Table of Contents

序言 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
- 推荐书籍 1.2.6

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

第三章 问题求解系统 1.4
- Sklearn介绍 1.4.1
- Sklearn的代码架构 1.4.2
- Tensorflow介绍 1.4.3
- Tensorflow的代码架构 1.4.4

第四章 推荐系统 1.5
- 计算广告学概论 1.5.1
- 推荐系统概述 1.5.2
- FM(Factorization Machines),FFM 1.5.3
- Learning To Rank LTR 1.5.4

第五章 算法 1.6
- 第一节 插值 1.6.1
- 第二节 最长递增子序列 1.6.2
- LU分解 1.6.3
- SVD 1.6.4
- 霍夫曼编码 1.6.5
- 最小生成树Prim算法 1.6.6
- 最短路径Dijkstra算法 1.6.7

快速排序与二叉树 1.6.8
医学图像重建算法 1.6.9
第六章 最优化 1.7
最优化的优缺点及其改进方案 1.7.1
Levenberg-Marquardt算法 1.7.2
Qusi-Netwon 拟牛顿法 1.7.3
信赖域方法 1.7.4
参数调节 1.7.5
ResiNN 1.7.6
linear search 1.7.7
最速下降法，牛顿法，LBFGS 1.7.8
线性与非线性方程组解的稳定性分析 1.7.9
第七章 机器学习 1.8
机器学习面试问题集 1.8.1
机器学习学习任务 1.8.2
机器学习原理 1.8.3
SVM，KKT条件与核函数方法 1.8.4
SVM 1.8.4.1
KKT条件与对偶 1.8.4.2
核函数方法 1.8.4.3
决策树与集成学习 1.8.5
决策树 1.8.5.1
集成学习 1.8.5.2
贝叶斯方法 1.8.6
逻辑回归与最大熵模型 1.8.7
高斯混合模型 1.8.8
隐马尔科夫模型 1.8.9
DNN-HMM混合系统 1.8.10
马尔科夫链蒙特卡洛方法 1.8.11
降维与无监督学习 1.8.12
分类问题与方法 1.8.13
回归问题与方法 1.8.14
K均值EM等聚类算法 1.8.15
正则化方法原理与实践 1.8.16
VC维 1.8.17

机器学习项目 1.8.18
多元回归分析 1.8.19
矩阵微分 1.8.20
正则化 1.8.21
训练神经网络的经验 1.8.22
Notes 1.8.23
特征工程 1.8.24
CRF 1.8.25
第八章 深度学习 1.9
CNN 1.9.1
RNN 1.9.2
第九章 项目工作 1.10
Titanic 1.10.1
Residual NN 1.10.2
Fast项目 1.10.3
Miura项目 1.10.4
问题求解系统 1.10.5
任务 1.10.6
深度学习的发展脉络 1.10.7
第十章 问题 1.11
基本ML问题 1.11.1
模型评估问题 1.11.2
降维的方法与原理 1.11.3
非监督学习问题 1.11.4
集成学习问题 1.11.5
前向神经网络问题 1.11.6
循环神经网络问题 1.11.7
生成式对抗网络问题 1.11.8
强化学习问题 1.11.9
神经网络的种类适用场景与优缺点 1.11.10
基本C++问题 1.11.11
C++标准模板库 1.11.12
Windows C++ 1.11.13
常规算法问题 1.11.14
常见问题 1.11.15

STL 1.11.16
结束 1.12

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。

■自然科学的基础是数学，数学是让一门学科严格化，并能称之为科学的基础。因此，本书的主线是以数学为基础，以编程为技术工具来高效的解决自然科学(如物理，半导体，光学，声学，分子化学)，商业与经济活动（推荐系统，计算广告，图像处理，信号处理，金融）中的问题。要践行毕达哥拉斯学派所谓的“世界的本质是数的理念”，换句话说，数学是所有学科唯一的通用语言。不同学科会分享相同的数学方法，这也就本书要总结的一个方面。不同学科的区别只是研究，处理问题的不同，而没有数学方法上的不同。就像随机偏微分方程可以用于量子力学，也可以用于金融；时间序列分析可以用于语音，数字信号处理，也可以用于金融。

■本书的是自己学习的总结，更是自己思想体系不断进化的体现。因此，这本书会一直去增补，完善，没有停止的说法。在可以预见的未来，本书会经历几个阶段。第一阶段是知识的学习与记录；第二阶段是知识的运用与不同知识原理的提炼；第三阶段就是知识的创造过程，也就是为解决一类问题而创造出一个新的方法。三阶段简单来说，就是把书读厚，把书读薄，再把书读厚的阶段。就时间尺度来说，第一阶段是30周岁以前，第二阶段是30岁以后的三年，第三阶段就是33岁以后的岁月。正如“日知其所亡，月无忘其所能”，自己要做到每年一个总结，同时安排下一年的任务。

■李品品 2018年

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

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

如果想让章节有层次感的显示，则可以在summary.md中设置

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。

同步

在gitbook 客户端book->Repository Setting...中添加Gitbook中书的地址即可：
https://github.com/Li-Simon/Gitbook.git 一般是在gitbook中先创建一本书，然后再copy到本地。

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

推荐书籍

1.《深度学习核心技术与实践》猿辅导研究团队
2.《The Elements of Statistical Learning》Hastie
3.《Hands-On Machine learning with Scikit-Learn&Tensorflow》
4.《凸优化》Stephen Boyd
5.《最优化导论》Edwin K.P.Chong
6.《Pattern Recognition and Machine Learning》M.Bishop
7.《Neural Network for Pattern Recognition》M.Bishop
8.《深度学习》Ian Goodfellow
9.《剑指Offer》

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_IMMIEvents>,
    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_IE84HOAVBLEvents>,
    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@@QAEHQAHH@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> Alg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 exampl
e, if you’re trying to use a Foo<int>, the compiler must see both the Foo template and the fa
ct 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 definit
ely 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. Suppo
se 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 definiti
on 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 intentio
nal 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 que
stion/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

```
AtlAdvise(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++ Introp，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传送的数据
```

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

#include "stddcls.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", stat
us, 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
```

第三章 问题求解系统

这一章，主要是构建自己的ML，DL问题求解系统,基于的是scikit-learn与tensorflow,目的是通过构建自己的问题系统，来方便的训练与使用模型;最终会把这个系统打造成自己的AutoML系统，当然不止是ML，还有DL。需要包含如下几部分。这个系统可以参照阿里的云栖。最终，本书中很多文档都是这个系统的注脚。

1. Scikit-learn的代码框架
2. tensorflow的代码框架
3. 自己系统的框架
    i. 数据清洗
    ii. 模型的选择
    iii. 模型的验证
    iv. 模型的使用
4. 系统处理的问题
    i. 基于sklearn，对不同的主流方向(主流问题，如分类，回归，降维，特征工程，推荐系统)，构建不同的父类
    ii. 数据清洗是一个麻烦的工程，这个工程也该包含在我们的系统里面
    iii. 接下来就是模型的选择
    iv. 特征工程是一个大的方面
    v. 模型结果的验证
5. 问题求解系统的几大部分
    i. 数据预处理
    ii. 特征工程
    iii. 统计分析
    iv. 机器学习
    v. 深度学习
    vi. 将包含的部分
        i. 时间序列分析
        ii. 文本分析
        iii. 网络分析
        iv. 推荐系统
        v. 自动参数调节
6. 需要思考的问题
    i. 最终是否要做成一个PC端的应用程序？
    ii. 用什么语言写做这个界面？
    iii. 底层用什么语言实现？
    iv. AutoML的当前市场，未来的趋势以及怎么盈利？
7. 思考自己的框架怎么给业界带来便利，加速业界各项研究的发展？

i. 自己框架怎么让机器学习，语言识别，NLP的研究变得更容易？
ii. 怎么解决非一二线互联网工作没有优秀的ML，DL人才的问题？了解他们应用AI存在的困难，以及探索解决之道。(PC端这样方便)

8. 实现问题求解系统中将要遇到的困难？只有当自己能预测自己的进度的时候iu，也就表明自己对整个系统成竹在胸了。每一步需要的知识，怎么建构都有自己的想法的是，自己的解决方案的时候，余下的就只是体力活了。

Sklearn介绍

sklearn的算法与模块

如下图所示，sklearn的定位是作为数据挖掘与数据分析的有效工具。它提供了这四大类算法，分类，回归，聚类，降维；两个模块：模型选择，预处理。每类算法包含有不同的算法。

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ... — Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ... — Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, ... — Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization. — Examples

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics. — Examples

Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction. — Examples

1. 分类
    i. SVM，最近邻，随机森林
2. 回归
    i. SVR，ridge regression, Lasso regression
3. Clustering
    i. k-Means,spectral clustering,man-shift
4. 降维
    i. PCA,特征提取，非负矩阵分解
5. 模型选择
    i. grid search,cross validation
6. Preprocessing
    i. 预处理，特征提取

User Guide

细节的东西我们可以看scikit-learn的User Guide。
User Guide包含如下几大模块：

1. Supervised learning
2. Unsupervised learning
3. Model selection and evaluation
4. Dataset transformations
5. Dataset loading utilities
6. Computing with scikit-learn

Sklearn的代码架构

API Reference

1. `sklearn.base` : Base classes and utility functions
2. `sklearn.calibration` : Probability Calibration
3. `sklearn.cluster` : Clustering
4. `sklearn.compose` : Composite Estimators
5. `sklearn.covariance` : Covariance Estimators
6. `sklearn.cross_decomposition` : Cross decomposition
7. `sklearn.datasets` : Datasets
8. `sklearn.decomposition` : Matrix Decomposition
9. `sklearn.discriminant_analysis` : Discriminant Analysis
10. `sklearn.dummy` : Dummy estimators
11. `sklearn.ensemble` : Ensemble Methods
12. `sklearn.exceptions` : Exceptions and warnings
13. `sklearn.feature_extraction` : Feature Extraction
14. `sklearn.feature_selection` : Feature Selection
15. `sklearn.gaussian_process` : Gaussian Processes
16. `sklearn.isotonic` : Isotonic regression

17. `sklearn.impute` : **Impute**

18. `sklearn.kernel_approximation` **Kernel Approximation**

19. `sklearn.kernel_ridge` **Kernel Ridge Regression**

20. `sklearn.linear_model` : **Generalized Linear Models**

21. `sklearn.manifold` : **Manifold Learning**

22. `sklearn.metrics` : **Metrics**

23. `sklearn.mixture` : **Gaussian Mixture Models**

24. `sklearn.model_selection` : **Model Selection**

25. `sklearn.multiclass` : **Multiclass and multilabel classification**

26. `sklearn.multioutput` : **Multioutput regression and classification**

27. `sklearn.naive_bayes` : **Naive Bayes**

28. `sklearn.neighbors` : **Nearest Neighbors**

29. `sklearn.neural_network` : **Neural network models**

30. `sklearn.pipeline` : **Pipeline**

31. `sklearn.preprocessing` : **Preprocessing and Normalization**

32. `sklearn.random_projection` : **Random projection**

33. `sklearn.semi_supervised` **Semi-Supervised Learning**

34. `sklearn.svm` : **Support Vector Machines**

35. `sklearn.tree` : **Decision Trees**

36. `sklearn.utils` : **Utilities**

python类的继承

```
class Person(object):
    pass

class Child(Person):                # Child 继承 Person
    pass

May = Child()
Peter = Person()

print(isinstance(May,Child))        # True
print(isinstance(May,Person))       # True
print(isinstance(Peter,Child))      # False
print(isinstance(Peter,Person))     # True
print(issubclass(Child,Person))     # True
```

class Child(Person)表示Child继承于Person，任何类最终都可以追溯到根类object，这与Java,C#是一样的。

多态

定义：允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。(发送消息就是函数调用)
实现多态的技术称为:动态绑定，是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。
多态的作用：消除类型之间的耦合关系。
子类覆盖父类的函数实现，也就是子类重新实现父类的函数。不同的子类对相同的父类函数可以有不同的实现，因此多态就是"一个接口，多种实现"。
多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编译时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能确定。

多态的好处

继承和堕胎本质的目的是为了代码重用和解耦。而重用和解耦是为了减少开发时间和减少错误。最终达到降低成本的目的。
多态体现的就是“对扩展开放，对修改封闭”。封闭就是子类不能直接修改父类的其它函数，扩展开放也就是子类可以重写父类的函数。

1. 可替换性(substitutability)。多态对已经存在代码具有可替换性。
2. 可扩充性(extensibility)。多态对代码具有可扩充性，增加新的子类不影响已经存在类的多态性，继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现圆锥，半圆锥以及半球体的多态基础上，很容易增加球体类的多态性。
3. 接口性(interface-ability)。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。
4. 灵活性(flexibility)。它在应用中体现了灵活多样的操作，提高了使用效率。
   5.简化性(simplicity)。多态简化对应用软件的编写和修改过程，尤其在处理大量对象和操作时，这个特点尤为突出和重要。
5. 提高了代码的维护性(继承保证)
6. 提高了代码的扩展性(由多态保证)

Tensorflor介绍

Tensorflow的代码架构

第八章 推荐系统

计算广告学概论

一些概念

CPM:Cost per Mille,千次展览付费 RPM:Revenue per Mille千次展览收益
CPC:Cost per Click按点击付费
CPT:Cost Per Time按时间付费
CPS:Cost Per Sale按销售额付费
CPA:Cost per Action按转化付费
ROI:Return On Investment 投入产出比
RTB:Real Time Bidding
ADN：ad Net-work 广告网络
Auction-based advertising:竞价广告
Search ad:搜索广告
GSP:Generalized Second Price广义第二高价
DSP:Demand Side Platform 需求方平台

优化目标

eCPM:excepted Cost Per Mille,千次展示期望收入
ePCM = 点击率x点击价值
流量塑形：在有些情况下，我们可以主动地影响流量，以利于合约的达成。
搜索广告是典型的竞价广告，是整个在线广告中份额最大的部分。

定价问题

常见的两种：GSP与基于纳什均衡的VCG(Vickrey-Clarke-Groves)定价策略。
GSP(广义第二高价)：对赢得每一个位置的广告主，都按照他下一位的广告位置出价来收取费用。
基于纳什均衡。
ePCM等于点击率乘以出价 $r = u.v$

一般有如下变形： $r = u^k.v$ 其中k是价格挤压（Squashing）因子.

重定向

把那些曾经对广告主服务发生明确兴趣的用户找出来，向他们投放该广告主的广告。

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lantent vector
- $\mathbf{V_i} = [v_{i1}, v_{i2}, ..., v_{ik}]^T$

,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。

- $\sum_{i=1}^{n} \sum_{j=i+1}^{n} (V_i, V_j) x_i x_j$
- $= \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (V_i, V_j) x_i x_j - \frac{1}{2} \sum_{i=1}^{n} (V_i, V_i) x_i x_i$
- $= \frac{1}{2} \sum_{f=1}^{n} (\sum_{i=1}^{n} v_{if} x_i)^2 - \sum_{i=1}^{n} v_{if}^2 x_i^2)$

通过对每个特征引入lantent 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。

第五章:算法

这一章主要介绍传统的算法与数据结构，比如排序，动态规划，最短路径，插值;二叉树，红黑书，链表，队列，堆栈。还有就是线性代数与矩阵计算中的算法，比如:LU分解，Householder变换，矩阵求本征值，矩阵求逆，PCA，SVD，Lanczos等。未来可能会把图像与信号处理的算法加在这里，但是不急。

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 $\hat{z} = \sum_{i=0}^{n} \frac{1}{d^{\alpha}} z_i$
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$\hat{z_0}$是点(x_o, y_o)处的估计值，即：$z_o = z(x_o, y_o)$
假设条件:

1. 无偏约束条件 $E(\hat{z_0} - z_0) = 0$
2. 优化目标/代价函数 $J = Var((\hat{z_0} - z_0))$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这两组数拟合出来，因此可以用距离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],...,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MaxLen[i] 满足i < j, 那么我们可以遍历i从1到j-1来求得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：$a_{1j} = u_{1j}, j = 1, 2, .., n. => u_{1j} = a_{1j}$。

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

...。

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:$y_k = \sum_{i=k}^{i=n} u_{ki}x_i => x_k = [y_n - \sum_{i=k+1}^{i=n} u_{ki}x_i]/u_{kk}$
这样通过LU分解矩阵就求得了线性方程组的就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mxn(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->x+e，求梯度等于0的x),可以得到解为:$x=(X^TA)^{-1}A^Tb$

QR分解

定理:设A是mxn阶矩阵，m>=n,假设A为满秩的，则存在一个唯一的正交矩阵$Q(Q^TQ=I)$和唯一的具有正对角元$r_{ij}>0$的nxn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

DotNetWallet - Microsoft Visual Studio
File Edit View VAssistX Project Build Incredibuild Debug Team Tools Test Analyze
Debug
Any CPU
DotNetWallet
Toolbox
NuGet - Solution
DotNetWallet*
Config.cs
QBitNinjaJutsus.cs
AddressHistory
Browse
Installed
Updates
Consolidate
NBitcoin
Include prerelease
NBitcoin by NBitcoin, 132K downloads
The C# Bitcoin Library
NBitcoin.Indexer by NBitcoin.Indexer, 32.7K downloads
Library for querying data indexed by NBitcoin.Indexer on Azure Storage

研究这个

Toolbox
NuGet - Solution
DotNetWallet*
Config.cs
QBitNinjaJutsus.cs
AddressHistoryRecord.cs
Browse
Installed
Updates
Consolidate
Bitcoin
Include prerelease
Bitcoin by Bitcoin, 1.12K downloads
v1.0.0
A fully featured C# library for making and receiving payments using the Bitcoin network.
Bitcoin.Net by Christian Findlay, 353 downloads
v1.2.0
Bitcoin.Net
BitCoinSharp by Nathan Baulch, 5.83K downloads
v0.2.166.49
A C# implementation of a BitCoin client-only node
Stratis.Bitcoin by Stratis Platform, 2.04K downloads
v1.0.0
Stratis Bitcoin FullNode

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

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

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

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

根据中心切片定理，我们可以用 P 来代替 F：

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

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

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

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

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

或

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

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

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

2. 投影函数一维傅里叶变换

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

3. 密度函数二维傅里叶变换

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

4. 在v=0时，傅里叶变换表示

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

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。

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

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

4. 对（3）式右面进行坐标变换x，y

$$\mathrm{d}s\mathrm{d}t=J\mathrm{d}x\mathrm{d}y=\begin{vmatrix}\partial t/\partial x & \partial s/\partial x\\ \partial t/\partial y & \partial s/\partial y\end{vmatrix}\mathrm{d}x\mathrm{d}y=\mathrm{d}x\mathrm{d}y$$

雅克比转换

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

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。

5. f(x,y)二维傅里叶变换

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

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

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

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：

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

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

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

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

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

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

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

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

$\delta(g(x))f(x)=\sum_n \frac{1}{|g'(\lambda_n)|}\delta(x-\lambda_n)$ ，其中 λ_n 为 $g(x)$ 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 $|grad(g)|=\sqrt{\left(\frac{\partial g}{\partial x}\right)^2+\left(\frac{\partial g}{\partial y}\right)^2}$ 和 $|grad(g)|=\sqrt{\left(\frac{\partial g}{\partial x}\right)^2+\left(\frac{\partial g}{\partial y}\right)^2+\left(\frac{\partial g}{\partial z}\right)^2}$ 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

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

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

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。

最优化方法

这章主要讨论常见的最优化方法，以及优化技巧与不同方法的优缺点。

最优化的优缺点及其改进方案

常见的损失函数

分类问题损失函数

y是标签，f是预测结果。
对于二分类问题$Y=[-1,1]$
0-1损失函数：

- $L_{0-1}(f,y)=1_{fy\leq 0}$

Hinge损失函数：

- $L_{hinge}(f,y)=\max(0,1-fy)$

Logistic损失函数：

- $L_{logistic}(f,y)=\log_2(1+\exp(-fy))$

当预测值$f\in[-1,1]$时，可以定义如下损失函数：
交叉熵（Cross Entropy）：

- $L_{crossentropy}=-\log_2(\frac{1+fy}{2})$

回归问题损失函数

对于回归问题，Y=R,我们希望$f(x_i,\theta)\approx y_i$,最常用的损失函数是平方损失函数：

- $L_{square}(f,y)=(f-y)^2$

平方损失函数对异常点惩罚很大，因此对异常点比较敏感。为了解决这个问题，引入了绝对损失函数：

- $L_{square}(f,y)=|f-y|$

但是绝对损失函数在0点不可以求导。
一个方法是在0点进行函数光滑化也就是在0点附件用平方函数代替。$\delta->0$

- $L_{square}(f,y)=\sqrt{(f-y)^2+\delta^2}$

或者用Huber损失函数：
Huber损失函数：

- $L_{square}(f,y)=(f-y)^2,|f-y|>\delta$

- $L_{square}(f,y)=2\delta|f-y|-\delta^2,|f-y|>\delta$

还得补充与思考文本处理，信号处理中的损失函数。

机器学习中哪些是凸优化问题，哪些是非凸优化问题

凸函数条件：Hassian矩阵为半正定。
逻辑回归，线性回归，SVM是凸优化问题。PCA，矩阵分解以及NN是非凸优化问题。

梯度下降

问题1：怎样验证自己代码中梯度公式的正确性？

- $|\frac{L(\theta+he_i)-L(\theta+he_i)}{2h}-\frac{\partial L(\theta)}{\theta_i}|\approx Mh^2$

若$M\leq 10^7$,则:

- $|\frac{L(\theta+he_i)-L(\theta+he_i)}{2h}-\frac{\partial L(\theta)}{\theta_i}|\leq h$

因此取$h=10^{-7}$看,看上式是否成立，不成立则$M\geq 10^7$,则取$h=10^{-8}$，看上式得值是否变为原来的10^{-2}。如果是则，求导正确，不是则求导错误。

问题2：当数据量特别大的时候经典梯度下降法存在什么问题，需要怎么改进？

因为经典的梯度下降是对整个训练集进行的，如果训练集特别大，每次更新要针对整个数据集，因此需要很大的计算力，计算很费时间。
一般采用随机梯度下降发，每次训练一个数据。但是该方法很难收敛，会在极值点振荡，因此一般采用mini-batch Gradient Descent方法。一般batch的size m取2的整数次幂。
为了充分利用数据以及避免数据的特定顺序给算法收敛带来的印象，一般每次遍历数据前，对数据先随机排序，然后每次取m个数据，直到遍历所有数据。有必要再按新规则排序，重复以上步骤。
如何选择学习率α，为了加快学习速度与提高精度。一开始取大的学习速率，误差曲线进入平台期后，再减小学习速率做更精细的调整。

问题3：深度学习中常用的优化方法是随机梯度下降法，但是SGD偶尔也会失效，无法给出满意的训练结果，这是为什么？

当SGD遇到山谷的时候，由于沿山谷方向的梯度比较小，因此结果会在山谷方向来回移动，而不能像整体梯度下降一样能沿着山谷有一个确定方向的移动。其次是遇到平坦区域，有些方向的梯度接近于0，因此需要很长世间才能走出这片区域。

问题4：为了改进SGD，研究者都做了哪些改动？提出了哪些方法？他们各有哪些特点？

动量**(Momentum)**方法

为了解决SGD在山谷振荡，鞍点停滞的现象。我们引入了带动量的SGD。普通SGD是：

$$\theta_{t+1} = \theta_t - \eta g_t$$

带动量的SGD公式是：

$$v_t = \gamma v_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - v_t$$

通过$0 < \gamma < 1$，能实现梯度的累积，也就是速度的累积，因此能更好的冲出山谷与平坦（平原）区域。

AdaGrad方法

惯性的获得是基于历史的，除了从过去的步伐中获得一股子冲劲，我们还想知道不同参数更新的步伐，这样，对于更新频率低的参数能加快更新步伐，而更新频率高的参数减小步伐。因此就有了AdaGrad方法，他采用"历史梯度平均和"来衡量不同参数的梯度的稀疏性。，取值越小表明越稀疏。更新公式如下：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{k=0}^{t} g_{k,i}^2 + \epsilon}} g_{t,i}$$

$\theta_{t+1,i}$表示t+1时刻，参数向量的第i个参数，$g_{k,i}$表示k时刻，参数向量的第i个方向。此外，分母的求和新式实现了退火过程，意味着随着时间推移，学习速率越来越小，保证算法最终收敛。

Adam方法

Adam方法将惯性保持与环境感知这两个有点集合在一起。
通过记录梯度的一阶矩来保持惯性，通过记录二阶矩来实现环境感知能力。通过指数衰减平均技术来实现时间久远的梯度对当前平均值的贡献呈指数衰减。计算公式如下：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Adam更新公式：

$$\theta_{t+1} = \theta_t - \frac{}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

其中:$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$

问题**5**：**L1**正则化使得模型参数具有稀疏性的原理是什么？

角度**1**：通过做等高线图

由图可知，L1更容易在轴上与登高线相交，而L2更容易在两轴之间相交。其实加L1,L2正则就是加一个约束，也就是带约束的优化问题，通过这个约束，我们就知道了解空间的限定范围。

角度**2**：贝叶斯先验

L1正则化相当于对模型参数引入了拉普拉斯先验，L2正则化相当于对模型参数引入了高斯先验。因为拉普拉斯先验分布中，参数取值为0的概率更高，因此更容易产生稀疏性。而高斯先验分布只会让所有参数趋于0，并且在趋于0的不同位置是等概率的，因此不会让参数等于0。

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Hassian阵是正定的。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$diag(J^T J)$,因此迭代变成：

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

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 fucntion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没有是损失函数下降，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。

$$
\begin{aligned}
& k := 0; x := x_0; v := 2; A := J(x)^T J(x) g := J(x)^T f(x) \\
& u := \tau * max(a_{ii}) \\
& found := (||g||_\infty \le \epsilon_1) \\
& while(notfound) and (k < k_{max}) \\
& k := k + 1; solve- > (A + uI) h_{lm} = -g \\
& if \quad ||h_{lm}|| \le \epsilon(||x|| + \epsilon_2) \\
& found := true \\
& else: \\
& x_{new} := x + h_{lm} \\
& \rho := (F(x) - F(x + h_{dl})/(L(0) - L(h_{dl}))) \\
& if \quad \rho \ge 0 \\
& x := x_{new}; g := J(x)^T f(x) \\
& A := J(x)^T J(x) g := J(x)^T f(x) \\
& found := (||g||_\infty \le \epsilon_1) \\
& u := u * max(1/3, 1 - (2\rho - 1)^3); v := 2 \\
& else
\end{aligned}
$$

