构较复杂的情景中执行上述工作，标准库提供 `weak_ptr`、`bad_weak_ptr` 和 `enable_shared_from_this` 等辅助类。
6. Class `unique_ptr` 实现独占式拥有(exclusive ownership)或严格拥有(strict ownership)概念，保证同一时间内只有一个智能指针可以指向该对象。你可以移交拥有权。它对于避免内存泄漏(resource leak)——如 `new` 后忘记 `delete` ——特别有用。

shared_ptr

多个智能指针可以共享同一个对象，对象的最末一个拥有着有责任销毁对象，并清理与该对象相关的所有资源。

- 支持定制型删除器(custom deleter)，可防范 Cross-DLL 问题(对象在动态链接库(DLL)中被 `new` 创建，却在另一个 DLL 内被 `delete` 销毁)、自动解除互斥锁

weak_ptr

`weak_ptr` 允许你共享但不拥有某对象，一旦最末一个拥有该对象的智能指针失去了所有权，任何 `weak_ptr` 都会自动成空(empty)。因此，在 `default` 和 `copy` 构造函数之外，`weak_ptr` 只提供“接受一个 `shared_ptr`”的构造函数。

- 可打破环状引用(cycles of references)，两个其实已经没有被使用的对象彼此互指，使之看似还在“被使用”的状态)的问题

unique_ptr

`unique_ptr` 是 C++11 才开始提供的类型，是一种在异常时可以帮助避免资源泄漏的智能指针。采用独占式拥有，意味着可以确保一个对象和其相应的资源同一时间只被一个 pointer 拥有。一旦拥有着被销毁或编程 `empty`，或开始拥有另一个对象，先前拥有的那个对象就会被销毁，其任何相应资源亦会被释放。

- `unique_ptr` 用于取代 `auto_ptr`

auto_ptr

被 C++11 弃用，原因是缺乏语言特性如“针对构造和赋值”的 `std::move` 语义，以及其他瑕疵。

auto_ptr 与 unique_ptr 比较

- `auto_ptr` 可以赋值拷贝，复制拷贝后所有权转移；`unique_ptr` 无拷贝赋值语义，但实现了 `move` 语义；
- `auto_ptr` 对象不能管理数组(析构调用 `delete`)，`unique_ptr` 可以管理数组(析构调用 `delete[]`)；

强制类型转换运算符

[MSDN . 强制转换运算符](#)

static_cast

- 用于非多态类型的转换
- 不执行运行时类型检查(转换安全性不如 dynamic_cast)
- 通常用于转换数值数据类型(如 float -> int)
- 可以在整个类层次结构中移动指针, 子类转化为父类安全(向上转换), 父类转化为子类不安全(因为子类可能有不在父类的字段或方法)

| 向上转换是一种隐式转换。

dynamic_cast

- 用于多态类型的转换
- 执行行运行时类型检查
- 只适用于指针或引用
- 对不明确的指针的转换将失败(返回 nullptr), 但不引发异常
- 可以在整个类层次结构中移动指针, 包括向上转换、向下转换

const_cast

- 用于删除 const、volatile 和 __unaligned 特性(如将 const int 类型转换为 int 类型)

reinterpret_cast

- 用于位的简单重新解释
- 滥用 reinterpret_cast 运算符可能很容易带来风险。除非所需转换本身是低级别的, 否则应使用其他强制转换运算符之一。
- 允许将任何指针转换为任何其他指针类型(如 char* 到 int* 或 One_class* 到 Unrelated_class* 之类的转换, 但其本身并不安全)
- 也允许将任何整数类型转换为任何指针类型以及反向转换。
- reinterpret_cast 运算符不能丢掉 const、volatile 或 __unaligned 特性。
- reinterpret_cast 的一个实际用途是在哈希函数中, 即, 通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

bad_cast

- 由于强制转换为引用类型失败, dynamic_cast 运算符引发 bad_cast 异常。
- bad_cast 使用

运行时类型信息 (RTTI)

dynamic_cast

- 用于多态类型的转换

typeid

- typeid 运算符允许在运行时确定对象的类型
- type_id 返回一个 type_info 对象的引用
- 如果想通过基类的指针获得派生类的数据类型，基类必须带有虚函数
- 只能获取对象的实际类型

type_info

- type_info 类描述编译器在程序中生成的类型信息。此类的对象可以有效存储指向类型的名称的指针。type_info 类还可存储适合比较两个类型是否相等或比较其排列顺序的编码值。类型的编码规则和排列顺序是未指定的，并且可能因程序而异。
- 头文件：typeinfo

► typeid、type_info 使用

Effective C++

1. 视 C++ 为一个语言联邦(C、Object-Oriented C++、Template C++、STL)
2. 尽量以 `const`、`enum`、`inline` 替换 `#define` (宁可使用编译器替换预处理器)
3. 尽可能使用 `const`
4. 确定对象被使用前已先被初始化(构造时赋值(`copy` 构造函数)比 `default` 构造后赋值(`copy assignment`)效率高)
5. 了解 C++ 默默编写并调用哪些函数(编译器暗自为 class 创建 `default` 构造函数、`copy` 构造函数、`copy assignment` 操作符、析构函数)
6. 若不想使用编译器自动生成的函数，就应该明确拒绝(将不想使用的成员函数声明为 `private`，并且不予实现)
7. 为多态基类声明 `virtual` 析构函数(如果 class 带有任何 `virtual` 函数，它就应该拥有一个 `virtual` 析构函数)
8. 别让异常逃离析构函数(析构函数应该吞下不传播异常，或者结束程序，而不是吐出异常；如果要处理异常应该在非析构的普通函数处理)
9. 绝不在构造和析构过程中调用 `virtual` 函数(因为这类调用从不下降至 derived class)
10. 令 `operator=` 返回一个 `reference to *this` (用于连锁赋值)
11. 在 `operator=` 中处理“自我赋值”
12. 赋值对象时应确保复制“对象内的所有成员变量”及“所有 base class 成分”(调用基类复制构造函数)

13. 以对象管理资源(资源在构造函数获得, 在析构函数释放, 建议使用智能指针, 资源取得时机便是初始化时机(Resource Acquisition Is Initialization, RAII))
14. 在资源管理类中小心 copying 行为(普遍的 RAII class copying 行为是:抑制 copying、引用计数、深度拷贝、转移底部资源拥有权(类似 auto_ptr))
15. 在资源管理类中提供对原始资源(raw resources)的访问(对原始资源的访问可能经过显式转换或隐式转换, 一般而言显示转换比较安全, 隐式转换对客户比较方便)
16. 成对使用 new 和 delete 时要采取相同形式(new 中使用 [] 则 delete [], new 中不使用 [] 则 delete)
17. 以独立语句将 newed 对象存储于(置入)智能指针(如果不这样做, 可能会因为编译器优化, 导致难以察觉的资源泄漏)
18. 让接口容易被正确使用, 不易被误用(促进正常使用的办法:接口的一致性、内置类型的行为兼容;阻止误用的办法:建立新类型, 限制类型上的操作, 约束对象值、消除客户的资源管理责任)
19. 设计 class 犹如设计 type, 需要考虑对象创建、销毁、初始化、赋值、值传递、合法值、继承关系、转换、一般化等等。
20. 宁以 pass-by-reference-to-const 替换 pass-by-value (前者通常更高效、避免切割问题(slicing problem), 但不适用于内置类型、STL迭代器、函数对象)
21. 必须返回对象时, 别妄想返回其 reference(绝不返回 pointer 或 reference 指向一个 local stack 对象, 或返回 reference 指向一个 heap-allocated 对象, 或返回 pointer 或 reference 指向一个 local static 对象而有可能同时需要多个这样的对象。)
22. 将成员变量声明为 private(为了封装、一致性、对其读写精确控制等)
23. 宁以 non-member、non-friend 替换 member 函数(可增加封装性、包裹弹性(packaging flexibility)、机能扩充性)
24. 若所有参数(包括被this指针所指的那个隐喻参数)皆须要类型转换, 请为此采用 non-member 函数
25. 考虑写一个不抛异常的 swap 函数
26. 尽可能延后变量定义式的出现时间(可增加程序清晰度并改善程序效率)
27. 尽量少做转型动作(旧式: (T)expression、T(expression); 新式: const_cast<T>(expression)、dynamic_cast<T>(expression)、reinterpret_cast<T>(expression)、static_cast<T>(expression); 尽量避免转型、注重效率避免 dynamic_casts、尽量设计成无需转型、可把转型封装成函数、宁可用新式转型)
28. 避免使用 handles(包括 引用、指针、迭代器)指向对象内部(以增加封装性、使 const 成员函数的行为更像 const、降低“虚吊号码牌”(dangling handles, 如悬空指针等)的可能性)

Google C++ Style Guide

- ▶ Google C++ Style Guide 图

STL

索引

[STL 方法含义](#)

容器

容器	底层数据结构	有序 无序	可不可重 复	其他
array	数组	无序	可重 复	支持快速随机访问
vector	数组	无序	可重 复	支持快速随机访问
list	双向链表	无序	可重 复	支持快速增删
deque	双端队列(一个中央控制器+多个缓冲区)	无序	可重 复	支持首尾快速增删, 支持随机访问
stack	deque 或 list 封闭头端开口	无序	可重 复	不用 vector 的原因应该是容量大小有限制, 扩容耗时
queue	deque 或 list 封闭底端出口和前端入口	无序	可重 复	不用 vector 的原因应该是容量大小有限制, 扩容耗时
priority_queue	vector	无序	可重 复	vector容器+heap处理规则
set	红黑树	有序	不可重 复	
multiset	红黑树	有序	可重 复	
map	红黑树	有序	不可重 复	
multimap	红黑树	有序	可重 复	
hash_set	hash表	无序	不可重 复	
hash_multiset	hash表	无序	可重 复	
hash_map	hash表	无序	不可重 复	
hash_multimap	hash表	无序	可重 复	

数据结构

顺序结构

顺序栈 (**Sequence Stack**)

[SqStack.cpp](#)

- ▶ 顺序栈数据结构和图片

队列 (**Sequence Queue**)

- ▶ 队列数据结构

非循环队列

- ▶ 非循环队列图片

循环队列

- ▶ 循环队列图片

顺序表 (**Sequence List**)

[SqList.cpp](#)

- ▶ 顺序表数据结构和图片

链式结构

[LinkList.cpp](#)

[LinkList_with_head.cpp](#)

- ▶ 链式数据结构

链队列 (**Link Queue**)

- ▶ 链队列图片

线性表的链式表示

单链表 (**Link List**)

- ▶ 单链表图片

双向链表 (**Du-Link-List**)

► 双向链表图片

循环链表(Cir-Link-List)

► 循环链表图片

哈希表

[HashTable.cpp](#)

概念

哈希函数: $H(key): K \rightarrow D$, $key \in K$

构造方法

- 直接定址法
- 除留余数法
- 数字分析法
- 折叠法
- 平方取中法

冲突处理方法

- 链地址法: key 相同的用单链表链接
- 开放定址法
 - 线性探测法: key 相同 -> 放到 key 的下一个位置, $H_i = (H(key) + i) \% m$
 - 二次探测法: key 相同 -> 放到 $D_i = 1^2, -1^2, \dots, \pm(k)^2, (k \leq m/2)$
 - 随机探测法: $H = (H(key) + \text{伪随机数}) \% m$

线性探测的哈希表数据结构

► 线性探测的哈希表数据结构和图片

递归

概念

函数直接或间接地调用自身

递归与分治

- 分治法
 - 问题的分解
 - 问题规模的分解
- 折半查找(递归)
- 归并查找(递归)
- 快速排序(递归)

递归与迭代

- 迭代:反复利用变量旧值推出新值
- 折半查找(迭代)
- 归并查找(迭代)

广义表

头尾链表存储表示

► 广义表的头尾链表存储表示和图片

扩展线性链表存储表示

► 扩展线性链表存储表示和图片

二叉树

[BinaryTree.cpp](#)

性质

1. 非空二叉树第 i 层最多 $2^{(i-1)}$ 个结点 ($i \geq 1$)
2. 深度为 k 的二叉树最多 $2^k - 1$ 个结点 ($k \geq 1$)
3. 度为 0 的结点数为 n_0 , 度为 2 的结点数为 n_2 , 则 $n_0 = n_2 + 1$
4. 有 n 个结点的完全二叉树深度 $k = \lfloor \log_2(n) \rfloor + 1$
5. 对于含 n 个结点的完全二叉树中编号为 i ($1 \leq i \leq n$) 的结点
 - i. 若 $i = 1$, 为根, 否则双亲为 $\lfloor i / 2 \rfloor$
 - ii. 若 $2i > n$, 则 i 结点没有左孩子, 否则孩子编号为 $2i + 1$
 - iii. 若 $2i + 1 > n$, 则 i 结点没有右孩子, 否则孩子编号为 $2i + 1$

存储结构

► 二叉树数据结构

顺序存储

► 二叉树顺序存储图片

链式存储

► 二叉树链式存储图片

遍历方式

- 先序遍历
- 中序遍历
- 后续遍历
- 层次遍历

分类

- 满二叉树
- 完全二叉树(堆)
 - 大顶堆:根 \geq 左 && 根 \geq 右
 - 小顶堆:根 \leq 左 && 根 \leq 右
- 二叉查找树(二叉排序树):左 $<$ 根 $<$ 右
- 平衡二叉树(AVL树): $|$ 左子树树高 - 右子树树高 $| \leq 1$
- 最小失衡树:平衡二叉树插入新结点导致失衡的子树:调整:
 - LL型:根的左孩子右旋
 - RR型:根的右孩子左旋
 - LR型:根的左孩子左旋, 再右旋
 - RL型:右孩子的左子树, 先右旋, 再左旋

其他树及森林

树的存储结构

- 双亲表示法
- 双亲孩子表示法
- 孩子兄弟表示法

并查集

一种不相交的子集所构成的集合 $S = \{S_1, S_2, \dots, S_n\}$

平衡二叉树(AVL树)

性质

- $| \text{左子树树高} - \text{右子树树高} | \leq 1$
- 平衡二叉树必定是二叉搜索树，反之则不一定
- 最小二叉平衡树的节点的公式： $F(n)=F(n-1)+F(n-2)+1$ (1 是根节点, $F(n-1)$ 是左子树的节点数量, $F(n-2)$ 是右子树的节点数量)

► 平衡二叉树图片

最小失衡树

平衡二叉树插入新结点导致失衡的子树

调整：

- LL 型：根的左孩子右旋
- RR 型：根的右孩子左旋
- LR 型：根的左孩子左旋，再右旋
- RL 型：右孩子的左子树，先右旋，再左旋

红黑树

红黑树的特征是什么？

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色(叶子是 NIL 节点)。
4. 每个红色节点必须有两个黑色的子节点。(从每个叶子到根的所有路径上不能有两个连续的红色节点。)(新增节点的父节点必须相同)
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。(新增节点必须为红)

调整

1. 变色
2. 左旋
3. 右旋

应用

- 关联数组：如 STL 中的 map、set

红黑树、B 树、B+ 树的区别？

- 红黑树的深度比较大，而 B 树和 B+ 树的深度则相对要小一些
- B+ 树则将数据都保存在叶子节点，同时通过链表的形式将他们连接在一起。

B 树 (B-tree)、B+ 树 (B+-tree)

► B 树、B+ 树图片

特点

- 一般化的二叉查找树(binary search tree)
- “矮胖”，内部(非叶子)节点可以拥有可变数量的子节点(数量范围预先定义好)

应用

- 大部分文件系统、数据库系统都采用B树、B+树作为索引结构

区别

- B+树中只有叶子节点会带有指向记录的指针(ROWID)，而B树则所有节点都带有，在内部节点出现的索引项不会再出现在叶子节点中。
- B+树中所有叶子节点都是通过指针连接在一起，而B树不会。

B树的优点

对于在内部节点的数据，可直接得到，不必根据叶子节点来定位。

B+树的优点

- 非叶子节点不会带上 ROWID，这样，一个块中可以容纳更多的索引项，一是可以降低树的高度。二是一个内部节点可以定位更多的叶子节点。
- 叶子节点之间通过指针来连接，范围扫描将十分简单，而对于B树来说，则需要在叶子节点和内部节点不停的往返移动。

| B 树、B+ 树区别来自：[differences-between-b-trees-and-b-trees](#)、[B树和B+树的区别](#)

八叉树

► 八叉树图片

八叉树(octree)，或称八元树，是一种用于描述三维空间(划分空间)的树状数据结构。八叉树的每个节点表示一个正方体的体积元素，每个节点有八个子节点，这八个子节点所表示的体积元素加在一起就等于父节点的体积。一般中心点作为节点的分叉中心。

用途

- 三维计算机图形
- 最邻近搜索

图

算法

排序

排序算法	平均时间复杂度	最差时间复杂度	空间复杂度	数据对象稳定性
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	数组不稳定、链表稳定
插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n \cdot \log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
堆排序	$O(n \cdot \log_2 n)$	$O(n^2)$	$O(1)$	不稳定
归并排序	$O(n \cdot \log_2 n)$	$O(n \cdot \log_2 n)$	$O(1)$	稳定
希尔排序	$O(n \cdot \log^2 n)$	$O(n^2)$	$O(1)$	不稳定
计数排序	$O(n+m)$	$O(n+m)$	$O(n+m)$	稳定
桶排序	$O(n)$	$O(n)$	$O(m)$	稳定
基数排序	$O(k \cdot n)$	$O(n^2)$		稳定

