# 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）*valdt = 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）

# 任务计划 = 任务预估时间 \* 3左右

和领导要工作千万不要预估多少就多少，把问题深入思考一下，多要些，降低他的预期。

预估7天最后5天完成，和预估3天最后5天完成，虽然是同一件事但在领导那里这两个效果是截然相反的。

不过要想要理由。

# 做探索性的和复杂性的任务要以进度为导向

1. 避免最后一无所获，老板不接受
2. 开始前尽量声明这是一个探索性的任务，技术上是有风险的，目前市面上有的框架或技术暂时做不到兼容\*\*\*\*或同时既保证\*\*\*又保证\*\*\*\*
3. 比如要做A，我可以先做A----，报告一下进度，再做A---，因为老板可能随时监督
4. 最后即使探索，发现这条路不行，至少做了A---，同时尽可能的将积累的技术

# 老手和新手的差异可能很大部分是动作快慢

也就是把想法变为做法的过程。

# 一些机器学习需要了解的知识

作为数据建模的人员我们需要什么：

1. 了解我们所用到的算法的大致原理，适用于什么场景，它们识别什么模式，不能识别什么模式
2. 我们要做的工作最大的部分不是调参，而是特征提取和误差分析。特征提取：如何对生产环境中产生的数据进行合理的筛选、变换，做成有效的性状喂给算法。误差分析：如何分析算法是否恰当，是否达到了最优，如果出现不理想的情况造成的原因可能是什么，应该怎样调整什么（特征提取过程，更换算法，调节参数）

# Loop时判定条件的位置决定是至少执行一次还是可能不执行

While(判定条件){

}

// 可能不执行

Var a = true

While(a){

A=判定条件

}

// 至少执行一次

# 多和工作上的人沟通

1. 自己工作内容有变动的需要及时和相关交接的人沟通，公司有个平台维护同事，平台的机器换了都不说一声，自己做自己的，我弄起费了半天劲，发现平台换了。
2. 如果发现对方有问题，及时沟通，有个测试同事，一个很简单的问题，只是理解上不一致，reopen了好几个bug。沟通下理解不就好了，没发现问题吗？

# Scala中如果频繁涉及到多个项的计算，尽量用case class而不用Tuple

这样代码易维护一些，否则时间长了一会tup.\_3一会tup.\_5的记不清，而且涉及到两个以上tup合并的时候更加复杂。

# 培训、产品设计、教学等一定要化繁为简

1）切记一大堆，以教会别认为目的

2）口齿清晰、语速慢、有重点

3）从目的出发，从功能出发，不要直接将规则或使用方式

4）不要高估使用者的耐心，一分钟学不会就不学了

5）交流

现在公司的培训就是这样，一堆程序员，本来最容易接受这种使用方式，培训都培训不好。

# 做项目的节奏把握

1）揣摩需求，有不太明白的点一定要沟通，除非需求方本身也不太清楚

2）查看数据、环境等，了解做项目的环境，对象，材料

3）大框架设计要粗，但是要审慎

4）对大框架的技术难点进行评估，如果不能在时间、资源的限制条件下做好就调整

5）具体实施一定要快，先做出效果

6）一层一层优化和功能细化

7）测试

# 长期项目或者重复性工作可以写一个工具类的包

对于新加入一个公司或者加入一个较长期的项目组，可以一开始就构建一个工具类的包，原则是只被应用而不引用项目文件，工具类如果是函数式编程最好能够直接提供异常报错。

另外长期从事数据处理可以建一个自己的工程，内置一些写好的接口（可用性和通用性要高）和工具类，不过要分语言建立。目录可以如下：

utilStack

r

python

scala

java

c

等等

# 短期项目快速实现，迅速落地，然后才是调节优化，做到敏捷开发

做短期项目一定要注意，要快速实现一版，注意这里实现并不是探路，而是可以落地的，以该版本为底版继续进行分析。

切忌一上来就按部就班的开发一小块一小块的细节，琢磨细节的时间越多，进行真正流程开发测试的时间越少，后期一旦开发条件变了（平台不行，断网了等等），或者突然有意想不到的bug，就没法按期实现。一定要迅速落地。

然后保有该版本，进行发散和优化。

一个经验教训就是：在为期5天的招标过程中，我拖到验收前半小时才做完，几乎拖了团队的后腿，而且这个我是有准备的。

# 增量执行流程

## 测试的增量数据源制作

1.数据列表生成

1）输入一个唯一的id；

2）将数据源中的数据按一定方式分成若干份增量，依次存入到一个列表中，列表中的数据需要时同一个schema；

3）读取时通过增量数据源读取，读取需要第一步中的标识；

2.增量数据源队列化

1）输入增量数据生成中的唯一的id，读取数据中的列表并生成队列；

2）每个增量数据源需要一个唯一的标识，将队列以该标识输出；

3）开放一个选项，当没有新数据时报不报错，如果不报错需要做一个空的df

3.增量数据源读取

1）输入增量数据生成中的唯一的id，读取数据中的列表并生成队列；

2）每个增量数据源需要一个唯一的标识，将队列以该标识输出；

3）开放一个选项，当没有新数据时报不报错，如果不报错需要做一个空的df

## 处理过程

4.增量数据处理

要支持空数据的处理（如果2中“没有新数据时报错”这一步可以不必支持）

## 新旧结果合并

5.增量结果读取

1）输入增量结果存储（6）的id号，根据结果获取id号，如果没有就输出None，如果有就输入Some(DataFrame)；

5.新旧结果合并

获取Some

## 结果存储

6.增量结果存储

1）输入一个唯一的id号

2）将结果输出

# 一个数据挖掘和可视化框架

orange

# junit和单元测试

测试驱动开发

除了单元测试之外，还有功能测试和系统测试等粒度较粗的测试。

学习的时候可以参考junit5和scalaTest以及github spark自带的测试文件

# apache poi文件读写的api

另外aspose

# 部署管理

持续集成(CI):Jenkins

# Datalore

https://datalore.io/

jetBrain的一个在线数据挖掘平台。

## 基本处理函数工具化的一些思考

1）功能模块化：将功能拆分成多个模块，这个不用多说；

2）基本处理函数工具化

有些功能是由一些基本函数组成的，比如时间分箱，每次用到时间分箱功能的时候都要写一个，比较麻烦。如果将Timestamp => Timestamp形式的分箱，写成一个工具类，让共同调用，其他只要各自实现数据类型转换接口就行了。

这样做的好处：

1. 代码复用
2. 可以确保代码最新，不会改了一个时间分箱算法，其他也要改，有些地方可能根本记不住
3. 可以使用一个单元测试管控算法。只要保证时间分箱算法的单元测试能跑就可以确保时间分箱算法正确。其他不用管了。
4. 综上：省时、省心、省力、还保证正确。