# 执行力

# 如何与客户沟通

# 书到用时方恨少

学的时候总抱怨方法、原理太多，真正用的时候发现一个也用不上。学的时候总感觉能做好大的项目，用的时候一个特征汇总就搞掉你。

介绍一些架构。

# 不平衡分类

# 有限总体稀有事件抽样

# 做事要做漂亮

在规定时间内做事，宁愿做的不要那么多也要那么精。

# 数据归一化对算法的影响

1. 一些算法天然的要求数据量纲一致：kmeans、knn、pca等
2. 除此之外，数据归一化还能够影响梯度下降等最优化方法的收敛。如神经网络在进行训练之前经常需要对量纲差异较大的数据进行归一化。

# 做一些项目的基本点

1）产品设计

对分布式来说，产品设计往往是其短板，由于其数据处理过程都是基于rdd的因此，在中间交互、结果展示等方面都受限

2）数据处理

3）展示

# 对于不定参数的对象

在应用中有些参数需要不定长的参数,比如有的需要列名/时间/分箱,有的需要列名/时间就可以.

此时两种可以解决上述需求:

1)将类写为class A(private var参数1: T){以set形式传递参数}的形式

2)类正常写为class A(val参数1: Option[T]）,以Option包括参数，直接apply

# 常用的工程项目目录结构和理念

## 理念

低内聚高耦合

## 目录方式1

Utils

Test

Models

Objects

# 为什么很多团队都强调结果导向

具备较强沟通能力，善于团队合作，交付能力强，自我驱动，结果导向。

不要过程导向或情感导向，只有结果很差，没有功劳时才谈苦劳。

1. 注重最终效果
2. 考量的内容首先是需求，其次才是实现。

# 产品设计

在设计的基本想法有了的时候，可以尝试用简短的几句话看看能不能把框架说清楚，如果不行再改一下。

# 对于只有很少数目范围的参数可以放在某个Object类中声明

以case class(fieldFormat: String, …)

fieldFormat可以为string、long等类型。

1. 一种方法是把“string”、“long”做成独立的类型，共同继承一个父类型。
2. 另一种是在case class实例化之前加入判定条件，fieldFormat必须在某个范围（“string”、“long”）内
3. 可以做另一个object

Object A extends Serializable{

Val firstType = “string”

Val secondType = “long”

}

Case class实例化时注意只输入A中的某个变量

# 需求管理

用的需求>开发需求>设计需求

设计时考虑的东西是少的，随着开发过程不断的思考，不断改正增加需求，在用的过程中又不断更新开发阶段的一些bug并提出一些新的设计

# 专业性

到一个地方应该注重显示自己的专业性，规范性，有其他人不合规范，即使略有棱角也要显示出来。虽然一开始可能还有点龃龉但后面他很快会正视你。

# 框架设计的一些注意事项

1. 里层的不能改变外层的变量

否则乱了。外层稍微一边动，里层的某个就不能运行了。牵一发动全身。最主要的是有时候忘记里层的某个组件改动外层的东西了，没有发现bug就是隐患，发现了bug也不好排查。

1. 高内聚，低耦合

目的也是和1一致，防止某个模块的稍一变动会导致其他模块跟着大的改动。

模块内可以大改，高内聚以充分利用代码和功能，避免一个脚本一个功能，而是要做成组合式的。

1. 模块划分要恰当

不要过大，会导致大的范围内内聚，牵一发动全身。

不要过小，会导致模块过多，耦合过多。

# 千万别不小心被Excel自动填充的功能坑掉

Excel会自动进行数字的递增填充（向下递增，向上递减）。不小心就会出现如下情况：

Jw2c

Jw3c

Jw4c

在工作中一定要注意这种细节，否则给人一种不靠谱、不认真的印象。

# 与客户沟通的一条思路

注重客户需要什么

思考我们现有产品能给客户提供什么？

我们的产品可以做什么样的改进从而更大的满足客户需求？

那些需求是我们做不到的或做不好的。

关键一条就是寻找交集。在交集的基础上再谈问题。

# 没做好或做错了不要扭

不要潜意识里认为自己对了.

今天领导请吃饭,但人事没有通知到我,因为她没发消息而是口头通知,但当时我不在,中午我回家吃饭,结果领导来了又被叫回来.她说她通知了,结果又在群里说了一声”再次通知一下”,明明群里没有消息记录.

我以前更是这样,反思一下,没错的时候有主张要表达,错了要用于承认.