```
    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)
        Hassian = 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 = Hassian.max()
        lamda_v = 2

        matrix_I = np.identity(Hassian.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(Hassian + 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)
                        Hassian = 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
```

```
                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Hassian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:

- $\alpha_k = argmin_\alpha f(x_k + \alpha d_k)$

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$x_{k+1} = x_k - \alpha_k d_k$

Qusi-Netwon methods

BFGS

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

比较出名的Qusi-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

- $B_k p_k = -\nabla f(x_k)$
- B_k就是近似的Hassian 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

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

这个条件就是简单的求$f(x)$的二阶导数。
令：

- $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$，$s_k = x_{k+1} - x_k$，因此
- B_{k+1}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Hassian阵以如下方式更新，

- $B_{k+1} = B_k + U_k + V_k$

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：

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

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：

- $\alpha = \frac{1}{y_k^T s_k}$
- $\beta = \frac{1}{s_k^T B_k s_k}$

最终我们得到Hassian阵的更新方程：

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

利用 Sherman–Morrison formula，

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

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。

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

DFP

参考 DFP的矫正公式如下：

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

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：

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

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

Broyden族算法

之前BFGS和DFP矫正都是由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：

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

$\phi_k \in [0,1]$，v_k由下式定义：

$$\blacksquare v_k = \sqrt{y_k^T H_k y_k}\left(\frac{s_k}{y_k^T s_k} - \frac{H_k y_k}{y_k^T H_k y_k}\right)$$

Gauss-Netwon方法

对于函数$f(x)$的小邻域展开

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

我们需要选择步长h_{gn}来最小化$L(h)$

- $h_{gn} = argmin_h L(h)$
- $(J^T J)h_{gn} = -J^T f$

信赖域方法

Powell's Dog Leg Method是一种信赖域方法
在Gauss-Netwon迭代中

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

最陡的方向由下面公式给出:

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

但是这只是给出了方向，而没有给出步长。
考虑线性模型

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

当α取如下值得时候，以上函数取最小值

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

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

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

else:

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

选择β 使得$||h_{dl}|| = \Delta$
在L-M算法中我们定义了增益因子，

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

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

$$
\begin{aligned}
& k := 0; x := x_0; \Delta := \Delta_0; g := J(x)^T f(x) \\
& found := (||f(x)||_\infty \le \epsilon_3) or (||g||_\infty \le \epsilon_1) \\
& while(not found) and (k < k_{max}) \\
& k := k + 1; computer \alpha by(2.4) \\
& h_{sd} := -\alpha g; solve J(x) h_{gn} \simeq -f(x) \\
& computer h_{dl} by(4.5) \\
& if ||h_{dl}|| \le \epsilon(||x|| + \epsilon_2) \\
& found := true \\
& else: \\
& x_{new} := x + h_{dl} \\
& \rho := (F(x) - F(x + h_{dl})/(L(0) - L(h_{dl}))) \\
& if \quad \rho \ge 0 \\
& x := x_{new}; g := J(x)^T f(x) \\
& found := (||f(x)||_\infty \le \epsilon_3) or (||g||_\infty \le \epsilon_1) \\
& if \quad \rho \ge 0.75 \\
& \Delta := max(\Delta, 3 * ||h_{dl}||) \\
& elseif \quad \rho < 0.25 \\
& \Delta := \Delta/2; found := (\Delta \le \epsilon_2(||x|| + \epsilon)
\end{aligned}
$$

```
    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
        Hassian = 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(Hassian), -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(Hassian, 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)
                        Hassian = 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() < epsi
lon_3:
                            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Hassian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Hassian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Hassian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

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

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

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Hassian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，计算二劫导数太费时间，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 Rn 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，我们任然需要建立物理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，亚像素的偏离），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，我们都在用用记忆卷积，反卷积的深度神经网络，都市没有发现很好的效果。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，并不比没有处理额图像好，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.

$\alpha_k = argmin_{\alpha \geq 0} f(x_{(k)} - \alpha \nabla f(x^{(k)}))$

Proposition 8.1 : if $x^{(k)}$ is the steepest descent sequence for f: R^n->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->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:

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

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

so

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

where:

- $\alpha_k = argmin_{\alpha \geq 0} f(x_{(k)} - \alpha g^{(k)}) = argmin_{\alpha \geq 0} (\frac{1}{2}(x^k - \alpha g^{(k)})^T Q (x^k - \alpha g^{(k)}) - (x^k - \alpha$

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:

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

Submit α_k into γ_k, then

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

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:

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

Theorem 8.2 In the steepest descent algorithm, we have

- $x^{(k)}$ ->x^* for any $x^{(0)}$

Theorem 8.3 For the fixed step size gradient algorithm, $x^{(k)}$ ->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 :

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

Let:

- $r = \frac{\lambda_{max}(Q)}{\lambda_{min}(Q)} = ||Q||||Q^{-1}||$ 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 $\lambda_{max}(Q) = \lambda_{min}(Q)$, corresonding to circular contours of f.
You can using the convergence ratio r to judge the speed of convergence.

Definition 8.1 Given a sequence $x^{(k)}$ thst converges to x^*, that is, $lim_{k->\infty} ||x^{(k)} - x^*|| = 0$, we

say that the order of convergence is o, where $p \in R$, if $0 < lim_{k->\infty} \frac{x^{(k+1)} - x^*}{||x^{(k)} - x^*||^p} < \infty$,

if for all $p > 0$, $lim_{k->\infty} \frac{x^{(k+1)} - x^*}{||x^{(k)} - x^*||^p} = 0$,

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, ehere exist a function f and an initial $x^{(0)}$ such that the order of convergence of $x^{(k)}$ is equal 1.

Newtow's Method

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

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^*, Netwon's method is weel defined for all k, and converges to x^* with order of convergence at least 2.

At stated in the aboved theorem, Newton's methods has superies convergence properties if the starting point is near the solution. However, the method is not guaranteed to converge to the soulution 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 $a \in (a, \alpha)$,

$f(x^{(k)} = \alpha d^{(k)}) < f(x^{(k)})$.

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

- $x^{(k+1)} = x^{(k)} - \alpha_k F(x^{(k)})^{-1} g^{(k)}$ where
- $\alpha_k = argmin_{\alpha > 0} f(x^{(k) - \alpha_k F(x^{(k)})^{-1} g^{(k)}})$

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:

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

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

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

Let $s(x) = r_i(x)\frac{\partial^2 r_i}{\partial x_j \partial x_k}(x)$
So the Hessian matrix as

- $F(x) = 2(J(x)^T J(x) + S(x))$.

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

- $x^{(k+1)} = x^{(k)} - (J(x)^T J(x) + S(x))^{-1} J(x)^T r(x)$. In some case, s(x) can ne ignored because its components are negligibly small. In this case, the above Newtons's algorithm reduces to what is commonly called the Gauss-Newtons 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 matri $J(x)^T J(x)$ may not be positive definite. As described before, this problem can be overcome using a Levenberg-Marquardt modification:

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

Conjugate Direction Methods

The conjugate direction methods typically perrform better than the method of steepest descent, but not as weel 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 nxn matrix. The directions $d^{(0)}, d^{(1)}, d^{(2)}, ..., d^{(m)}$ are Q-conjugate if, for all $i \neq j$, we have $d(i)TQd^{(j)} = 0$.

Lemma 10.1 Let Q be as symmetric positive definite nxn matrix. If the diretions $d^{(0)}, d^{(1)}, d^{(2)}, ..., d^{(m)} \in R^n, k \leq n-1$, are nonzero and Q-conjugate, then they are linearly independent.

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

- $d^{(0)}, d^{(1)}, d^{(2)}, ..., d^{(m)}, for k \geq 0,$
- $g^{(k)} = \nabla f(x^{(k)}) = Qx^{(k)} - b,$
- $\alpha_k = -\frac{g^{(k)T} d^{(k)}}{d^{(k)T} Q d^{(k)}}$
- $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$

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, $g^{(k+1)T} d^{(i)} = 0$ 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. $g^{(0)} = \nabla f(x^{(0)})$. If $g^{(0)} = 0$, stop, else set $d^{(0)} = -g^{(0)}$. 3. $\alpha_k = -\frac{g^{(k)T} d^{(k)}}{d^{(k)T} Q d^{(k)}}$
3. $x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}$
4. $g^{(k+1)} = \nabla f(x^{k+1})$. If $g^{(k+1)} = 0$,stop.
5. $\beta_k = -\frac{g^{(k+1)T} Q d^{(k)}}{d^{(k)T} Q d^{(k)}}$ 7. $d^{(k+1)} = -g^{(k+1)} + \beta_k d^{(k)}$
6. Set k:= k+1; go to step 3.

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

最优化方法

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

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，theta是参数，要迭代求解的值，theta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。

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

//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*n2。

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n2，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。

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。

```
C:\Windows\system32\cmd.exe
Paras:  1.99795  0.998981 -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

```
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*i
nit_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_d
ata_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。牛顿法最大的特点就在于它的收敛速度很快。[1]
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，每一步都需要求解目标函数的**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需要求解复杂的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：

- $m_k(p) = f(x_k) + \bigtriangledown f(x_k)^T p + \frac{p^T B_k p}{2}$
- $p_k = -B_k^{-1} \bigtriangledown f(x_k)$

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Bk 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（Conjecture 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，

$\mathbf{u^T A v}=0$，则称矢量$\mathbf{u},\mathbf{v}$是共轭的。假设：$P=[\mathbf{p_1},\mathbf{p_2},\ldots,\mathbf{p_n}]$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：

$\mathbf{Ax}_*=\sum_{i=1}^{n}\alpha_i\mathbf{AP_i}$ 左乘$\mathbf{P_k^T}$：$\mathbf{P_k^T Ax}_*=\sum_{i=1}^{n}\alpha_i P_k^T\mathbf{AP_i}$ 代入：

$\mathbf{P_k^T Ax}_*=\mathbf{b}$,$\mathbf{u^T Av}=\langle\mathbf{u},\mathbf{v}\rangle_{\mathbf{A}}$，

利用$i!=k$，有$\langle\mathbf{p_k},\mathbf{p_i}\rangle_{\mathbf{A}}=\mathbf{0}$ 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：$\mathbf{f(x)}=\frac{\mathbf{1}}{\mathbf{2}}\mathbf{x^T Ax}-\mathbf{x^T b}$，他存在唯一的最小值，因为他的二阶导是正定的：

$\mathbf{D^2 f(x)}=\mathbf{A}$，解就是一阶导数等于0的地方 $\mathbf{Df(x)}=\mathbf{Ax}-\mathbf{b}$，假设第一个基矢$\mathbf{p_0}$就是$f(x)$在$\mathbf{x}=\mathbf{x_0}$处的负梯度，我们可以假设

$\mathbf{x_0}=\mathbf{0}$.因此：$\mathbf{p_0}=\mathbf{b}-\mathbf{Ax_0}$. 令$\mathbf{r_0}$表示第k步的残差：

$\mathbf{r_k}=\mathbf{b}-\mathbf{Ax_k}$,$\mathbf{r_k}$是$\mathbf{f(x)}$在$\mathbf{x}=\mathbf{x_k}$处的负梯度。为了让$\mathbf{p_k}$与前面的所有的$\mathbf{p_i}$相互共轭，

$\mathbf{p_k}$可以如下构造：$\mathbf{p_k}=\mathbf{r_k}-\sum_{\mathbf{i<k}}\frac{\mathbf{p_i^T Ar_k}}{\mathbf{p_i^T Ap_i}}$ 沿着这个方向，因此下一步的优化方向就是：

$\mathbf{x_{k+1}}=\mathbf{x_k}+\alpha_\mathbf{k}\mathbf{p_k}$,满足：$g'(\alpha_k=0)$ 因此：

$\mathbf{f(x_{k+1})}=\mathbf{f(x_k}+\alpha_\mathbf{k}\mathbf{p_k})=:\mathbf{g}(\alpha_\mathbf{k})$，$\alpha_\mathbf{k}=\frac{\mathbf{p_k^T b}}{\mathbf{p_k^T Ap_k}}=\frac{\mathbf{p_k^T(r_k+Ax_k)}}{\mathbf{p_k^T Ap_k}}=\frac{\mathbf{p_k^T r_k}}{\mathbf{p_k^T Ap_k}}$
算法：

$\mathbf{r_0}:=\mathbf{b}-\mathbf{Ax_0}$

$\mathbf{p_0}:=\mathbf{r_0}$
$\mathbf{k}:=\mathbf{0}$
repeat:

$\alpha_\mathbf{k}:=\mathbf{r_0}$

$\alpha_\mathbf{k}:=\frac{\mathbf{r_k^T r_k}}{\mathbf{p_k^T Ap_k}}$

$\mathbf{x_{k+1}}:=\mathbf{x_k}+\alpha_\mathbf{k}\mathbf{p_k}$

$\mathbf{r_{k+1}}:=\mathbf{r_k}+\alpha_\mathbf{k}\mathbf{Ap_k}$

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

- $\beta_{\mathbf{k}} := \frac{\mathbf{r_{k+1}^T r_{k+1}}}{\mathbf{r_k^T r_k}}$
- $\mathbf{p_{k+1}} := \mathbf{r_{k+1}} + \beta_{\mathbf{k}} \mathbf{p_k}$
- $\mathbf{k} := \mathbf{k} + \mathbf{1}$

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 $\mathbf{A_{11}}$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-conditiond:

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 $\mathbf{x} + \mathbf{\Delta x}$ is the solution when $\mathbf{b}$ changes from $\mathbf{b}$ to $\mathbf{b} + \mathbf{\Delta b}$.

Then we can write:

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

or

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

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

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

So:

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

Using the matrix norm properties:

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

We can get:

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

Also, we can get:

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

Using equations 2.5 and 2.6 we can get:

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

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

we can rewrite equation 2.7 as:

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

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

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

$\mathbf{K(A)}$ is a measure of the relative sensitivity of the solution to changes in the right-hand side vector b.When the condition number $\mathbf{K(A)}$ 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。

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

y_i是观测值，x_i是已知变量，一共m组观测值。

当$\mathbf{f}(\mathbf{x}, \beta)$是线性方程组时

即 $\mathbf{f}(\mathbf{X},\beta)=\mathbf{A}\beta$
cost function 可以表示如下:

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

其中$\mathbf{Y}=[\mathbf{y_1},\mathbf{y_2}\cdots\mathbf{y_m}]^{\mathbf{T}}$
cost function 对β

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

求导,

- $-2A^TY+2(A^TA)\beta=0$

再让导数等于0,就可以求得解为:β

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

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。

- $f(x+\alpha\mathbf{d})=f(x_0)+\alpha f'(x_0)\mathbf{d}+O(\alpha^2)$
- $x_{t+1}=x_t-\alpha f'(x_t),$

牛顿法

把cost function进行展开到二阶:

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

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

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

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:

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

根据牛顿法:

$vx_{t+1}=x_t-H^{-1}g,$
梯度表示为:

∂r

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

对方程(4.9)求二阶导，我们可以得到Hessian矩阵：

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

当残差r_i很小的时候,我们就可以去掉最后一项，因此

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

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$diag(J^T J)$,因此迭代变成：

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

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 fucntion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没有是损失函数下降，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 fucntion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

这章主要讨论传统的机器学习方法，涉及到分类，回归，聚类，降维，决策树与集成学习

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，问问bagging, boosting
8. 生成模型，判别模型举个例子
9. 聚类方法的掌握，问问kmeans的EM推导思路，谱聚类和Grapg-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. factorizatuon 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 bayer, 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 Spilines
    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 Iteraction 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$\mathbf{y_D}$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：

- $E(f;D) = E_D[(f(\mathbf{x},D) - y_D)^2]$
  - $= E_D[(f(\mathbf{x},D) - \hat{f}(\mathbf{x}) + \hat{f}(\mathbf{x}) - y_D)^2]$
  - $= E_D[(f(\mathbf{x},D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y_D)2] + E_D[2(f(\mathbf{x},D) - \hat{f}(\mathbf{x}))(\hat{f}(\mathbf{x}$
  - $= E_D[(f(\mathbf{x},D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y_D)^2]$
  - $= E_D[(f(\mathbf{x},D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y + y - y_D)^2]$
  - $= E_D[(f(\mathbf{x},D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y)^2] + E_D[(y - y_D)^2] + E_D[2(\hat{f}(\mathbf{x})$
  - $= E_D[(f(\mathbf{x},D) - \hat{f}(\mathbf{x}))^2] + E_D[(\hat{f}(\mathbf{x}) - y)^2] + E_D[(y - y_D)^2]$ 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，可以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：$Y \in (1, 2, ..., K)$

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

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

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$f(\mathbf{X}, \mathbf{\Theta})$,其中$\mathbf{\Theta}$是模型参数，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：

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

等价于：

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

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

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

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

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，转化成极大化似然函数的求解。
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,...,K),输入变量$\mathbf{X}\in R^n$
则，模型如下：

$$P(Y=k|x)=\frac{exp(\mathbf{w_k x})}{1+\sum_{k=1}^{K-1}exp(\mathbf{w_k x})},k=1,2,...,K-1$$

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

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：

$$L(\mathbf{W})=\prod_{n=1}^{N}P(Y=Y_n|X=X_n)$$ 这样，我们需要求的参数是$W=(\mathbf{w_1},\mathbf{w_2},...,\mathbf{w_{n+1}})$.

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$(x_i, y_i), i = 1, 2...N, y_i \in +1, -1, x_i \in R^m$
SVM算法是为了求得一个分割超平面$\mathbf{wx} + b = 0$ 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: $\gamma_i = y_i(\frac{\mathbf{wx_i}}{||\mathbf{w}||} + \frac{b}{||\mathbf{w}||})$
除以$||w||$是因为x,b同时乘以一个因子，超平面不变。

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

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:

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

$$s.t. \qquad y_i(\frac{\mathbf{wx_i}}{||\mathbf{w}||} + \frac{b}{||\mathbf{w}||}) \geq \gamma$$

转化为函数间隔:

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

$$s.t. \qquad y_i(\mathbf{wx_i} + b) \geq ||w||\gamma = \hat{\gamma}$$

为了简化计算，我们取$\hat{\gamma} = 1$
因此问题转化为:

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

$$s.t. \qquad y_i(\mathbf{wx_i} + b) \geq 1, i = 1, 2..., N$$

转化为求极小问题:

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

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:

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

其中：$f: R^n->R, \mathbf{h}: R^n->R^m, m \leq n; \mathbf{g}: R^n->R^p$;
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：$\bigtriangledown h_i(\mathbf{x}^*)$,$\bigtriangledown g_j(\mathbf{x}^*), i \leq i \leq m, j \in J(\mathbf{x}^*)$

是线性无关的，则称$\mathbf{x}^*$是一个正则点。
KKT条件:某个点是局部技校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$\mathbf{h} = 0, \mathbf{g} \leq 0$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：

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

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

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

λ^*为拉格朗日乘子向量，$\mathbf{u}^*$是iKKT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f在$\mathbf{x}$处的Hassian矩阵;

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

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

其中$G_k(\mathbf{x})$是g_k处的Hassian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. $\mathbf{u^*}^{\mathbf{T}}\mathbf{g}(\mathbf{x}*) = 0$

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

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$D=((x_1,y_1),(x_2,y_2)...(x_N,y_N))$其中$x_i=(x_i^{(1)},x_i^{(2)}...x_i^{(n)})$是n维变量，n表示特征的数目，$y_i\in(1,2...K)$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:

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

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。

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

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$H(A)=\sum_{i=1}^{i=K}p_i\ln p_i$,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。

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

```
        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。

- $C(T)=\sum_{t=1}^{t=|T|}N_tH_t$

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$k\in(1,2...K)$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

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

假设已将输入空间划分为M个单元$R_1,R_2,...,R_M$,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：

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

当输入空间的划分确定时，可以用平方误差

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

来表示回归书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

$$\hat{c}_m=ave(y_i|x_i\in R_m)。$$

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：

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

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

若为离散特征，则找到第j个特征下的取值s:

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

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

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:

$$min_{j,s}[\sum_{x_i\in R_1}(y_i-c_1)^2+\sum_{x_i\in R_2}(y_i-c_2)^2]$$,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^*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 $D=((x_1,y_1),(x_2,y_2)...(x_N,y_N)).$

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,

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

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

其中：

- $\hat{c}_m = ave(y_i|x_i \in R_m)$。

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:

- $min_{j,s}[\sum_{x_i \in R_1}(y_i - c_1)^2 + \sum_{x_i \in R_2}(y_i - c_2)^2]$,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^*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 $D = ((x_1, y_1), (x_2, y_2)...(x_N, y_N))$.