- 均按从小到大排列
- k: 代表数值中的“数位”个数
- n: 代表数据规模
- m: 代表数据的最大值减最小值
- 来自: [wikipedia . 排序算法](#)

查找

查找算法	平均时间复杂度	空间复杂度	查找条件
顺序查找	$O(n)$	$O(1)$	无序或有序
二分查找(折半查找)	$O(\log_2 n)$	$O(1)$	有序
插值查找	$O(\log_2(\log_2 n))$	$O(1)$	有序
斐波那契查找	$O(\log_2 n)$	$O(1)$	有序
哈希查找	$O(1)$	$O(n)$	无序或有序
二叉查找树(二叉搜索树查找)	$O(\log_2 n)$		
红黑树	$O(\log_2 n)$		
2-3树	$O(\log_2 n - \log_3 n)$		
B树/B+树	$O(\log_2 n)$		

图搜索算法



索算法	据结构	间复杂度	复杂度														
BFS 广度优先搜索	邻接矩阵 邻接链表	$O(V)$	$\sqrt{ V }$	$O(V ^2)$	$O(V)$	$\sqrt{ V }$	$+ E $	$ E $	$)$	$O(V)$	$\sqrt{ V }$	$O(V ^2)$	$O(V)$	$\sqrt{ V }$	$+ E $	$ E $	$)$
DFS 深度优先搜索	邻接矩阵 邻接链表	$O(V)$	$\sqrt{ V }$	$O(V ^2)$	$O(V)$	$\sqrt{ V }$	$+ E $	$ E $	$)$	$O(V)$	$\sqrt{ V }$	$O(V ^2)$	$O(V)$	$\sqrt{ V }$	$+ E $	$ E $	$)$

其他算法

算法	思想	应用
分治法	把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并	循环赛日程安排问题、排序算法(快速排序、归并排序)
动态规划	通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法，适用于有重叠子问题和最优子结构性质的问题	背包问题、斐波那契数列
贪心法	一种在每一步选择中都采取在当前状态下最好或最优(即最有利)的选择，从而希望导致结果是最好或最优的算法	旅行推销员问题(最短路径问题)、最小生成树、哈夫曼编码

Problems

Single Problem

- [Chessboard Coverage Problem](#)(棋盘覆盖问题)
- [Knapsack Problem](#)(背包问题)
- [Neumann Neighbor Problem](#)(冯诺依曼邻居问题)
- [Round Robin Problem](#)(循环赛日程安排问题)
- [Tubing Problem](#)(输油管道问题)

Leetcode Problems

- [Github . haoel/leetcode](#)
- [Github . pezy/LeetCode](#)

剑指 Offer

- [Github . zhedahht/CodingInterviewChinese2](#)
- [Github . gatieme/CodingInterviews](#)

Cracking the Coding Interview 程序员面试金典

- [Github . careercup/ctci](#)
- [牛客网 . 程序员面试金典](#)

牛客网

- [牛客网 . 在线编程专题](#)

操作系统

进程与线程

对于有线程系统：

- 进程是资源分配的独立单位
- 线程是资源调度的独立单位

对于无线程系统：

- 进程是资源调度、分配的独立单位

进程之间的通信方式以及优缺点

- 管道(PIPE)
 - 有名管道:一种半双工的通信方式, 它允许无亲缘关系进程间的通信
 - 优点:可以实现任意关系的进程间的通信
 - 缺点:
 1. 长期存于系统中, 使用不当容易出错
 2. 缓冲区有限
 - 无名管道:一种半双工的通信方式, 只能在具有亲缘关系的进程间使用(父子进程)
 - 优点:简单方便

- 缺点:
 - 1. 局限于单向通信
 - 2. 只能创建在它的进程以及其有亲缘关系的进程之间
 - 3. 缓冲区有限
- 信号量(Semaphore):一个计数器,可以用来控制多个线程对共享资源的访问
 - 优点:可以同步进程
 - 缺点:信号量有限
- 信号(Signal):一种比较复杂的通信方式,用于通知接收进程某个事件已经发生
- 消息队列(Message Queue):是消息的链表,存放在内核中并由消息队列标识符标识
 - 优点:可以实现任意进程间的通信,并通过系统调用函数来实现消息发送和接收之间的同步,无需考虑同步问题,方便
 - 缺点:信息的复制需要额外消耗CPU的时间,不适宜于信息量大或操作频繁的场合
- 共享内存(Shared Memory):映射一段能被其他进程所访问的内存,这段共享内存由一个进程创建,但多个进程都可以访问
 - 优点:无须复制,快捷,信息量大
 - 缺点:
 - 1. 通信是通过将共享空间缓冲区直接附加到进程的虚拟地址空间中来实现的,因此进程间的读写操作的同步问题
 - 2. 利用内存缓冲区直接交换信息,内存的实体存在于计算机中,只能同一个计算机系统中的诸多进程共享,不方便网络通信
- 套接字(Socket):可用于不同及其间的进程通信
 - 优点:
 - 1. 传输数据为字节级,传输数据可自定义,数据量小效率高
 - 2. 传输数据时间短,性能高
 - 3. 适合于客户端和服务器端之间信息实时交互
 - 4. 可以加密,数据安全性强
 - 缺点:需对传输的数据进行解析,转化成应用级的数据。

线程之间的通信方式

- 锁机制:包括互斥锁/量(mutex)、读写锁(reader-writer lock)、自旋锁(spin lock)、条件变量(condition)
 - 互斥锁/量(mutex):提供了以排他方式防止数据结构被并发修改的方法。
 - 读写锁(reader-writer lock):允许多个线程同时读共享数据,而对写操作是互斥的。
 - 自旋锁(spin lock)与互斥锁类似,都是为了保护共享资源。互斥锁是当资源被占用,申请者进入睡眠状态;而自旋锁则循环检测保持着是否已经释放锁。
 - 条件变量(condition):可以以原子的方式阻塞进程,直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。
- 信号量机制(Semaphore)
 - 无名线程信号量
 - 命名线程信号量

- 信号机制(Signal):类似进程间的信号处理
- 屏障(barrier):屏障允许每个线程等待,直到所有的合作线程都达到某一点,然后从该点继续执行。

线程间的通信目的主要是用干线程同步,所以线程没有像进程通信中的用于数据交换的通信机制

进程之间的通信方式以及优缺点来源于:[进程线程面试题总结](#)

进程之间私有和共享的资源

- 私有:地址空间、堆、全局变量、栈、寄存器
- 共享:代码段,公共数据,进程目录,进程ID

线程之间私有和共享的资源

- 私有:线程栈,寄存器,程序寄存器
- 共享:堆,地址空间,全局变量,静态变量

多进程与多线程间的对比、优劣与选择

对比

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂,需要用IPC;数据是分开的,同步简单	因为共享进程数据,数据共享简单,但也是因为这个原因导致同步复杂	各有优势
内存、CPU	占用内存多,切换复杂,CPU利用率低	占用内存少,切换简单,CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂,速度慢	创建销毁、切换简单,速度很快	线程占优
编程、调试	编程简单,调试简单	编程复杂,调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布	适应于多核、多机分布式;如果一台	适应于多核分布式	进程

式	机器不够，扩展到多台机器比较简单	适应于多核分布式	占优
---	------------------	----------	----

优劣

优劣	多进程	多线程
优点	编程、调试简单，可靠性较高	创建、销毁、切换速度快，内存、资源占用小
缺点	创建、销毁、切换速度慢，内存、资源占用大	编程、调试复杂，可靠性较差

选择

- 需要频繁创建销毁的优先用线程
- 需要进行大量计算的优先使用线程
- 强相关的处理用线程，弱相关的处理用进程
- 可能要扩展到多机分布的用进程，多核分布的用线程
- 都能满足需求的情况下，用你最熟悉、最拿手的方式

多进程与多线程间的对比、优劣与选择来自：[多线程还是多进程的选择及区别](#)

Linux 内核的同步方式

原因

在现代操作系统里，同一时间可能有多个内核执行流在执行，因此内核其实象多进程多线程编程一样也需要一些同步机制来同步各执行单元对共享数据的访问。尤其是在多处理器系统上，更需要一些同步机制来同步不同处理器上的执行单元对共享的数据的访问。

同步方式

- 原子操作
- 信号量(semaphore)
- 读写信号量(rw_semaphore)
- 自旋锁(spinlock)
- 大内核锁(BKL, Big Kernel Lock)
- 读写锁(rwlock)
- 大读者锁(brlock-Big Reader Lock)
- 读-拷贝修改(RCU, Read-Copy Update)
- 顺序锁(seqlock)

来自：[Linux 内核的同步机制, 第 1 部分](#)、[Linux 内核的同步机制, 第 2 部分](#)

死锁

原因

- 系统资源不足
- 资源分配不当
- 进程运行推进顺序不合适

产生条件

- 互斥
- 请求和保持
- 不剥夺
- 环路

预防

- 打破互斥条件:改造独占性资源为虚拟资源,大部分资源已无法改造。
- 打破不可抢占条件:当一进程占有一独占性资源后又申请一独占性资源而无法满足,则退出原占有的资源。
- 打破占有且申请条件:采用资源预先分配策略,即进程运行前申请全部资源,满足则运行,不然就等待,这样就不会占有且申请。
- 打破循环等待条件:实现资源有序分配策略,对所有设备实现分类编号,所有进程只能采用按序号递增的形式申请资源。
- 有序资源分配法
- 银行家算法

文件系统

- Windows:FCB 表 + FAT + 位图
- Unix:inode + 混合索引 + 成组链接

主机字节序与网络字节序

主机字节序(CPU 字节序)

概念

主机字节序又叫 CPU 字节序,其不是由操作系统决定的,而是由 CPU 指令集架构决定的。主机字节序分为两种:

- 大端字节序(Big Endian):高序字节存储在低位地址,低序字节存储在高位地址

- 小端字节序(Little Endian) : 高序字节存储在高位地址, 低序字节存储在低位地址

存储方式

32位整数 `0x12345678` 是从起始位置为 `0x00` 的地址开始存放, 则:

内存地址	<code>0x00</code>	<code>0x01</code>	<code>0x02</code>	<code>0x03</code>
大端	12	34	56	78
小端	78	56	34	12

▶ 大端小端图片

判断大端小端

▶ 判断大端小端

各架构处理器的字节序

- x86(Intel、AMD)、MOS Technology 6502、Z80、VAX、PDP-11 等处理器为小端序;
- Motorola 6800、Motorola 68000、PowerPC 970、System/370、SPARC(除 V9 外)等处理器为大端序;
- ARM(默认小端序)、PowerPC(除 PowerPC 970 外)、DEC Alpha、SPARC V9、MIPS、PA-RISC 及 IA64 的字节序是可配置的。

网络字节序

网络字节顺序是 TCP/IP 中规定好的一种数据表示格式, 它与具体的 CPU 类型、操作系统等无关, 从而可以保证数据在不同主机之间传输时能够被正确解释。

网络字节顺序采用:大端(Big Endian)排列方式。

页面置换算法

在地址映射过程中, 若在页面中发现所要访问的页面不在内存中, 则产生缺页中断。当发生缺页中断时, 如果操作系统内存中没有空闲页面, 则操作系统必须在内存选择一个页面将其移出内存, 以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

分类

- 全局置换:在整个内存空间置换
- 局部置换:在本进程中进行置换

算法

全局：

- 工作集算法
- 缺页率置换算法

局部：

- 最佳置换算法(OPT)
- 先进先出置换算法(FIFO)
- 最近最久未使用(LRU)算法
- 时钟(Clock)置换算法

计算机网络

计算机经网络体系结构：



各层作用及协议

分层	作用	协议
物理层	通过媒介传输比特，确定机械及电气规范(比特 Bit)	RJ45、CLOCK、IEEE802.3(中继器，集线器)
数据链路层	将比特组装成帧和点到点的传递(帧 Frame)	PPP、FR、HDLC、VLAN、MAC(网桥，交换机)
网络层	负责数据包从源到宿的传递和网际互连(包 Packet)	IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP(路由器)
运输层	提供端到端的可靠报文传递和错误恢复(段Segment)	TCP、UDP、SPX
会话层	建立、管理和终止会话(会话协议数据单元 SPDU)	NFS、SQL、NETBIOS、RPC
表示层	对数据进行翻译、加密和压缩(表示协议数据单元 PPDU)	JPEG、MPEG、ASII
应用层	允许访问OSI环境的手段(应用协议数据单元 APDU)	FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS

物理层

- 传输数据的单位——比特
- 数据传输系统：源系统(源点、发送器) --> 传输系统 --> 目的系统(接收器、终点)

通道：

- 单向通道(单工通道)：只有一个方向通信，没有反方向交互，如广播

- 双向交替通行(半双工通信):通信双方都可发消息,但不能同时发送或接收
- 双向同时通信(全双工通信):通信双方可以同时发送和接收信息

通道复用技术:

- 频分复用(FDM, Frequency Division Multiplexing):不同用户在不同频带,所用用户在同样时间占用不同带宽资源
- 时分复用(TDM, Time Division Multiplexing):不同用户在同一时间段的不同时间片,所有用户在不同时间占用同样的频带宽度
- 波分复用(WDM, Wavelength Division Multiplexing):光的频分复用
- 码分复用(CDM, Code Division Multiplexing):不同用户使用不同的码,可以在同样时间使用同样频带通信

数据链路层

主要信道:

- 点对点信道
- 广播信道

点对点信道

- 数据单元——帧

三个基本问题:

- 封装成帧:把网络层的IP数据报封装成帧,SOH - 数据部分 - EOT
- 透明传输:不管数据部分什么字符,都能传输出去;可以通过字节填充方法解决(冲突字符前加转义字符)
- 差错检测:降低误码率(BER, Bit Error Rate),广泛使用循环冗余检测(CRC, Cyclic Redundancy Check)

点对点协议(Point-to-Point Protocol):

- 点对点协议(Point-to-Point Protocol):用户计算机和ISP通信时所使用的协议

广播信道

广播通信:

- 硬件地址(物理地址、MAC地址)
- 单播(unicast)帧(一对一):收到的帧的MAC地址与本站的硬件地址相同
- 广播(broadcast)帧(一对全体):发送给本局域网上所有站点的帧
- 多播(multicast)帧(一对多):发送给本局域网上一部分站点的帧

网络层

- IP(Internet Protocol, 网际协议)是为计算机网络相互连接进行通信而设计的协议。
- ARP(Address Resolution Protocol, 地址解析协议)
- ICMP(Internet Control Message Protocol, 网际控制报文协议)
- IGMP(Internet Group Management Protocol, 网际组管理协议)

IP 网际协议

IP 地址分类:

- IP 地址 ::= {<网络号>, <主机号>}

IP 地址类别	网络号	网络范围	主机号	IP 地址范围
A 类	8bit, 第一位固定为 0	0 —— 127	24bit	1.0.0.0 —— 127.255.255.255
B 类	16bit, 前两位固定为 10	128.0 —— 191.255	16bit	128.0.0.0 —— 191.255.255.255
C 类	24bit, 前三位固定为 110	192.0.0 —— 223.255.255	8bit	192.0.0.0 —— 223.255.255.255
D 类	前四位固定为 1110, 后面为多播地址			
E 类	前五位固定为 11110, 后面保留为今后所用			

IP 数据报格式:



ICMP 网际控制报文协议

ICMP 报文格式:



应用:

- PING(Packet InterNet Groper, 分组网间探测)测试两个主机之间的连通性
 - TTL(Time To Live, 生存时间)该字段指定 IP 包被路由器丢弃之前允许通过的最大网段数量

内部网关协议

- RIP(Routing Information Protocol, 路由信息协议)

- OSPF(Open Sortest Path First, 开放最短路径优先)

外部网关协议

- BGP(Border Gateway Protocol, 边界网关协议)

IP多播

- IGMP(Internet Group Management Protocol, 网际组管理协议)
- 多播路由选择协议

VPN 和 NAT

- VPN(Virtual Private Network, 虚拟专用网)
- NAT(Network Address Translation, 网络地址转换)

路由表包含什么？

1. 网络 ID(Network ID, Network number) : 就是目标地址的网络 ID。
2. 子网掩码(subnet mask) : 用来判断 IP 所属网络
3. 下一跳地址/接口(Next hop / interface) : 就是数据在发送到目标地址的旅途中下一站的地址。其中 interface 指向 next hop(即为下一个 route)。一个自治系统(AS, Autonomous system)中的 route 应该包含区域内所有的子网络，而默认网关(Network id: 0.0.0.0 , Netmask: 0.0.0.0)指向自治系统的出口。

根据应用和执行的不同，路由表可能含有如下附加信息：

1. 花费(Cost) : 就是数据发送过程中通过路径所需要的花费。
2. 路由的服务质量
3. 路由中需要过滤的出/入连接列表

运输层

协议：

- TCP(Transmission Control Protocol, 传输控制协议)
- UDP(User Datagram Protocol, 用户数据报协议)

端口：

应用程序	FTP	TELNET	SMTP	DNS	TFTP	HTTP	HTTPS	SNMP
端口号	21	23	25	53	69	80	443	161

TCP

- TCP(Transmission Control Protocol, 传输控制协议)是一种面向连接的、可靠的、基于字节流的传输层通信协议，其传输的单位是报文段。