# 让别人清晰的知道你的边界在哪里

# 不要说一大堆,别人不听的

# 注意Exception不要下太多的断言

否则可能本来不是这样导致的异常，你再Exception中说是。像：

**java.lang.Exception: 您输入的数据列numbers不能转为double类型，cannot resolve 'numbers' given input columns features, notEqualLengthVector, equalDense;**

就比较尴尬。

比较好的方式是，在处理“xxx”中出现了异常，具体信息为：xxxx

# 初值选取

有些机器学习的模型，要求最好初值选取接近数据的分布方向——比如SOM。此时如何选取初值成为一个问题。

# 有个在线绘制流程图的工具——processOn

不错，适合远程交流和现场交流

# 一个想到的开发流程

1）拿到需求 -> 写文档 -> 试开发 -> 沟通需求 + 修改文档 -> 沟通需求决定什么能改什么不能改，什么重要什么不重要，什么能做好什么可能有疑问 + 决定开发框架

2）开发 –> 疑问 -> 沟通 + 解决 –> 开发 –> 测试

3）需求修改 -> 判断大小 -> 开发 -> 测试

4）发布

# 关于平台资源调度的优化

这是今天CTO发的他关于平台的优化，基本听不懂，说明这正是欠缺的：

11.34上的Spark2.0版引擎已升级为1.2版，主要更使用Ice重新实现了引擎总线，包括：  
1.使用Ice总线替换了原有Thfit总线；  
2.使用全**异步**交互模式，替换了原有**同步**模式；  
3.在事件交互上，使用**Push模式**替换了**Pull模式**，实时性更好；  
4.在模块交互上使用**Callback模式**替换了原有**等待模式**；  
5.使用**网络连接共享**、**主动连接池管理**、**动态线程池管理**；  
6.可配置使用高压缩比，以加快数据交互；  
7.所有关键资源可配置；  
8.容错机制得到完善和优化；  
9.更改了部份Bug;  
优点是：平台自身性能更高，并发性更好，吞吐量成倍增长（是以前的3倍）；系统印记更小，更为稳定。

# 拓扑六边形

游戏中常用：

每个六边形有六个最近的六边形，每个顶点有三个最近的顶点。

六边形的数据结构方法：

1. 标记坐标 => 由标记坐标计算像素坐标

六边形的寻路算法

## Dijkstra算法与最佳优先搜索

参考文献：https://blog.csdn.net/yulijunzj/article/details/18139085

## A\*算法

参考文献：https://blog.csdn.net/yulijunzj/article/details/18139085

# 启发式算法

参考文献：https://blog.csdn.net/yulijunzj/article/details/18139085

# 一些公开资料的时候考虑系统版本

比如top4fun的竞赛数据就有了mac和window两个版本，不错，很贴心，这才是做实事的人。

# 对于数据挖掘还是R社区和Python做的好一些

1. 它们方法更全而且实现的更好
2. 它们社区的应用非常多
3. Spark相对来说要弱很多，spark在性能等方面要强于R，而且能够处理别人处理不了的数据量，但对于方法探究和模式寻找阶段来说spark做的还是很不细，API和应用太少了。
4. Spark在基本统计方面和两者是差不多的，但复杂的模式探究上，建议将大的数据筛选为多个合适的数据，通过R或python进行模式探究，在并行化部分算法后通过spark实现。

# 准动态处理

1. 已经通过文件读写+基于数据库的文件信息管理，实现的一个版本是

mbtj项目中的动态统计部分。

1. 通过streaming实现

Streaming可以读取目录中晚于当前streaming时间戳的数据

1. 通过saveAsTable+基于数据库的文件信息管理，实现

这是一个parquet形式存储的，而且可以append、overwrite、errorIfexists等选项

但parquet文件写起来比txt慢很多，虽然读起来很快、占用空间更小（约六七分之一）

可以对比下性能：

# 写一些工具类时门槛要低

写过判定列名是否存在和列名类型是否一致的工具类，一个是基于列名+表+类型名称的判定方式，另一个是基于DataType+类型名称的判定方式，虽然第二个用到的方法比较高端，但是我还是喜欢用第一个，因为门槛低，因为获得DataType还需要一行代码。

1）Tools.*columnTypesIn*(name, rawDataFrame, *Array*("string", "double", "int", "float", "long"), **true**)

2）*val dt = df.schema("id").dataType  
\* import DataTypeImpl.\_  
\* dt in Seq("string", "int", "decimal(20, 0)")*

这说明并不是形式越高级越好用，而是门槛和附加信息越少越好用

不过后者功能更强大起始（因为里面加入了更多的元素，这个和设计无关了）