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$R_1, R_2, ..., R_M$,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，$0 \le \alpha_1 \le \alpha_2 \le ... \le \alpha_n$，

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t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)：

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

以t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)：

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

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：

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

当α增大时，在某一α有：

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

当α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：

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

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α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：

$C(t)-C(T)$

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：

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

当α增大时，在某一α有：

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

当α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：

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

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α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：

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

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 . ↩

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

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

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。
{% math %}\kern{10 em}H(x) = sign(\sum _{i=1}^{T}h_i(x){% 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：

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

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：

AdaBoost

Boosting系列算法的代表就是AdaBoost。

算法如下：

输入：训练数据集 $T=((x_1,y_1),(x_2,y_2)...(x_N,y_N))$,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

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

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

(2)对m=1,2,...,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$G_m(x)G_m(x)$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：

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

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

这里，Z_m是归一化因子：

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

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:

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

得到最终分类器

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

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:

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

定理**(二分类问题AdaBoost的训练误差界)**

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

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(decsion 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。

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

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:

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

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

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

对于一个训练数据集 $T=((x_1,y_1),(x_2,y_2),...,(x_N,y_N)),x_i\in\chi\in R^N$, 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$R_1,R_2,...,R_J$，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:

- $T(x;\theta)=\sum_{j=1}^{J}c_jI(x\in R_j).$

其中，参数$\Theta=((R_1,c_1),(R_2,c_2),...,(R_J,c_J))$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:

- $f_0(x)=0$
- $f_m(x)=f_{m-1}(x)+T(x;\Theta_m),m=1,2,...,M$
- $f_m(x)=\sum_{m=1}^{M}T(x;\Theta_m)$

在前向分步算法的第m步，给定当前模型$f_{m-1}(x)$，需求解:

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

得到$\hat{\Theta}_m$，即第m颗树的参数。
采用平方误差损失函数时:

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

其损失变成:

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

这里,

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

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。

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

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 $T=((x_1,y_1),(x_2,y_2),...,(x_N,y_N)),x_i\in\chi\in R^N$，$\chi$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

- $f_0(x)=argmin_c\sum_{i=1}^{N}L(y_i,c).$

(2)对m=1,2,...,M

(a)对i=1,2,...,N，计算

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

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

(c)对j=1,2,..,J，计算

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

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

(3)得到回归树

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

函数空间的数值优化：

XGBoost[1]

模型复杂度惩罚是XGBoost相对于MART的提升。

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

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 $\hat{f}^{(0)}(x) = \hat{f}_{(0)}(x) = \hat{\theta}_0 = argmin_\theta \sum_{i=1}^{n} L(y_i, \theta)$;
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. $\hat{\phi}_m = argmin_{\phi \in \Phi} \sum_{i=1}^{n} \frac{1}{2} \hat{h}_m(x_i)[(-\frac{\hat{g}_m(x_i)}{\hat{h}_m(x_i)}) - \phi(x_i)]^2$
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),每一次迭代都市最小化如下损失函数:

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

对于梯度提升算法，基函数是如下的树:

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

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:

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

代入:

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

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

由于每个样本点只能属于一个叶子节点，因此:

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

因此上面可以简化成:

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

$$= \sum_{j=1}^{T} \sum_{i \in I_{jm}} [\hat{g}_m(x_i) w_{jm} + \frac{1}{2}\hat{h}_m(x_i) w_{jm}^2]$$

定义:

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

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

因此，可以把cost function写成：

$$J_m(\phi_m) = \sum_{j=1}^{T} [G_{jm} w_{jm} + \frac{1}{2} H_{jm} w_{jm}^2]$$

$$\blacksquare \geq -\sum_{j=1}^{T} \frac{1}{2} \frac{G_{jm}^2}{H_{jm}}$$

成立条件是：

$w_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, 2..., T$

为了寻找最佳分裂点j，也就是最大化如下的Gain:

$$\blacksquare Gain = \frac{1}{2}[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}]$$

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 $\hat{f}^{(0)}(x) = \hat{f}_{(0)}(x) = \hat{\theta}_0 = argmin_\theta \sum_{i=1}^{n} L(y_i, \theta)$;
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. Determin the structure $\hat{R}_{jm}, j = 1, ..., T$ by selecting splits which maximize

   $$\blacksquare Gain = \frac{1}{2}[\frac{G_L^2}{H_L} + \frac{G_R^2}{H_R} - \frac{G_{jm}^2}{H_{jm}}]$$

6. Determine the leaf weights $w_{jm}, j = 1, ..., T$ for the learnt structure by

   $\hat{w}_{jm} = -\frac{G_{jm}}{H_{jm}}, j = 1, 2..., T$
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

   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。

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颗树的期望与单颗树的期望一样，如果输之间是IID(independent identically distributed),每颗树的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$c \neq 0$,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。

1

> [1]. Tree Boosting With XGBoost ↩

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：$Y \in (1, 2, ..., K)$

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

条件独立假设：

$$P(X = x|Y = y) = P(X = x|Y = C_k) = P(X^{(1)} = x^{(1)}, X^{(2)} = x^{(2)}, ..., X^{(n)} = x^{(n)}|Y = C_k)$$
$$= \prod_{j=1}^{n} P(X^{(j)} = x^{(j)}|Y = C_k)$$

在上面的公式中，先验概率计算如下：

$$P(Y = C_k) = \frac{\sum_{i=1}^{N} I(y_i = c_k)}{N}, k = 1, 2..., K$$

条件概率计算

假设第j个特征$x^{(j)}$的可能取值集合是$(a_{j1}, a_{j2}, ..., a_{js_j})$，条件概率计算如下：

$$P(X^{(j)} = a_{jl}|Y = C_k) = \frac{\sum_{i=1}^{N} I(x_i^{(j)} = a_{jl}, y_i = c_k)}{\sum_{i=1}^{N} I(y_i = c_k)}$$

$j = 1, 2, ..., n; l = 1, 2, ..., S_j; k = 1, 2..., K$

后验概率计算

基于贝叶斯定理计算后验概率：

$P(Y=C_k|X=x)=\frac{P(X=x|Y=C_k)P(Y=C_k)}{\sum_k P(X=x|Y=C_k)P(Y=C_k)}$

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：

$y=f(x)=argmax_{c_k}P(Y=C_k)\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=C_k)$

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。

$$P(X^{(j)}=a_{jl}|Y=C_k)=\frac{\sum_{i=1}^{N}I(x_i^{(j)}=a_{jl},y_i=c_k)+\lambda}{\sum_{i=1}^{N}I(y_i=c_k)+s_j\lambda}$$

$\lambda>0, j=1,2,...,n; l=1,2,...,S_j; k=1,2...,K$

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：

$$\blacksquare P(Y=k|x) = \frac{exp(\mathbf{w_k x})}{1+\sum_{k=1}^{K-1} exp(\mathbf{w_k x})}, k=1,2,...,K-1$$

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

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：

$$\blacksquare L(\mathbf{W}) = \prod_{n=1}^{N} P(Y=Y_n|X=X_n)$$ 这样，我们需要求的参数是$W = (\mathbf{w_1}, \mathbf{w_2}, ..., \mathbf{w_{n+1}})$.

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：
$$\blacksquare P(Y=1|x) = \pi(x) = \frac{exp(\mathbf{w*x})}{1+exp(\mathbf{w*x})}$$
$$\blacksquare P(Y=0|x) = 1-\pi(x)$$

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：

$$\blacksquare L(\mathbf{W}) = \sum_{n=1}^{N} [y_i \log \pi(x_i) + (1-y_i) \log[1-\pi(x_i)]$$
$$\blacksquare = \sum_{n=1}^{N} [y_i(w*x_i) + \log(1+exp(w*x_i))$$

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Hassian矩阵比较麻烦，而拟牛顿法只需要构造Hassian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：$T=((x_1,y_1),(x_2,y_2),...,(x_N,y_N))$

以及特征函数$f_i(\mathbf{x},\mathbf{y}),i=1,2,...,n$，最大熵模型的学习等价于约束最优化问题：

$$\max_{P\in C} H(P)=\sum_{x,y}\hat{P}(x)P(y|x)\log P(y|x)$$

$$S.t.E_P(f_i)=E_{\hat{P}}(f_i),i=1,2,...,n \qquad \sum_y P(y|x)=1$$

可以转化为求极小问题：

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

$$S.t.E_P(f_i)-E_{\hat{P}}(f_i)=0,i=1,2,...,n \qquad \sum_y P(y|x)=1$$

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：$w_0,w_1,...,w_n$,定义拉格朗日函数$L(P,\mathbf{w})$;

$$L(P,\mathbf{w})=-H(P)+w_0(1-\sum_y P(y|x))+\sum_{i=1}^n w_i(E_P(f_i)-E_{\hat{P}}(f_i))$$

$$=-\sum_{x,y}\hat{P}(x)P(y|x)\log P(y|x)+w_0(1-\sum_y P(y|x))$$

$$+\sum_{i=1}^n w_i(\sum_{x,y}\hat{P}(x,y)f_i(x,y)-\sum_{x,y}\hat{P}(x)P(y|x)f_i(x,y))$$

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.

$$\blacksquare \frac{\partial L(P,W)}{\partial P(y|x)} = \sum_{x,y} \hat{P}(x)(\log P(y|x) + 1) - \sum_{y} w_0 - \sum_{x,y}(\hat{P}(x) \sum_{i=1}^{n} w_i f_i(x,y))$$

$$\blacksquare = \sum_{x,y} \hat{P}(x)(\log P(y|x) + 1 - w_0 - \sum_{i=1}^{n} w_i f_i(x,y))$$

令其偏导数等于0，在$\hat{P}(x) > 0$的情况下，解得：

$$\blacksquare P(y|x) = \exp(\sum_{i=1}^{n} w_i f_i(x,y)) + w_0 - 1)$$

由于：$\sum_{y} P(y|x) = 1$,得：

$$\blacksquare P_w(y|x) = \frac{1}{Z_w(x)} \exp(\sum_{i=1}^{n} w_i f_i(x,y)))$$

其中：

$$\blacksquare Z_w(x) = \sum_{i=1} \exp(\sum_{y}^{n} w_i f_i(x,y)))$$

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：

$$\blacksquare w^* = arg \max_{w} \Psi(\mathbf{w})$$

高斯混合模型

三硬币模型

三枚硬币A, B, C正面出现的正面的概率是π, p, q.进行如下实验：先掷硬币A，正面选硬币B，反面选硬币C；然后掷选出的硬币，掷硬币的结果出现正面记作1，反面记作0.独立地重复n次实验，得到一个观察序列如下：
1,1,0,1,0,0,1,0,1,1
三硬币模型可以写作：

$$P(y|\theta) = \sum_z P(y,z|\theta) = \sum_z P(z|\theta)P(y|z,\theta)$$

$$= \pi p^y(1-p)^{1-y} + (1-\pi)q^y(1-q)^{1-y}$$

$\Theta = (\pi, p, q)$是模型参数，随机变量y的数据可以观察，随机变量z的数据不可以观察。

最大似然函数

寻找一组模型参数，使得在该参数下，观测序列出现的概率极大。

将可观察数据表示为$Y = (Y_1, Y_2, ..., Y_n)^T$，为观测数据表示为 $Z = (Z_1, Z_2, .., Z_n)^T$.则观察数据的似然函数是：

$$P(Y|\theta) = \sum_Z P(Z|\theta)P(Y|Z,\theta)$$

$$= \prod_{j=1}^{n}[\pi p^y(1-p)^{1-y} + (1-\pi)q^y(1-q)^{1-y}]$$

考虑求模型参数 $\Theta = (\pi, p, q)$的极大似然估计，即：

$$\hat{\theta} = arg\max_\theta log(P(Y|\theta))$$

只能通过迭代来求解模型参数。EM算法就是该问题的一种方法。

EM算法

E步：计算模型参数$\pi^{(i)}, p^{(i)}, q^{(i)}$下观测数据$y_j$来自掷硬币B的概率是(无论是正面还是方面，都可以用下面公式进行计算)：

$$u^{(i+1)} = \frac{\pi^{(i)}(p^{(i)})^{y_j}(1-p^{(i)})^{1-y_j})}{\pi^{(i)}(p^{(i)})^{y_j}(1-p^{(i)})^{1-y_j}) + (1-\pi^{(i)})(q^{(i)})^{y_j}(1-q^{(i)})^{1-y_j})}$$

M步：计算模型参数的新估计值：
之所以抛硬币B是因为抛硬币A出现了正面，因此π(抛A出现正面的几率如下计算)

$$\pi^{(i+1)} = \frac{1}{n}\sum_{j=1}^{n} u^{(i+1)}$$

在抛硬币B的前提下，根据抛B得到正面的频率得到P(B正面出现的概率)

$$p^{(i+1)} = \frac{\sum_{j=1}^{n} u^{(i+1)} y_j}{\sum_{j=1}^{n} u^{(i+1)}}$$

在抛硬币C的前提下(1-来自抛B的概率)，根据抛C得到正面的频率得到P(C正面出现的概率)

$$p^{(i+1)} = \frac{\sum_{j=1}^{n} (1-u^{(i+1)}) y_j}{\sum_{j=1}^{n} (1-u^{(i+1)})}$$

（EM算法）

输入：观察变量数据Y，隐变量数据Z，联合分布$P(Y,Z|\theta)$,条件分布$P(Z|Y,\theta)$;
输出：模型参数θ.

(1)选择参数的初始值$\theta^{(0)}$,开始迭代；

(2)E步：记$\theta^{(i)}$为第i次迭代参数θ的估计值，在第i+1次迭代的E步，计算：

$$Q(\theta,\theta^{(i)}) = E_z[\log P(Y,Z|\theta)|Y,\theta^{(i)}] = \sum_z P(Z|Y,\theta^{(i)}) \log P(Y,Z|\theta)$$

这里，$P(Y,Z|\theta^{(i)})$是在给定

观测数据Y和当前的参数估计$\theta^{(i)}$下隐变量数据Z的条件概率分布：

(3)M步（求导令其等于0得极大）：求使$Q(\theta,\theta^{(i)})$极大化的θ,确定第i+1次迭代的参数的估计值

$\theta^{(i+1)}$:

$$\theta^{(i+1)} = arg\max_\theta Q(\theta,\theta^{(i)})$$

(4)重复第(2)(3)步，直到收敛。

Q函数

完全数据的对数似然函数$\log P(Y,Z|\theta)$关于给定观测数据Y和当前参数θ下对未观察数据Z的条件概率分布$\log P(Z|Y,\theta^{(i)})$的期望成为Q函数，即：

$$Q(\theta,\theta^{(i)}) = E_z[\log P(Y,Z|\theta)|Y,\theta^{(i)}]$$

高斯混合模型

高斯混合模型是指的具有如下形式的概率分布模型:

$$P(y|\theta)=\sum_{k=1}^{K}\alpha_k\phi(y|\theta_k)$$

其中α_k是系数$\alpha_k\geq 0,\sum_{k=1}^{K}\alpha_k=1$;$\phi(y|\theta_k)$是高斯分布函数, $\theta_k=(u_k,\sigma_k^2)$,

$$P(y|\theta)=\frac{1}{\sqrt{2\pi}\sigma_k}exp(-\frac{(y-u_k)^2}{2\sigma_k^2})$$

称为第k个分模型。

高斯混合模型参数股记得EM算法

可以用EM算法来求解GMM的参数α_k,u_k,σ_k.

$\gamma_{jk}=1$表示第j个观测来自第k个分模型, $\gamma_{jk}=0$表示第j个观测不是来自第k个分模型。

有了观察数据y_j以及未观测数据γ_{jk},则完全数据是:

$$(y_j,\gamma_{j1},\gamma_{j2},...,\gamma_{jK}),j=1,2,...,K$$

似然函数为:

$$P(y,\gamma|\theta)=\prod_{j=1}^{N}P(y_j,\gamma_{j1},\gamma_{j2},...,\gamma_{jK}|\theta)$$

$$=\prod_{k=1}^{K}\prod_{j=1}^{N}[\alpha_k\phi(y_j|\theta_k)]^{\gamma_{jk}}$$

$$=\prod_{k=1}^{K}\alpha_k^{n_k}\prod_{j=1}^{N}[\phi(y_j|\theta_k)]^{\gamma_{jk}}$$

$$=\prod_{k=1}^{K}\alpha_k^{n_k}\prod_{j=1}^{N}[\frac{1}{\sqrt{2\pi}\sigma_k}\exp(-\frac{(y-u_k)^2}{2\sigma_k^2})]^{\gamma_{jk}}$$

式中$\sum_{j=1}^{N}\gamma_{jk}=n_k,\sum_{k=1}^{K}n_k=N$。

转化为完全数据的对数似然函数。

隐马尔科夫模型

定义

HMM类似于量子力学中的路径积分。马尔科夫模型是关于时序的概率模型，描述由一个隐藏的马尔科夫链生成不可观测的状态随机序列，再由各个状态生成一个观测而产生观测随机序列的过程。隐藏的马尔科夫链随机生成的状态的序列成为状态序列（state sequence）;每个状态生成一个观测，而由次产生的观测的随机序列成为观测序列（observation sequence）.序列的每一个位置又可以看成一个时刻。
马尔科夫模型由初始概率分布，状态转移概率分布以及观测概率分布确定。
设Q是所有可能的状态集合，V是所有可能的观测的集合。

$$Q=(q_1,q_2,...,q_N),V=(v_1,v_2,...,v_M)$$

其中N是可能的状态数，M是可能的观测数。
I是长度为T的状态序列，O是对应的观测序列。

$$I=(i_1,i_2,...,i_T),O=(o_1,o_2,...,o_T)$$

A是状态转移矩阵：

$$A=[a_{ij}]_{NxN}$$

其中：

$$a_{ij}=P(i_{t+1}=q_j|i_t=q_i),i=1,2,...,N,j=1,2,..,N$$

是在t时刻处于q_i的条件下在时刻t+1转移到q_j的概率。
B是观察概率矩阵：

$$A=[b_j(k)]_{NxM}$$

其中：

$$b_j(k)=P(o_t=v_k|i_t=q_j),k=1,2,..,M;j=1,2,...,N$$

是在时刻t处于状态q_j的条件下生成观测v_k的概率。
π是初始状态概率向量：

$$\pi=(\pi_i)$$

其中：

$$\pi_i=P(i_1=q_i),i=1,2,...,N$$

是在时刻t=1处于状态q_i的概率。
马尔科夫模型λ可以用三元符号来表示：

$$\lambda=(A,B,\pi)$$

A,B,π称为马尔科夫模型的三要素。

马尔科夫模型的两个假设:

(1)齐次马尔科夫假设。任一时刻只与前一时刻有关系

- $P(i_t|i_{t-1},o_{t-1},...,i_1,o_1)=P(i_t|i_{t-1}),t=1,2,...,T$

(2)观测独立性假设:任一时刻的观测只依赖于该时刻的马尔科夫链的状态，而与其他观测以及状态没有关系。

- $P(o_t|i_T,o_T,...,i_{t+1},o_{t+1},i_t,i_{t-1},o_{t-1},i_1,o_1)=P(o_t|i_t)$

HMM的三个问题

1. 概率计算问题:给定模型$\lambda=(A,B,\pi)$,和观测序列$O=o_1,o_2,...,o_T$,计算在模型λ下观测序列O出现的概率$P(O|\lambda)$。
2. 学习问题:已知观测序列$O=o_1,o_2,...,o_T$,估计模型$\lambda=(A,B,\pi)$,使得$P(O|\lambda)$最大。即用极大似然法的方法估计参数。
3. 预测问题(也称为解码(decoding)问题):已知观测序列$O=o_1,o_2,...,o_T$和模型

   $\lambda=(A,B,\pi)$,求给定观测序列条件概率$P(I|O)$最大的序列$I=(i_1,i_2,...,i_T)$,即给定观测序列，求最有可能的对应的状态序列。

概率计算问题

直接计算方法:

问题:给定模型$\lambda=(A,B,\pi)$,和观测序列$O=o_1,o_2,...,o_T$,计算在模型λ下观测序列O出现的概率$P(O|\lambda)$。

对于状态序列$I=(i_1,i_2,...,i_T)$的概率是:$P(I|\lambda)=\pi_{i_1}a_{i_1i_2}a_{i_2i_3}...a_{i_{T-1}i_T}$,

对于上面这种状态序列，产生观测序列$O=o_1,o_2,...,o_T$的概率是:

$P(O|I,\lambda)=b_{i_1}(o_1)b_{i_2}(o_2)...b_{i_T}(o_T)$,
因此I和O的联合概率是:

$P(O,I|\lambda)=P(O|I,\lambda)P(I|\lambda)=b_{i_1}(o_1)b_{i_2}(o_2)...b_{i_T}(o_T)\pi_{i_1}a_{i_1i_2}a_{i_2i_3}...a_{i_{T-1}i_T}$
对所有可能的I求和，得到:

- $P(O|\lambda)=\sum_I P(O,I|\lambda)$

直接计算，时间复杂度是$O(TN^T)$.为了降低计算的时间复杂度，引入了前向算法。

前向算法

给定模型λ，定义到时刻t部分观测序列为$O=o_1,o_2,...,o_T$且状态为q_i的概率为前向概率。记作：

$$\alpha_t(i)=P(o_1,o_2,...,o_t,i_t=q_i|\lambda)$$

有了前向概率就可以通过递归的方法来计算任何一个观测序列的概率。就如下图所示：

前向算法使用**前向概率**的概念，记录每个时间下的前向概率，使得在递推计算下一个前向概率时，只需要上一个时间点的所有前向概率即可。原理上也是用空间换时间。这样的**时间复杂度是** $O(N^2T)$。

上面就是状态数目为5，时间T=4的图。一个问题是，前向算法是否可以转化为矩阵求解问题？

$$\alpha_t=(\alpha_t(1),\alpha_t(2),...,\alpha_t(N))^T$$

初始状态是π,转移矩阵是A，因此时刻t的概率分布是一个NxM的矩阵，也就是N个状态，每个状态有M个分立值，计算如下：问题是怎么降低运算复杂度？N^T的复杂度是$O(N^{2.373}\log T)$

后向概率

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, ..., o_T | i_t = q_i, \lambda)$$

定义 $\beta_T(i) = 1, i = 1, 2, ..., N$

对$t = T-1, T-2, ..., 1$有:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j), i = 1, 2, ..., N$$

$$\beta_t = A^{T-t} \prod_{j=t+1}^{T} xB[o_j], t = T-1, T-2, ..., 1$$

$$\beta_t = (\beta_t(1), \beta_t(2), ..., \beta_t(N))^T$$

一些概率与期望值

(1) 给定模型λ以及观测O，在时刻t处于状态q_i的概率为:

$$\gamma_t(i) = P(i_t = q_i | O, \lambda) = \frac{P(i_t = q_i, O|\lambda)}{P(O|\lambda)}$$

因为 $\alpha_t(i)\beta_t(i) = P(i_t = q_i, O|\lambda)$

所以: $$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)}$$

(2)给定模型λ以及观测O，在时刻t处于状态q_i且在t+1时刻处以q_j的概率为:

$$\xi_t(i, j) = P(i_t = q_i, i_{t+1} = g_j | O, \lambda)$$

$P(i = q, i = g, O|\lambda)$ $P(i = q, i = g, O|\lambda)$

- $\xi_t(i,j)=\frac{P(i_t=q_i,i_{t+1}=g_j,O|\lambda)}{P(O|\lambda)}=\frac{P(i_t=q_i,i_{t+1}=g_j,O|\lambda)}{\sum_{i=1}^{N}\sum_{j=1}^{N}P(i_t=q_i,i_{t+1}=g_j,O|\lambda)}$

而:

- $P(i_t=q_i,i_{t+1}=g_j,O|\lambda)=\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)$

所以:

- $\xi_t(i,j)=\frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}$

(3)将$\gamma_t(i),\xi_t(i,j)$对不同时刻t求和，可以得到一些有用的期望:

(a)在观测O下状态i出现的期望值:

- $\sum_{i=1}^{T}\gamma_t(i)$

(b)在观测O下由状态i转移的期望值:

- $\sum_{i=1}^{T-1}\gamma_t(i)$

(c)在观测O下由状态i转移到状态j的期望值:

- $\sum_{i=1}^{T-1}\xi_t(i,j)$

学习算法

监督学习方法

我们有观测序列和对应的状态序列$[(O_1,I_1),(O_2,I_2),...,(O_S,I_S)]$

怎么可以通过统计方法来得到转移矩阵A，观测概率B，以及初始状态概率π

- $\hat{a}_{ij}=\frac{A_{ij}}{\sum_{j=1}^{N}A_{ij}},i=1,2,..,N;j=1,2,...,N$
- $\hat{b}_j(k)=\frac{B_{jk}}{\sum_{k=1}^{M}B_{jk}},j=1,2,..,N;k=1,2,...,M$

初始状态概率π_i的估计值$\hat{\pi}_i$为S个样本中初始状态为q_i的频率。

人工标注的成本很高，因此就会利用非监督学习的方法。

Baum-Welch算法思想

假设给定训练数据只包含S个长度为T的观察序列$(O_1,O_2,...,O_s)$而没有对应的状态序列，目

标是学习HMM模型$\lambda=(A,B,\pi)$的参数。我们将观测序列数据堪称观察数据O,状态序列数据看作不可观测的隐数据I,那么隐马尔科夫模型事实上是一个含有隐变量的概率模型:

$$P(O|\lambda)=\sum_I P(O|I,\lambda)P(I|\lambda)$$

他的参数学习可以由EM算法来实现。

4. 确定完全数据的对数似然函数

所有观测数据写成$O=(o_1,o_2,...,o_T)$,所有隐数据写成$I=(i_1,i_2,...,i_T)$,完成数据是

$(O,I)=(o_1,o_2,...,o_T,i_1,i_2,...,i_T)$.完全数据的对数似然函数是$\log P(O,I|\lambda)$

5. EM算法的E步，求Q函数$Q(\lambda,\hat{\lambda})$

$$Q(\lambda,\hat{\lambda})=E_I[\log P(O,I|\lambda)|O,\hat{\lambda}]$$

$$Q(\lambda,\hat{\lambda})=\sum_I \log P(O,I|\lambda)P(O,I|\hat{\lambda})$$

其中,$\hat{\lambda}$是隐马尔科夫模型参数的当前估计值，λ是要极大化的马尔科夫模型参数。

$$P(O,I|\lambda)=\pi_{i_1}b_{i_1}(o_1)b_{i_2}(o_2)...b_{i_T}(o_T)\pi_{i_1}a_{i_1i_2}a_{i_2i_3}...a_{i_{T-1}i_T}$$