特征：

- 面向连接
- 只能点对点(一对一)通信
- 可靠交互
- 全双工通信
- 面向字节流

TCP 如何保证可靠传输：

- 确认和超时重传
- 数据合理分片和排序
- 流量控制
- 拥塞控制
- 数据校验

TCP 报文结构



TCP 首部



TCP:状态控制码(Code, Control Flag), 占 6 比特, 含义如下:

- URG:紧急比特(urgent), 当 `URG=1` 时, 表明紧急指针字段有效, 代表该封包为紧急封包。它告诉系统此报文段中有紧急数据, 应尽快传送(相当于高优先级的数据), 且上图中的 Urgent Pointer 字段也会被启用。
- ACK:确认比特(Acknowledge)。只有当 `ACK=1` 时确认号字段才有效, 代表这个封包为确认封包。当 `ACK=0` 时, 确认号无效。
- PSH:(Push function)若为 1 时, 代表要求对方立即传送缓冲区内的其他对应封包, 而无需等缓冲满了才送。
- RST:复位比特(Reset), 当 `RST=1` 时, 表明 TCP 连接中出现严重差错(如由于主机崩溃或其他原因), 必须释放连接, 然后再重新建立运输连接。
- SYN:同步比特(Synchronous), SYN 置为 1, 就表示这是一个连接请求或连接接受报文, 通常带有 SYN 标志的封包表示『主动』要连接到对方的意思。
- FIN:终止比特(Final), 用来释放一个连接。当 `FIN=1` 时, 表明此报文段的发送端的数据已发送完毕, 并要求释放运输连接。

UDP

- UDP (User Datagram Protocol, 用户数据报协议) 是 OSI (Open System Interconnection 开放式系统互联) 参考模型中一种无连接的传输层协议, 提供面向事务的简单不可靠信息传送服务, 其传输的单位是用户数据报。

特征:

- 无连接
- 尽最大努力交付
- 面向报文
- 没有拥塞控制
- 支持一对一、一对多、多对一、多对多的交互通信
- 首部开销小

UDP 报文结构



UDP 首部



TCP/UDP 图片来源于: <https://github.com/JerryC8080/understand-tcp-udp>

TCP 与 UDP 的区别

1. TCP 面向连接, UDP 是无连接的;
2. TCP 提供可靠的服务, 也就是说, 通过 TCP 连接传送的数据, 无差错, 不丢失, 不重复, 且按序到达; UDP 尽最大努力交付, 即不保证可靠交付
3. TCP 的逻辑通信信道是全双工的可靠信道; UDP 则是不可靠信道
4. 每一条 TCP 连接只能是点到点的; UDP 支持一对一, 一对多, 多对一和多对多的交互通信
5. TCP 面向字节流(可能出现黏包问题), 实际上是 TCP 把数据看成一连串无结构的字节流; UDP 是面向报文的(不会出现黏包问题)
6. UDP 没有拥塞控制, 因此网络出现拥塞不会使源主机的发送速率降低(对实时应用很有用, 如 IP 电话, 实时视频会议等)
7. TCP 首部开销20字节; UDP 的首部开销小, 只有 8 个字节

TCP 黏包问题

原因

TCP 是一个基于字节流的传输服务(UDP 基于报文的), “流”意味着 TCP 所传输的数据是没有边界的。所以可能会出现两个数据包黏在一起的情况。

解决

- 发送定长包。如果每个消息的大小都是一样的, 那么在接收对等方只要累计接收数据, 直到

数据等于一个定长的数值就将它作为一个消息。

- 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包头长度，依据包头长度来接收包体。
- 在数据包之间设置边界，如添加特殊符号 `\r\n` 标记。FTP 协议正是这么做的。但问题在于如果数据正文中文中也含有 `\r\n`，则会误判为消息的边界。
- 使用更加复杂的应用层协议。

TCP 流量控制

概念

流量控制(flow control)就是让发送方的发送速率不要太快，要让接收方来得及接收。

方法

- ▶ 利用可变窗口进行流量控制

TCP 拥塞控制

概念

拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。

方法

- 慢开始(slow-start)
- 拥塞避免(congestion avoidance)
- 快重传(fast retransmit)
- 快恢复(fast recovery)

- ▶ TCP的拥塞控制图

TCP 传输连接管理

因为 TCP 三次握手建立连接、四次挥手释放连接很重要，所以附上《计算机网络(第 7 版)-谢希仁》书中对此章的详细描述：<https://github.com/huihut/interview/blob/master/images/TCP-transport-connection-management.png>

TCP 三次握手建立连接



【TCP 建立连接全过程解释】

1. 客户端发送 SYN 给服务器，说明客户端请求建立连接；
2. 服务端收到客户端发的 SYN，并回复 SYN+ACK 给客户端(同意建立连接)；

3. 客户端收到服务端的 SYN+ACK 后, 回复 ACK 给服务端(表示客户端收到了服务端发的同意报文);
4. 服务端收到客户端的 ACK, 连接已建立, 可以数据传输。

TCP 为什么要进行三次握手 ?

【答案一】因为信道不可靠, 而 TCP 想在不可靠信道上建立可靠地传输, 那么三次通信是理论上的最小值。(而 UDP 则不需建立可靠传输, 因此 UDP 不需要三次握手。)

Google Groups . TCP 建立连接为什么是三次握手?{技术}{网络通信}

【答案二】因为双方都需要确认对方收到了自己发送的序列号, 确认过程最少要进行三次通信。

知乎 . TCP 为什么是三次握手, 而不是两次或四次?

【答案三】为了防止已失效的连接请求报文段突然又传送到了服务端, 因而产生错误。

《计算机网络(第 7 版)-谢希仁》

TCP 四次挥手释放连接



【TCP 释放连接全过程解释】

1. 客户端发送 FIN 给服务器, 说明客户端不必发送数据给服务器了(请求释放从客户端到服务器的连接);
2. 服务器接收到客户端发的 FIN, 并回复 ACK 给客户端(同意释放从客户端到服务器的连接);
3. 客户端收到服务端回复的 ACK, 此时从客户端到服务器的连接已释放(但服务端到客户端的连接还未释放, 并且客户端还可以接收数据);
4. 服务端继续发送之前没发完的数据给客户端;
5. 服务端发送 FIN+ACK 给客户端, 说明服务端发送完了数据(请求释放从服务端到客户端的连接, 就算没收到客户端的回复, 过段时间也会自动释放);
6. 客户端收到服务端的 FIN+ACK, 并回复 ACK 给客户端(同意释放从服务端到客户端的连接);
7. 服务端收到客户端的 ACK 后, 释放从服务端到客户端的连接。

TCP 为什么要进行四次挥手 ?

【问题一】TCP 为什么要进行四次挥手 ? / 为什么 TCP 建立连接需要三次, 而释放连接则需要四次 ?

【答案一】因为 TCP 是全双工模式, 客户端请求关闭连接后, 客户端向服务端的连接关闭(一二次挥手), 服务端继续传输之前没传完的数据给客户端(数据传输), 服务端向客户端的连接关闭(三四次挥手)。所以 TCP 释放连接时服务器的 ACK 和 FIN 是分开发送的(中间隔着数据传输), 而 TCP 建立连接时服务器的 ACK 和 SYN 是一起发送的(第二次握手), 所以 TCP 建立连接需要三次, 而释放连接则需要四次。

【问题二】为什么 TCP 连接时可以 ACK 和 SYN 一起发送, 而释放时则 ACK 和 FIN 分开发送呢? (ACK 和 FIN 分开是指第二次和第三次挥手)

【答案二】因为客户端请求释放时, 服务器可能还有数据需要传输给客户端, 因此服务端要先响应客户端 FIN 请求(服务端发送 ACK), 然后数据传输, 传输完成后, 服务端再提出 FIN 请求(服务端发送 FIN); 而连接时则没有中间的数据传输, 因此连接时可以 ACK 和 SYN 一起发送。

【问题三】为什么客户端释放最后需要 TIME-WAIT 等待 2MSL 呢?

【答案三】

1. 为了保证客户端发送的最后一个 ACK 报文能够到达服务端。若未成功到达, 则服务端超时重传 FIN+ACK 报文段, 客户端再重传 ACK, 并重新计时。
2. 防止已失效的连接请求报文段出现在本连接中。TIME-WAIT 持续 2MSL 可使本连接持续的时间内所产生的所有报文段都从网络中消失, 这样可使下次连接中不会出现旧的连接报文段。

TCP 有限状态机

► TCP 有限状态机图片

应用层

DNS

- DNS(Domain Name System, 域名系统)是互联网的一项服务。它作为将域名和 IP 地址相互映射的一个分布式数据库, 能够使人更方便地访问互联网。DNS 使用 TCP 和 UDP 端口 53。当前, 对于每一级域名长度的限制是 63 个字符, 域名总长度则不能超过 253 个字符。

域名:

- 域名 ::= {<三级域名>.<二级域名>.<顶级域名>} , 如: blog.huihut.com

FTP

- FTP(File Transfer Protocol, 文件传输协议)是用于在网络上进行文件传输的一套标准协议, 使用客户/服务器模式, 使用 TCP 数据报, 提供交互式访问, 双向传输。
- TFTP(Trivial File Transfer Protocol, 简单文件传输协议)一个小且易实现的文件传输协议, 也使用客户-服务器方式, 使用 UDP 数据报, 只支持文件传输而不支持交互, 没有列目录, 不能对用户进行身份鉴定

TELNET

- TELNET 协议是 TCP/IP 协议族中的一员，是 Internet 远程登陆服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机工作的能力。
- HTTP(HyperText Transfer Protocol, 超文本传输协议)是用于从 WWW(World Wide Web, 万维网)服务器传输超文本到本地浏览器的传送协议。
- SMTP(Simple Mail Transfer Protocol, 简单邮件传输协议)是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。SMTP 协议属于 TCP/IP 协议簇，它帮助每台计算机在发送或中转信件时找到下一个目的地。
- Socket 建立网络通信连接至少要一对端口号(Socket)。Socket 本质是编程接口(API)，对 TCP/IP 的封装，TCP/IP 也要提供可供程序员做网络开发所用的接口，这就是 Socket 编程接口。

WWW

- WWW(World Wide Web, 环球信息网, 万维网)是一个由许多互相链接的超文本组成的系统，通过互联网访问

URL

- URL(Uniform Resource Locator, 统一资源定位符)是因特网上标准的资源的地址(Address)

标准格式：

- 协议类型:[//服务器地址[:端口号]][/资源层级UNIX文件路径]文件名[?查询][#片段ID]

完整格式：

- 协议类型:[//[访问资源需要的凭证信息@]服务器地址[:端口号]][/资源层级UNIX文件路径]文件名[?查询][#片段ID]

其中【访问凭证信息@;端口号;?查询;#片段ID】都属于选填项

如：`https://github.com/huihut/interview#cc`

HTTP

HTTP(HyperText Transfer Protocol, 超文本传输协议)是一种用于分布式、协作式和超媒体信息系统的应用层协议。HTTP 是万维网的数据通信的基础。

请求方法

方法	意义
OPTIONS	请求一些选项信息，允许客户端查看服务器的性能
GET	请求指定的页面信息，并返回实体主体
	类似于 get 请求，只不过返回的响应中没有具体的内容，用于获取报头

	类似于 get 请求, 只不过返回的响应中没有具体的内容, 用于获取报头
POST	向指定资源提交数据进行处理请求(例如提交表单或者上传文件)。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改
PUT	从客户端向服务器传送的数据取代指定的文档的内容
DELETE	请求服务器删除指定的页面
TRACE	回显服务器收到的请求, 主要用于测试或诊断

状态码(Status-Code)

- 1xx: 表示通知信息, 如请求收到了或正在进行处理
 - 100 Continue: 继续, 客户端应继续其请求
 - 101 Switching Protocols 切换协议。服务器根据客户端的请求切换协议。只能切换到更高级的协议, 例如, 切换到 HTTP 的新版本协议
- 2xx: 表示成功, 如接收或知道了
 - 200 OK: 请求成功
- 3xx: 表示重定向, 如要完成请求还必须采取进一步的行动
 - 301 Moved Permanently: 永久移动。请求的资源已被永久的移动到新 URL, 返回信息会包括新的 URL, 浏览器会自动定向到新 URL。今后任何新的请求都应使用新的 URL 代替
- 4xx: 表示客户的差错, 如请求中有错误的语法或不能完成
 - 400 Bad Request: 客户端请求的语法错误, 服务器无法理解
 - 401 Unauthorized: 请求要求用户的身份认证
 - 403 Forbidden: 服务器理解请求客户端的请求, 但是拒绝执行此请求(权限不够)
 - 404 Not Found: 服务器无法根据客户端的请求找到资源(网页)。通过此代码, 网站设计人员可设置“您所请求的资源无法找到”的个性页面
 - 408 Request Timeout: 服务器等待客户端发送的请求时间过长, 超时
- 5xx: 表示服务器的差错, 如服务器失效无法完成请求
 - 500 Internal Server Error: 服务器内部错误, 无法完成请求
 - 503 Service Unavailable: 由于超载或系统维护, 服务器暂时的无法处理客户端的请求。延时的长度可包含在服务器的 Retry-After 头信息中
 - 504 Gateway Timeout: 充当网关或代理的服务器, 未及时从远端服务器获取请求

更多状态码: [菜鸟教程 . HTTP状态码](#)

其他协议

- SMTP(Simple Mail Transfer Protocol, 简单邮件传输协议)是在 Internet 传输 Email 的标准, 是一个相对简单的基于文本的协议。在其之上指定了一条消息的一个或多个接收者(在大多数情况下被确认是存在的), 然后消息文本会被传输。可以很简单地通过 Telnet 程序来测试一个 SMTP 服务器。SMTP 使用 TCP 端口 25。
- DHCP(Dynamic Host Configuration Protocol, 动态主机设置协议)是一个局域网的网络协议, 使用 UDP 协议工作, 主要有两个用途:

- 用于内部网络或网络服务供应商自动分配 IP 地址给用户
- 用于内部网络管理员作为对所有电脑作中央管理的手段
- SNMP(Simple Network Management Protocol, 简单网络管理协议)构成了互联网工程工作小组(IETF, Internet Engineering Task Force)定义的 Internet 协议族的一部分。该协议能够支持网络管理系统, 用以监测连接到网络上的设备是否有任何引起管理上关注的情况。

网络编程

Socket

[Linux Socket 编程\(不限 Linux\)](#)



Socket 中的 `read()`、`write()` 函数

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

`read()`

- `read` 函数是负责从 `fd` 中读取内容。
- 当读成功时, `read` 返回实际所读的字节数。
- 如果返回的值是 0 表示已经读到文件的结束了, 小于 0 表示出现了错误。
- 如果错误为 `EINTR` 说明读是由中断引起的; 如果是 `ECONNRESET` 表示网络连接出了问题。

`write()`

- `write` 函数将 `buf` 中的 `nbytes` 字节内容写入文件描述符 `fd`。
- 成功时返回写的字节数。失败时返回 -1, 并设置 `errno` 变量。
- 在网络程序中, 当我们向套接字文件描述符写时有俩种可能。
 - (1)`write` 的返回值大于 0, 表示写了部分或者是全部的数据。
 - (2)返回的值小于 0, 此时出现了错误。
- 如果错误为 `EINTR` 表示在写的时候出现了中断错误; 如果为 `EPIPE` 表示网络连接出现了问题(对方已经关闭了连接)。

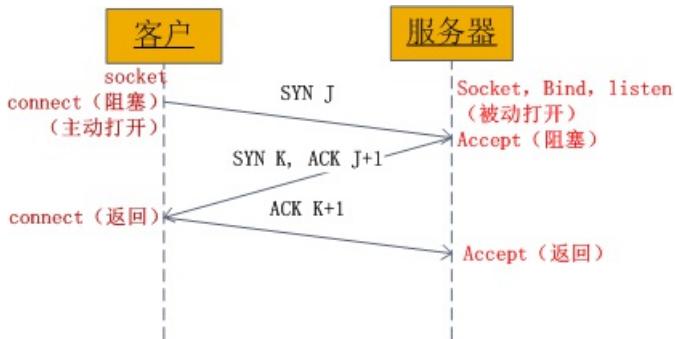
Socket 中 TCP 的三次握手建立连接

我们知道 TCP 建立连接要进行“三次握手”, 即交换三个分组。大致流程如下:

1. 客户端向服务器发送一个 `SYN J`
2. 服务器向客户端响应一个 `SYN K`, 并对 `SYN J` 进行确认 `ACK J+1`

3. 客户端再想服务器发一个确认 ACK K+1

只有就完了三次握手，但是这个三次握手发生在 Socket 的那几个函数中呢？请看下图：

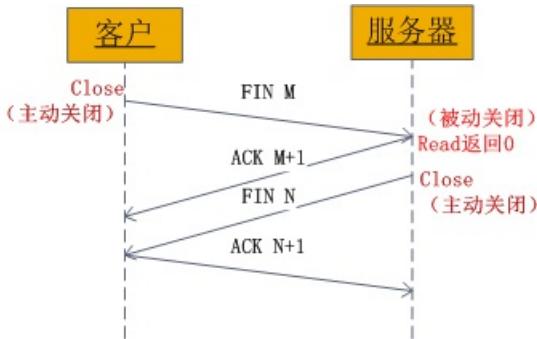


从图中可以看出：

1. 当客户端调用 `connect` 时，触发了连接请求，向服务器发送了 `SYN J` 包，这时 `connect` 进入阻塞状态；
2. 服务器监听到连接请求，即收到 `SYN J` 包，调用 `accept` 函数接收请求向客户端发送 `SYN K`，`ACK J+1`，这时 `accept` 进入阻塞状态；
3. 客户端收到服务器的 `SYN K`，`ACK J+1` 之后，这时 `connect` 返回，并对 `SYN K` 进行确认；
4. 服务器收到 `ACK K+1` 时，`accept` 返回，至此三次握手完毕，连接建立。