于是函数$Q(\lambda,\hat{\lambda})$可以写成:

$$Q(\lambda,\hat{\lambda})=\sum_I \log \pi_{i_1}P(O,I|\hat{\lambda})+\sum_I(\sum_{t=1}^{T-1}\log a_{i_ti_{t+1}})P(O,I|\hat{\lambda})+\sum_I(\sum_{t=1}^{T-1}\log b_{i_t}(o_t))P$$

式中求和都是对所有训练数据的序列总长度T进行的。

6. EM算法的M步:极大化Q函数$Q(\lambda,\hat{\lambda})$求模型参数A,B,λ。极大化，满足梯度为0，也就对A，B,λ中的元素分别求导令其为0。

在$Q(\lambda,\hat{\lambda})$中的三项中，分别只包含了$\pi,a_{ij},b_j(k)$,因此求解起来比较方便。此外，我们还可以利用概率和为1的限定条件，也就是利用拉格朗日乘子来进行计算。

对$Q(\lambda,\hat{\lambda})$的第一项，我们求λ:

$$\sum_I \log \pi_{i_1}P(O,I|\hat{\lambda})=\sum_{i=1}^N \log \pi_i P(O,i_1=i|\hat{\lambda})$$

因为π_i满足约束条件$\sum_{i=1}^N \pi_i=1$,利用拉格朗日乘子，写出拉格朗日函数:

$$\sum_{i=1}^N \log \pi_i P(O,i_1=i|\hat{\lambda})+\gamma(\sum_{i=1}^N \pi_i-1)$$

对其求偏导并令其为0:

$$\frac{\partial}{\partial \pi_i}[\sum_{i=1}^N \log \pi_i P(O,i_1=i|\hat{\lambda})+\gamma(\sum_{i=1}^N \pi_i-1)]=0$$

得到:

$$P(O,i_1=i|\hat{\lambda})+\gamma\pi_i=0$$

对i求和得到γ:

$$\gamma = -P(O|\hat{\lambda})$$

代入上上式，得到：

$$\pi_i = \frac{P(O,i_1=i|\hat{\lambda})}{P(O|\lambda)}$$

对$Q(\lambda,\hat{\lambda})$的第二项，我们求a_{ij}：

$$\sum_I(\sum_{t=1}^{T-1}\log a_{i_t i_{t+1}})P(O,I|\hat{\lambda}) = \sum_{i=1}^{N}\sum_{j=1}^{T-N}\sum_{t=1}^{T-1}\log a_{ij}P(O,i_t=i,i_{t+1}=j|\hat{\lambda})$$

利用约束条件$\sum_{i=1}^{N}a_{ij}=1$，利用拉格朗日乘子可以得到：

$$a_{ij} = \frac{\sum_{t=1}^{T-1}P(O,i_t=i,i_{t+1}=j|\hat{\lambda})}{\sum_{t=1}^{T-1}P(O,i_t=i|\lambda)}$$

对$Q(\lambda,\hat{\lambda})$的第三项，我们求$b_j(k)$：

$$\sum_I(\sum_{t=1}^{T-1}\log b_{i_t}(o_t))P(O,I|\hat{\lambda}) = \sum_{j=1}^{N}\sum_{t=1}^{T}\log b_j(o_t))P(O,i_t=j|\hat{\lambda})$$

利用$\sum_{k=1}^{M}b_j(k)=1$. 注意只有在$o_t=v_k$时，$b_j(o_t)$对$b_j(k)$的偏导数才不为0，以$I(o_t=v_k)$表示，求得：

$$b_j(k) = \frac{\sum_{t=1}^{T}P(O,i_t=j|\hat{\lambda})I(o_t=v_k)}{\sum_{t=1}^{T}P(O,i_t=j|\hat{\lambda})}$$

$\hat{\lambda}$是当前(A,B,π)的估计值，因此可以通过迭代，得到 (A,B,π)的收敛解。

Baum-Welch算法

输入：观测数据$O=(o_1,o_2,...,o_T)$;

输出：隐马尔科夫模型的参数

(1)初始化

对n=0,选取$a_{ij}^{(0)},b_j(k)^{(0)},\pi_i^{(0)}$得到模型$\lambda^{(0)}=(A^{(0)},B^{(0)},\pi^{(0)})$

(2).递推对n=1,2,..,

$$a_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\xi_t(i)}$$

$$b_j(k) = \frac{\sum_{t=1,o_t=v_k}^{T}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j)}$$

$$\pi_i = \gamma_1(i)$$

右端的值按照观测$O=(o_1,o_2,...,o_T)$和模型$\lambda^{(n)}=(A^{(n)},B^{(n)},\pi^{(n)})$,以及式中$\xi_t(i,j)$按照中“一些概率与期望值”中定义进行计算。

(3)终止，得到模型参数$\lambda^{(n+1)}=(A^{(n+1)},B^{(n+1)},\pi^{(n+1)})$

预测算法

前面已经讨论了给定观测序列$O=(o_1,o_2,...,o_T)$的概率计算问题以及通过观测序列学习模型参数问题，还余下HMM三个问题中的给定观测下隐变量的预测问题。
前面，我们在“一些概率与期望值”中通过前向概率和后向概率，得到了当前模型参数以及观察序列下，某一时刻隐变量的取值概率$\xi_t(i)$。

$$\gamma_t(i)=\frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N}\alpha_t(i)\beta_t(i)}$$

计算每一时刻t最可能出现的状态i,然后得到最有可能出现的序列。这就是近似算法.

近似算法

$$i_t^*=arg\max_{1\le i\le N}[\gamma_t(i)],t=1,2,..,T$$

从而得到状态序列$I^*=(i_1^*,i_2^*,..,i_T^*)$。近似算法简单，但是会预测出一些实际上不发生的状态，比如转移矩阵等于0的两个相邻状态。尽管如此，近似算法还是有用的。

维特比算法思想

维特比算法是用动态规划解马尔科夫模型预测问题，即用动态规划求概率最大路径(最优路径)，这时，一条路径对应一个状态序列。

根据动态规划原理，最优路径具有这样的特性：如果最优路径在时刻t通过节点i_t^*,那么这一路径从结点i_t^*到终点i_T^* 的部分路径，对于从 i_t^*到i_T^*的所有可能部分路径来说，必须是最优的。

因为假如不是这样，那么从i_t^*到i_T^*就有另外一条更好的部分路径存在，如果他和从i_t^*到i_T^*的部分路径连接起来，就会形成一条比原来的路径更优的路径，这是矛盾的。
根据这一原理，我们

首先定义两个变量δ,ψ,定义在时刻t状态为i的所有单个路径$(i_1,i_2,...,i_t)$中概率最大值为：

$$\delta_t(i)=\max_{i_1,i_2,...,i_{t-1}}P(i_t=i,i_{t-1},i_{t-2},...,i_1,o_t,...,o_1|\lambda),i=1,2,..,N$$

由定义可以得到变量δ的递推公式：

- $\delta_{t+1}(i) = \max\limits_{i_1,i_2,...,i_t} P(i_{t+1}=i, i_{t-1}, i_{t-2}, ..., i_1, o_{t+1}, ..., o_1|\lambda)$

  - $= \max\limits_{1\le j\le N}[\delta_t(j)a_{ji}]b_i(o_{t+1}), i=1,2,..,N; t=1,2,...,T-1$

定义在时刻t状态i的所有单个路径$(i_1, i_2, ..., i_{t-1}, i)$中概率最大的路径的第t-1个节点为:

- $\psi_t(i) = \arg\max\limits_{1\le j\le N}[\delta_{t-1}(j)a_{ji}], i=1,2,..,N$

维特比算法步骤

输入:模型$\lambda=(A,B,\pi)$和观测数据$O=(o_1,o_2,...,o_T)$;

输出: 最优化路径$I^*=(i_1^*, i_2^*, .., i_T^*)$。

(1). 初始化

- $\delta_1(i) = \pi_i b_i(o_1), i=1,2,..,N$

- $\psi_1(i) = 0, i=1,2,..,N$

(2). 递推, 对t=2,3,...,T

- $\delta_{t+1}(i) = \max\limits_{1\le j\le N}[\delta_t(j)a_{ji}]b_i(o_{t+1}), i=1,2,...,N$

- $\psi_t(i) = \arg\max\limits_{1\le j\le N}[\delta_{t-1}(j)a_{ji}], i=1,2,..,N$

(3). 终止

- $P^* = \max\limits_{1\le j\le N}\delta_T(i)$

- $i_T^* = \arg\max\limits_{1\le j\le N}\delta_T(i)$

(4)最优路径回溯。对t=T-1,T-2,....,1

- $i_t^* = \psi_{t+1}(i_{t+1}^*)$

求得最优路径$I^*=(i_1^*, i_2^*, .., i_T^*)$.

DNN-HMM混合系统

DNN-HMM混合系统利用DNN强大的表示学习能力以及HMM的序列化建模能力，他在大词汇连续语音识别任务重的表现远优于传统的(GMM-)HMM.

马尔科夫链蒙特卡洛方法

参考这篇文章

基本思想

对于一个给定的概率分布P(X)，若要得到其样本，通过上述的马尔科夫链的概念，我们可以构造一个转移矩阵P的马尔科夫链，使得该MC的平稳分布是P(X),这样，无论初始状态是何值，比如是x_0,那么随着马尔科夫过程的转移，得到了一系列的状态值，如:$x_0, x_1, ..., x_n, x_{n+1}, ...$,如果这个马尔科夫过程在第n步已经收敛，那么分布P(X)的样本就是$x_n, x_{n+1}, ...$。

细致平衡条件

假设一个各态便利的马尔科夫过程，其转移矩阵是P,分布是$\pi(x)$,若满足：

$$\pi(i)P_{i,j} = \pi(j)P_{j,i}$$

则$\pi(x)$是马尔科夫链的平稳分布，上式称为细致平衡条件。

Metropolos采样算法

Metropolos采样算法的基本原理

假设需要从目标概率密度$p(\theta)$中进行采样，同时，θ满足$-\infty < \theta < \infty$.Metropolis采样算法根据马尔科夫链去生成一个序列：

$\theta^{(1)} \rightarrow \theta^{(2)} \rightarrow ...\theta^{(t)} \rightarrow$

其中，$\theta^{(t)}$表示的是马尔科夫链在第t时刻的状态。

在Metropolis采样算法过程中，首先初始化状态值$\theta^{(1)}$，然后利用一个已知的分布$q(\theta|\theta^{(t-1)})$生成一个新的候选状态$\theta^{(*)}$,随后根据一定的概率选择接受这个新值还是拒绝这个新值，在Metropolos采样中，概率为：

$$\alpha = \min(1, \frac{p(\theta^{(*)})}{p(\theta^{(t-1)})})$$

这样的过程一直持续到采样过程的收敛，当收敛以后，样本$\theta^{(t)}$即为目标分布$P(\theta)$中的样本。

Metropolos采样算法的流程

基于以上的分析，可以总结出如下的Metropolis采样算法的流程
初始化时间t=1

设置u的值，并初始化初始状态$\theta^{(t)} = u$
重复以下的过程：

- 令t=t+1
- 从已知分布$q(\theta|\theta^{(t-1)})$中生成一个候选状态$\theta^{(*)}$
- 计算接受的概率：$\alpha = \min(1, \frac{p(\theta^{(*)})}{p(\theta^{(t-1)})})$
- 从均匀分布Uniform(0,1)生成一个随机值a.
- 如果$a < \alpha$,则接受新生成的值：$\theta^{(t)} = \theta^{(*)}$;

否则$\theta^{(t)} = \theta^{(t-1)}$ 直到t=T.

Metropolos采样算法的解释

要证明Metropolis采样算法的正确性，最重要的是要证明构造的马尔科夫过程满足细致平衡条件，即：

- $\pi(i)P_{i,j} = \pi(j)P_{j,i}$

对于上面的过程，分布为$p(\theta)$,从状态i转移到状态j的转移概率为：

- $P_{i,j} = \alpha_{i,j}Q_{i,j}$

其中，$Q_{i,j}$为上述已知分分布，且是对称的分布$Q_{i,j} = Q_{j,i}$。即:

- $q(\theta = \theta^{(t)}|\theta^{(t-1)}) = q(\theta = \theta^{(t-1)}|\theta^{(t)})$ 接下来，我们需要证明Metropolis采样算法中构造的马尔科夫链满足细致平衡条件：

- $p(\theta^{(i)}P_{i,j} = p(\theta^{(i)}\alpha_{i,j}Q_{i,j}$
  - $= p(\theta^{(i)}\min(1, \frac{p(\theta^{(j)})}{p(\theta^{(i)})})Q_{i,j}$
  - $= \min(p(\theta^{(i)}Q_{i,j}, p(\theta^{(j)}Q_{i,j})$
  - $= p(\theta^{(j)}\min(\frac{p(\theta^{(i)})}{p(\theta^{(j)})}, 1)Q_{j,i}$
  - $= p(\theta^{(i)}\alpha_{i,j}Q_{i,j}$
  - $= p(\theta^{(j)}P_{j,i}$

因此，通过上面方法构造出来的马尔科夫链满足细致平衡条件。

MCMC的使用场景

MCMC——Metropolis-Hasting算法
MCMC——Gibbs Sampling算法

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$\mathbf{X}$近似表示为:
$\mathbf{X} \approx \mathbf{WH}$

其中:$\mathbf{W} \in R^{Nxr}, \mathbf{H} \in R^{rxp}, r \leq max(N,p)$,我们假定$x_{ij}, w_{ik}, h_{kj} \geq 0$.
矩阵$\mathbf{W}, \mathbf{H}$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:

$$\hat{\beta}^{ridge} = \arg min_{\beta} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$$

$$subjectto: \sum_{j=1}^{p} \beta_j^2 \leq t.$$

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X进行SVD分解:

$$\mathbf{X} = \mathbf{UDV^T}$$

我们重写没有正则化的最小二乘拟合:

$$\mathbf{X}\beta^{ls} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{UU^Ty}$$

对于L2正则化的最小二乘法:

$$\mathbf{X}\beta^{ls} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{UD}(\mathbf{D^2} + \lambda\mathbf{I})^{-1}\mathbf{DU^Ty}$$

$$= \sum_{j=1}^{p} \mathbf{u_j} \frac{d_j^2}{d_j^2 + \lambda} \mathbf{u_j}^T\mathbf{y}$$

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:

- $\hat{\beta}^{ridge} = \arg min_{\beta} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2$
- $subjectto: \sum_{j=1}^{p} |\beta_j| \leq t.$

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。

VC维

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。

$y(x) = ax^3 + bx^2 + cx + d + random$ tensorflownews

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classfication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

- $\hat{y} = \lambda_0 + \lambda_1 x_1 + \lambda_2 x_2 + ... + \lambda_m x_m$

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。

- $L(b) = \sum_{i=0}^{n-1}[y_i - (\lambda_0 + \lambda_1 x_{1i} + \lambda_2 x_{2i} + ... + \lambda_m x_{mi})]^2$

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。这只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

- $\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1m} & 1 \\ x_{21} & x_{22} & \cdots & x_{2m} & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nm} & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \cdots \\ \lambda_n \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} y_{1o} \\ y_{2o} \\ \cdots \\ y_{no} \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_{2o} \\ \cdots \\ \epsilon_{no} \end{bmatrix}$

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

- $L(\lambda) = \frac{1}{2m} \sum_{i=0}^{n-1} (y_i - \sum_{k=0}^{m} \lambda_k x_i^k)^2$ 令：
- $\lambda = [\lambda_0, \lambda_1, \cdots, \lambda_m]^T$

- $X_i = [x_i^0, x_i^1, \cdots, x_i^m]$
- $Y = [Y_0, Y_1, \cdots, Y_{n-1}]^T$

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：

- $$\begin{bmatrix} \sum_{j=0}^{n-1} x_j^0 & \sum_{j=0}^{n-1} x_j^1 & \cdots & \sum_{j=0}^{n-1} x_j^m \\ \sum_{j=0}^{n-1} x_j^1 & \sum_{j=0}^{n-1} x_j^2 & \cdots & \sum_{j=0}^{n-1} x_j^{m+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{j=0}^{n-1} x_j^m & \sum_{j=0}^{n-1} x_j^{m+1} & \cdots & \sum_{j=0}^{n-1} x_j^{2m} \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \cdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n-1} y_j * x_j^0 \\ \sum_{j=0}^{n-1} y_j * x_j^1 \\ \cdots \\ \sum_{j=0}^{n-1} y_j * x_j^m \end{bmatrix} \text{即:} X * \lambda = Y$$

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：

- $$\begin{bmatrix} \sum_{j=0}^{n-1} x_j^0 - 2n & \sum_{j=0}^{n-1} x_j^1 & \cdots & \sum_{j=0}^{n-1} x_j^m \\ \sum_{j=0}^{n-1} x_j^1 & \sum_{j=0}^{n-1} x_j^2 - 2n & \cdots & \sum_{j=0}^{n-1} x_j^{m+1} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{j=0}^{n-1} x_j^m & \sum_{j=0}^{n-1} x_j^{m+1} & \cdots & \sum_{j=0}^{n-1} x_j^{2m} - 2n \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \cdots \\ \lambda_m \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{n-1} y_j * x_j^0 \\ \sum_{j=0}^{n-1} y_j * x_j^1 \\ \cdots \\ \sum_{j=0}^{n-1} y_j * x_j^m \end{bmatrix}$$

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 y 对标量 x 求导

我们假定所有的向量都是列向量 $\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} & \frac{\partial y_2}{\partial x} & \cdots & \frac{\partial y_m}{\partial x} \end{bmatrix}$

标量 y 对向量 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

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

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 Y是mxn矩阵

$$\frac{\partial \mathbf{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_{11}}{\partial x} & \frac{\partial y_{21}}{\partial x} & \cdots & \frac{\partial y_{m1}}{\partial x} \\ \frac{\partial y_{12}}{\partial x} & \frac{\partial y_{22}}{\partial x} & \cdots & \frac{\partial y_{m2}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{1n}}{\partial x} & \frac{\partial y_{2n}}{\partial x} & \cdots & \frac{\partial y_{mn}}{\partial x} \end{bmatrix}$$

用维度分析来解决求导的形式问题

向量对向量的微分 $\frac{\partial(\mathbf{Ax})}{\partial\mathbf{x}}=\begin{bmatrix}\frac{\partial(\mathbf{Ax})_1}{\partial x_1} & \frac{\partial(\mathbf{Ax})_2}{\partial x_1} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_1}\\ \frac{\partial(\mathbf{Ax})_1}{\partial x_2} & \frac{\partial(\mathbf{Ax})_2}{\partial x_2} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_2}\\ \vdots & \vdots & \ddots & \vdots\\ \frac{\partial(\mathbf{Ax})_1}{\partial x_n} & \frac{\partial(\mathbf{Ax})_2}{\partial x_n} & \cdots & \frac{\partial(\mathbf{Ax})_m}{\partial x_n}\end{bmatrix}=\begin{bmatrix}a_{11} & a_{21} & \cdots & a_{m1}\\ a_{12} & a_{22} & \cdots & a_{m2}\\ \vdots & \vdots & \ddots & \vdots\\ a_{1n} & a_{2n} & \cdots & a_{mn}\end{bmatrix}=\mathbf{A}^{\mathrm{T}}$

考虑$\frac{\partial\mathbf{Au}}{\partial\mathbf{x}}$,$\mathbf{A}$与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：

- $\frac{\partial\mathbf{Au}}{\partial\mathbf{x}}=\frac{\partial\mathbf{u}}{\partial\mathbf{x}}\mathbf{A}^{\mathrm{T}}$

- $dL=\frac{\partial\mathbf{L}^{\mathbf{T}}}{\partial\mathbf{w}}d\mathbf{w}$

再考虑如下问题：

- $\frac{\partial\mathbf{x}^{\mathrm{T}}\mathbf{Ay}}{\partial\mathbf{x}},\mathbf{x}\in\mathbb{R}^{m\times 1},\mathbf{y}\in\mathbb{R}^{n\times 1}$ 其中A与x无关，我们知道 $\frac{\partial\mathbf{x}^{\mathrm{T}}\mathbf{Ay}}{\partial\mathbf{x}}\in\mathbb{R}^{m\times 1}$ 因此
- $\frac{\partial\mathbf{x}^{\mathrm{T}}\mathbf{Ay}}{\partial\mathbf{x}}=f(A,y)+g(\frac{\partial\mathbf{y}}{\partial\mathbf{x}},x^TA)$ $\mathbf{A}\in\mathbf{R}^{m\times n},\mathbf{y}\in\mathbf{R}^{n\times 1},\mathbf{x}^{\mathbf{T}}\mathbf{A}\in\mathbf{R}^{1\times n},\frac{\partial\mathbf{y}}{\partial\mathbf{x}}\in\mathbf{R}^{m\times n}$ 因

此，为了满足$\frac{\partial\mathbf{x}^{\mathrm{T}}\mathbf{Ay}}{\partial\mathbf{x}}\in\mathbb{R}^{m\times 1}$，我们可以知道函数f,g的形式。最终有：

- $\frac{\partial(\mathbf{x}^{\mathrm{T}}\mathbf{A})\mathbf{y}}{\partial\mathbf{x}}=\frac{\partial\mathbf{y}}{\partial\mathbf{x}}\mathbf{A}^{\mathrm{T}}\mathbf{x}+\mathbf{Ay}$ 当$\mathbf{x}=\mathbf{y}$时，
- $\frac{\partial(\mathbf{x}^{\mathrm{T}}\mathbf{A})\mathbf{y}}{\partial\mathbf{x}}=(\mathbf{A}^{\mathrm{T}}+\mathbf{A})\mathbf{x}$

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 error(Mi)/(1-error(Mi))，其中error(Mi)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-error(Mi))/error(Mi)) 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MINI）,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. 通过特征组合后再来选择特征：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集特征的主要来源，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)

- $MIC: I(X,Y)=\sum_{y\in Y}\sum_{x\in X}p(x,y)\log(\frac{p(x,y)}{p(x)p(y)})$

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:$(X_k,Y_k),k=1,2,...,n$
定义距离矩阵:

- $a_{j,k}=||\mathbf{X_j}-\mathbf{X_k}||,\quad j,k=1,2,...,n$
- $b_{j,k}=||\mathbf{Y_j}-\mathbf{Y_k}||,\quad j,k=1,2,...,n$ $||*||$是欧氏距离。取所有的双中心距离。
- $\mathbf{A_{j,k}}:=a_{j,k}-\hat{a}_{j.}-\hat{a}_{.k}+\hat{a}_{..}$
- $\mathbf{B_{j,k}}:=b_{j,k}-\hat{b}_{j.}-\hat{b}_{.k}+\hat{b}_{..}$

$\hat{a}_{j.}$是j-th row的平均，$\hat{a}_{.k}$是k-th column的平均，$\hat{a}_{..}$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

特征
数据预处理
标准化
scale（零均值单位方差）
StandardScaler
区间缩放
MinMaxScaler(最小最大值标准化)
MaxAbsScaler（绝对值最大标准化）
对稀疏数据进行标准化
对离群点进行标准化
正则化
normalize
Normalizer
范数
补充
二值化
特征二值化
Binarizer
补充
对类别特征进行编码
OneHotEncoder
n_values
categorical_features
dtype
sparse
handle_unknown
缺失值计算
Imputer
missing_values
strategy
axis
verbose
copy
生成多项式特征
PolynomialFeatures
degree
interaction_only
include_bias
属性
方法
自定义转换
特征提取
从字典类型加载特征
DictVectorizer
特征散列
特征散列
FeatureHasher
实现细节
文本特征提取
话语表示
稀疏
常用 Vectorizer 使用
CountVectorizer
Tf-idf 项加权
TfidfTransformer
解码文本文件
chardet
应用和实例
词语表示的限制
用哈希技巧矢量化大文本语料库
使用HashingVectorizer执行外核缩放
自定义矢量化器类
图像特征提取
补丁提取
图像的连接图
特征选择
Filter
方差选择法
相关系数法
卡方检验
互信息法
Wrapper
递归特征消除法
Embedded
基于惩罚项的特征选择法
基于树模型的特征选择法
降维
主成分分析法（PCA）
线性判别分析法（LDA）