Socket 中 TCP 的四次握手释放连接

上面介绍了 socket 中 TCP 的三次握手建立过程，及其涉及的 socket 函数。现在我们介绍 socket 中的四次握手释放连接的过程，请看下图：



图示过程如下：

1. 某个应用进程首先调用 `close` 主动关闭连接，这时 TCP 发送一个 `FIN M`；
2. 另一端接收到 `FIN M` 之后，执行被动关闭，对这个 `FIN` 进行确认。它的接收也作为文件结束符传递给应用进程，因为 `FIN` 的接收意味着应用进程在相应的连接上再也接收不到额外数据；
3. 一段时间之后，接收到文件结束符的应用进程调用 `close` 关闭它的 `socket`。这导致它的 TCP 也发送一个 `FIN N`；
4. 接收到这个 `FIN` 的源发送端 TCP 对它进行确认。

这样每个方向上都有一个 FIN 和 ACK。

数据库

- 数据库事务四大特性:原子性、一致性、分离性、持久性
- 数据库索引:顺序索引、B+ 树索引、hash 索引 [MySQL 索引背后的数据结构及算法原理](#)
- [SQL 约束 \(Constraints\)](#)

范式

- 第一范式(1NF):属性(字段)是最小单位不可再分
- 第二范式(2NF):满足 1NF, 每个非主属性完全依赖于主键(消除 1NF 非主属性对码的部分函数依赖)
- 第三范式(3NF):满足 2NF, 任何非主属性不依赖于其他非主属性(消除 2NF 主属性对码的传递函数依赖)
- 鲍依斯-科得范式(BCNF):满足 3NF, 任何非主属性不能对主键子集依赖(消除 3NF 主属性对码的部分和传递函数依赖)
- 第四范式(4NF):满足 3NF, 属性之间不能有非平凡且非函数依赖的多值依赖(消除 3NF 非平凡且非函数依赖的多值依赖)

设计模式

各大设计模式例子参考:[CSDN专栏 . C++ 设计模式](#) 系列博文

设计模式工程目录

单例模式

[单例模式例子](#)

抽象工厂模式

[抽象工厂模式例子](#)

适配器模式

[适配器模式例子](#)

桥接模式

[桥接模式例子](#)

观察者模式

[观察者模式例子](#)

设计模式的六大原则

- 单一职责原则(SRP, Single Responsibility Principle)
- 里氏替换原则(LSP, Liskov Substitution Principle)
- 依赖倒置原则(DIP, Dependence Inversion Principle)
- 接口隔离原则(ISP, Interface Segregation Principle)
- 迪米特法则(LoD, Law of Demeter)
- 开放封闭原则(OCP, Open Close Principle)

链接装载库

内存、栈、堆

一般应用程序内存空间有如下区域：

- **栈**:由操作系统自动分配释放,存放函数的参数值、局部变量等的值,用于维护函数调用的上下文
- **堆**:一般由程序员分配释放,若程序员不释放,程序结束时可能由操作系统回收,用来容纳应用程序动态分配的内存区域
- **可执行文件映像**:存储着可执行文件在内存中的映像,由装载器装载是将可执行文件的内存读取或映射到这里
- **保留区**:保留区并不是一个单一的内存区域,而是对内存中受到保护而禁止访问的内存区域的总称,如通常C语言讲无效指针赋值为0(NULL),因此0地址正常情况下不可能有效的访问数据

栈

栈保存了一个函数调用所需要的维护信息,常被称为堆栈帧(Stack Frame)或活动记录(Activate Record),一般包含以下几方面:

- 函数的返回地址和参数
- 临时变量:包括函数的非静态局部变量以及编译器自动生成的其他临时变量
- 保存上下文:包括函数调用前后需要保持不变的寄存器

堆

堆分配算法：

- 空闲链表(Free List)
- 位图(Bitmap)
- 对象池

“段错误(segment fault)”或“非法操作，该内存地址不能 read/write”

典型的非法指针解引用造成的错误。当指针指向一个不允许读写的内存地址，而程序却试图利用指针来读或写该地址时，会出现这个错误。

普遍原因：

- 将指针初始化为 NULL，之后没有给它一个合理的值就开始使用指针
- 没用初始化栈中的指针，指针的值一般会是随机数，之后就直接开始使用指针

编译链接

各平台文件格式

平台	可执行文件	目标文件	动态库/共享对象	静态库
Windows	exe	obj	dll	lib
Unix/Linux	ELF、out	o	so	a
Mac	Mach-O	o	dylib、tbd、framework	a、framework

编译链接过程

1. 预编译(预编译器处理如 `#include`、`#define` 等预编译指令，生成 `.i` 或 `.ii` 文件)
2. 编译(编译器进行词法分析、语法分析、语义分析、中间代码生成、目标代码生成、优化，生成 `.s` 文件)
3. 汇编(汇编器把汇编码翻译成机器码，生成 `.o` 文件)
4. 链接(连接器进行地址和空间分配、符号决议、重定位，生成 `.out` 文件)

现在版本 GCC 把预编译和编译合成一步，预编译编译程序 cc1、汇编器 as、连接器 ld

MSVC 编译环境，编译器 cl、连接器 link、可执行文件查看器 dumpbin

目标文件

编译器编译源代码后生成的文件叫做目标文件。目标文件从结构上讲，它是已经编译后的可执行文件格式，只是还没有经过链接的过程，其中可能有些符号或有些地址还没有被调整。

可执行文件(Windows 的 .exe 和 Linux 的 ELF)、动态链接库(Windows 的 .dll 和 Linux 的 .so)、静态链接库(Windows 的 .lib 和 Linux 的 .a)都是按照可执行文件格式存储(Windows 按照 PE-COFF, Linux 按照 ELF)

目标文件格式

- Windows 的 PE(Portable Executable), 或称为 PE-COFF, .obj 格式
- Linux 的 ELF(Executable Linkable Format), .o 格式
- Intel/Microsoft 的 OMF(Object Module Format)
- Unix 的 a.out 格式
- MS-DOS 的 .com 格式

PE 和 ELF 都是 COFF(Common File Format)的变种

目标文件存储结构

段	功能
File Header	文件头, 描述整个文件的文件属性(包括文件是否可执行、是静态链接或动态连接及入口地址、目标硬件、目标操作系统等)
.text section	代码段, 执行语句编译成的机器代码
.data section	数据段, 已初始化的全局变量和局部静态变量
.bss section	BSS 段(Block Started by Symbol), 未初始化的全局变量和局部静态变量(因为默认值为 0, 所以只是在此预留位置, 不占空间)
.rodata section	只读数据段, 存放只读数据, 一般是程序里面的只读变量(如 const 修饰的变量)和字符串常量
.comment section	注释信息段, 存放编译器版本信息
.note.GNU-stack section	堆栈提示段

其他段略

链接的接口——符号

在链接中, 目标文件之间相互拼合实际上是目标文件之间对地址的引用, 即对函数和变量的地址的引用。我们将函数和变量统称为符号(Symbol), 函数名或变量名就是符号名(Symbol Name)。

如下符号表(Symbol Table) :

Symbol(符号名)	Symbol Value (地址)
main	0x100
Add	0x123
...	...

Linux 的共享库 (Shared Library)

Linux 下的共享库就是普通的 ELF 共享对象。

共享库版本更新应该保证二进制接口 ABI(Application Binary Interface)的兼容

命名

`libname.so.x.y.z`

- x: 主版本号, 不同主版本号的库之间不兼容, 需要重新编译
- y: 次版本号, 高版本号向后兼容低版本号
- z: 发布版本号, 不对接口进行更改, 完全兼容

路径

大部分包括 Linux 在内的开源系统遵循 FHS(File Hierarchy Standard)的标准, 这标准规定了系统文件如何存放, 包括各个目录结构、组织和作用。

- `/lib` : 存放系统最关键和最基础的共享库, 如动态链接器、C 语言运行库、数学库等
- `/usr/lib` : 存放非系统运行时所必要的关键性的库, 主要是开发库
- `/usr/local/lib` : 存放跟操作系统本身并不十分相关的库, 主要是一些第三方应用程序的库

动态链接器会在 `/lib`、`/usr/lib` 和由 `/etc/ld.so.conf` 配置文件指定的, 目录中查找共享库

环境变量

- `LD_LIBRARY_PATH` : 临时改变某个应用程序的共享库查找路径, 而不会影响其他应用程序
- `LD_PRELOAD` : 指定预先装载的一些共享库甚至是目标文件
- `LD_DEBUG` : 打开动态链接器的调试功能

so 共享库的编写

▶ 使用 CLion 编写共享库

so 共享库的使用(被可执行项目调用)

▶ 使用 CLion 调用共享库

Windows 应用程序入口函数

- GUI(Graphical User Interface)应用, 链接器选项: `/SUBSYSTEM:WINDOWS`
- CUI(Console User Interface)应用, 链接器选项: `/SUBSYSTEM:CONSOLE`

▶ _tWinMain 与 _tmain 函数声明

应用程序类型	入口点函数	嵌入可执行文件的启动函数
处理ANSI字符(串)的GUI应用程序	_tWinMain(WinMain)	WinMainCRTStartup
处理Unicode字符(串)的GUI应用程序	_tWinMain(wWinMain)	wWinMainCRTStartup
处理ANSI字符(串)的CUI应用程序	_tmain(Main)	mainCRTStartup
处理Unicode字符(串)的CUI应用程序	_tmain(wMain)	wmainCRTStartup
动态链接库(Dynamic-Link Library)	DllMain	_DllMainCRTStartup

Windows 的动态链接库(Dynamic-Link Library)

知识点来自《Windows核心编程(第五版)》

用处

- 扩展了应用程序的特性
- 简化了项目管理
- 有助于节省内存
- 促进了资源的共享
- 促进了本地化
- 有助于解决平台间的差异
- 可以用于特殊目的

注意

- 创建 DLL, 事实上是在创建可供一个可执行模块调用的函数
- 当一个模块提供一个内存分配函数(malloc、new)的时候, 它必须同时提供另一个内存释放函数(free、delete)
- 在使用 C 和 C++ 混编的时候, 要使用 extern "C" 修饰符
- 一个 DLL 可以导出函数、变量(避免导出)、C++ 类(导出导入需要同编译器, 否则避免导出)
- DLL 模块:cpp 文件中的 __declspec(dllexport) 写在 include 头文件之前
- 调用 DLL 的可执行模块:cpp 文件的 __declspec(dllimport) 之前不应该定义 MYLIBAPI

加载 Windows 程序的搜索顺序

1. 包含可执行文件的目录
2. Windows 的系统目录, 可以通过 GetSystemDirectory 得到
3. 16 位的系统目录, 即 Windows 目录中的 System 子目录
4. Windows 目录, 可以通过 GetWindowsDirectory 得到

5. 进程的当前目录
6. PATH 环境变量中所列出的目录

DLL 入口函数

- ▶ DllMain 函数

载入卸载库

- ▶ LoadLibrary、LoadLibraryExA、LoadPackagedLibrary、FreeLibrary、FreeLibraryAndExitThread 函数声明

显示地链接到导出符号

- ▶ GetProcAddress 函数声明

DumpBin.exe 查看 DLL 信息

在 vs 的开发人员命令提示符 使用 DumpBin.exe 可查看 DLL 库的导出段(导出的变量、函数、类名的符号)、相对虚拟地址(RVA, relative virtual address)。如：

```
DUMPBIN -exports D:\mydll.dll
```

LoadLibrary 与 FreeLibrary 流程图

- ▶ LoadLibrary 与 FreeLibrary 流程图

DLL 库的编写(导出一个 DLL 模块)

- ▶ DLL 库的编写(导出一个 DLL 模块)

DLL 库的使用(运行时动态链接 DLL)

- ▶ DLL 库的使用(运行时动态链接 DLL)

运行库(Runtime Library)

典型程序运行步骤

1. 操作系统创建进程，把控制权交给程序的入口(往往是运行库中的某个入口函数)
2. 入口函数对运行库和程序运行环境进行初始化(包括堆、I/O、线程、全局变量构造等等)。

3. 入口函数初始化后，调用 main 函数，正式开始执行程序主体部分。
4. main 函数执行完毕后，返回到入口函数进行清理工作（包括全局变量析构、堆销毁、关闭I/O 等），然后进行系统调用结束进程。

一个程序的 I/O 指代程序与外界的交互，包括文件、管道、网络、命令行、信号等。更广义地讲，I/O 指代操作系统理解为“文件”的事物。

glibc 入口

```
_start -> __libc_start_main -> exit -> _exit
```

其中 `main(argc, argv, __environ)` 函数在 `__libc_start_main` 里执行。

MSVC CRT 入口

```
int mainCRTStartup(void)
```

执行如下操作：

1. 初始化和 OS 版本有关的全局变量。
2. 初始化堆。
3. 初始化 I/O。
4. 获取命令行参数和环境变量。
5. 初始化 C 库的一些数据。
6. 调用 main 并记录返回值。
7. 检查错误并将 main 的返回值返回。

C 语言运行库(CRT)

大致包含如下功能：

- 启动与退出：包括入口函数及入口函数所依赖的其他函数等。
- 标准函数：有 C 语言标准规定的C语言标准库所拥有的函数实现。
- I/O：I/O 功能的封装和实现。
- 堆：堆的封装和实现。
- 语言实现：语言中一些特殊功能的实现。
- 调试：实现调试功能的代码。

C 语言标准库(ANSI C)

包含：

- 标准输入输出(`stdio.h`)
- 文件操作(`stdio.h`)
- 字符操作(`ctype.h`)

- 字符串操作(string.h)
- 数学函数(math.h)
- 资源管理(stdlib.h)
- 格式转换(stdlib.h)
- 时间/日期(time.h)
- 断言(assert.h)
- 各种类型上的常数(limits.h & float.h)
- 变长参数(stdarg.h)
- 非局部跳转(setjmp.h)

海量数据处理

- 海量数据处理面试题集锦
- 十道海量数据处理面试题与十个方法大总结

音视频

- 最全实时音视频开发要用到的开源工程汇总
- 18个实时音视频开发中会用到开源项目

其他

- Bjarne Stroustrup 的常见问题
- Bjarne Stroustrup 的 C++ 风格和技巧常见问题

书籍

语言

- 《C++ Primer》
- 《Effective C++》
- 《More Effective C++》
- 《深度探索 C++ 对象模型》
- 《深入理解 C++11》
- 《STL 源码剖析》

算法

- 《剑指 Offer》
- 《编程珠玑》
- 《程序员面试宝典》

系统

- 《深入理解计算机系统》
- 《Windows 核心编程》
- 《Unix 环境高级编程》

网络

- 《Unix 网络编程》
- 《TCP/IP 详解》

其他

- 《程序员的自我修养》

复习刷题网站

- leetcode
- 牛客网
- 慕课网
- 菜鸟教程

招聘时间岗位

- 牛客网 . 2019 IT名企校招指南

面试题目经验

牛客网

- 牛客网 . 2017秋季校园招聘笔经面经专题汇总
- 牛客网 . 史上最全2017春招面经大合集！！
- 牛客网 . 面试题干货在此

知乎

- 知乎 . 互联网求职路上, 你见过哪些写得很好、很用心的面经?最好能分享自己的面经、心路历程。
- 知乎 . 互联网公司最常见的面试算法题有哪些?
- 知乎 . 面试 C++ 程序员, 什么样的问题是好问题?