模型的超参数调节

Model	Parameters to optimize	Good range of values
Linear Regression	• fit_intercept • normalize	• True / False • True / False
Ridge	• alpha • Fit_intercept • Normalize	• 0.01, 0.1, 1.0, 10, 100 • True/False • True/False
k-neighbors	• N_neighbors • p	• 2, 4, 8, 16 …. • 2, 3
SVM	• C • Gamma • class_weight	• 0.001, 0.01…..10...100...1000 • 'Auto', RS* • 'Balanced' , None
Logistic Regression	• Penalty • C	• L1 or l2 • 0.001, 0.01…..10...100
Naive Bayes (all variations)	NONE	NONE
Lasso	• Alpha • Normalize	• 0.1, 1.0, 10 • True/False
Random Forest	• N_estimators • Max_depth • Min_samples_split • Min_samples_leaf • Max features	• 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	• Eta • Gamma • Max_depth • Min_child_weight • Subsample • Colsample_bytree • Lambda • alpha	• 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

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/
```

深度学习

这一章主要讨论深度学习模型。主要包含CNN，RNN，DNN(MLP)。
CNN侧重于空间，主要用在图像分类模型，比如LeNet-5,AlexNet,VGG,GoogLeNet,Residual Net,Dense Net,Two path Net。
RNN侧重于时间序列，主要用于语音识别，自然语言处理，包含的网络有传统的RNN，LSTM以及GRU。
DNN(MLP)主要指有多个隐藏层的神经网络，也称为多层感知机(Multi-layer perceptron,MLP)。
在介绍完CNN与RNN之后，就是介绍这两种网络的实际用途了，比如RNN在语音识别方面的应用。GAN是一大块，但是不知道能用在什么地方，有空也必须添加进去。

CNN

CNN主要是处理空间数据，比如图像数据。CNN主要包含LetNet-5,AlexNet,GoogLeNet,VGG,Residual Net,Dense Net。
CNN核心：局部感受野，权值共享，时间或空间亚采样这三种思想来保证某种程度的平移，尺度，形变不变性。
介绍每种模型的时候，要说明这种模型的优点，最好能阐述为啥会有这种优点。

CNN基础

如何理解CNN中的卷积？

CNN相对于全连接网络的好处

减少模型参数数量，减少计算时间。

CNN的卷积核与特征提取

CNN中卷积核数目以及参数数据的计算。根据的是相邻两层feature maps个数，以及每个map的大小来计算出所用卷积核的数目。

CNN的卷积核大小为啥是奇数?

不能保持对称性，比如原来图像长度宽是MxM，卷积核大小是nxn，则卷积作用后图像大小是(M-n+1)x(M-n+1)，如果n是偶数，则n-1是奇数，因此处理后的图像不再对称了。
在这里，我们将要计算模型中，两层之间的参数数目。

如何计算CNN两层之间参数数目

C1层是卷积层，单通道下用了6个卷积核，这样就得到了6个feature map，其中每个卷积核的大小为5*5，用每个卷积核与原始的输入图像进行卷积，这样feature map的大小为(32-5+1)×(32-

5+1)= 28×28，所需要的参数的个数为(5×5+1)×6= 156(其中5×5为卷积模板参数，1为偏置参数)，连接数为(5×5+1)×28×28×6=122304(其中28×28为卷积后图像的大小)。

S2层为 pooling 层，也可以说是池化或者特征映射的过程，拥有6个 feature map，每个feature map的大小为14*14，每个feature map的隐单元与上一层C1相对应的feature map的 2×2 单元相连接，这里没有重叠。计算过程是：2×2 单元里的值相加然后再乘以训练参数w，再加上一个偏置参数b(每一个feature map共享相同w和b)，然后取sigmoid 值，作为对应的该单元的值。所以S2层中每 feature map 的长宽都是上一层C1的一半。S2层需要2×6=12个参数，连接数为(4+1)×14×14×6 = 5880。注：这里池化的过程与ufldl教程中略有不同。下面为卷积操作与池化的示意图：

计算CNN相邻两层之间参数数目，假设上层有大小是NxMxM，其中n是通道数目，MxM是图像大小。如果连接下一层的卷积核大小是nxn，我们可以对上一层的a个通道做卷积，则每次操作需要a个卷积核，再加上一个偏置，则需要axnxn+1个参数，假设下一层有K个通道，则两层之间所有的参数数目是K(axnxn+1)。这里a可以取不同值，也可以是不同值的组合，比如在LeNet-5中，S2到C3之间就存在不同通道的组合，C3的前6个特征图以S2中3个相邻的特征图子集为输入。接下来6个特征图以S2中4个相邻特征图子集为输入。然后的3个以不相邻的4个特征图子集为输入。最后一个将S2中所有特征图为输入。这样C3层有6(3*25+1)+6*(4*25+1)+3_(4_25+1)+(25*6+1)=1516个可训练参数和10*10*1516=151600个连接。

CNN的pooling的作用

降维与信息提取。

下采样与上采样的误差怎么传递与反向？

卷积神经网络(CNN)反向传播算法
CNN中一些特殊环节的反向传播

CNN网络中另外一个不可导的环节就是Pooling池化操作，因为Pooling操作使得feature map的尺寸变化，假如做2×22×22 \times 2的池化，假设那么第l+1l+1l+1层的feature map有16个梯度，那么第lll层就会有64个梯度，这使得梯度无法对位的进行传播下去。其实解决这个问题的思想也很简单，就是把1个像素的梯度传递给4个像素，但是需要保证传递的**loss**（或者梯度）总和不变。根据这条原则，mean pooling和max pooling的反向传播也是不同的。

1、mean pooling

mean pooling的前向传播就是把一个patch中的值求取平均来做pooling，那么反向传播的过程也就是把某个元素的梯度等分为n份分配给前一层，这样就保证池化前后的梯度（残差）之和保持不变，还是比较理解的，
mean pooling比较容易让人理解错的地方就是会简单的认为直接把梯度复制N遍之后直接反向传播回去，但是这样会造成loss之和变为原来的N倍，网络是会产生梯度爆炸的。

2、max pooling

max pooling也要满足梯度之和不变的原则，max pooling的前向传播是把patch中最大的值传递给后一层，而其他像素的值直接被舍弃掉。那么反向传播也就是把梯度直接传给前一层某一个像素，而其他像素不接受梯度，也就是为0。所以max pooling操作和mean pooling操作不同点在于需要记录下池化操作时到底哪个像素的值是最大，也就是max id，这个可以看caffe源码的pooling_layer.cpp，下面是caffe框架max pooling部分的源码

```
// If max pooling, we will initialize the vector index part.
if (this->layer_param_.pooling_param().pool() == PoolingParameter_PoolMethod_MAX && top.size(
) == 1)
{
    max_idx_.Reshape(bottom[0]->num(), channels_, pooled_height_,pooled_width_);
  }
```

源码中有一个max_idx_的变量，这个变量就是记录最大值所在位置的，因为在反向传播中要用到，那么假设前向传播和反向传播的过程就如下图所示

ImageNet top5 error record history

网络层数随历史的变化。

AlexNet

AlexNet的优点，

1. 使用ReLU作为激活函数，消除了sigmoid,tanh造成的梯度消失问题。同时也减小了激活函数的计算量。
2. 百万数据和多GPU训练减小了计算时间。
3. 局部相应归一化(LRN,Local Response Normalization)
4. 重叠池化，也就是pooling的尺寸小于步长。

5. 通过数据增强(裁剪，平移，尺度变换，水平翻转，加随机光照)与Dropout降低过拟合

VGG

相对于以前的网络而言，VGG与GoogLeNet的网络变得更深了。VGGNet在整个网络中使用3x3的小感受野，以步长1进行逐像素卷积，因此两个3x3卷积相当于一个5x5,三个3x3卷积核相当于一个7x7的卷积核，这样大大减小了模型的参数个数。使用了小尺寸的卷积核，增加了网络深度并不会带来明显的参数膨胀，却能在更深的网络中获得更高的精度。

GoogLeNet

特点：

1. 不适用全连接层，而是用平均池化代替全连接层，减小模型的参数数量,这种想法来自于NIN。
2. 采用一种高效的机器视觉深度神经网络结构，称为"Inception",这种模块化结构方便增加与修改，现在inception版本到了V6/V7。GoogLeNet中采用了9个inception模块。
3. 为了避免梯度小时，网络额外增加了2个辅助的softmax用于向前传导梯度。

GoogLeNet中的Inception

(a) Inception module, naïve version (b) Inception module with dimension reductions

GoogLeNet的参数 :可以看出里面用7x7的平均池代替了全连接。

后续改进的版本：

1. Inception-v2在之前的版本中主要加入了batch Normalization；另外也借鉴VGGNet的思想，用两个3x3的卷积代替5x5的卷积，不仅降低了训练参数，也提升了速度。
2. Inception-3在v2的基础上进一步分解大的卷积，比如把nxn的卷积拆分成两个一维卷积：1xn,nx1。

3. Inception-V4借鉴了ResNet可以构建更深层网络的思想，设计了一个更深更优化的模型。

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

softmax2
SoftmaxActivation
FC
AveragePool 7x7+1(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
AveragePool 5x5+3(V)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
DepthConcat
Conv 1x1+1(S)
Conv 3x3+1(S)
Conv 5x5+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
Conv 1x1+1(S)
MaxPool 3x3+1(S)
MaxPool 3x3+2(S)
LocalRespNorm
Conv 3x3+1(S)
Conv 1x1+1(V)
LocalRespNorm
MaxPool 3x3+2(S)
Conv 7x7+2(S)
input
softmax1
SoftmaxActivation
FC
FC
Conv 1x1+1(S)
softmax0
SoftmaxActivation
FC
FC
Conv 1x1+1(S)

Residual Net

详见第九章项目工作，Residual NN.

Dense Net

DenseNet的示意图，主要表现实任意两层间可以有short connection。
在ResNets中，只在相邻两层有skip connection.在激活函数是恒等函数的前提下，有：

$$x_l = H_l(x_{l-1}) + x_{l-1}$$

对于Dense Net,l层可以与0，1，...,l-1中任意多层相连。因此有：

$$x_l = H_l(x_0, x_1, ..., x_{l-1})$$

DenseNet的一个优点是网络更窄，参数更少，大一部分原因是得益于这种dense block的设计，后面有提到在dense block中每个卷积层的输出feature map数量都很小(小于100)，而不像其他网络一样动不动就是几百上千的宽度。同样这种连接方式使得特征和梯度的传递更加有效，网络也更容易训练。前面提到过梯度消失问题是网络深度越深的时候越容易出现，原因就是输入信息和梯度信息在很多层之间传递导致的，而现在这种dense connection相当于每一层都直接连接input和loss,因此就可以减轻梯度消失现象，这样更深网络不是问题。
Dense net主要通过建立不同层之间的直接连续，充分利用了feature，进一步减轻了梯度消失问题，加深了网络深度，而且训练效果非常好。另外利用bottlenech layer,translation layer以及较小的growth rate使得网络变窄，参数减小，有效抑制了过拟合，同时减小了计算量。

Dense Blocks之间通过pooling与convolution来改变图像尺寸。

Figure 2. A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

http://blog.csdn.net/u014380165

问题集

1. 为什么1x1的卷积核可以用来降维？
   1、降维（dimension reductionality）。比如，一张500 *500*且厚度*depth*为*100* 的图片在*20*个*filter*上做*1*1的卷积，那么结果的大小为500*500*20。2、加入非线性。卷积层之后经过激励层，1*1的卷积在前一层的学习表示上添加了非线性激励（non-linear activation），提升网络的表达能力。

参考

RNN

循环神经网络主要用于序列模型，为了捕获CNN捕获不了的序列的长程关联。LSTM的提出是为了弥补传统RNN梯度消失的问题，GRU是为了弥补LSTM的 ..

RNN

循环神经网络

循环神经网络中一些重要的设计模式包含如下几种：
(1)每个时间步都有输出，并且隐藏单元之间有循环连接的循环网络，如下图所示。
RNN的输入到隐藏的连接由权值矩阵U参数化，隐藏到隐藏的循环由权重矩阵W参数化，隐藏到输出层的连接由权重矩阵V参数化。$L^{(t)}$是损失函数。

前向传播公式：假设激活函数是tanh，我们有如下的更新方程：

- $\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}x^{(t)}$
- $\mathbf{h}^{(t)} = \tanh(\mathbf{a})$
- $\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$
- $\hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)})$

其中$\mathbf{b}, \mathbf{c}$是偏置向量,$\mathbf{U}, \mathbf{V}, \mathbf{W}$是权重矩阵。
这一循环网络将一个输入序列映射到相同长度的输出序列。与$\mathbf{x}$序列配对的$\mathbf{y}$的总的损失就是所有时间步的损失之和。例如，$L^{(t)}$为给定的$x^{(1)}, x^{(2)}, ..., x^{(t)}$后$y^{(t)}$的负对数似然，则：

$$\blacksquare L((x^{(1)}, x^{(2)}, ..., x^{(\tau)}), (y^{(1)}, y^{(2)}, ..., y^{(\tau)})) = \sum_t L^{(t)}$$

$$\blacksquare = -\sum_t \log p_{model}(y^{(t)}|(x^{(1)}, x^{(2)}, ..., x^{(t)})$$

其中$p_{model}(y^{(t)}|(x^{(1)}, x^{(2)}, ..., x^{(t)})$需要读取模型输出向量$\hat{\mathbf{y}}^{(t)}$对应于$\mathbf{y}^{(t)}$的项。
应用于展开图且代价为$O(\tau)$的反向传播算法是通过时间反向传播(back-propagation through time, BPTT)。
(2)每个时间步都产生一个输出，只有当前时刻的输出到下个时刻的隐藏单元之间有循环连接的循环网络。如下图所示：
该图中，RNN被训练将特定输出值放入o中，并且o是被允许传播到未来的唯一信息。此处没有从h前向传播的直接连接。之前的h仅通过产生的预测间接地连接到当前。o通常缺乏过去的重要信息，除非它非常高维并且内容丰富。这使得该图中的RNN不那么强大，但是它更容易训练，因为每个时间步可以与其他时间步分离训练，允许训练期间更多的并行化。

Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $\boldsymbol{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

LSTM

由于Vanilla RNN具有梯度消失问题，对长关系的依赖的建模能力不够强大，也就是很长时刻以前的输入，对现在的网络影响非常小，后向传播那些梯度也很难影响很早以前的输入，即会出现题都消失的问题。而LSTM通过构建一些门，让网络能记住那些非常重要的信息，而这个核心的结构就是cell state。比如遗忘门，来选择性清空过去的记忆和更新较新的信息。
两种常见的LSTM结构。
第一种是带遗忘门的Traditional LSTM。公式如下：

- $f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$
- $i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$
- $o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$
- $c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$
- $h_t = o_t * \sigma_h(c_t)$

前三行是三个门，分别是遗忘门f,输入门i,输出门o,输入都是x_t, h_{t-1},只是参数不同，然后要经过一个激活函数把值缩放到[0,1]之间，第四行c_t是cell state,由上一时刻的c_{t-1}和输入得到,两者相互独立。如果遗忘门f_t取0的话，那么上一时刻的状态就会全部被清空，然后只关注此时刻的输入。输入门i_t决定是否接收此时刻的输入，最后输出门o_t决定是否输出cell state。

LSTM

由于Vanilla RNN具有梯度消失问题，对长关系的依赖的建模能力不够强大，也就是很长时刻以前的输入，对现在的网络影响非常小，后向传播那些梯度也很难影响很早以前的输入，即会出现题都消失的问题。而LSTM通过构建一些门，让网络能记住那些非常重要的信息，而这个核心的结构就是cell state。比如遗忘门，来选择性清空过去的记忆和更新较新的信息。
两种常见的LSTM结构。
第一种是带遗忘门的Traditional LSTM。公式如下：

- $f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$
- $i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$
- $o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$
- $c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$
- $h_t = o_t * \sigma_h(c_t)$

前三行是三个门，分别是遗忘门f,输入门i,输出门o,输入都是x_t, h_{t-1},只是参数不同，然后要经过一个激活函数把值缩放到[0,1]之间，第四行c_t是cell state,由上一时刻的c_{t-1}和输入得到,两者相互独立。如果遗忘门f_t取0的话，那么上一时刻的状态就会全部被清空，然后只关注此时刻的输入。输入门i_t决定是否接收此时刻的输入，最后输出门o_t决定是否输出cell state。

- $\hat{c} = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$

因此有$c_t = f_t * c_{t-1} + i_t * \hat{c}$. 第二种是带遗忘门的Peephole LSTM，公式如下：

- $f_t = \sigma_g(W_f x_t + U_f c_{t-1} + b_f)$
- $i_t = \sigma_g(W_i x_t + U_i c_{t-1} + b_i)$
- $o_t = \sigma_g(W_o x_t + U_o c_{t-1} + b_o)$
- $c_t = f_t * c_{t-1} + i_t * \sigma_c(W_c x_t + b_c)$
- $h_t = o_t * \sigma_h(c_t)$

和上面的公式比较，发现只是把h_{t-1}换成了c_{t-1}，即三个门的输入都改成了[x_t, c_{t-1}],因为是从cell state里取得信息，所以叫窥视管(peephole)。
把这两种结构结合起来，可以用如下图描述：

图中连着门的那些虚线都市peephole。三个输入都是$[x_t, h_{t-1}, c_{t-1}]$

GRU

第九章 Kaggle

这章主要通过Kaggle实战来提升自己的运用的能力，最主要的还是特征提取的能力。

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

关联系数，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_traint = 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_traint, sex_train, train.Pclass, train.Sib
Sp, 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, tes
t.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颗树时效果最好，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
```

模型融合

对多个模型的结果取平均(投票)，作为最终的结果。比如拿SVM，LR，RF，GBDT的平均结果作为最终结果。

用几个模型筛选出较为重要的特征

```
def get_top_n_features(titanic_train_data_X, titanic_train_data_Y, top_n_features):
        # 随机森林
        rf_est = RandomForestClassifier(random_state=42)
        rf_param_grid = {'n_estimators': [500], 'min_samples_split': [2, 3], 'max_depth': [20
]}
```

```
        rf_grid = model_selection.GridSearchCV(rf_est, rf_param_grid, n_jobs=25, cv=10, verbo
se=1)
        rf_grid.fit(titanic_train_data_X,titanic_train_data_Y)
        #将feature按Importance排序
        feature_imp_sorted_rf = pd.DataFrame({'feature': list(titanic_train_data_X), 'importa
nce': rf_grid.best_estimator_.feature_importances_}).sort_values('importance', ascending=False
)
        features_top_n_rf = feature_imp_sorted_rf.head(top_n_features)['feature']
        print('Sample 25 Features from RF Classifier')
        print(str(features_top_n_rf[:25]))

        # AdaBoost
        ada_est = ensemble.AdaBoostClassifier(random_state=42)
        ada_param_grid = {'n_estimators': [500], 'learning_rate': [0.5, 0.6]}
        ada_grid = model_selection.GridSearchCV(ada_est, ada_param_grid, n_jobs=25, cv=10, ve
rbose=1)
        ada_grid.fit(titanic_train_data_X, titanic_train_data_Y)
        #排序
        feature_imp_sorted_ada = pd.DataFrame({'feature': list(titanic_train_data_X),'importa
nce': ada_grid.best_estimator_.feature_importances_}).sort_values( 'importance', ascending=Fa
lse)
        features_top_n_ada = feature_imp_sorted_ada.head(top_n_features)['feature']

        # ExtraTree
        et_est = ensemble.ExtraTreesClassifier(random_state=42)
        et_param_grid = {'n_estimators': [500], 'min_samples_split': [3, 4], 'max_depth': [15
]}
        et_grid = model_selection.GridSearchCV(et_est, et_param_grid, n_jobs=25, cv=10, verbo
se=1)
        et_grid.fit(titanic_train_data_X, titanic_train_data_Y)
        #排序
        feature_imp_sorted_et = pd.DataFrame({'feature': list(titanic_train_data_X), 'importa
nce': et_grid.best_estimator_.feature_importances_}).sort_values('importance', ascending=False
)
        features_top_n_et = feature_imp_sorted_et.head(top_n_features)['feature']
        print('Sample 25 Features from ET Classifier:')
        print(str(features_top_n_et[:25]))

        # 将三个模型挑选出来的前features_top_n_et合并
        features_top_n = pd.concat([features_top_n_rf, features_top_n_ada, features_top_n_et]
, ignore_index=True).drop_duplicates()

        return features_top_n
```

深度学习在晶圆检测中的应用研究

要从这个项目中深挖出整个深度学习，也就是从这个项目中涉及到CNN，RNN,DNN.以及训练技巧以及ML。图像处理与信号处理。也就是要体现出深度与广度。三段经历代表深度学习，编程，最优化(机器学习)三大领域，要会发现亮点，也就是自己的特色。注意深度，附带是广度。

项目背景与目的

通过深度学习来提升光刻模型的性能。随着芯片制造中芯片节点(晶体管栅极的最小线宽（栅宽）)变小，光成像过程中光的波动性越来越显著，几何光学要被波动光学所修正。光刻成像的模型也越来越复杂，但是，即使模型越来越复杂，我们通过光刻的成像模型也与显示的成像结果存在一些差距，尤其在边角处。这些由成像模型导致的错误，很大部分可以由有经验的工程师来辨别出来，这些错误区别于真实的印刷错误。我们的目的是想通过深度学习来减小我们的模型错误而保留真实的印刷错误。可以认为是把模型导致的错误当成噪声，我们需要抑制这些噪声，而把真正的印刷错误当成真实的信号来对待，我们需要保留这些信号而抑制噪声。
一个问题是:不加特殊的处理，你在抑制噪声的时候，也同时抑制了真正的信号。那你的网络要怎么处理才能保留住信号而抑制噪声？

问题的延伸

我们真正想要找的是印刷错误(真正的错误)，而模型导致的错误是虚假的错误。因此，我们可以当成一个二分类问题去求解。但是不能漏掉一个真实的错误。

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，因此就涉及到一些优化算法，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 Rn 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，我们任然需要建立物理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，亚像素的偏离)，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，我们都在用用记忆卷积，反卷积的深度神经网络，都市没有发现很好的效果。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，并不比没有处理额图像好，这就尴尬了，因此，我们一直加深网络。

工具

Quardo 4000; Tensorflow

图像预处理

主要困难还是在于图像的收集，我们可以通过降低判定real-false的阈值来增加模型错误的图像数量，我们也需要人工来判定哪些错误是真实的印刷错误，哪些是由模型导致的错误。而人工标注是一件很费时间的事情。
我们输入的是我们模型产生的图像，而我们的target是真是拍摄到的芯片图像。这些都是数字图像，我们必须确保输入与输出的图像是同一个地方的，也就是图像必须真正的对齐，不能差任何一点，半个pixel也不能(查一点就会导致两图相减在边缘地方出现巨变)，但是最后发现我们的图像差了半个pixel，我们必须通过信号处理的方法来对齐我们的输入与输出图像。这些是数据预处理过程中遇到的真实问题。

数据集

图像大小:48*48,
数据集:一共31400张图片，训练集90%,一共28260张图，测试集10%,一共3140张图
batch size大小5.
一个epcoh(训练完28260张图)的时间是600s,即10分钟左右。

```
#batch_size =5
#learn_rate = 0.0001
net = tflearn.input_data(shape=[None, 48, 48, 1])
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 3, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 1, 64)
net = tflearn.conv_2d(net, 1, 1)
Training Step: 1000  | total loss: 6.56844 | time: 118.655s
| Adam | epoch: 001 | loss: 6.56844 - acc: 0.7033 | val_loss: 8.57394 - val_acc: 0.6723 -- it
er: 05000/28260
--
Training Step: 2000  | total loss: 6.26063 | time: 234.816s
| Adam | epoch: 001 | loss: 6.26063 - acc: 0.6907 | val_loss: 8.11802 - val_acc: 0.6762 -- it
er: 10000/28260
--
Training Step: 3000  | total loss: 5.61156 | time: 351.264s
| Adam | epoch: 001 | loss: 5.61156 - acc: 0.7085 | val_loss: 7.88475 - val_acc: 0.6794 -- it
er: 15000/28260
--
Training Step: 4000  | total loss: 6.61628 | time: 469.786s
| Adam | epoch: 001 | loss: 6.61628 - acc: 0.7000 | val_loss: 9.18316 - val_acc: 0.6804 -- it
er: 20000/28260
--
Training Step: 5000  | total loss: 5.81782 | time: 586.015s
| Adam | epoch: 001 | loss: 5.81782 - acc: 0.6987 | val_loss: 7.58499 - val_acc: 0.6852 -- it
er: 25000/28260
--
```

```
Training Step: 6000  | total loss: 5.99239 | time: 50.356ss
| Adam | epoch: 002 | loss: 5.99239 - acc: 0.7008 | val_loss: 6.78956 - val_acc: 0.6852 -- it
er: 01740/28260
```

网络选择

从一开始的VGG到

```
#batch_size =5
#learn_rate = 0.0001
net = tflearn.input_data(shape=[None, 48, 48, 1])
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.residual_block(net, 1, 64)
net = tflearn.conv_2d(net, 64, 1)
net = tflearn.residual_block(net, 1, 64)
net = tflearn.residual_block(net, 2, 64)
net = tflearn.conv_2d(net, 1, 1)
Training Step: 1000  | total loss: 6.41052 | time: 88.615s
```

效果

通过我们的ResidualNN,我们能很大的降低我们模型得到的图像与实际拍摄到的图像之间的差别，也就是在Max2x2。(RMS怎么定义忘记了？)

Max 2x2 Residual	Original Ref	ResNet Ref
<20	0	14.3%
<25	0	55.3%
<30	0.9%	79.1%
<35	7.7%	89.0%
>35	92.3%	11%

原因是，我们输入的图像已经很接近输出的图像，因此，我们只需要训练残差就可以了。

图像与信号处理

Residual NN

Residual NN提出来的背景：

随着网络变深，训练误差与测试误差得提高了。这是违反我们的训练的初衷的，因为即使我们把26层后面的网络变成恒等映射，效果也不会变差。

Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

存在这种随着网络层数增加，会出现如下两个问题：

1.梯度消失或者爆炸，导致训练难以收敛。这个问题可以通过norimalized initialization 和 intermediate normalization layers解决。

2.随着深度增加，模型的训练误差与测试误差会迅速下滑，这不是overfit造成的。这种现象在CIFAR-10和ImageNet中都有提及。

Residual-NN的实现

参考He Kaiming的这篇文章

Residual Block的设计如下：

Figure 2. Residual learning: a building block.