CSDN

- CSDN . 全面整理的C++面试题
- CSDN . 百度研发类面试题(C++方向)
- CSDN . c++常见面试题30道
- CSDN . 腾讯2016实习生面试经验(已经拿到offer)

cnblogs

- cnblogs . C++面试集锦(面试被问到的问题)
- cnblogs . C/C++ 笔试、面试题目大汇总
- cnblogs . 常见C++面试题及基本知识点总结(一)

Segmentfault

- segmentfault . C++常见面试问题总结

STL

网站

- [github . huihut/note/STL.md](#)
- [cplusplus . stl](#)
- [cppreference . C++ 参考手册](#)
- [CSDN专栏:STL学习笔记](#)

组成

- 容器(containers)
- 算法(algorithms)
- 迭代器(iterators)
- 仿函数(functors)
- 配接器(adapters)
- 空间配置器(allocator)

容器 (containers)

- 序列式容器(sequence containers) : 元素都是可序(ordered), 但未必是有序(sorted)
- 关联式容器(associative containers)

array

array是固定大小的顺序容器, 它们保存了一个以严格的线性顺序排列的特定数量的元素。

在内部, 一个数组除了它所包含的元素(甚至不是它的大小, 它是一个模板参数, 在编译时是固定的)以外不保存任何数据。存储大小与用语言括号语法([])声明的普通数组一样高效。这个类只是增加了一层成员函数和全局函数, 所以数组可以作为标准容器使用。

与其他标准容器不同, 数组具有固定的大小, 并且不通过分配器管理其元素的分配:它们是封装固定大小数组元素的聚合类型。因此, 他们不能动态地扩大或缩小。

零大小的数组是有效的, 但是它们不应该被解除引用(成员的前面, 后面和数据)。

与标准库中的其他容器不同, 交换两个数组容器是一种线性操作, 它涉及单独交换范围内的所有元素, 这通常是相当低效的操作。另一方面, 这允许迭代器在两个容器中的元素保持其原始容器关联。

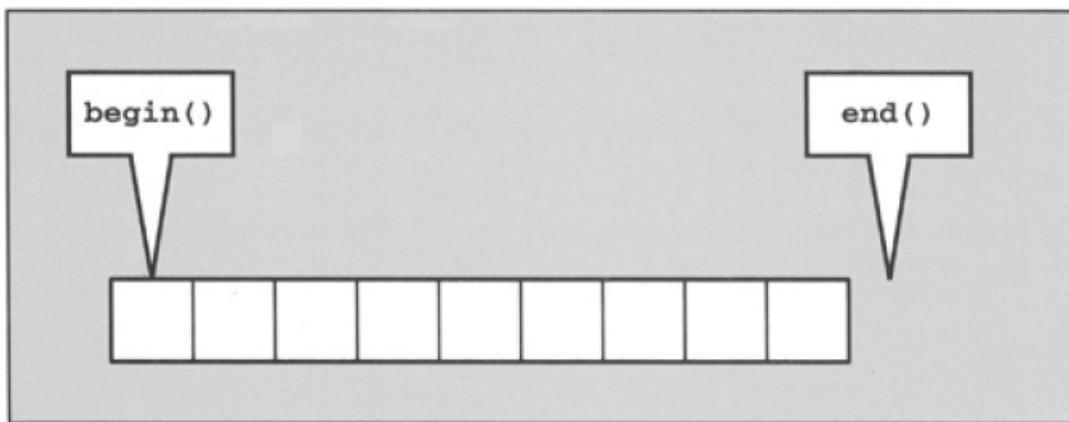
数组容器的另一个独特特性是它们可以被当作元组对象来处理：array头部重载get函数来访问数组元素，就像它是一个元组，以及专门的tuple_size和tuple_element类型。

```
template < class T, size_t N > class array;
```



array::begin

返回指向数组容器中第一个元素的迭代器。



```
iterator begin() noexcept;
const_iterator begin() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main()
{
    std::array<int, 5> myarray = {2, 16, 77, 34, 50};
    std::cout << "myarray contains:";

    for(auto it = myarray.begin(); it != myarray.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::end

返回指向数组容器中最后一个元素之后的理论元素的迭代器。

```
iterator end() noexcept;
const_iterator end() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> myarray = { 5, 19, 77, 34, 99 };

    std::cout << "myarray contains:";
    for ( auto it = myarray.begin(); it != myarray.end(); ++it )
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 5 19 77 34 99
```

array::rbegin

返回指向数组容器中最后一个元素的反向迭代器。

```
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,4> myarray = {4, 26, 80, 14} ;
```

```
    for(auto rit = myarray.rbegin(); rit < myarray.rend(); ++rit)
        std::cout << ' ' << *rit;

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 14 80 26 4
```

array::rend

返回一个反向迭代器，指向数组中第一个元素之前的理论元素(这被认为是它的反向结束)。

```
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,4> myarray = {4, 26, 80, 14};
    std::cout << "myarray contains";
    for(auto rit = myarray.rbegin(); rit < myarray.rend(); ++rit)
        std::cout << ' ' << *rit;

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 14 80 26 4
```

array::cbegin

返回指向数组容器中第一个元素的常量迭代器(const_iterator)；这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

```
const_iterator cbegin() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> myarray = {2, 16, 77, 34, 50};

    std::cout << "myarray contains:";

    for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
        std::cout << ' ' << *it; // cannot modify *it

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::cend

返回指向数组容器中最后一个元素之后的理论元素的常量迭代器 (const_iterator)。这个迭代器可以增加和减少，但是不能用来修改它指向的内容。

```
const_iterator cend() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> myarray = { 15, 720, 801, 1002, 3502 };

    std::cout << "myarray contains:";

    for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
        std::cout << ' ' << *it; // cannot modify *it

    std::cout << '\n';
```

```
    return 0;
}
```

Output

```
myarray contains: 2 16 77 34 50
```

array::crbegin

返回指向数组容器中最后一个元素的常量反向迭代器(const_reverse_iterator)

```
const_reverse_iterator crbegin() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,6> myarray = {10, 20, 30, 40, 50, 60} ;

    std::cout << "myarray backwards:";
    for ( auto rit=myarray.crbegin(); rit < myarray.crend(); ++rit )
        std::cout << ' ' << *rit;    // cannot modify *rit

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray backwards: 60 50 40 30 20 10
```

array::crend

返回指向数组中第一个元素之前的理论元素的常量反向迭代器(const_reverse_iterator), 它被认为与其反向结束。

```
const_reverse_iterator crend() const noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,6> myarray = {10, 20, 30, 40, 50, 60} ;

    std::cout << "myarray backwards:";
    for ( auto rit=myarray.crbegin() ; rit < myarray.crend(); ++rit )
        std::cout << ' ' << *rit; // cannot modify *rit

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray backwards: 60 50 40 30 20 10
```

array::size

返回数组容器中元素的数量。

```
constexpr size_type size() noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> myints;
    std::cout << "size of myints:" << myints.size() << std::endl;
    std::cout << "sizeof(myints):" << sizeof(myints) << std::endl;

    return 0;
}
```

Possible Output

```
size of myints: 5
sizeof(myints): 20
```

array::max_size

返回数组容器可容纳的最大元素数。数组对象的max_size与其size一样，始终等于用于实例化数组模板类的第二个模板参数。

```
constexpr size_type max_size() noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myints;
    std::cout << "size of myints: " << myints.size() << '\n';
    std::cout << "max_size of myints: " << myints.max_size() << '\n';

    return 0;
}
```

Output

```
size of myints: 10
max_size of myints: 10
```

array::empty

返回一个布尔值，指示数组容器是否为空，即它的size()是否为0。

```
constexpr bool empty() noexcept;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,0> first;
    std::array<int,5> second;
    std::cout << "first " << (first.empty() ? "is empty" : "is not empty") << '\n';
    std::cout << "second " << (second.empty() ? "is empty" : "is not empty") << '\n';
    return 0;
}
```

```
}
```

Output:

```
first is empty
second is not empt
```

array::operator[]

返回数组中第n个位置的元素的引用。与array::at相似，但array::at会检查数组边界并通过抛出一个out_of_range异常来判断n是否超出范围，而array::operator[]不检查边界。

```
reference operator[] (size_type n);
const_reference operator[] (size_type n) const;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myarray;
    unsigned int i;

    // assign some values:
    for(i=0; i<10; i++)
        myarray[i] = i;

    // print content
    std::cout << "myarray contains:";
    for(i=0; i<10; i++)
        std::cout << ' ' << myarray[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```

array::at

返回数组中第n个位置的元素的引用。与array::operator[]相似，但array::at会检查数组边界并通过抛出一个out_of_range异常来判断n是否超出范围，而array::operator[]不检查边界。

```
reference at ( size_type n );
const_reference at ( size_type n ) const;
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,10> myarray;
    unsigned int i;

    // assign some values:
    for(i=0; i<10; i++)
        myarray[i] = i;

    // print content
    std::cout << "myarray contains:";
    for(i=0; i<10; i++)
        std::cout << ' ' << myarray[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 0 1 2 3 4 5 6 7 8 9
```

array::front

返回对数组容器中第一个元素的引用。array::begin返回的是迭代器，array::front返回的是直接引用。

在空容器上调用此函数会导致未定义的行为。

```
reference front();
const_reference front() const;
```

Example

```
#include <iostream>
```

```

#include <array>

int main ()
{
    std::array<int,3> myarray = {2, 16, 77};

    std::cout << "front is: " << myarray.front() << std::endl; // 2
    std::cout << "back is: " << myarray.back() << std::endl; // 77

    myarray.front() = 100;

    std::cout << "myarray now contains:";
    for ( int& x : myarray ) std::cout << ' ' << x;

    std::cout << '\n';

    return 0;
}

```

Output

```

front is: 2
back is: 77
myarray now contains: 100 16 77

```

array::back

返回对数组容器中最后一个元素的引用。array::end返回的是迭代器，array::back返回的是直接引用。

在空容器上调用此函数会导致未定义的行为。

```

reference back();
const_reference back() const;

```

Example

```

#include <iostream>
#include <array>

int main ()
{
    std::array<int,3> myarray = {5, 19, 77};

    std::cout << "front is: " << myarray.front() << std::endl; // 5
    std::cout << "back is: " << myarray.back() << std::endl; // 77

    myarray.back() = 50;

```

```

    std::cout << "myarray now contains:";
    for ( int& x : myarray ) std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}

```

Output

```

front is: 5
back is: 77
myarray now contains: 5 19 50

```

array::data

返回指向数组对象中第一个元素的指针。

由于数组中的元素存储在连续的存储位置，所以检索到的指针可以偏移以访问数组中的任何元素。

```

    value_type* data() noexcept;
const value_type* data() const noexcept;

```

Example

```

#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char,12> charray;

    std::memcpy (charray.data(),cstr,12);

    std::cout << charray.data() << '\n';

    return 0;
}

```

Output

```

Test string

```

array::fill

用val填充数组所有元素，将val设置为数组对象中所有元素的值。

```
void fill (const value_type& val);
```

Example

```
#include <iostream>
#include <array>

int main () {
    std::array<int,6> myarray;

    myarray.fill(5);

    std::cout << "myarray contains:";
    for ( int& x : myarray) { std::cout << ' ' << x; }

    std::cout << '\n';

    return 0;
}
```

Output

```
myarray contains: 5 5 5 5 5 5
```

array::swap

通过x的内容交换数组的内容，这是另一个相同类型的数组对象(包括相同的大小)。

与其他容器的交换成员函数不同，此成员函数通过在各个元素之间执行与其大小相同的单独交换操作，以线性时间运行。

```
void swap (array& x) noexcept(noexcept(swap(declval<value_type&>(),declval<value_type&>())));
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> first = {10, 20, 30, 40, 50};
```

```

    std::array<int,5> second = {11, 22, 33, 44, 55};

    first.swap (second);

    std::cout << "first:";

    for (int& x : first) std::cout << ' ' << x;
    std::cout << '\n';

    std::cout << "second:";

    for (int& x : second) std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}

```

Output

```

first: 11 22 33 44 55
second: 10 20 30 40 50

```

get(array)

形如:std::get<0>(myarray);传入一个数组容器, 返回指定位置元素的引用。

```

template <size_t I, class T, size_t N> T& get(array <T, N>&arr)noexcept;
template <size_t I, class T, size_t N> T && get(array <T, N> && arr)noexcept;
template <size_t I, class T, size_t N> const T& get(const array <T, N>&arr)noexcept;

```

Example

```

#include <iostream>
#include <array>
#include <tuple>

int main ()
{
    std::array<int,3> myarray = {10, 20, 30};
    std::tuple<int,int,int> mytuple (10, 20, 30);

    std::tuple_element<0,decltype(myarray)>::type myelement; // int myelement

    myelement = std::get<2>(myarray);
    std::get<2>(myarray) = std::get<0>(myarray);
    std::get<0>(myarray) = myelement;

    std::cout << "first element in myarray: " << std::get<0>(myarray) << "\n";
    std::cout << "first element in mytuple: " << std::get<0>(mytuple) << "\n";
}

```

```
    return 0;
}
```

Output

```
first element in myarray: 30
first element in mytuple: 10
```

relational operators (array)

形如:arrayA != arrayB、arrayA > arrayB;依此比较数组每个元素的大小关系。

```
(1)
template <class T, size_T N>
    bool operator ==(const array <T, N>&lhs, const array <T, N>&rhs);
(2)
template <class T, size_T N>
    bool operator !=(const array <T, N>&lhs, const array <T, N>&rhs);
(3)
template <class T, size_T N>
    bool operator <(const array <T, N>&lhs, const array <T, N>&rhs);
(4)
template <class T, size_T N>
    bool operator <=(const array <T, N>&lhs, const array <T, N>&rhs);
(5)
template <class T, size_T N>
    bool operator >(const array <T, N>&lhs, const array <T, N>&rhs);
(6)
template <class T, size_T N>
    bool operator >=(const array <T, N>&lhs, const array <T, N>&rhs);
```

Example

```
#include <iostream>
#include <array>

int main ()
{
    std::array<int,5> a = {10, 20, 30, 40, 50};
    std::array<int,5> b = {10, 20, 30, 40, 50};
    std::array<int,5> c = {50, 40, 30, 20, 10};

    if (a==b) std::cout << "a and b are equal\n";
    if (b!=c) std::cout << "b and c are not equal\n";
    if (b<c) std::cout << "b is less than c\n";
    if (c>b) std::cout << "c is greater than b\n";
    if (a<=b) std::cout << "a is less than or equal to b\n";
    if (a>=b) std::cout << "a is greater than or equal to b\n";
```

```
    return 0;
}
```

Output

```
a and b are equal
b and c are not equal
b is less than c
c is greater than b
a is less than or equal to b
a is greater than or equal to b
```

vector

vector是表示可以改变大小的数组的序列容器。

就像数组一样, vector为它们的元素使用连续的存储位置, 这意味着它们的元素也可以使用到其元素的常规指针上的偏移来访问, 而且和数组一样高效。但是与数组不同的是, 它们的大小可以动态地改变, 它们的存储由容器自动处理。

在内部, vector使用一个动态分配的数组来存储它们的元素。这个数组可能需要重新分配, 以便在插入新元素时增加大小, 这意味着分配一个新数组并将所有元素移动到其中。就处理时间而言, 这是一个相对昂贵的任务, 因此每次将元素添加到容器时矢量都不会重新分配。

相反, vector容器可以分配一些额外的存储以适应可能的增长, 并且因此容器可以具有比严格需要包含其元素(即, 其大小)的存储更大的实际容量。库可以实现不同的策略的增长到内存使用和重新分配之间的平衡, 但在任何情况下, 再分配应仅在对数生长的间隔发生尺寸, 使得在所述载体的末端各个元件的插入可以与提供分期常量时间复杂性。

因此, 与数组相比, 载体消耗更多的内存来交换管理存储和以有效方式动态增长的能力。

与其他动态序列容器(deques, lists和forward_lists)相比, vector非常有效地访问其元素(就像数组一样), 并相对有效地从元素末尾添加或移除元素。对于涉及插入或移除了结尾之外的位置的元素的操作, 它们执行比其他位置更差的操作, 并且具有比列表和forward_lists更不一致的迭代器和引用。

针对 vector 的各种常见操作的复杂度(效率)如下:

- 随机访问 - 常数 O(1)
- 在尾部增删元素 - 平摊(amortized)常数 O(1)}
- 增删元素 - 至 vector 尾部的线性距离 O(n)}

```
template < class T, class Alloc = allocator<T> > class vector;
```



vector::vector

(1) empty容器构造函数(默认构造函数) 构造一个空的容器, 没有元素。(2) fill构造函数 用n个元素构造一个容器。每个元素都是val的副本(如果提供)。(3) 范围(range)构造器 使用与[range, first, last]范围内的元素相同的顺序构造一个容器, 其中的每个元素都是emplace -从该范围内相应的元素构造而成。(4) 复制(copy)构造函数(并用分配器复制) 按照相同的顺序构造一个包含x中每个元素的副本的容器。(5) 移动(move)构造函数(和分配器移动) 构造一个获取x元素的容器。如果指定了alloc并且与x的分配器不同, 那么元素将被移动。否则, 没有构建元素(他们的所有权直接转移)。x保持未指定但有效的状态。(6) 初始化列表构造函数 构造一个容器中的每个元件中的一个拷贝的IL, 以相同的顺序。

```
default (1)
explicit vector (const allocator_type& alloc = allocator_type());
fill (2)
explicit vector (size_type n);
    vector (size_type n, const value_type& val,
            const allocator_type& alloc = allocator_type());
range (3)
template <class InputIterator>
vector (InputIterator first, InputIterator last,
        const allocator_type& alloc = allocator_type());
copy (4)
vector (const vector& x);
vector (const vector& x, const allocator_type& alloc);
move (5)
vector (vector&& x);
vector (vector&& x, const allocator_type& alloc);
initializer list (6)
vector (initializer_list<value_type> il,
        const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    // constructors used in the same order as described above:
    std::vector<int> first;           // empty vector of ints
    std::vector<int> second(4, 100);   // four ints with value 100
    std::vector<int> third(second.begin(), second.end()); // iterating through second
    std::vector<int> fourth(third);    // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16, 2, 77, 29};
    std::vector<int> fifth(myints, myints + sizeof(myints) / sizeof(int));
```

```
    std::cout << "The contents of fifth are:";  
    for(std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)  
        std::cout << ' ' << *it;  
    std::cout << '\n';  
  
    return 0;  
}
```

Output

```
The contents of fifth are: 16 2 77 29
```

vector::~vector

销毁容器对象。这将在每个包含的元素上调用allocator_traits::destroy，并使用其分配器释放由矢量分配的所有存储容量。

```
~vector();
```

vector::operator=

将新内容分配给容器，替换其当前内容，并相应地修改其大小。

```
copy (1)  
vector& operator= (const vector& x);  
move (2)  
vector& operator= (vector&& x);  
initializer list (3)  
vector& operator= (initializer_list<value_type> il);
```

Example

```
#include <iostream>  
#include <vector>  
  
int main ()  
{  
    std::vector<int> foo (3,0);  
    std::vector<int> bar (5,0);  
  
    bar = foo;  
    foo = std::vector<int>();  
  
    std::cout << "Size of foo: " << int(foo.size()) << '\n';  
    std::cout << "Size of bar: " << int(bar.size()) << '\n';
```

```
    return 0;
}
```

Output

```
Size of foo: 0
Size of bar: 3
```

vector::begin

vector::end

vector::rbegin

vector::rend

vector::cbegin

vector::cend

vector::rbegin

vector::rend

vector::size

返回vector中元素的数量。

这是vector中保存的实际对象的数量，不一定等于其存储容量。

```
size_type size() const noexcept;
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myints;
    std::cout << "0. size: " << myints.size() << '\n';
```

```

    for (int i=0; i<10; i++) myints.push_back(i);
    std::cout << "1. size: " << myints.size() << '\n';

    myints.insert (myints.end(),10,100);
    std::cout << "2. size: " << myints.size() << '\n';

    myints.pop_back();
    std::cout << "3. size: " << myints.size() << '\n';

    return 0;
}

```

Output

```

0. size: 0
1. size: 10
2. size: 20
3. size: 19

```

vector::max_size

返回该vector可容纳的元素的最大数量。由于已知的系统或库实现限制，

这是容器可以达到的最大潜在大小，但容器无法保证能够达到该大小：在达到该大小之前的任何时间，仍然无法分配存储。

```
size_type max_size() const noexcept;
```

Example

```

#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;

    // set some content in the vector:
    for (int i=0; i<100; i++) myvector.push_back(i);

    std::cout << "size: " << myvector.size() << "\n";
    std::cout << "capacity: " << myvector.capacity() << "\n";
    std::cout << "max_size: " << myvector.max_size() << "\n";
    return 0;
}

```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

vector::resize

调整容器的大小，使其包含n个元素。

如果n小于当前的容器size，内容将被缩小到前n个元素，将其删除（并销毁它们）。

如果n大于当前容器size，则通过在末尾插入尽可能多的元素以达到大小n来扩展内容。如果指定了val，则新元素将初始化为val的副本，否则将进行值初始化。

如果n也大于当前的容器的capacity（容量），分配的存储空间将自动重新分配。

注意这个函数通过插入或者删除元素的内容来改变容器的实际内容。

```
void resize (size_type n);
void resize (size_type n, const value_type& val);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;

    // set some initial content:
    for (int i=1;i<10;i++) myvector.push_back(i);

    myvector.resize(5);
    myvector.resize(8,100);
    myvector.resize(12);

    std::cout << "myvector contains:";
    for (int i=0;i<myvector.size();i++)
        std::cout << ' ' << myvector[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0
```

vector::capacity

返回当前为vector分配的存储空间的大小, 用元素表示。这个capacity(容量)不一定等于vector的size。它可以相等或更大, 额外的空间允许适应增长, 而不需要重新分配每个插入。

```
size_type capacity() const noexcept;
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;

    // set some content in the vector:
    for (int i=0; i<100; i++) myvector.push_back(i);

    std::cout << "size: " << (int) myvector.size() << '\n';
    std::cout << "capacity: " << (int) myvector.capacity() << '\n';
    std::cout << "max_size: " << (int) myvector.max_size() << '\n';
    return 0;
}
```

A possible output for this program could be:

```
size: 100
capacity: 128
max_size: 1073741823
```

vector::empty

返回vector是否为空(即, 它的size是否为0)

```
bool empty() const noexcept;
```

Example

```
#include <iostream>
#include <vector>
```

```

int main ()
{
    std::vector<int> myvector;
    int sum (0);

    for (int i=1;i<=10;i++) myvector.push_back(i);

    while (!myvector.empty())
    {
        sum += myvector.back();
        myvector.pop_back();
    }

    std::cout << "total: " << sum << '\n';

    return 0;
}

```

Output

```
total: 55
```

vector::reserve

请求vector容量至少足以包含n个元素。

如果n大于当前vector容量，则该函数使容器重新分配其存储容量，从而将其容量增加到n（或更大）。

在所有其他情况下，函数调用不会导致重新分配，并且vector容量不受影响。

这个函数对vector大小没有影响，也不能改变它的元素。

```
void reserve (size_type n);
```

Example

```

#include <iostream>
#include <vector>

int main ()
{
    std::vector<int>::size_type sz;

    std::vector<int> foo;
    sz = foo.capacity();
    std::cout << "making foo grow:\n";
    for (int i=0; i<100; ++i) {

```

```

foo.push_back(i);
if (sz!=foo.capacity()) {
    sz = foo.capacity();
    std::cout << "capacity changed: " << sz << '\n';
}
}

std::vector<int> bar;
sz = bar.capacity();
bar.reserve(100); // this is the only difference with foo above
std::cout << "making bar grow:\n";
for (int i=0; i<100; ++i) {
    bar.push_back(i);
    if (sz!=bar.capacity()) {
        sz = bar.capacity();
        std::cout << "capacity changed: " << sz << '\n';
    }
}
return 0;
}

```

Possible output

```

making foo grow:
capacity changed: 1
capacity changed: 2
capacity changed: 4
capacity changed: 8
capacity changed: 16
capacity changed: 32
capacity changed: 64
capacity changed: 128
making bar grow:
capacity changed: 100

```

vector::shrink_to_fit

要求容器减小其capacity(容量)以适应其尺寸。

该请求是非绑定的，并且容器实现可以自由地进行优化，并且保持capacity大于其size的vector。这可能导致重新分配，但对矢量大小没有影响，并且不能改变其元素。

```
void shrink_to_fit();
```

Example

```
#include <iostream>
#include <vector>
```

```

int main ()
{
    std::vector<int> myvector (100);
    std::cout << "1. capacity of myvector: " << myvector.capacity() << '\n';

    myvector.resize(10);
    std::cout << "2. capacity of myvector: " << myvector.capacity() << '\n';

    myvector.shrink_to_fit();
    std::cout << "3. capacity of myvector: " << myvector.capacity() << '\n';

    return 0;
}

```

Possible output

```

1. capacity of myvector: 100
2. capacity of myvector: 100
3. capacity of myvector: 10

```

vector::operator[]

vector::at

vector::front

vector::back

vector::data

vector::assign

将新内容分配给vector, 替换其当前内容, 并相应地修改其大小。

在范围版本(1)中, 新内容是从第一个和最后一个范围内的每个元素按相同顺序构造的元素。

在填充版本(2)中, 新内容是n个元素, 每个元素都被初始化为一个val的副本。

在初始化列表版本(3)中, 新内容是以相同顺序作为初始化列表传递的值的副本。

所述内部分配器被用于(通过其性状), 以分配和解除分配存储器如果重新分配发生。它也习惯于摧毁所有现有的元素, 并构建新的元素。

```

range (1)

```

```

template <class InputIterator>
void assign (InputIterator first, InputIterator last);
fill (2)
void assign (size_type n, const value_type& val);
initializer list (3)
void assign (initializer_list<value_type> il);

```

Example

```

#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> first;
    std::vector<int> second;
    std::vector<int> third;

    first.assign (7,100);           // 7 ints with a value of 100

    std::vector<int>::iterator it;
    it=first.begin()+1;

    second.assign (it,first.end()-1); // the 5 central values of first

    int myints[] = {1776,7,4};
    third.assign (myints,myints+3);   // assigning from array.

    std::cout << "Size of first: " << int (first.size()) << '\n';
    std::cout << "Size of second: " << int (second.size()) << '\n';
    std::cout << "Size of third: " << int (third.size()) << '\n';
    return 0;
}

```

Output

```

Size of first: 7
Size of second: 5
Size of third: 3

```

补充:vector::assign 与 vector::operator= 的区别:

1. vector::assign 实现源码

```

void assign(size_type __n, const _Tp& __val) { __M_fill_assign(__n, __val); }

template <class _Tp, class _Alloc>
void vector<_Tp, _Alloc>::__M_fill_assign(size_t __n, const value_type& __val)
{

```

```

    if (__n > capacity()) {
        vector<_Tp, _Alloc> __tmp(__n, __val, get_allocator());
        __tmp.swap(*this);
    }
    else if (__n > size()) {
        fill(begin(), end(), __val);
        _M_finish = uninitialized_fill_n(_M_finish, __n - size(), __val);
    }
    else
        erase(fill_n(begin(), __n, __val), end());
}

```

1. vector::operator= 实现源码

```

template <class _Tp, class _Alloc>
vector<_Tp,_Alloc>&
vector<_Tp,_Alloc>::operator=(const vector<_Tp, _Alloc>& __x)
{
    if (&__x != this) {
        const size_type __xlen = __x.size();
        if (__xlen > capacity()) {
            iterator __tmp = _M_allocate_and_copy(__xlen, __x.begin(), __x.end());
            destroy(_M_start, _M_finish);
            _M_deallocate(_M_start, _M_end_of_storage - _M_start);
            _M_start = __tmp;
            _M_end_of_storage = _M_start + __xlen;
        }
        else if (size() >= __xlen) {
            iterator __i = copy(__x.begin(), __x.end(), begin());
            destroy(__i, _M_finish);
        }
        else {
            copy(__x.begin(), __x.begin() + size(), _M_start);
            uninitialized_copy(__x.begin() + size(), __x.end(), _M_finish);
        }
        _M_finish = _M_start + __xlen;
    }
    return *this;
}

```

vector::push_back

在vector的最后一个元素之后添加一个新元素。val的内容被复制(或移动)到新的元素。

这有效地将容器size增加了一个，如果新的矢量size超过了当前vector的capacity，则导致所分配的存储空间自动重新分配。

```

void push_back (const value_type& val);
void push_back (value_type&& val);

```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;
    int myint;

    std::cout << "Please enter some integers (enter 0 to end):\n";

    do {
        std::cin >> myint;
        myvector.push_back (myint);
    } while (myint);

    std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";

    return 0;
}
```

vector::pop_back

删除vector中的最后一个元素，有效地将容器size减少一个。

这破坏了被删除的元素。

```
void pop_back();
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;
    int sum (0);
    myvector.push_back (100);
    myvector.push_back (200);
    myvector.push_back (300);

    while (!myvector.empty())
    {
        sum+=myvector.back();
        myvector.pop_back();
    }
}
```

```

    }

    std::cout << "The elements of myvector add up to " << sum << '\n';

    return 0;
}

```

Output

```
The elements of myvector add up to 600
```

vector::insert

通过在指定位置的元素之前插入新元素来扩展该vector, 通过插入元素的数量有效地增加容器大小。这会导致分配的存储空间自动重新分配, 只有在新的vector的size超过当前的vector的capacity的情况下。

由于vector使用数组作为其基础存储, 因此除了将元素插入到vector末尾之后, 或vector的begin之前, 其他位置会导致容器重新定位位置之后的所有元素到他们的新位置。与其他种类的序列容器(例如list或forward_list)执行相同操作的操作相比, 这通常是低效的操作。

```

single element (1)
iterator insert (const_iterator position, const value_type& val);
fill (2)
iterator insert (const_iterator position, size_type n, const value_type& val);
range (3)
template <class InputIterator>
iterator insert (const_iterator position, InputIterator first, InputIterator last);
move (4)
iterator insert (const_iterator position, value_type&& val);
initializer list (5)
iterator insert (const_iterator position, initializer_list<value_type> il);

```

Example

```

#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector (3,100);
    std::vector<int>::iterator it;

    it = myvector.begin();
    it = myvector.insert ( it , 200 );

    myvector.insert (it,2,300);
}

```

```

// "it" no longer valid, get a new one:
it = myvector.begin();

std::vector<int> anothervector (2,400);
myvector.insert (it+2,anothervector.begin(),anothervector.end());

int myarray [] = { 501,502,503 };
myvector.insert (myvector.begin(), myarray, myarray+3);

std::cout << "myvector contains:";
for (it=myvector.begin(); it<myvector.end(); it++)
    std::cout << ' ' << *it;
std::cout << '\n';

return 0;
}

```

Output

```
myvector contains: 501 502 503 300 300 400 400 200 100 100 100
```

补充:insert 迭代器野指针错误:

```

int main()
{
    std::vector<int> v(5, 0);
    std::vector<int>::iterator vi;

    // 获取vector第一个元素的迭代器
    vi = v.begin();

    // push_back 插入元素之后可能会因为 push_back 的骚操作(创建一个新vector把旧vector的值复制到新vector), 导致vector迭代器iterator的指针变成野指针, 而导致insert出错
    v.push_back(10);

    v.insert(vi, 2, 300);

    return 0;
}

```

改正:应该把 `vi = v.begin();` 放到 `v.push_back(10);` 后面

vector::erase

从vector中删除单个元素(position)或一系列元素([first, last])。

这有效地减少了被去除的元素的数量, 从而破坏了容器的大小。

由于vector使用一个数组作为其底层存储，所以删除除vector结束位置之后，或vector的begin之前的元素外，将导致容器将段被擦除后的所有元素重新定位到新的位置。与其他种类的序列容器（例如list或forward_list）执行相同操作的操作相比，这通常是低效的操作。

```
iterator erase (const_iterator position);
iterator erase (const_iterator first, const_iterator last);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector;

    // set some values (from 1 to 10)
    for (int i=1; i<=10; i++) myvector.push_back(i);

    // erase the 6th element
    myvector.erase (myvector.begin() + 5);

    // erase the first 3 elements:
    myvector.erase (myvector.begin(), myvector.begin() + 3);

    std::cout << "myvector contains:";
    for (unsigned i=0; i<myvector.size(); ++i)
        std::cout << ' ' << myvector[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
myvector contains: 4 5 7 8 9 10
```

vector::swap

通过x的内容交换容器的内容，x是另一个相同类型的vector对象。尺寸可能不同。

在调用这个成员函数之后，这个容器中的元素是那些在调用之前在x中的元素，而x的元素是在这个元素中的元素。所有迭代器，引用和指针对交换对象保持有效。

请注意，非成员函数存在具有相同名称的交换，并使用与此成员函数相似的优化来重载该算法。

```
void swap (vector& x);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> foo (3,100); // three ints with a value of 100
    std::vector<int> bar (5,200); // five ints with a value of 200

    foo.swap(bar);

    std::cout << "foo contains:";
    for (unsigned i=0; i<foo.size(); i++)
        std::cout << ' ' << foo[i];
    std::cout << '\n';

    std::cout << "bar contains:";
    for (unsigned i=0; i<bar.size(); i++)
        std::cout << ' ' << bar[i];
    std::cout << '\n';

    return 0;
}
```

Output

```
foo contains: 200 200 200 200 200
bar contains: 100 100 100
```

vector::clear

从vector中删除所有的元素(被销毁), 留下size为0的容器。

不保证重新分配, 并且由于调用此函数, vector的capacity不保证发生变化。强制重新分配的典型替代方法是使用swap: `vector<T>().swap(x); // clear x reallocating`

```
void clear() noexcept;
```

Example

```
#include <iostream>
#include <vector>

void printVector(const std::vector<int> &v)
```

```

{
    for (auto it = v.begin(); it != v.end(); ++it)
    {
        std::cout << *it << ' ';
    }
    std::cout << std::endl;
}

int main()
{
    std::vector<int> v1(5, 50);

    printVector(v1);
    std::cout << "v1 size = " << v1.size() << std::endl;
    std::cout << "v1 capacity = " << v1.capacity() << std::endl;

    v1.clear();

    printVector(v1);
    std::cout << "v1 size = " << v1.size() << std::endl;
    std::cout << "v1 capacity = " << v1.capacity() << std::endl;

    v1.push_back(11);
    v1.push_back(22);

    printVector(v1);
    std::cout << "v1 size = " << v1.size() << std::endl;
    std::cout << "v1 capacity = " << v1.capacity() << std::endl;

    return 0;
}

```

Output

```

50 50 50 50 50
v1 size = 5
v1 capacity = 5

v1 size = 0
v1 capacity = 5
11 22
v1 size = 2
v1 capacity = 5

```

vector::emplace

通过在position位置处插入新元素args来扩展容器。这个新元素是用args作为构建的参数来构建的。

这有效地增加了一个容器的大小。

分配存储空间的自动重新分配发生在新的vector的size超过当前向量容量的情况下。

由于vector使用数组作为其基础存储，因此除了将元素插入到vector末尾之后，或vector的begin之前，其他位置会导致容器重新定位位置之后的所有元素到他们的新位置。与其他种类的序列容器（例如list或forward_list）执行相同操作的操作相比，这通常是低效的操作。

该元素是通过调用allocator_traits::construct来转换args来创建的。插入一个类似的成员函数，将现有对象复制或移动到容器中。

```
template <class... Args>
iterator emplace (const_iterator position, Args&&... args);
```