对于每一个Residual Units：

- $y_l = h(x_l) + F(x_l, W_l)$
- $x_{l+1} = f(y_l)$

其中函数h一般取恒等映射$h(x_l) = x_l$，实验发现f取恒等映射能实现最快的误差下降与最低的训练误差；f是激活函数，取ReLU。F的残差函数。

一个推荐的网络结构:

如果激活函数f取恒等映射$y_l = f(y_l)$, 则有:

$$x_{l+1} = x_l + F(x_l, W_l)$$

$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i, W_i)$$

因此对于误差反向传播:

$$\frac{\partial \epsilon}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial \epsilon}{\partial x_L}(1 + \frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F(x_i, W_i))$$

因为$\frac{\partial}{\partial x_l}\sum_{i=l}^{L-1} F(x_i, W_i)$不会常等于-1,因此梯度不会消失, 即使当权重非常小的时候。

因此信号对于正向与反向传播, 可以直接的从一个单元传到另外一个单元。条件就是skip connection 函数h,激活函数f是恒等映射。函数h是恒等映射时保证梯度不消失或者爆炸的关键。他们使用的mini-batch是128, 用了2GPUs,权重衰减是0.0001, 动量是0.9, 权重初始化了。详细的可以看上面给的论文链接。权重每30个epoch变成原来的十分之一。

Residual NN模型效果:

Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Resi-NN能解决以往随着模型加深训练与测试误差变大的现象。与其它网络的结果比较。

CIFAR-10	error (%)
NIN [13]	8.81
DSN [14]	8.22
FitNet [15]	8.39
Highway [7]	7.72
ELU [12]	6.55
FitResNet, LSUV [16]	5.84
ResNet-110 [1] (1.7M)	6.61
ResNet-1202 [1] (19.4M)	7.93
ResNet-164 [ours] (1.7M)	5.46
ResNet-1001 [ours] (10.2M)	4.92 (4.89±0.14)
ResNet-1001 [ours] (10.2M)†	**4.62** (4.69±0.20)

CIFAR-100	error (%)
NIN [13]	35.68
DSN [14]	34.57
FitNet [15]	35.04
Highway [7]	32.39
ELU [12]	24.28
FitNet, LSUV [16]	27.66
ResNet-164 [1] (1.7M)	25.16
ResNet-1001 [1] (10.2M)	27.82
ResNet-164 [ours] (1.7M)	24.33
ResNet-1001 [ours] (10.2M)	**22.71** (22.68±0.22)

模型的改进（Simons）

是否可以通过逐渐加深网络来实现浅层的网络而具有强大的功能。具体步骤就是：

1. 一个浅层的网络进行end-end的训练，训练到一定程度的时候，进行第二步
2. 沿用第一步的权重，通过以Residual Block来微调网络，减小误差，优化方法是SGD，但是我们要保证，每增加一层网络，总的模型的效果会更好。
3. 循环第二步，直到结果符合我们期望的为止(误差到noise层)
   理论上，这类似于机器学习中的Adaboost，是否可以分析它的误差会指数衰减？希望能像ML算法一样，能有数学理论来分析深度学习。通过理论指导，像搭积木一样来构建深度神经网络。

三大神经网络

CNN

怎么实现最简单的CNN?

RNN

LSTM

DNN

自编码

强化学习

Fast项目

要从这个项目中体现出你对语言的驾驭能力，也就是编程能力。各大语言的区别，主要是C++，Java与python。OOP，还有就是模式设计这些。

指针与引用

C++的OOP

C++的继承与多态

C++与Java的最大区别

模式设计

Windows的消息机制

多线程与同步

进程间通讯

Socket

泛型编程

STL--标准模板库

智能指针

数据结构

排序算法

Miura项目

要从这个项目中你对这最优化问题的理解与实践。也就是能对不同的最优化算法能如数家珍，并能知道其适用场景。从最优化扯到ML，DL。还是多与机器学习靠拢，这就形成自己的三个方面。

最优化

一阶方法--梯度下降法

随机梯度下降法

动量方法

Adagrad

Adam

特殊的一阶方法--共轭梯度法

二阶方法

牛顿法

quasi-Netwon法

BFGS

DFP

线性搜索

Trust-region方法

L-M算法

Dog-Leg算法

约束最优化问题

拉格朗日乘子理论

罚函数法

二次规划

对偶性与方法

问题求解系统

分类

回归

支持向量机

决策树

集成学习与随机森林

维数约化

特征工程

Updates Log

ToDo

- Tensorflow - char embedding layer
- Tensorflow - Pointer Layer
- 机器学习 - 专题-XGBoost
- 深度学习 - GAN 与 RL 基本模型
- CRF 条件随机场 / 图模型
- Linux Shell 常用命令
  - wc/sed/awk/grep/sort/uniq/paste/cat/head/tail
- DL - 专题 - 加速方法
- 机器学习基础相关
  - 深度学习与机器学习的异同及联系
- 机器学习实践相关
  - 数据预处理
  - 缺失值
  - 特征工程，特征选择的标志（为什么选择这些特征）
  - XGB 调参的方法、LightGBM、CatBoost
    - gbdt,xgboost,lgbm的区别
  - 模型融合的方法
  - 数据可视化分析 - 流形学习
- 算法专题-排列与组合
- 吉布斯采样
- 算法-矩阵乘法
  - 稀疏矩阵
- 显著性检测
- 机器学习的体系、学派
- 朴素贝叶斯与逻辑回归的区别
- 为什么 RNN 中采用 tanh 作为激活函数

  > https://www.zhihu.com/question/61265076

- 修订 FastText 中词向量生成的描述
- 自定义用于快速测试代码的数据结构及相关方法
- 并查集
- 什么是梯度消失
- HR面常见问题
- 超参数调优
- 机器学习实战-部分代码（K-Means等）

- 判断相似二叉树及优化
- `max(x, y)` 的期望
- 加速网络收敛的方法
- 回归树、基尼指数
- 判断两个链表是否相交（CSDN）
- 一阶 二阶泰勒展开
- 梯度下降往往没有指向

2018-9-27

- 笔试 - iHandy

2018-9-26

- 算法 - 专题 - 双指针

2018-9-25

- 算法 - 专题 - 双指针
- 笔试 - 作业帮

2018-9-24

- 算法 - 专题 - 双指针

2018-9-23

- 算法 - 专题 - 双指针

2018-9-21

- 算法 - 专题 - 双指针

2018-9-15

- 笔试 - 爱奇艺

- 笔试 - 搜狐畅游
- NLP - 专题-句向量

2018-9-8

- 算法 - 高级数据结构
  - 树状数组
- papers
  - 视觉问答

2018-9-7

- 算法 - 区间问题
- 算法 - 数学
  - 最大共线点的个数

2018-9-5

- 算法 - 排列组合专题整理

2018-9-4

- 算法 - 数据结构 - 设计
  - LRU 缓存淘汰算法
- 算法 - 排列组合
  - 下一个排列
  - 上一个排列
  - 全排列（无重复）
  - 全排列（有重复）
- 算法 - 面试真题 - n个[0,n)的数，求每个数的出现次数（不能开辟额外空间）

2018-9-3

- 深度学习基础 - 正则化 - Batch Normalization（修订）

2018-9-2

- 集成学习专题整理

2018-9-1

- 笔试面经 - 头条 4 面小结
- 算法 - 字符串 - 进制转换/长短地址转换
- 算法 - LeetCode - 连续子数组和

2018-8-31

- 数据结构 - 字符串 - 中缀表达式转后缀表达式（逆波兰式）
- 数据结构 - 二叉树 - 最大路径和
- 深度学习 - 基础 - 批量归一化（BN）
- 算法 - 动态规划 - 最大正方形

2018-8-30

- 算法 - 海量数据处理专题 - 判断一个数是否在 40 亿个数中出现
- 笔试面经 - 招行 - 推倒吧骨牌

2018-8-29

- DL-专题-RNN
  - RNN 的 3 种设计模式
- DL-专题-序列建模
  - Seq2Seq
    - Beam Search

2018-8-28

- 完善 K-Means 注释
- 算法-随机数
  - 用 `rand_m()` 生成 `rand_n()`
- DL-专题-RNN

2018-8-27

- code
  - K-Means
- 360 笔试
- 算法-数据结构

2018-8-26

- 深度学习基础
  - 梯度下降及其优化（修订）

2018-8-25

- 深度学习基础
  - 过拟合与欠拟合
  - 梯度下降（修订）

2018-8-24

- DL-专题-CNN
  - 卷积的内部实现
  - 卷积的反向传播
- NLP-专题-词向量修订
- 算法-专题-数据结构
  - 二叉树
    - 最近公共祖先
    - 打印节点路径
  - 链表
    - 链表快排
    - 两个链表的第一个公共节点

2018-8-23

- 算法-动态规划
  - 完全背包
  - 01背包、完全背包的一维 DP 优化、滚动数组优化

2018-8-22

- proj-阅读理解-完形填空（改进三）
- LeetCode
  - 暴力搜索
    - 岛屿的个数（DFS + BFS）
- 字节跳动 0812 笔试题回顾
  - 3 积分卡牌游戏（未测试）
  - 5 直播爱好者（未测试）
- 算法-随机数
  - 利用不等概率方法产生等概率的随机数
  - 如何采样，使 n-1 被采样 n 次？
- 算法-动态规划
  - 01 背包

2018-8-21

- proj-阅读理解-完形填空
- LeetCode
  - 数组
    - 删除排序数组中的重复项
    - 我的日程安排表 I
- 字节跳动 0812 笔试题回顾
  - 1 世界杯开幕式（DFS 搜索联通区域）
  - 2 文章病句标识（多级排序，区间合并）

2018-8-20

- DP-鹰蛋问题
- 基础-术语表
  - 指数衰减平均的说明
- LeetCode
  - 数据-双指针
    - 三数之和
    - 最接近的三数之和

2018-8-19

- 算法-洗牌与采样
  - 带权采样（查表法）
- DL-专题-优化算法

    - 带动量的 SGD 算法
    - 自适应学习率的优化方法
        - AdaGrad
        - RMSProp
        - AdaDelta
        - Adam

2018-8-18

- DL-专题-优化算法
    - 梯度下降法
    - SGD、小批量 SGD
    - 牛顿法与梯度下降
- 题解-LeetCode
    - 数组
        - 三数之和
        - 最接近的三数之和

2018-8-17

- 机器学习基础 - 方差与偏差
- 机器学习基础 - 判别模型与生成模型

2018-8-16

- 算法 - 洗牌与采样

深度学习的发展脉络

写这一节的目的是为了对深度学习的发展有一个认识。可以写成一部深度学习的发展史。里面要描述出当时存在的问题，以及为解决这些问题而发展出什么模型；为解决不同领域的问题如图像识别方向，语音识别方向，自然语言处理方向而发展出的不同模型。以及为解决特定领域的问题而发展特定的方法，比如从RNN到LSTM。最总会成为一个树形结构。
另外一个目的就是要时刻对最前沿的领域有一个了解，及时去拜读arxiv上最新的深度学习的论文。了解当前存在的主要问题是什么，然会看存在的可能的解决方案是什么？自己有什么想法等。看CVPR的前三名，就基本对计算机视觉的最新工作有个基本的了解了。
格式：论文是为了解决什么问题，解决的怎么样了，相对于其它方法的优势是什么，附上论文的地址。

第十章 面试

这章涉及到C++,传统算法, ML, DL方面的常见问题。

机器学习类面试问题集

算法理论方面

2.1 LR，SVM，KNN，GBDT，XGB推导，算法细节(LR为何是sigmod，理论推导出sigmod,KNN距离度量方式，XGBoost为什么要用二阶信息不用一阶，LR和SVM对比，GBDT和XGB和LightGBM对比)。2.2 CNN DNN RNN 细节以及相关问题(poll层，激活函数，梯度消失弥散问题，LSTM结构图，深度网络优势及缺点)。2.3 常见排序算法的复杂度和一些细节以及改进优化。2.4 树模型建模过程。2.5 特征选择方法。2.6 模型训练停止方法。2.7 正则化作用。2.8 模型效果评价指标。2.9 AUC理解和计算方法。2.10 Hadoop，Hive，Spark相关理论。2.11 L_BFGS，DFP推导。2.12 弱分类器组合成强分类器的理论证明。2.13 FM，FMM，Rank_SVM算法细节。2.14 map_reduce基本概念以及常见处理代码。2.15 过拟合的解决方法。2.16 各个损失函数之间区别。2.17 L1,L2正则化相关问题。

模型评估问题

降维的方法与原理

非监督学习问题

集成学习问题

前向神经网络问题

为啥Sigmoid和Tanh激活函数会导致梯度消失的现象？

因为在参数很大或者很小的时候，他们的导数会趋于0，因此造成了梯度消失。

ReLU系列激活函数相对于Sigmoid和Tanh激活函数的有点事什么？它们有什么局限性以及如何改进？

优点

(1)计算梯度简单，因为Sigmod和Tanh涉及到指数运算
(2)ReLU能有效解决后两者的题都消失问题，提供相对宽的激活边界
(3)ReLU的单侧抑制提供了网络的稀疏表达能力

缺点

ReLU的局限性在于其训练过程中会导致神经元死亡的问题，由于激活函数和是$mf(x) = \max(0, z)$导致梯度在经过该ReLU单元时，被置为0。且之后再也不能被任何数据激活，因为流经这个神经元的梯度永远为0，因此不对任何数据产生响应。因此会导致一定比例的神经元不可逆的死亡，进而参数梯度永远无法更新，整个训练过程失败。
为了解决这一问题，人们设计了Leaky ReLU(LReLU)其表达式是：

- $f(z) = z; z > 0$
- $f(z) = az; z \leq 0$

一般a是一个很小的正数，但是怎么选取a也是一个问题。为解决这个问题，人们发明了参数化的PReLU(Parametric ReLU),它与LReLU的主要区别是，斜率参数a作为网络中一个可学习的参数。而另一个LReLU的变种增加了"随机化"机制，具体的，在训练过程中，斜率a作为一个满足某种分布的随机采样；测试时再固定下来。Random ReLU(RReLU)在一定程度上能起到正则化的作用。

平方误差损失函数和交叉熵损失函数分别适合什么场景？

平方损失函数:回归问题，输出是连续值，并且最后一层不含Sigmoid或者Softmax激活函数的NN。因为会导致梯度很小而发生梯度消失的问题
交叉熵：分类问题，

写出多层感知机的平方误差和交叉熵损失函数

给定包含m个样本的集合$((x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)}))$，

代价函数

平方误差其整体代价函数是:

$$J(W,b)=[\frac{1}{m}\sum_{i=1}^{m}J(W,b;x^{(i)},y^{(i)})]+\frac{\lambda}{2}\sum_{l=1}^{N-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ij}^{(l)})^2$$

$$=[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}||y^{(i)}-L_{W,b}(x^{(i)})||^2]+\frac{\lambda}{2}\sum_{l=1}^{N-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ij}^{(l)})^2$$

二分类交叉熵误差其整体代价函数是:

$$J(W,b)=-[\frac{1}{m}\sum_{i=1}^{m}J(W,b;x^{(i)},y^{(i)})]+\frac{\lambda}{2}\sum_{l=1}^{N-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ij}^{(l)})^2$$

$$=-[\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\ln o^{(i)}+(1-y^{(i)})\ln(1-o^{(i)}))]+\frac{\lambda}{2}\sum_{l=1}^{N-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ij}^{(l)})^2$$

多分类交叉熵误差其整体代价函数是:

$$J(W,b)=-[\frac{1}{m}\sum_{i=1}^{m}J(W,b;x^{(i)},y^{(i)})]+\frac{\lambda}{2}\sum_{l=1}^{N-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ij}^{(l)})^2$$

$$=-[\frac{1}{m}\sum_{i=1}^{m}y_k^{(i)}\ln o_k^{(i)}]+\frac{\lambda}{2}\sum_{l=1}^{N-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ij}^{(l)})^2$$

其中$o_k^{(i)}$ 代表第i个样本的预测属于类别k的概率, $y_k^{(i)}$为实际的概率(如果第i个样本的真实类别为k, 则$y_k^{(i)}=1$, 否则为0)。

梯度计算公式

第(l)层的参数为$W^{(l)},b^{(l)}$, 每一层的线性变换为:

$$\mathbf{z}^{(l)}=\mathbf{W}^{(l)}\mathbf{x}^{(l)}+\mathbf{b}^{(l)}$$ 输出为:

$$\mathbf{a}^{(l)}=f(\mathbf{z}^{(l)})$$

其中f是非线性激活函数(比如Sigmoid,Tanh, ReLu等); $\mathbf{a}^{(l)}$直接作为下一层的输入.即:

$\mathbf{x}^{(l+1)}=\mathbf{a}^{(l)}$。我们通过批量梯度下降法来优化网络参数。

$$\mathbf{W_{ij}}^{(l)}=\mathbf{W_{ij}}^{(l)}-\alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(\mathbf{W},\mathbf{b}),$$

$$\mathbf{b_i}^{(l)}=\mathbf{b_i}^{(l)}-\alpha\frac{\partial}{\partial b_i^{(l)}}J(\mathbf{W},\mathbf{b}),$$

问题的核心是求$\frac{\partial}{\partial W_{ij}^{(l)}}J(\mathbf{W},\mathbf{b}),\frac{\partial}{\partial b_i^{(l)}}J(\mathbf{W},\mathbf{b})$
然后从最后一层, 反向迭代。

$$\frac{\partial}{\partial W_{ij}^{(l)}}J(\mathbf{W},\mathbf{b})=\frac{\partial J(\mathbf{W},\mathbf{b})}{\partial z_j^{(l+1)}}\frac{\partial z_j^{(l+1)}}{\partial \mathbf{W}_{ij}^{(l)}}$$

神经网络训练技巧

主要涉及的问题是“过拟合”：解决方法包括数据集增强，正则化，模型集成。其中dropout是模型集成方法中最高效与常用的技巧。此外，DNN训练中设计的手动调参，如学习率，权重衰减系数，Dropout比例等，这些参数的选择会显著影响模型最终的训练效果。Batch Normalization(BN)方法有效规避了这些复杂参数对网络训练产生的影响，在加速训练收敛的同时也提升了网络的泛化能力。

神经网络训练时是否可以将全部参数初始化为0？

因为所有参数都是相同的值，所有神经元都是对称的，因此没有办法打破这种对称，导致无论学习多久，所有参数依然是相同的。因此我们需要随机地初始化NN参数的值，来打破这种对称性。

为啥Dropout可以抑制过拟合？它的工作原理和实现？

还得先介绍Dropout的实际步骤，也就是训练的时候是多少个神经元？预测的时候是多少个？Dropout作用于小批量训练数据，由于其随机丢弃部分神经元的机制，相当于每次迭代都是在训练不同结构的神经网络，类比于Bagging方法，Dropout可被认为书一种实用的大规模深度神经网络的模型集成算法。这是由于传统意义上的Bagging涉及多个模型的同时训练和测试评估，当网络与参数规模庞大时，这种集成方式需要消耗大量的运算时间与空间。Dropout在小批量级别上的操作，提供了一种轻量级的Bagging集成近似，能够实现指数级数量神经网络的训练与评测。

Dropout在具体实现中，某个神经元以概率p被丢弃，因此N个神经元在Dropout下相当于是2^N个模型的集成。这2^N个模型可认为是原始网络的子网络，它们共享部分权值，并且具有相同的网络层数，而模型的整体参数数目不变，这就大大简化了运算。对于任意神经元，每次训练中都与一组随机挑选的不同的神经元集合共同进行优化，这个过程会减弱全体神经元之间的联合适应性，减小过拟合的风险，增强泛化能力。

批量归一化的基本动机与原理是什么？在卷积神经网络中如何使用？

神经网络训练过程的本质是学习数据分布，如果训练数据与测试数据的分布不同将大大降低网络的泛化能力，因此我们需要在训练开始前对所有输入数据进行归一化处理。
归一化方法是针对每一批数据（比如m个样本），在网络的每一层的输入之前增加归一化处理（均值为0，方差为1），将所有批处理数据强制在统一的数据分布下，即对该层的任一神经元（假设是第k维）$\hat{x}^{(k)}$采用如下公式：

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

其中$x^{(k)}$为该层第k个神经元的原始输入数据，$E[x^{(k)}]$为这一批输入数据在第k个神经元的均值，$\sqrt{Var[x^{(k)}]}$为这一批数据在k个审计元的标准差。

卷积操作的本质特性包括系数交互和参数共享，具体解释这两种特性以及作用。

稀疏交互

对于全连接网络，如果第k层有m个神经元，第k+1层有n个神经元，怎两层间的权重数目是mn,对于卷积网络，如果卷积核大小为k,则这两层间的权重个数是kn,因为后面一层，每一个神经元只与前一层k个神经元相连。一般k远远小于m，因此相对于全连接网络而言，参数数量大大减少了，因此具有稀疏交互的特征。
稀疏交互的物理意义是，通常图像，文本，语音等现实世界中的数据都具有局部的特征结构，我看可以先学习局部的特征，再将局部的特征组合起来形成复杂和抽象的特征。以人脸识别为例，最底层的神经元可以检测出各个角度的边缘特征，中间层的神经元可以将边缘组合起来得到眼睛，鼻子，嘴巴等复特征；最后，位于上层的神经元可以根据各个器官的组合检测出人脸的特征。

参数共享

参数共享是指在同一个模型的不同模块中使用相同的参数，它是卷积运算的固有属性。在学习的过程中，我们学习到的就是卷积核（的参数）。根据参数共享的思想，我们只需要学习一组参数集合，而不需要针对每个位置的每个参数都进行优化，从而大大降低了模型的储存需求。
参数共享的物理意义是使得卷积具有平移不变性。假如图像中有一只猫，那么无论它出现图像中的什么位置，都应该将它识别为猫，也就是说神经网络的输出对于平移变换具有不变性。

常用的池化操作有哪些？池化的作用是什么？

常用的池化操作主要针对非重叠区域，包括均值（mean pooling）,最大池（max pooling）。mean pooling是用一个区域的平均值代替这个区域，能够抑制由于领域大小受限造成估计值方差增大的现象，特点是对背景的保留效果更好；max pooling是用一个区域的最大值代替这个区域本身，能够抑制网络参数误差造成估计均值偏移现象，特点是更好地提取纹理信息。
池化操作的本质是降采样。除了能保显著降低参数数量，还能够保持对平移，伸缩，旋转操作的不变性。平移不变性是对小量平移而言，尺度一般是pooling的尺寸。伸缩（尺度）不变性，因为pooling就是一个能保持区域主要信息（如最大值）的压缩变换，因此具有伸缩不变性。旋转不变性，在配合多重pooling以后，对于max pooling，对于旋转的图像，我们仍能得到相同的结果，比如图像数字5。

卷积神经网络如何用于文本分类任务？

假设文本总单词量是N，每个单词可以转化为一个K维向量，向量维度K可以预先在其它语料库中获得，也可以作为未知的参数有网络训练得到。这样就组成了一个NXK的输入矩阵，可以看成是一个图像。