Example

```
#include <iostream>
#include <vector>

int main ()
{
    std::vector<int> myvector = {10,20,30};

    auto it = myvector.emplace ( myvector.begin() + 1, 100 );
    myvector.emplace ( it, 200 );
    myvector.emplace ( myvector.end(), 300 );

    std::cout << "myvector contains:";
    for (auto& x: myvector)
        std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}
```

Output

```
myvector contains: 10 200 100 20 30 300
```

vector::emplace_back

在vector的末尾插入一个新的元素，紧跟在当前的最后一个元素之后。这个新元素是用args作为构造函数的参数来构造的。

这有效地将容器大小增加了一个，如果新的矢量大小超过了当前的vector容量，则导致所分配的存储空间自动重新分配。

该元素是通过调用allocator_traits :: construct来转换args来创建的。

与push_back相比, emplace_back可以避免额外的复制和移动操作。

```
template <class... Args>
void emplace_back (Args&&... args);
```

Example

```
#include <vector>
#include <string>
#include <iostream>

struct President
{
    std::string name;
    std::string country;
    int year;

    President(std::string p_name, std::string p_country, int p_year)
        : name(std::move(p_name)), country(std::move(p_country)), year(p_year)
    {
        std::cout << "I am being constructed.\n";
    }
    President(President&& other)
        : name(std::move(other.name)), country(std::move(other.country)), year(other.year)
    {
        std::cout << "I am being moved.\n";
    }
    President& operator=(const President& other) = default;
};

int main()
{
    std::vector<President> elections;
    std::cout << "emplace_back:\n";
    elections.emplace_back("Nelson Mandela", "South Africa", 1994);

    std::vector<President> reElections;
    std::cout << "\npush_back:\n";
    reElections.push_back(President("Franklin Delano Roosevelt", "the USA", 1936));

    std::cout << "\nContents:\n";
    for (President const& president: elections) {
        std::cout << president.name << " was elected president of "
              << president.country << " in " << president.year << ".\n";
    }
    for (President const& president: reElections) {
        std::cout << president.name << " was re-elected president of "
              << president.country << " in " << president.year << ".\n";
    }
}
```

```
}
```

Output

```
emplace_back:  
I am being constructed.  
  
push_back:  
I am being constructed.  
I am being moved.  
  
Contents:  
Nelson Mandela was elected president of South Africa in 1994.  
Franklin Delano Roosevelt was re-elected president of the USA in 1936.
```

vector::get_allocator

返回与vector关联的构造器对象的副本。

```
allocator_type get_allocator() const noexcept;
```

Example

```
#include <iostream>  
#include <vector>  
  
int main ()  
{  
    std::vector<int> myvector;  
    int * p;  
    unsigned int i;  
  
    // allocate an array with space for 5 elements using vector's allocator:  
    p = myvector.get_allocator().allocate(5);  
  
    // construct values in-place on the array:  
    for (i=0; i<5; i++) myvector.get_allocator().construct(&p[i],i);  
  
    std::cout << "The allocated array contains:";  
    for (i=0; i<5; i++) std::cout << ' ' << p[i];  
    std::cout << '\n';  
  
    // destroy and deallocate:  
    for (i=0; i<5; i++) myvector.get_allocator().destroy(&p[i]);  
    myvector.get_allocator().deallocate(p,5);  
  
    return 0;  
}
```

Output

```
The allocated array contains: 0 1 2 3 4
```

relational operators (vector)

swap (vector)

vector

deque

deque(['deq]) (双端队列) 是double-ended queue 的一个不规则缩写。deque是具有动态大小的序列容器，可以在两端(前端或后端)扩展或收缩。

特定的库可以以不同的方式实现deques，通常作为某种形式的动态数组。但是在任何情况下，它们都允许通过随机访问迭代器直接访问各个元素，通过根据需要扩展和收缩容器来自动处理存储。

因此，它们提供了类似于vector的功能，但是在序列的开始部分也可以高效地插入和删除元素，而不仅仅是在结尾。但是，与vector不同，deques并不保证将其所有元素存储在连续的存储位置：deque通过偏移指向另一个元素的指针访问元素会导致未定义的行为。

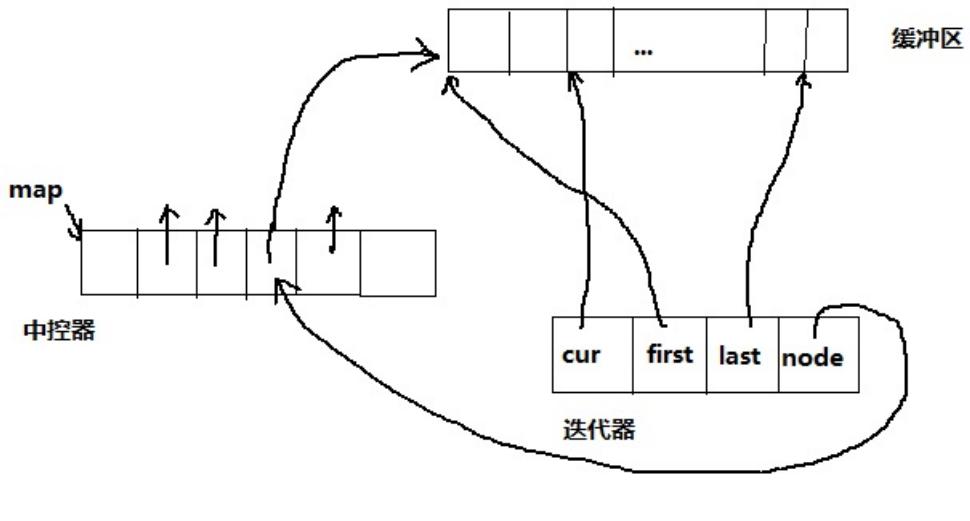
两个vector和deques提供了一个非常相似的接口，可以用于类似的目的，但内部工作方式完全不同：虽然vector使用单个数组需要偶尔重新分配以增长，但是deque的元素可以分散在不同的块的容器，容器在内部保存必要的信息以提供对其任何元素的持续时间和统一的顺序接口（通过迭代器）的直接访问。因此，deques在内部比vector更复杂一点，但是这使得他们在某些情况下更有效地增长，尤其是在重新分配变得更加昂贵的很长序列的情况下。

对于频繁插入或删除开始或结束位置以外的元素的操作，deques表现得更差，并且与列表和转发列表相比，迭代器和引用的一致性更低。

deque上常见操作的复杂性(效率)如下：

- 随机访问 - 常数O(1)
- 在结尾或开头插入或移除元素 - 摊销不变O(1)
- 插入或移除元素 - 线性O(n)

```
template < class T, class Alloc = allocator<T> > class deque;
```



deque::deque

构造一个deque容器对象，根据所使用的构造函数版本初始化它的内容：

Example

```
#include <iostream>
#include <deque>

int main ()
{
    unsigned int i;

    // constructors used in the same order as described above:
    std::deque<int> first;                                // empty deque of ints
    std::deque<int> second (4,100);                         // four ints with value 100
    std::deque<int> third (second.begin(),second.end());   // iterating through second
    std::deque<int> fourth (third);                          // a copy of third

    // the iterator constructor can be used to copy arrays:
    int myints[] = {16,2,77,29};
    std::deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

    std::cout << "The contents of fifth are:";
    for (std::deque<int>::iterator it = fifth.begin(); it!=fifth.end(); ++it)
        std::cout << ' ' << *it;

    std::cout << '\n';

    return 0;
}
```

Output

```
The contents of fifth are: 16 2 77 29
```

deque::push_back

在当前的最后一个元素之后，在deque容器的末尾添加一个新元素。val的内容被复制(或移动)到新的元素。

这有效地增加了一个容器的大小。

```
void push_back (const value_type& val);
void push_back (value_type&& val);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
    std::deque<int> mydeque;
    int myint;

    std::cout << "Please enter some integers (enter 0 to end):\n";

    do {
        std::cin >> myint;
        mydeque.push_back (myint);
    } while (myint);

    std::cout << "mydeque stores " << (int) mydeque.size() << " numbers.\n";

    return 0;
}
```

deque::push_front

在deque容器的开始位置插入一个新的元素，位于当前的第一个元素之前。val的内容被复制(或移动)到插入的元素。

这有效地增加了一个容器的大小。

```
void push_front (const value_type& val);
void push_front (value_type&& val);
```

Example

```

#include <iostream>
#include <deque>

int main ()
{
    std::deque<int> mydeque (2,100);      // two ints with a value of 100
    mydeque.push_front (200);
    mydeque.push_front (300);

    std::cout << "mydeque contains:";
    for (std::deque<int>::iterator it = mydeque.begin(); it != mydeque.end(); ++it)
        std::cout << ' ' << *it;
    std::cout << '\n';

    return 0;
}

```

Output

```
300 200 100 100
```

deque::pop_back

删除deque容器中的最后一个元素，有效地将容器大小减少一个。

这破坏了被删除的元素。

```
void pop_back();
```

Example

```

#include <iostream>
#include <deque>

int main ()
{
    std::deque<int> mydeque;
    int sum (0);
    mydeque.push_back (10);
    mydeque.push_back (20);
    mydeque.push_back (30);

    while (!mydeque.empty())
    {
        sum+=mydeque.back();
        mydeque.pop_back();
    }
}

```

```
    std::cout << "The elements of mydeque add up to " << sum << '\n';

    return 0;
}
```

Output

```
The elements of mydeque add up to 60
```

deque::pop_front

删除deque容器中的第一个元素，有效地减小其大小。

这破坏了被删除的元素。

```
void pop_front();
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
    std::deque<int> mydeque;

    mydeque.push_back (100);
    mydeque.push_back (200);
    mydeque.push_back (300);

    std::cout << "Popping out the elements in mydeque:";
    while (!mydeque.empty())
    {
        std::cout << ' ' << mydeque.front();
        mydeque.pop_front();
    }

    std::cout << "\nThe final size of mydeque is " << int(mydeque.size()) << '\n';

    return 0;
}
```

Output

```
Popping out the elements in mydeque: 100 200 300
The final size of mydeque is 0
```

deque::emplace_front

在deque的开头插入一个新的元素，就在其当前的第一个元素之前。这个新的元素是用args作为构建的参数来构建的。

这有效地增加了一个容器的大小。

该元素是通过调用allocator_traits::construct来转换args来创建的。

存在一个类似的成员函数push_front，它可以将现有对象复制或移动到容器中。

```
template <class... Args>
void emplace_front (Args&&... args);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
    std::deque<int> mydeque = {10, 20, 30};

    mydeque.emplace_front (111);
    mydeque.emplace_front (222);

    std::cout << "mydeque contains:";
    for (auto& x: mydeque)
        std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}
```

Output

```
mydeque contains: 222 111 10 20 30
```

deque::emplace_back

在deque的末尾插入一个新的元素，紧跟在当前的最后一个元素之后。这个新的元素是用args作为构建的参数来构建的。

这有效地增加了一个容器的大小。

该元素是通过调用allocator_traits::construct来转换args来创建的。

存在一个类似的成员函数push_back, 它可以将现有对象复制或移动到容器中

```
template <class... Args>
void emplace_back (Args&&... args);
```

Example

```
#include <iostream>
#include <deque>

int main ()
{
    std::deque<int> mydeque = {10, 20, 30};

    mydeque.emplace_back (100);
    mydeque.emplace_back (200);

    std::cout << "mydeque contains:";
    for (auto& x: mydeque)
        std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}
```

Output

```
mydeque contains: 10 20 30 100 200
```

forward_list

forward_list(单向链表)是序列容器, 允许在序列中的任何地方进行恒定的时间插入和擦除操作。

forward_list(单向链表)被实现为单链表; 单链表可以将它们包含的每个元素存储在不同和不相关的存储位置中。通过关联到序列中下一个元素的链接的每个元素来保留排序。forward_list容器和列表

之间的主要设计区别容器是第一个内部只保留一个到下一个元素的链接, 而后者每个元素保留两个链接:一个指向下一个元素, 一个指向前一个元素, 允许在两个方向上有效的迭代, 但是每个元素消耗额外的存储空间并且插入和移除元件的时间开销略高。因此, forward_list对象比列表对象更有效率, 尽管它们只能向前迭代。

与其他基本的标准序列容器(array, vector和deque), forward_list通常在插入, 提取和移动容器内任何位置的元素方面效果更好, 因此也适用于密集使用这些元素的算法, 如排序算法。

的主要缺点修饰符Modifiers S和列表相比这些其它序列容器s是说，他们缺乏可以通过位置的元素的直接访问；例如，要访问forward_list中的第六个元素，必须从开始位置迭代到该位置，这需要在这些位置之间的线性时间。它们还消耗一些额外的内存来保持与每个元素相关联的链接信息（这可能是大型小元素列表的重要因素）。

该修饰符Modifiers class模板的设计考虑到效率：按照设计，它与简单的手写C型单链表一样高效，实际上是唯一的标准容器，为了效率的考虑故意缺少尺寸成员函数：由于其性质作为一个链表，具有一个需要一定时间的大小的成员将需要它保持一个内部计数器的大小（如列表所示）。这会消耗一些额外的存储空间，并使插入和删除操作效率稍低。要获取forward_list对象的大小，可以使用距离算法的开始和结束，这是一个需要线性时间的操作。



forward_list::forward_list

```
default (1)
explicit forward_list (const allocator_type& alloc = allocator_type());
fill (2)
explicit forward_list (size_type n);
explicit forward_list (size_type n, const value_type& val,
                      const allocator_type& alloc = allocator_type());
range (3)
template <class InputIterator>
forward_list (InputIterator first, InputIterator last,
              const allocator_type& alloc = allocator_type());
copy (4)
forward_list (const forward_list& fwdlst);
forward_list (const forward_list& fwdlst, const allocator_type& alloc);
move (5)
forward_list (forward_list&& fwdlst);
forward_list (forward_list&& fwdlst, const allocator_type& alloc);
initializer list (6)
forward_list (initializer_list<value_type> il,
              const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
    // constructors used in the same order as described above:

    std::forward_list<int> first;                                // default: empty
    std::forward_list<int> second (3,77);                          // fill: 3 seventys
    std::forward_list<int> third (second.begin(), second.end()); // range initialization
    std::forward_list<int> fourth (third);                         // copy constructor
```

```

    std::forward_list<int> fifth (std::move(fourth)); // move ctor. (fourth wasted)
    std::forward_list<int> sixth = {3, 52, 25, 90}; // initializer_list constructor

    std::cout << "first:" ; for (int& x: first) std::cout << " " << x; std::cout << '\n';
    std::cout << "second:"; for (int& x: second) std::cout << " " << x; std::cout << '\n';
    std::cout << "third:"; for (int& x: third) std::cout << " " << x; std::cout << '\n';
    std::cout << "fourth:"; for (int& x: fourth) std::cout << " " << x; std::cout << '\n';
    std::cout << "fifth:"; for (int& x: fifth) std::cout << " " << x; std::cout << '\n';
    std::cout << "sixth:"; for (int& x: sixth) std::cout << " " << x; std::cout << '\n';

    return 0;
}

```

Possible output

```

forward_list constructor examples:
first:
second: 77 77 77
third: 77 77 77
fourth:
fifth: 77 77 77
sixth: 3 52 25 90

```

forward_list::~forward_list

forward_list::before_begin

返回指向容器中第一个元素之前的位置的迭代器。

返回的迭代器不应被解除引用:它是为了用作成员函数的参数`emplace_after`, `insert_after`, `erase_after`或`splice_after`, 指定序列, 其中执行该动作的位置的开始位置。

```

iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;

```

Example

```

#include <iostream>
#include <forward_list>

int main ()
{
    std::forward_list<int> mylist = {20, 30, 40, 50};

    mylist.insert_after ( mylist.before_begin(), 11 );

    std::cout << "mylist contains:" ;

```

```
    for ( int& x: mylist ) std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}
```

Output

```
mylist contains: 11 20 30 40 50
```

forward_list::cbefore_begin

返回指向容器中第一个元素之前的位置的const_iterator。

一个常量性是指向常量内容的迭代器。这个迭代器可以增加和减少(除非它本身也是const)，就像forward_list::before_begin返回的迭代器一样，但不能用来修改它指向的内容。

返回的值不得解除引用。

```
const_iterator cbefore_begin() const noexcept;
```

Example

```
#include <iostream>
#include <forward_list>

int main ()
{
    std::forward_list<int> mylist = {77, 2, 16};

    mylist.insert_after ( mylist.cbefore_begin(), 19 );

    std::cout << "mylist contains:";
    for ( int& x: mylist ) std::cout << ' ' << x;
    std::cout << '\n';

    return 0;
}
```

Output

```
mylist contains: 19 77 2 16
```

list

stack

queue

priority_queue

set

multiset

map

map 是关联容器，按照特定顺序存储由 key value (键值) 和 mapped value (映射值) 组合形成的元素。

在映射中，键值通常用于对元素进行排序和唯一标识，而映射的值存储与此键关联的内容。该类型的键和映射的值可能不同，并且在部件类型被分组在一起VALUE_TYPE，这是一种对类型结合两种：

```
typedef pair<const Key, T> value_type;
```

在内部，映射中的元素总是按照由其内部比较对象(比较类型)指示的特定的严格弱排序标准按键排序。映射容器通常比unordered_map容器慢，以通过它们的键来访问各个元素，但是它们允许基于它们的顺序对子集进行直接迭代。在该映射值地图可以直接通过使用其相应的键来访问括号运算符((操作符[]))。映射通常如实施

```
template < class Key,                                     // map::key_type
          class T,                                         // map::mapped_type
          class Compare = less<Key>,                      // map::key_compare
          class Alloc = allocator<pair<const Key,T> >    // map::allocator_type
        > class map;
```

map::map

构造一个映射容器对象，根据所使用的构造器版本初始化其内容：

(1) 空容器构造函数(默认构造函数)

构造一个空的容器，没有元素。

(2) 范围构造函数

构造具有一样多的元素的范围内的容器[第一, 最后一个), 其中每个元件布设构造的从在该范围内它的相应的元件。

(3) 复制构造函数(并用分配器复制)

使用x中的每个元素的副本构造一个容器。

(4) 移动构造函数(并与分配器一起移动)

构造一个获取x元素的容器。如果指定了alloc并且与x的分配器不同, 那么元素将被移动。否则, 没有构建元素(他们的所有权直接转移)。x保持未指定但有效的状态。

(5) 初始化列表构造函数

用il中的每个元素的副本构造一个容器。

```
empty (1)
explicit map (const key_compare& comp = key_compare(),
              const allocator_type& alloc = allocator_type());
explicit map (const allocator_type& alloc);
range (2)
template <class InputIterator>
map (InputIterator first, InputIterator last,
      const key_compare& comp = key_compare(),
      const allocator_type& alloc = allocator_type());
copy (3)
map (const map& x);
map (const map& x, const allocator_type& alloc);
move (4)
map (map&& x);
map (map&& x, const allocator_type& alloc);
initializer list (5)
map (initializer_list<value_type> il,
      const key_compare& comp = key_compare(),
      const allocator_type& alloc = allocator_type());
```

Example

```
#include <iostream>
#include <map>

bool fncomp (char lhs, char rhs) {return lhs<rhs;}

struct classcomp {
    bool operator() (const char& lhs, const char& rhs) const
    {return lhs<rhs;}
};

int main ()
{
```

```

std::map<char,int> first;

first['a']=10;
first['b']=30;
first['c']=50;
first['d']=70;

std::map<char,int> second (first.begin(),first.end());

std::map<char,int> third (second);

std::map<char,int,classcomp> fourth;           // class as Compare

bool(*fn_pt)(char,char) = fncomp;
std::map<char,int,bool(*)(char,char)> fifth (fn_pt); // function pointer as Compare

return 0;
}

```

map::begin

返回引用map容器中第一个元素的迭代器。

由于map容器始终保持其元素的顺序，所以开始指向遵循容器排序标准的元素。

如果容器是空的，则返回的迭代器值不应被解除引用。

```

iterator begin() noexcept;
const_iterator begin() const noexcept;

```

Example

```

#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;

    mymap['b'] = 100;
    mymap['a'] = 200;
    mymap['c'] = 300;

    // show content:
    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';

    return 0;
}

```

Output

```
a => 200  
b => 100  
c => 300
```

map::key_comp

返回容器用于比较键的比较对象的副本。

```
key_compare key_comp() const;
```

Example

```
#include <iostream>  
#include <map>  
  
int main ()  
{  
    std::map<char,int> mymap;  
  
    std::map<char,int>::key_compare mycomp = mymap.key_comp();  
  
    mymap['a']=100;  
    mymap['b']=200;  
    mymap['c']=300;  
  
    std::cout << "mymap contains:\n";  
  
    char highest = mymap.rbegin()->first;      // key value of last element  
  
    std::map<char,int>::iterator it = mymap.begin();  
    do {  
        std::cout << it->first << " => " << it->second << '\n';  
    } while ( mycomp((*it++).first, highest) );  
  
    std::cout << '\n';  
  
    return 0;  
}
```

Output

```
mymap contains:  
a => 100  
b => 200  
c => 300
```

map::value_comp

返回可用于比较两个元素的比较对象，以获取第一个元素的键是否在第二个元素之前。

```
value_compare value_comp() const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;

    mymap['x']=1001;
    mymap['y']=2002;
    mymap['z']=3003;

    std::cout << "mymap contains:\n";

    std::pair<char,int> highest = *mymap.rbegin();           // last element

    std::map<char,int>::iterator it = mymap.begin();
    do {
        std::cout << it->first << " => " << it->second << '\n';
    } while ( mymap.value_comp()(*it++, highest) );
}

return 0;
}
```

Output

```
mymap contains:
x => 1001
y => 2002
z => 3003
```

map::find

在容器中搜索具有等于k的键的元素，如果找到则返回一个迭代器，否则返回map::end的迭代器。

如果容器的比较对象自反地返回假(即，不管元素作为参数传递的顺序)，则两个key被认为是等同的。

另一个成员函数map::count可以用来检查一个特定的键是否存在。

```
iterator find (const key_type& k);
const_iterator find (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;
    std::map<char,int>::iterator it;

    mymap['a']=50;
    mymap['b']=100;
    mymap['c']=150;
    mymap['d']=200;

    it = mymap.find('b');
    if (it != mymap.end())
        mymap.erase (it);

    // print content:
    std::cout << "elements in mymap:" << '\n';
    std::cout << "a => " << mymap.find('a')->second << '\n';
    std::cout << "c => " << mymap.find('c')->second << '\n';
    std::cout << "d => " << mymap.find('d')->second << '\n';

    return 0;
}
```

Output

```
elements in mymap:
a => 50
c => 150
d => 200
```

map::count

在容器中搜索具有等于k的键的元素，并返回匹配的数量。

由于地图容器中的所有元素都是唯一的，因此该函数只能返回1(如果找到该元素)或返回零(否则)。

如果容器的比较对象自反地返回错误(即，不管按键作为参数传递的顺序)，则两个键被认为是等同的。

```
size_type count (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;
    char c;

    mymap ['a']=101;
    mymap ['c']=202;
    mymap ['f']=303;

    for (c='a'; c<'h'; c++)
    {
        std::cout << c;
        if (mymap.count(c)>0)
            std::cout << " is an element of mymap.\n";
        else
            std::cout << " is not an element of mymap.\n";
    }

    return 0;
}
```

Output

```
a is an element of mymap.
b is not an element of mymap.
c is an element of mymap.
d is not an element of mymap.
e is not an element of mymap.
f is an element of mymap.
g is not an element of mymap.
```

map::lower_bound

将迭代器返回到下限

返回指向容器中第一个元素的迭代器，该元素的键不会在k之前出现(即，它是等价的或者在其后)。

该函数使用其内部比较对象(key_comp)来确定这一点，将迭代器返回到key_comp(element_key, k)将返回false的第一个元素。

如果map类用默认的比较类型(less)实例化，则函数返回一个迭代器到第一个元素，其键不小于k。

一个类似的成员函数upper_bound具有相同的行为lower_bound，除非映射包含一个key值等于k的元素：在这种情况下，lower_bound返回指向该元素的迭代器，而upper_bound返回指向下一个元素的迭代器。

```
iterator lower_bound (const key_type& k);
const_iterator lower_bound (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;
    std::map<char,int>::iterator itlow,itup;

    mymap['a']=20;
    mymap['b']=40;
    mymap['c']=60;
    mymap['d']=80;
    mymap['e']=100;

    itlow=mymap.lower_bound ('b'); // itlow points to b
    itup=mymap.upper_bound ('d'); // itup points to e (not d!)

    mymap.erase(itlow,itup);      // erases [itlow,itup)

    // print content:
    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';

    return 0;
}
```

Output

```
a => 20
e => 100
```

map::upper_bound

将迭代器返回到上限

返回一个指向容器中第一个元素的迭代器，它的关键字被认为是在k之后。

该函数使用其内部比较对象(key_comp)来确定这一点，将迭代器返回到key_comp(k, element_key)将返回true的第一个元素。

如果map类用默认的比较类型(less)实例化，则函数返回一个迭代器到第一个元素，其键大于k。

类似的成员函数lower_bound具有与upper_bound相同的行为，除了map包含一个元素，其键值等于k：在这种情况下，lower_bound返回指向该元素的迭代器，而upper_bound返回指向下一个元素的迭代器。

```
iterator upper_bound (const key_type& k);
const_iterator upper_bound (const key_type& k) const;
```

Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char,int> mymap;
    std::map<char,int>::iterator itlow,itup;

    mymap['a']=20;
    mymap['b']=40;
    mymap['c']=60;
    mymap['d']=80;
    mymap['e']=100;

    itlow=mymap.lower_bound ('b'); // itlow points to b
    itup=mymap.upper_bound ('d'); // itup points to e (not d!)

    mymap.erase(itlow,itup);      // erases [itlow,itup)

    // print content:
    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
        std::cout << it->first << " => " << it->second << '\n';

    return 0;
}
```

Output

```
a => 20
e => 100
```

map::equal_range

获取相同元素的范围

返回包含容器中所有具有与k等价的键的元素的范围边界。由于地图容器中的元素具有唯一键，所以返回的范围最多只包含一个元素。

如果没有找到匹配，则返回的范围具有零的长度，与两个迭代器指向具有考虑去后一个密钥对所述第一元件k根据容器的内部比较对象(key_comp)。如果容器的比较对象返回false，则两个键被认为是等价的。

```
pair<const_iterator, const_iterator> equal_range (const key_type& k) const;
pair<iterator, iterator> equal_range (const key_type& k);
```

Example

```
#include <iostream>
#include <map>

int main ()
{
    std::map<char, int> mymap;

    mymap['a']=10;
    mymap['b']=20;
    mymap['c']=30;

    std::pair<std::map<char, int>::iterator, std::map<char, int>::iterator> ret;
    ret = mymap.equal_range('b');

    std::cout << "lower bound points to: ";
    std::cout << ret.first->first << " => " << ret.first->second << '\n';

    std::cout << "upper bound points to: ";
    std::cout << ret.second->first << " => " << ret.second->second << '\n';

    return 0;
}
```

Output

```
lower bound points to: 'b' => 20
upper bound points to: 'c' => 30
```

multimap

无序容器 (Unordered Container) : `unordered_set`、 `unordered_multiset`、`unordered_map`、`unordered_multimap`

包括：

- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

都是以哈希表实现的。



`unordered_set`、`unodered_multiset`结构：



`unordered_map`、`unodered_multimap`结构：



`unordered_set`

`unordered_multiset`

`unordered_map`

`unordered_multimap`

tuple

元组是一个能够容纳元素集合的对象。每个元素可以是不同的类型。

```
template <class... Types> class tuple;
```

Example

```
#include <iostream>      // std::cout
#include <tuple>         // std::tuple, std::get, std::tie, std::ignore

int main ()
{
    std::tuple<int,char> foo (10,'x');
    auto bar = std::make_tuple ("test", 3.1, 14, 'y');
```

```

std::get<2>(bar) = 100; // access element

int myint; char mychar;

std::tie (myint, mychar) = foo; // unpack elements
std::tie (std::ignore, std::ignore, myint, mychar) = bar; // unpack (with ignore)

mychar = std::get<3>(bar);

std::get<0>(foo) = std::get<2>(bar);
std::get<1>(foo) = mychar;

std::cout << "foo contains: ";
std::cout << std::get<0>(foo) << ' ';
std::cout << std::get<1>(foo) << '\n';

return 0;
}

```

Output

```
foo contains: 100 y
```

tuple::tuple

构建一个 tuple(元组)对象。

这涉及单独构建其元素, 初始化取决于调用的构造函数形式:

(1)默认的构造函数

构建一个 元组对象的元素值初始化。

(2)复制/移动构造函数

该对象使用tpl的内容进行初始化 元组目的。tpl 的相应元素被传递给每个元素的构造函数。

(3)隐式转换构造函数

同上。tpl中的 所有类型都可以隐含地转换为构造中它们各自元素的类型元组 目的。

(4)初始化构造函数 用elems中的相应元素初始化每个元素。elems 的相应元素被传递给每个元素的构造函数。

(5)对转换构造函数

该对象有两个对应于pr.first和的元素pr.second。PR中的所有类型都应该隐含地转换为其中各自元素的类型元组 目的。

(6)分配器版本

和上面的版本一样，除了每个元素都是使用allocator alloc构造的。

```
default (1)
constexpr tuple();
copy / move (2)
tuple (const tuple& tpl) = default;
tuple (tuple&& tpl) = default;
implicit conversion (3)
template <class... UTypes>
    tuple (const tuple<UTypes...>& tpl);
template <class... UTypes>
    tuple (tuple<UTypes...>&& tpl);
initialization (4)
explicit tuple (const Types&... elems);
template <class... UTypes>
    explicit tuple (UTypes&&... elems);
conversion from pair (5)
template <class U1, class U2>
    tuple (const pair<U1,U2>& pr);
template <class U1, class U2>
    tuple (pair<U1,U2>&& pr);
allocator (6)
template<class Alloc>
    tuple (allocator_arg_t aa, const Alloc& alloc);
template<class Alloc>
    tuple (allocator_arg_t aa, const Alloc& alloc, const tuple& tpl);
template<class Alloc>
    tuple (allocator_arg_t aa, const Alloc& alloc, tuple&& tpl);
template<class Alloc, class... UTypes>
    tuple (allocator_arg_t aa, const Alloc& alloc, const tuple<UTypes...>& tpl);
template<class Alloc, class... UTypes>
    tuple (allocator_arg_t aa, const Alloc& alloc, tuple<UTypes...>&& tpl);
template<class Alloc>
    tuple (allocator_arg_t aa, const Alloc& alloc, const Types&... elems);
template<class Alloc, class... UTypes>
    tuple (allocator_arg_t aa, const Alloc& alloc, UTypes&&... elems);
template<class Alloc, class U1, class U2>
    tuple (allocator_arg_t aa, const Alloc& alloc, const pair<U1,U2>& pr);
template<class Alloc, class U1, class U2>
    tuple (allocator_arg_t aa, const Alloc& alloc, pair<U1,U2>&& pr);
```

Example

```
#include <iostream>      // std::cout
#include <utility>       // std::make_pair
#include <tuple>          // std::tuple, std::make_tuple, std::get

int main ()
{
    std::tuple<int,char> first;                      // default
    std::tuple<int,char> second (first);               // copy
```

```

    std::tuple<int,char> third (std::make_tuple(20,'b')); // move
    std::tuple<long,char> fourth (third); // implicit conversion
    std::tuple<int,char> fifth (10,'a'); // initialization
    std::tuple<int,char> sixth (std::make_pair(30,'c')); // from pair / move

    std::cout << "sixth contains: " << std::get<0>(sixth);
    std::cout << " and " << std::get<1>(sixth) << '\n';

    return 0;
}

```

Output

```
sixth contains: 30 and c
```

pair

这个类把一对值(values)结合在一起, 这些值可能是不同的类型(T1 和 T2)。每个值可以被公有的成员变量first、second访问。

pair是tuple(元组)的一个特例。

pair的实现是一个结构体, 主要的两个成员变量是first second 因为是使用struct不是class, 所以可以直接使用pair的成员变量。

应用:

- 可以将两个类型数据组合成一个如map
- 当某个函数需要两个返回值时

```
template <class T1, class T2> struct pair;
```

pair::pair

构建一个pair对象。

这涉及到单独构建它的两个组件对象, 初始化依赖于调用的构造器形式:

(1)默认的构造函数

构建一个对对象的元素值初始化。

(2)复制/移动构造函数(和隐式转换)

该对象被初始化为pr的内容 对目的。pr 的相应成员被传递给每个成员的构造函数。

(3)初始化构造函数

会员 第一是由一个和成员构建的第二与b。

(4) 分段构造

构造成员 first 和 second 到位, 传递元素first_args 作为参数的构造函数 first, 和元素 second_args 到的构造函数 second 。

```
default (1)
constexpr pair();
copy / move (2)
template<class U, class V> pair (const pair<U,V>& pr);
template<class U, class V> pair (pair<U,V>&& pr);
pair (const pair& pr) = default;
pair (pair&& pr) = default;
initialization (3)
pair (const first_type& a, const second_type& b);
template<class U, class V> pair (U&& a, V&& b);
piecewise (4)
template <class... Args1, class... Args2>
pair (piecewise_construct_t pwc, tuple<Args1...> first_args,
      tuple<Args2...> second_args);
```

Example

```
#include <utility>      // std::pair, std::make_pair
#include <string>        // std::string
#include <iostream>       // std::cout

int main () {
    std::pair <std::string,double> product1;                      // default constructor
    std::pair <std::string,double> product2 ("tomatoes",2.30);   // value init
    std::pair <std::string,double> product3 (product2);           // copy constructor

    product1 = std::make_pair(std::string("lightbulbs"),0.99);    // using make_pair (move)

    product2.first = "shoes";                                     // the type of first is string
    product2.second = 39.90;                                       // the type of second is double

    std::cout << "The price of " << product1.first << " is $" << product1.second << '\n';
    std::cout << "The price of " << product2.first << " is $" << product2.second << '\n';
    std::cout << "The price of " << product3.first << " is $" << product3.second << '\n';
    return 0;
}
```

Output

```
The price of lightbulbs is $0.99
The price of shoes is $39.9
The price of tomatoes is $2.3
```


结束