卷积层

在这个矩阵上，我们引入hXK大小的卷积核，进行卷积操作：

$$c_i = f(w * x_{i:i+h-1+b})$$

卷积后得到一个N-h+1维度大小的特征向量。通过不同大小的卷积核，提炼出不同的特征向量，构成卷积层的输出。

池化层

选用K-Max进行池化，也就是对每个卷积核作用之后得到的特征向量，进行k-Max池化，也就是选择特征向量中最大的K个特征，这样，每个特征向量转化成一个k维向量，效果就是，通过池化把不同长度的句子转化成定长的向量表示。若有M个卷积核，池化后就得到一个kXM的矩阵，k=1的话就是，1-Max,最终一个文本对应一个1XM的向量。
后面的网络结构就与具体的任务有关了，如果是人本分类，就最后接入一个全连接层，并使用softmax激活函数输出每个类别的概率。

ResNet的提出背景和核心理论是什么？

ResNet的提出背景是解决或缓解深层的神经网络训练中的梯度消失问题。当时的一个问题是，层数更深的神经网络反而会有更大的训练误差。这种反常，很大程度上归结于深度神经网络的梯度消失问题。随着网络层数的增加，很误差反向传递时，在每层间是相乘的，N层网络，从输出到输入，就是N次连乘，因此很容易造成梯度的消失与膨胀，影响参数的学习。下图就是传统深度神经网络与残差网络的对比。

如下图所示，输入x经过两个神经网络的变换得到F(x)，同时也短接到两层之后，最后这个包含两层神经网络模块输出H(x)=F(x)+x；这样一来，F(x)被设计为只需要拟合输入x与目标输出$\hat{H}(x)$的残差$\hat{H}(x)-x$，残差网络的名字因此而来。如果某一层的输出已经很好的拟合了期望结果，那么多加入一层不会使得模型变差，因为该层的输出将直接被短接到两层之后，直接相当于学习了一

个恒等映射，而跳过的两层只需要拟合上层输出和目标之间的残差即可。
ResNet可以有效改善深层的神经网络学习问题，使得训练更深的网络成为可能。

循环神经网络问题

建模序列化数据的一种主流深度学习模型。

设计目的

捕获长距离输入之间的依赖。

背景:

传统的前馈神经网络一般输入的都市一个定长的向量，无法处理变长的序列信息，即使通过一些方法把序列处理成定长的向量，模型也很难捕捉序列中的长距离依赖关系。RNN则通过将神经元串行起来处理序列化的数据。由于每个神经元能用它的内部变量保存之前输入的序列信息，因此整个序列被浓缩成抽象的表示，并可以据此进行分类或生成新的序列。近年来，得益于计算能力的大幅度提升和模型的改进，RNN在很多领域取得了突破性的进展--机器翻译，图像描述，推荐系统，智能聊天机器人，自动作词作曲等。

处理文本数据时，循环神经网络与前馈神经网络相比有什么特点？

Softmax function:

$$softmax(\mathbf{x}_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}$$

展开图计算

Figure 10.2: A recurrent network with no outputs. This recurrent network just processes information from the input $\boldsymbol{x}$ by incorporating it into the state $\boldsymbol{h}$ that is passed forward through time. *(Left)* Circuit diagram. The black square indicates a delay of 1 time step. *(Right)* The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

当前神经元的输入是，系统上一个的状态$h^{(t-1)}$，以及外部信号$x^{(t)}$,因此当前神经元的输出状态是：

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

可以看出，当前状态包含整个过去序列的信息。

从图可以看出，RNN可以处理变长度的输入问题，这是它区别于CNN的地方，他可以用来处理文本与语音问题。在最后的输入 $h^{(T)}$，我们可以再加一个softmax则可以用用来做文本分类任务。通过最小化损失误差(即输出的y和真实的类别之间的差距)，我们可以不断的训练网络，得到一个分类模型。相比于CNN等前馈网络，RNN由于具备对序列信息的刻画能力，往往能得到更准确的结果。

相对于循环图，展开图有如下两个优点：

(1)无论序列长度，学成的模型始终具有相同的输入大小；因为它指的是从一种状态到另外一种状态的转移，而不是在可变长度的历史状态上操作。

(2)我们可以在每个时间步使用相同参数的相同转移函数f。

这两个因素使得学习在所有时间步和所有序列长度上操作单一的模型f是可能的，而不需要在所有时间步学习独立的模型$g^{(t)}$.

$$h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, ..., x^{(2)}, x^{(1)})$$

$$= f(h^{(t-1)}, x^{(t)}; \theta)$$

循环神经网络

循环神经网络中一些重要的设计模式包含如下几种：

(1)每个时间步都有输出，并且隐藏单元之间有循环连接的循环网络，如下图所示。

RNN的输入到隐藏的连接由权值矩阵U参数化，隐藏到隐藏的循环由权重矩阵W参数化，隐藏到输出层的连接由权重矩阵V参数化。$L^{(t)}$是损失函数。

前向传播公式:假设激活函数是tanh，我们有如下的更新方程:

- $\mathbf{a}^{(t)} = \mathbf{b} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{U}x^{(t)}$
- $\mathbf{h}^{(t)} = \tanh(\mathbf{a})$
- $\mathbf{o}^{(t)} = \mathbf{c} + \mathbf{V}\mathbf{h}^{(t)}$
- $\hat{\mathbf{y}}^{(t)} = softmax(\mathbf{o}^{(t)})$

其中$\mathbf{b}, \mathbf{c}$是偏置向量,$\mathbf{U}, \mathbf{V}, \mathbf{W}$是权重矩阵。

这一循环网络将一个输入序列映射到相同长度的输出序列。与$\mathbf{x}$序列配对的$\mathbf{y}$的总的损失就是所有时间步的损失之和。例如，$L^{(t)}$为给定的$x^{(1)}, x^{(2)}, ..., x^{(t)}$后$y^{(t)}$的负对数似然，则:

- $L((x^{(1)}, x^{(2)}, ..., x^{(\tau)}), (y^{(1)}, y^{(2)}, ..., y^{(\tau)})) = \sum_t L^{(t)}$
- $= -\sum_t \log p_{model}(y^{(t)}|(x^{(1)}, x^{(2)}, ..., x^{(t)})$

其中$p_{model}(y^{(t)}|(x^{(1)}, x^{(2)}, ..., x^{(t)})$需要读取模型输出向量$\hat{\mathbf{y}}^{(t)}$对应于$\mathbf{y}^{(t)}$的项。应用于展开图且代价为$O(\tau)$的反向传播算法是通过时间反向传播(back-propagation through time, BPTT)。

(2)每个时间步都产生一个输出，只有当前时刻的输出到下个时刻的隐藏单元之间有循环连接的循环网络。如下图所示:

该图中，RNN被训练将特定输出值放入o中，并且o是被允许传播到未来的唯一信息。此处没有从h前向传播的直接连接。之前的h仅通过产生的预测间接地连接到当前。o通常缺乏过去的重要信息，除非它非常高维并且内容丰富。这使得该图中的RNN不那么强大，但是它更容易训练，因为每个时间步可以与其他时间步分离训练，允许训练期间更多的并行化。

Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$ and the loss is $L^{(t)}$. *(Left)* Circuit diagram.

(3)隐藏单元之间存在循环连接，但读取整个序列后产生单个输出的循环网络，如下图所示：

这样的网络可以用于概括序列并产生用于进一步处理的固定大小的表示。在结束处可能存在目标，或者通过更下游模块的反向传播来获得输出$o^{(t)}$上的梯度。

Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $\boldsymbol{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

这三种结构分别用于什么情况？

循环神经网络的梯度消失问题

RNN的求解可以采用BPTT(Back Propagation Through Time，基于事件的反向传播)算法实现。
梯度消失问题导致RNN不能捕获长距离输入之间的依赖关系。
传统的RNN梯度可以表示如下：

- $\frac{\partial net_t}{\partial net_1} = \frac{\partial net_t}{\partial net_{t-1}} \frac{\partial net_{t-1}}{\partial net_{t-2}} \cdots \frac{\partial net_2}{\partial net_1}$

其中：

- $net_t = Ux_t + Wh_{t-1}$
- $h_t = Ux_t + f(net_t)$
- $y = g(Vh_t)$
- $\frac{\partial net_t}{\partial net_{t-1}} = \frac{\partial net_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial net_{t-1}} = Wdiag[f'(net_{t-1})]$
  - $= W_{ij} f'(net_{t-1}^j)$

是一个nxn的矩阵，n是隐韩层net_{t-1}神经元的个数。

生成式对抗网络问题

强化学习问题

神经网络的种类适用场景与优缺点

基本C++问题

字符串指针赋值问题

如果指针没有分配空间，则会出现0xC0000005: Access violation writing location

```
void main(int n)
{
    //char pStr[] = "abc";//right
    //char pStr[4] = "abc";//right
        //char pStr[3] = "abc";//wrong,need >=4
    char* pStr = "abc"; //wrong, 没有分配空间,
    char temp = 'b';
    pStr[0] = temp;
    printf("%s\n",pStr);
}
```

sizeof

定义一个空的类型，里面没有任何成员变量和成员函数，但是对该类型求sizeof，得到的结果是多少？
答案：1；
不是0的原因是，当我们声明该类型的实例时，它必须在内存中有一定的空间，否则无法使用这些实例，至少占多少内存，由编译器决定，在VS中，每个空类型的实例占1字节的空间。
面试官：如果在该类型里面添加一个构造函数和析构函数，再对该类型求sizeof，得到的结果又是多少？
应聘者：1。因为调用构造函数与析构函数只需要知道函数的地址即可，而这些函数的地址只与类型相关，与类型的实例无关，编译器也不会因为这两个函数而在实例中添加额外的信息。
面试官：那如果把析构函数标记为虚函数呢？
应聘者：编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个实例中添加一个指向该虚函数表的指针，在32位的机器上，一个指针占4字节，因此求sizeof是4,不是4+1,因为只是指向虚函数表的指针的大小，在64位机器上，一个指针占8字节，因此求sizeof是8.

数组与指针的sizeof

sizeof对数组，得到整个数组所占空间的大小。
sizeof对指针，得到指针本身所占空间的大小。
数组的名字就是一个指针，该指针指向数组的第一个元素，我们可以用一个指针来访问数组。也就是数组名加元素下标来访问数组元素或者获取数组元素的值。

```
int data1[] = {1,4,7,9,6};
cout<<sizeof(data1)<<endl;
cout<<"1st of data1:"<<*data1<<endl;
cout<<"2st of data1:"<<*(data1 + 1)<<endl;
cout<<"1st of data1:"<<*(data1+3)<<endl;

20
1st of data1:1
2st of data1:4
1st of data1:9
```

class与struct区别

C++中Class成员变量或者成员函数默认private,而struct成员变量默认为public.
C#中，class与struct的成员变量与成员函数都默认是private.

```
#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;
}
size of myints: 5
sizeof(myints): 20
```

String 类

string的构造、拷贝构造、析构、赋值、输出、比较、字符串加、长度、子串

数据结构

数据结构一直是技术面试的重点，大多数面试题都是围绕数组，字符串，链表，树，栈以及队列这几种常用的数据结构展开的，因此每一个面试者毒药熟练掌握着集中数据结构。
数组和字符串是两种最基本的数据结构，他们用连续内存分别储存数据和字符。链表和树是面试中出现频率最高的数据结构，由于操作链表和树需要大量的指针，应聘者在解决相关问题的时候一定要留意代码的鲁棒性，否则容易出现程序崩溃的问题。栈是一个与递归密切相关的数据结构，同样队列也与广度优先遍历算法密切相关，深入理解这两种数据结构能帮助我们解决很多算法问题。

数组

它占用一块连续的内存空间并按照顺序存储数据，创建数据时，我们需要首先指定数组的容量大小，然后根据大小分配内存。即使我们只在数组中存储一个数字，也需要为所有的数据预先分配内存，因此数组的空间效率不是很好，经常会有空闲的区域没有得到充分利用。
数组中的内存是连续的，因此我们可以根据下标在O(1)时间读/写任何元素，因此它的时间效率是很高的。我们可以根据数组时间效率高的优点来实现简单的哈希表，把数组的下标设为哈希表的键值(key),数组中的数据设为哈希表的value。这样就可以在O(1)时间内实现查找，从而快速，高效地解决很多问题。

Vector

针对数组空间效率不高的问题，人们有设计实现了多种动态数组，比如STL中的vector,为了避免浪费，我们首先为数组开辟较小的空间，然后往数组中添加数据，当数据的数目超过数字的容量时，我们再重新分配一个更大的空间(STL中vector每次扩容时，新的容量都是前一次的两倍)，把之前的数据复制到新的数组中，再把之前的内存释放，这样就减小内存的浪费。每次扩容都需要大量的额外操作，因此尽量减小改变数组容量大小的次数。
简单设计Vector:一开始，新建一个大小为N的数组(弄成数组是保证内存是连续的)，当vector中的元素内存超出开始的数组内存大小时，就新建一个大小为2N的数值，把原来的数组copy到新的数组，再释放原来的内存。用这种节奏来创建更大的数组来容纳更多的vector。

```
void CList::ReplaceEmpty(char str[], int length)
{
    if(str == NULL || length <=0)
        return;
    int count = 0;
    int num = 0;
    while(str[num] != '\0')
    {
        if(str[num] == ' ')
            count++;
        num++;
    }
    int newStringLength = num + 2*count;
    int newNum = newStringLength;
    if(newStringLength >= length)
        return;
    for(int n = num; n >= 0; n--)
    {
        if(str[n] == ' ')
        {
            str[newNum--] = '0';
            str[newNum--] = '2';
            str[newNum--] = '%';
        }
        else
            str[newNum--] = str[n];
    }
}
```

二叉树:

树的遍历

1. 前序遍历:先访问根结点,再访问左子结点,最后访问右子结点。
2. 中序遍历:先访问左子结点,再访问根结点,最后访问右子结点。
3. 后序遍历:先访问左子结点,再访问右子结点,再访问根结点。
   前中后表示访问根结点的时间。

- 前序遍历：先访问根结点，再访问左子结点，最后访问右子结点。图 2.5 中的二叉树的前序遍历的顺序是 10、6、4、8、14、12、16。
- 中序遍历：先访问左子结点，再访问根结点，最后访问右子结点。图 2.5 中的二叉树的中序遍历的顺序是 4、6、8、10、12、14、16。
- 后序遍历：先访问左子结点，再访问右子结点，最后访问根结点。图 2.5 中的二叉树的后序遍历的顺序是 4、8、6、12、16、14、10。

图 2.5　一个二叉树的例子

根据前序,中序来重建树:

如图 2.7 所示，前序遍历序列的第一个数字 1 就是根结点的值。扫描中序遍历序列，就能确定根结点的值的位置。根据中序遍历特点，在根结点的值 1 前面的 3 个数字都是左子树结点的值，位于 1 后面的数字都是右子树结点的值。

图 2.6　根据前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}重建的二叉树

前序中,第一个点就是根结点,根据根结点的值,在中序中找到根结点的值,则根结点之前的是坐子树,根结点之后的是右子树。根据中序排列中左子树的大小(长度)L,在前序排列中找到左子树(从1到L),因此对于左子树,又得到了前序和中序排列。对右子树也可以得到前序与中序排列。

由此递归实现。

由于在中序遍历序列中，有 3 个数字是左子树结点的值，因此左子树总共有 3 个左子结点。同样，在前序遍历的序列中，根结点后面的 3 个数字就是 3 个左子树结点的值，再后面的所有数字都是右子树结点的值。这样我们就在前序遍历和中序遍历两个序列中，分别找到了左右子树对应的子序列。

图 2.7　在二叉树的前序遍历和中序遍历的序列中确定根结点的值、左子树结点的值和右子树结点的值

```
BinaryTreeNode* CTree::ConstructBinaryTree(int* preorderS,int* preorderEnd, int* inorderS, int
* inorderEnd)
{
    int rootValue = preorderS[0];
    int* rootNode = inorderS;

    while(rootNode <= inorderEnd && *rootNode != rootValue) //find root in inorder
    {
        ++rootNode;
    }
    int LTLen = rootNode - inorderS;
    BinaryTreeNode* RootTree = new BinaryTreeNode();
    RootTree->m_nValue = rootValue;
    if(LTLen >0)
    {
        RootTree->m_pLeft = ConstructBinaryTree(preorderS+1, preorderS+LTLen, inorderS, rootN
ode-1);
    }
    if(LTLen < preorderEnd - preorderS)
    {
        RootTree->m_pRight = ConstructBinaryTree(preorderS+LTLen + 1, preorderEnd, rootNode+1
, inorderEnd);
    }
    return RootTree;
}

void CTree::PreOrderPrint(BinaryTreeNode* BTree)
{
    if(BTree != NULL)//print root first
    {
        m_preorder.push_back(BTree->m_nValue);
        cout<<BTree->m_nValue<<endl;
```

```
    }
    if(BTree->m_pLeft != NULL)
    {
        PreOrderPrint(BTree->m_pLeft);
    }
    if(BTree->m_pRight != NULL)
    {
        PreOrderPrint(BTree->m_pRight);
    }
}

void CTree::InOrderPrint(BinaryTreeNode* BTree)
{
    if(BTree->m_pLeft != NULL)
    {
        InOrderPrint(BTree->m_pLeft);
    }
    if(BTree != NULL)//print root second
    {
        m_preorder.push_back(BTree->m_nValue);
        cout<<BTree->m_nValue<<endl;
    }
    if(BTree->m_pRight != NULL)
    {
        InOrderPrint(BTree->m_pRight);
    }
}

void CTree::PostOrderPrint(BinaryTreeNode* BTree)
{
    if(BTree->m_pLeft != NULL)
    {
        PostOrderPrint(BTree->m_pLeft);
    }

    if(BTree->m_pRight != NULL)
    {
        PostOrderPrint(BTree->m_pRight);
    }

    if(BTree != NULL)//print root last
    {
        m_postorder.push_back(BTree->m_nValue);
        cout<<BTree->m_nValue<<endl;
    }
}
```

栈与队列

栈:先进后出，压入push,弹出pop
队列:后进先出，压入push,弹出pop
用两个栈来实现队列。模板类的实现要在.h里面

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

template <class T>
class CStackDeque
{
public:
    CStackDeque(void){};
    ~CStackDeque(void){};
    void appendTail(const T& node);
    T deleteHead();

private:
    stack<T> m_stack1;
    stack<T> m_stack2;
};

template<class T>
void CStackDeque<T>::appendTail(const T& node)
{
    m_stack1.push(node);
}

template<class T>
T CStackDeque<T>::deleteHead()
{
    if(m_stack2.size() <= 0)
    {
        while(m_stack1.size() > 0)
        {
            T& node = m_stack1.top();
            m_stack1.pop();
            m_stack2.push(node);
        }
    }

    if(m_stack2.size() <= 0)
        throw new exception("queue is empty!");

    T node2 = m_stack2.top();
    m_stack2.pop();
    return node2;
}
void main()
{
    CStackDeque<int> queue;
```

```
    queue.appendTail(1);
    int value = queue.deleteHead();
}
```

排序算法

核心是在数组中选择一个值，重新排列数组，使得该值左边的元素都小于该值，右边的元素都大于该值。分成两组后，再继续这种操作，以此递归下去，最终得到全排序。
下面都是对数组操作，因此要注意下标问题。

```
int CSort::Partition(int data[], int length, int start, int end)
{
        if(start >= end)
                return start;
        if(data == NULL || start <0 || end > length || length <= 0)
                throw new exception("Invidate parameters!");
        int small = start;
        srand((unsigned)time(NULL));//random seed
        int randInt = rand()%(end - start);
        int value = data[start + randInt];
        cout<<"Value: "<<value<<endl;
        swap(data[start+randInt],data[end]);//all element start from start,
        for(int index = start; index < end; index++)
        {
                if(data[index] < value)
                {
                        swap(data[index], data[small]);
                        small++;
                }
        }
        swap(data[small], data[end]);
        return small;
}

void CSort::QucikSort(int data[], int length, int start, int end)
{
    if(end <= start)
        return;

    int index = Partition(data,length,start,end);
    if(index>start)
    {
        QucikSort(data,length,start, index-1);
    }
    if(index < end)
    {
        QucikSort(data, length, index + 1 ,end);
    }
}
```

与，或，异或

负数的二进制表示
举例：
-5 在计算机中表达为：11111111 11111111 11111111 11111011。转换为十六进制：0xFFFFFFFB。
-1在计算机中如何表示：
先取1的原码：00000000 00000000 00000000 00000001
得反码：11111111 11111111 11111111 11111110
得补码：11111111 11111111 11111111 11111111
可见，－1在计算机里用二进制表达就是全1。16进制为：0xFFFFFFFF。
注：十六进制前缀是0x。

表 2.1　与、或、异或的运算规律

与（&）	0 & 0 = 0	1 & 0 = 0	0 & 1 = 0	1 & 1 = 1
或（|）	0 | 0 =0	1 | 0 = 1	0 | 1 = 1	1 | 1 = 1
异或（^）	0 ^ 0 = 0	1 ^ 0 = 1	0 ^ 1 = 1	1 ^ 1 = 0

左移运算符 m << n 表示把 m 左移 n 位。左移 n 位的时候，最左边的 n 位将被丢弃，同时在最右边补上 n 个 0。比如：

00001010 << 2 = 00101000

10001010 << 3 = 01010000

计算二进制中1的个数

```
int CRecursion::CountNumber1(int n)
{
    int count = 0;
    while(n)
    {
        if(n&1)
            count++;
        n = n>>1;
    }
    return count;
}
```

链表

在O(1)时间内删除一个节点

方法：通过用节点下一个节点来覆盖这个节点，因此不需要查询将被删除节点前面的所有节点。

问题：

如果你传入一个指针，实际上你传入的就是一个物理地址，对于链表而言，你不能在链表头接入一个元素，因为最终出来的时候，指针（的地址）一直没变，因此技术你在函数里面对指针进行赋值，但是出了函数，有没有效果了。

一种会出错的删除做法：

```
void CList::DeleteNode(ListNode* pListHead, ListNode* pToBeDeleted)
{
    if(!pListHead || !pToBeDeleted)
        return;

    if(pToBeDeleted->m_pNext == NULL)
    {
        pToBeDeleted = NULL;
    }
    if(pToBeDeleted->m_pNext != NULL)
    {

        /*pToBeDeleted->m_value = pToBeDeleted->m_pNext->m_value;/出错的做法
        pToBeDeleted->m_pNext = pToBeDeleted->m_pNext->m_pNext;
        delete pToBeDeleted->m_pNext;*///因为删除了pToBeDeleted->m_pNext，因此索引到此，访问了空
地址，程序会崩溃。正确的做法如下

        ListNode* pNext = pToBeDeleted->m_pNext;//将要被删除的节点，在删除之前，复制它的值
        pToBeDeleted->m_value = pToBeDeleted->m_pNext->m_value;
        pToBeDeleted->m_pNext = pToBeDeleted->m_pNext->m_pNext;
        delete pNext;
    }
}
```

对数组中奇偶数据分类排序

```
void CCArray::ReorderOddEven(int* data, int length, bool (*Func)(int))
{
    if(data==NULL || length <= 0)
        return;
    int* pBegin = data;
    int* pEnd = data + length - 1;

    while(pBegin < pEnd)
    {
        while(!Func(*pBegin))
        {
            pBegin++;
        }

        while(Func(*pEnd))
```

```
        {
            pEnd--;
        }

        int temp = *pEnd;
        *pEnd = *pBegin;
        *pBegin = temp;
    }
}
void main()
{
    const int length = 6;
    int data[] = {4,1,5,3,6,7};
    CCArray* test_arr = new CCArray();
    test_arr->ReorderOddEven(data, length, IsEven);//函数指针的调用
    for(int i = 0; i < length; i++)
    {
        cout<<data[i]<<endl;
    }
}
bool IsEven(int value)
{
    return (value&1) == 0;
}
```

为啥要用C++智能指针

1. 内存泄漏：是指操作系统将空间分配给你，但是那个空间被你申请后并没有被使用，并且没有相应的释放语句。也就是该空间不再被任何指针或引用所引用，成为一个幽灵空间，操作系统以为你在控制它，但其实你并没有控制它。
2. 重复释放：程序通过free或delete语句释放已经不属于该程序的空间。这是很危险的，因为第二次free的空间已经被别的程序所使用。所以C++中把这种错误当成了致命错误。
   栈上的空间是由系统分配的。堆上的空间是由用户自己分配的。通过new创建，free释放。

C++什么时候该使用new?该注意什么来防止内存泄漏？

打印1到最大的n位数，比如n=2,怎打印1到99.

```
void PrintToMaxofNDigits(int n);//输入要打印的位数
void PrintNumber(char* number);//打印字符串
bool Increment(char* number);//每次增加1看是否越界
void CRecursion::PrintToMaxofNDigits(int n)
{
    if(n <= 0)
        return;

    char *number = new char[n+1];
```

```
    memset(number,'0',n);
    number[n] = '\0';
    while(!Increment(number))
    {
        PrintNumber(number);
    }
    delete []number;
}

bool CRecursion::Increment(char* number)
{
    bool isOverFlow = false;
    int numLength = strlen(number);
    int nTakeOver = 0;
    for(int i = numLength - 1; i >=0; i--)
    {
        int nSum = number[i] -'0' + nTakeOver;
        if(i == numLength -1)
            nSum++;

        if(nSum >= 10)
        {
            if(i == 0)
            {
                isOverFlow = true;
            }
            else
            {
                number[i] = nSum - 10 + '0';
                nTakeOver = 1;
            }
        }
        else
        {
            number[i] = nSum + '0';
            break;
        }
    }
    return isOverFlow;
}

void CRecursion::PrintNumber(char* number)
{
    bool isBeginning0 = true;
    int nLength = strlen(number);
    for(int i = 0; i < nLength; ++i)
    {
        if(isBeginning0 && number[i] != '0')
            isBeginning0 = false;
        if(!isBeginning0)
        {
            printf("%c",number[i]);
```

```
        }
    }
    printf("\n");
}
```

printf("%c",number[i]);打印字符
接下来是递归算法：
通过递归，打印只针对于个位数，也就是index == length -1的位置。

```
void CRecursion::PrintToMaxofNDigitsRecu(int n)
{
    if(n <= 0)
        return;

    char* number = new char[n+1];
    number[n] = '\0';
    for(int i =0; i < 10; i++)
    {
        number[0] = i+'0';
        PrintToMaxofNDigitsRecursively(number, n, 0);
    }
    delete[] number;
}

void CRecursion::PrintToMaxofNDigitsRecursively(char* number, int length, int index)
{
    if(index == length -1)
    {
        PrintNumber(number);
        return;
    }

    for(int i = 0; i < 10; i++)
    {
        number[index+1] = i + '0';
        PrintToMaxofNDigitsRecursively(number, length, index + 1);
    }
}
```

输入两颗二叉树A和B，判断B是不是A的子结构。

通过递归实现。如果两个头节点相同，则比较两者的左右子树。如果不相同，则把A的左右子树当成树，看他们是否含有B。

```
bool CTree::HasSubtree(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2)
{
    bool result;
    if(pRoot2 == NULL)
```

```
        return true;
    if(pRoot1 == NULL)
        return false;

    if(pRoot1->m_nValue == pRoot2->m_nValue)
    {
        result = DoesTree1HaveTree1(pRoot1, pRoot2);
    }
    if(!result)
    {
        result = DoesTree1HaveTree1(pRoot1->m_pLeft, pRoot2);
    }
    if(!result)
    {
        result = DoesTree1HaveTree1(pRoot1->m_pRight, pRoot2);
    }
    return result;
}

bool CTree::DoesTree1HaveTree1(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2)
{
    bool result;
    if(pRoot2 == NULL)
        return true;
    if(pRoot1 == NULL)
        return false;

    if(pRoot1->m_nValue != pRoot2->m_nValue)
    {
        return false;
    }

    return DoesTree1HaveTree1(pRoot1->m_pLeft, pRoot2->m_pLeft)&&DoesTree1HaveTree1(pRoot1->m
_pRight, pRoot2->m_pRight);
}
```

输入两个整数序列，第一个是栈的压入序列，第二个是不是栈的弹出序列

比如压入序列是1，2，3，4，5;则4，5，3，2，1是可能的弹出序列，而4，3，5，1，2不是弹出序列。

```
bool CTree::IsPopOrder(const int* pPush, const int* pPop, int nLength)
{
    stack<int> stackData;
    const int* pNextPush = pPush;
    const int* pNextPop = pPop;
    int pushLength = 0;
    while(pushLength < nLength)
    {
        int topData = *pNextPop;
```

```
        while(*pNextPush != topData)
        {
            if(pNextPush == NULL)//no this element in the push list
                return false;
            stackData.push(*pNextPush);
            pNextPush++;
            pushLength++;
        }
        pNextPush++;
        pushLength++;
        pNextPop++;
    }
    while(!stackData.empty())
    {
        if(stackData.top() != *pNextPop)
        {
            return false;
        }
        stackData.pop();
        pNextPop++;
    }
    return true;
}
```

标准模板库

STL这节包含两部分，一个是数据结构，一个是算法。对于这些数据结构，一方面要知道有哪些数据结构，然后要知道每一种数据结构的特征与功能，能实现哪些操作，然后是我们要知道每种数据结构使用的场景。然后要知道这些数据结构实现的原理是什么？比如红黑树这些。最终就是知道原理以后，我们能基于特定任务而创建自己的数据结构。
algorithms，要知道有哪些算法，这些算法的操作对象是什么(迭代器)？哪些数据结构。然后就是熟悉常用的一些算法。
Boost库包含150多个库，其中就有线性代数库uBLAS，Boost是STL的高阶版，当前自己用不到，因此，了解了STL就足够了。如果要用线性代数库，则Eigen可能是一个好的选择。遇到问题，需要开发库，原则来说先看已有的库，基本上已有的库就呢个满足自己的需求了。
参考这篇文章 C++ STL 一般总结
STL中体现了泛型程序设计的思想，泛型是一种软件的复用技术。

STL的六大组件：

容器（Container）

是一种数据结构，如list,vector,deques,以模板的方法提供，为了访问容器中的数据，可以使用由容器类输出的迭代器。
顺序容器：vector,lsit,deque,string.
关联容器：set,multiset, map, mulmap,hash_set,hash_map,hash_multiset,hash_multimap
杂项：stack,queue,valarray,bitset

迭代器(Iterator)

提供了访问容器中对象的方法，例如，可以使用一堆迭代器指定list或vector中的一定范围的对象。迭代器如同一个指针。事实上，C++指针也是一种迭代器。但是迭代器也可以是那些定义了operator*()以及其它类似于指针的操作符地方的类对象。

算法(algorithm)

是用来操作容器中数据的模板函数。列例如，STL中的sort()来对一个vector中的数据进行排序。用find()来搜索一个list中的对象，函数本身与他们操作的数据的结构和类型无关，因此他们可以在从简单数组到高级复杂容器的任何数据结构上使用。

仿函数(Function object)

仿函数(functor)又称为函数对象(function object)，其实就是重载了()操作符的struct，没有什么特别的地方。

迭代适配器(adaptor)

空间配置器(allocator)

其中主要工作包括1.对象的创建与销毁，2.内存的获取与释放

STL底层数据结构实现

容器的基本特征

概念：容器是储存其他对象的对象。被储存的对象必须是同一类型。

基本特征：以下用X表示容器类型（后面会讲到），T表示储存的对象类型（如int）；a和b表示为类型X的值；u表示为一个X容器的标识符（如果X表示vector<int>，则u是一个vector<int>对象。）

表达式	返回类型	说明	复杂度
X::iterator	指向T的迭代器类型	满足正向迭代器要求的任何迭代器	编译时间
X u		创建一个名为u的空容器	固定
X ()		创建一个匿名空容器	固定
X u(a)		同X u(a);	线性
a.begin()	迭代器	返回指向容器第一个元素的迭代器	固定
a.end()	迭代器	返回指向超尾值的迭代器	固定
a.size()	无符号整型	返回元素个数	固定
a.swap()	void	交换a和b内容	固定
a == b	可转换为bool	如果a和b长度相当且每个元素都相等，则为真	线性
a != b	可转换为bool	返回！（a == b）	线性

顺序容器的基本特征

概念：序列是对基本容器的一种改进，在保持其基础功能上增加一些我们需要的更为方便的功能。

要求：序列的元素必须是严格的线性顺序排序。因此序列中的元素具有确定的顺序，可以执行将值插入到特定位置、删除特定区间等操作。

序列容器基本特征：以下用t表示类型为T（储存在容器中的值的类型）的值，n表示整数，p、q、i和j表示迭代器。

表达式	返回类型	说明
X a(n,t)		声明一个名为a的由n个t值组成的序列
X(n,t)		创建一个由n个t值组成的匿名序列
X a(i,j)		声明一个名为a的序列，并将其初始化为区间[i,j)的内容
X(i,j)		创建一个匿名序列，并将其初始化为区间[i,j)的内容
a.insert(p,t)	迭代器	将t插入到p的前面
a.insert(p,n,t)	void	将n个t插入到p的前面
a.insert(p,i,j)	void	将区间[i,j)的元素插入到p前面
a.erase(p)	迭代器	删除p所指向的元素
a.erase(p,q)	迭代器	删除区间[p,q)中的元素
a.clear()	void	清空容器

不同容器特有的特征

不同容器特有的特征：

表达式	返回类型	含义	支持的容器
a.front()			vector、list、deque
a.back()			vector、list、deque
a.push_front(t)			list、deque
a.push_back(t)			vector、list、deque
a.pop_front(t)			list、deque
a.pop_back(t)			vector、list、deque
a[n]			vector、deque
a.at(t)			vector、deque

data structure

array

vector

deque

list

stack

queue

set

map

pair

algorithms

Windows C++

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 old

```
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;
}
```

树

二叉搜索树的后序遍历

输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果，是返回true,否则返回false。假设输入的数组的任意两个数字都不相同。
二叉搜索树的特征是结点的值大于所有左子树的值，且小于所有右子树的值。如果是后序遍历，则最后一个数是根节点的值，这个值通过一刀把它前面的数组切分成左右两部分，左部分的值都小于小于该值，为树的左子树，右边部分的值都大于该值，是右子树。如果这个划分不存在，则输入数组不是某二叉树的后序遍历。

例如输入数组{5, 7, 6, 9, 11, 10, 8}，则返回 true，因为这个整数序列是图 4.6 二叉搜索树的后序遍历结果。如果输入的数组是{7, 4, 6, 5}，由于没有哪棵二叉搜索树的后序遍历的结果是这个序列，因此返回 false。

图 4.6　后序遍历序列 5、7、6、9、11、10、8 对应的二叉搜索树

```
bool CTree::VerifySequenceOnBST(int sequence[], int length)
{
    if(sequence == NULL || length< 0)
        return false;

    int rootValue = sequence[length -1];
    int leftTreeLength = 0;
    for(int i = 0; i < length; i++)//finding the point can devide the sequence into 2 parts
    {
```

```
        if(sequence[i] > rootValue && leftTreeLength == 0)
        {
            leftTreeLength = i+1;
            break;
        }
    }
    //make sure the right part large than the root value
    for(int i = leftTreeLength; i < length; i++ )
    {
        if(sequence[i] < rootValue)
            return false;
    }

    bool left = true;
    bool right = true;
    if(leftTreeLength > 0)
        left = VerifySequenceOnBST(sequence, leftTreeLength);

    if(length - 1 - leftTreeLength > 0)
        left = VerifySequenceOnBST(sequence + leftTreeLength, length - leftTreeLength - 1);

    return (left&&right);
}
```

二叉树中和为某一值的路径

输入一颗二叉树和一个整数，打印出二叉树中结点值的和为输入整数的
所有路径。从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。
通过递归来实现。求和到叶结点，看是否符合。符合则打印，不符合则把改节点弹出并返回上一级
看右结点是否符合。都不符合则再返回上一级，看右边的子树。

```
void CTree::FindPath(BinaryTreeNode* pRoot, int exceptedSum)
{
    if(pRoot == NULL)
        return;

    vector<int> treePath;
    int currentSum = 0;
    FindPath(pRoot->m_pRight,treePath,currentSum,exceptedSum);
}

void CTree::FindPath(BinaryTreeNode* pRoot, vector<int>& path, int currentSum, int exceptedSu
m)
{
    currentSum += pRoot->m_nValue;
    if(currentSum > exceptedSum)//can't delete this node, because m_nValue can be negative.
    {
        pRoot =NULL;
        return;
    }
```

```
    path.push_back(pRoot->m_nValue);
    if(currentSum == exceptedSum && pRoot->m_pLeft == NULL && pRoot->m_pRight == NULL)
    {
        for(int i=0 ; i< path.size(); i++)
        {
            printf("%c\t",path[i]);
        }
        printf("\n");
    }
    if(pRoot->m_pLeft != NULL)
    {
        FindPath(pRoot->m_pLeft,path,currentSum,exceptedSum);
    }
    if(pRoot->m_pRight != NULL)
    {
        FindPath(pRoot->m_pRight,path,currentSum,exceptedSum);
    }
    path.pop_back();
}
```

从上到下打印二叉树

通过deque实现，在pop_front的同时，把该结点的左右子节点push_back到队列的尾部。这样打印完整个deque

```
void CTree::PrintFromTopToBottom(BinaryTreeNode* pRoot)
{
    if(!pRoot)
        return;

    deque<BinaryTreeNode*> dequeTreeNode;
    dequeTreeNode.push_back(pRoot);
    while(dequeTreeNode.size())
    {
        BinaryTreeNode* currentNode = dequeTreeNode.front();
        dequeTreeNode.pop_front();
        printf("%c",currentNode->m_nValue);

        if(currentNode->m_pLeft)
            dequeTreeNode.push_back(currentNode->m_pLeft);

        if(currentNode->m_pRight)
            dequeTreeNode.push_back(currentNode->m_pRight);
    }
}
```

之字形打印整个二叉树

通过两个stack来实现，比如第1层存于stack1，在打印第一层时，把第一层的子节点(也就是dierceng)存于stack2,打印完stack1,此时stack为空，再打印stack2,,在打印stack2时，把第二层的子节点(也就是第三层)存于stack1。以此循环下去，直到stack1,stack2为空为止。

复杂链表的复制

请实现函数ComplexListNode *Colne(ComplexListNode* pHead),复制一个复杂链表在复杂链表中除了有一个m_pNext指针指向下一个节点，还有一个m_pSibling指针指向链表中的任意节点或者nullptr。节点的C++定义如下。
分三步：
1.在原来链表每个结点后面复制前面的节点，做成一个map,也就是hash表。这样原来每个节点N后面都跟着一个N'。
2.为链表中新加的每个N'链接指针m_pSibling。
3.把链表结点根据序号的奇偶性分成两部分，这样就得到了我们想要的clone的链表。返回表头即可。

二叉搜索树与双向链表

将一个二叉搜索树转化成一个排序的双向链表。要求不能添加任何新的结点，只能调整树中结点指针的指向。如图：

图 4.12　一棵二叉搜索树及转换之后的排序双向链表

通过根结点，把树分为两部分，对每部分都转化成一个双向链表，再最终把这两部分与根节点拼起来，就得到最终的双向链表。问题转化为两个问题，1.怎么把树分成两部分再实现递归;2.怎么办把根结点与两个排序好的链表进行链接。

```
BinaryTreeNode* CTree::Convert(BinaryTreeNode* pRoot)
{
    if(pRoot == NULL)
        return;

    BinaryTreeNode* pLastNodeInList = NULL;
    ConvertNode(pRoot, &pLastNodeInList);
    BinaryTreeNode* pHeadofList =  pLastNodeInList;
    while(pHeadofList!=NULL &&pHeadofList->m_pLeft !=NULL)
    {
        pHeadofList = pHeadofList->m_pLeft;
    }
```

```
    return pHeadofList;
}

void CTree::ConvertNode(BinaryTreeNode* pRoot,BinaryTreeNode** pLastNodeInList)
{
    if(pRoot == NULL)
        return;

    BinaryTreeNode* pCurrentNode = pRoot;
    if(pCurrentNode->m_pLeft!=NULL)
        ConvertNode(pCurrentNode->m_pLeft, pLastNodeInList);
        //左结点转化为双向链表
    pCurrentNode->m_pLeft = *pLastNodeInList;//当前节点的右节点连接到双向链表
    if(*pLastNodeInList != NULL)
        (*pLastNodeInList)->m_pRight = pCurrentNode;
        //当前节点变成链表最后一个节点
    *pLastNodeInList = pCurrentNode;
    if(pCurrentNode->m_pRight!=NULL)
        ConvertNode(pCurrentNode->m_pRight, pLastNodeInList);
}
```

字符串排列

输入一个字符串，打印出该字符串中字符的所有排列，比如abc,则所有排列为：abc,acb,bca,bac,cba,cab.

```
void CRecursion::Permutation(char* pStr)
{
    if(pStr == nullptr)
        return;
    Permutation(pStr,pStr);
}

void CRecursion::Permutation(char* pStr, char* pBegin)
{
    if(*pBegin == '\0')
        printf("%s\n",pStr);
    else
    {
        for(char* pch = pBegin; *pch != '\0'; ++pch)
        {
            char temp = *pch;//pch与pBegin置换
            *pch = *pBegin;
            *pBegin = temp;
            Permutation(pStr, pBegin+1);
            temp = *pch;//还原到原来的样子
            *pch = *pBegin;
            *pBegin = temp;
        }
    }
```

```
}

void main(int n)
{
    CRecursion recu;
    char pStr[] = "abc";
    recu.Permutation(pStr);
    printf("Print end!\n");
}
```

序列化二叉树

请实现两个函数，分别用来序列化与反序列化二叉树。若结点的子节点为空，则用$来代替，如果两个子节点都不存在，则用两个$来代替。节点之间用","隔开。
为确保根节点在最前面，采取前序遍历来序列化数据。ostream用来文件写操作，从内存中写入到硬盘中。
反序列化则把字符串重构成二叉树。根据读取的是数还是$来判定是否需要再建子节点。建完左子节点再建右子节点。

```
void CTree::Serialize(BinaryTreeNode* pRoot, ostream& stream)
{
    if(pRoot == NULL)
    {
        stream <<"$,";
        return;
    }
    stream << pRoot->m_nValue<<',';
    Serialize(pRoot->m_pLeft,stream);
    Serialize(pRoot->m_pRight,stream);
}

void CTree::DeSerialize(BinaryTreeNode** pRoot, istream& stream)
{
    int number;
    if(ReadStream(stream, &number))
    {
        *pRoot = new BinaryTreeNode();
        (*pRoot)->m_nValue = number;
        (*pRoot)->m_pLeft = nullptr;
        (*pRoot)->m_pRight = nullptr;

        DeSerialize(&(*pRoot)->m_pLeft,stream);
        DeSerialize(&(*pRoot)->m_pRight,stream);
    }
}

bool CTree::ReadStream(istream& stream, int* number)
{
    if(stream.eof())
```

```
        return false;

    char buffer[32];
    buffer[0] = '\0';

    char ch;
    stream >> ch;
    int i = 0;
    while(!stream.eof() && ch != ',')
    {
        buffer[i++] = ch;
        stream >> ch;
    }
    bool isNumber = false;
    if(i>0 && buffer[0] != '$')
    {
        *number = atof(buffer);
        isNumber = true;
    }
    return isNumber;
}
```

寻找数组中第K大的数据

参考无序整数数组中找第k大的数
利用快速排序的思想。从数组中随机挑选一个数S,把数组分成大于S的数组SL，以及小于等于S的数组SR,第一次时间复杂度为N，如果SL数组大小大于K，则要找的数据在SL中，否则在SR中。假设在SL中，我们再一次分割数组，这一次平均时间复杂度为N/2，如果有第三次划分，怎第三次平均时间复杂度是N/4,这样下来总的时间是2N,因此时间复杂度是O(N).

寻找最小的K个数

利用堆排序来实现时间复杂度为O(Nlog(K+1))的算法。建立大小为K的堆，则查找的时间复杂度是O(K+1)。
一个问题是怎么维护一个最大堆？最大堆可以在O(1)时间内找到最大值，在O(log K)时间内插入。
下面实现利用了快速排序中的二分法。

```
void CCArray::SmallestKInList(int* numbers, int length, int K)
{
    if(numbers == NULL || length <= 0)
        return;

    int middle = length >> 1;
    int start = 0;
    int end = length -1;
    int index = 0;
```

```
    CSort* sort = new CSort();
    while(end > K-1)
    {
        index = sort->Partition(numbers,length,start,end);
        if(index == K-1)
            break;
        else if(index > K -1)
            end = index -1;
        else
            start = index + 1;
    }
    delete sort;
    return;
}
```

寻找频率超过半数的数

实际上就是中位数

```
int CCArray::MoreThanHalfNumSort(int* numbers, int length)
{
    if(numbers == NULL || length <= 0)
        return 0;

    int middle = length >> 1;
    int start = 0;
    int end = length -1;
    int index = 0;
    CSort* sort = new CSort();

    while(start < end)
    {
        index = sort->Partition(numbers,length,start,end);
        if(index == middle)
            return numbers[middle];
        else if(index > middle)
            end = index - 1;
        else
            start = index + 1;
    }
    return numbers[middle];
}
```

连续子数组最大和。

输入译者整型数组，其中数组里面有正数也有负数，数组中一个或者连续的多个数组组成一个子数组。求所有子数组的和。
通过定义一个中间变量，子数组以i为终点的连续子数组的最大和f[i]，则我们需要求的连续子数组

最大和也就是f[i]在i=1,...,n中的最大值。因为f[i]可以用如下递归公式简单的实现。因为问题可以简单的求解。

从1到整数n中1出现的次数。

思路:通过最高是1还是大于1来操作。以3154为例 如果最高位是3大于1,我们先计算1-3000之间1的数目。再计算3001-3154.

1-3000之间1的数目是千位含有1000-1999有10^3,还有当千位为0，1，2时，个位，十位，百位随机挑选一个为1，其它两位任意，一共有firstNum*(lenght-1)*pow(10,length-2) 个组合；
$firstNum > 1, numberOf1 = (firstNum) * (length - 1) * pow(10, length - 2) + pow(10, lengt$
$firstNum == 1, numberOf1 = (length - 1) * pow(10, length - 2) + residuleNum + 1;$
此时，高位为1，则余数+1个数字最高位含1.再加上低位含有1的情况，也就是有length-1位可选为1，其它lenght-2位任意。再把余数residuleNum 返回继续计算，因为residuleNum 相当于不计算最高位为1之后，从pow(10,length -1)到n含1的个数，也就是1-residuleNum含1的数目。

```
int CMath::NumberOf1InN(int n)
{
    if(n<10)
    {
        if(n >= 1)
            return 1;
        else
            return 0;
    }
    char str[32];
    sprintf(str,"%d",n);
```

```
    int length = strlen(str);
    int firstNum = str[0] - '0';
    int residuleNum = n%(pow(10,length-1));
    int numberOf1 = 0;
    if(firstNum > 1)
    {
        numberOf1 = (firstNum)*(length - 1)*pow(10,length-2) + pow(10,length-1);
    }
    else
    {
        numberOf1 = (length - 1)*pow(10,length-2) + residuleNum + 1;
    }
    return numberOf1 + NumberOf1InN(residuleNum);
}

int CMath::pow(int base, int n)
{
    int result = 1;
    if(n==0)
        return 1;
    if(n<0)
        return 0;
    for(int i = 0; i < n; i++)
    {
        result *= base;
    }
    return result;
}
```

把数组排成最小的数

输入一个正整数数组，把它拼接起来排成一个数，使得这个数最小。比如输入的是{3,32,321}则拼成的最小数是321323.
其实就是字典排序，把这些数字看成字符，然后像字典一样再排序，排在前面的放在前面来保证拼接的数最小。
通过调用STL algorithm中的qsort来对字符排序，自己定义compare函数，实际也是调用strcmp来排序。

```
int compare(const void* strNumber1, const void* strNumber2);
int g_MaxNumberLength = 10;
char* g_StrCombine1 = new char[g_MaxNumberLength * 2 + 1];
char* g_StrCombine2 = new char[g_MaxNumberLength * 2 + 1];

void CMath::PrintMinNUmber(int* numbers, int length)
{
    char** strNumbers = (char**)(new int[length]);
    for(int i = 0; i< length; i++)
    {
        strNumbers[i] = new char[g_MaxNumberLength];
```

```
        sprintf(strNumbers[i], "%d",numbers[i]);
    }
    qsort(strNumbers, length, sizeof(char*),compare);
    for(int i = 0; i< length; i++)
    {
        printf("%s",strNumbers[i]);
    }
    printf("%\n");
    for(int i= 0; i < length; i++)
        delete[] strNumbers[i];
    delete[] strNumbers;
}
int compare(const void* strNumber1, const void* strNumber2)
{
    strcpy(g_StrCombine1,*(const char**)strNumber1);
    strcpy(g_StrCombine2,*(const char**)strNumber2);
    return strcmp(g_StrCombine1,g_StrCombine2);
}
```

结束