系统架构

一、Hidden Technical Debt[2015]

1. 随着机器学习 machine learning: ML 社区在实时系统 live systems 方面不断积累多年的经验，出现了一种广泛而又令人不安的趋势：开发和部署机器学习系统相对快速 fast 且廉价 cheap，但是随着时间的推移维护机器学习系统既困难 difficult 又昂贵 expensive 。

   这种二分法 dichotomy 可以通过技术债 technical debt 的角度来理解。技术债是 Ward Cunningham 在 1992 年引入的一个比喻，它可以帮助你推断软件工程的快速发展所产生的长期代价。和财政债务一样，背负技术债通常有合理的战略原因。并非所有的债务都是坏的，但是所有的债务都需要偿还。

   技术债可以通过重构代码、改进单元测试、删除死代码 dead code 、减少依赖、收紧 API 、改进文档来偿还。偿还的目的不是添加新功能，而是支持未来的改进、减少错误、以及提高可维护性 maintainability 。推迟此类偿还会导致复合成本 compounding costs 。隐藏的债务 hidden debt 是危险的，因为它会悄无声息地复合 compounds (即利滚利，债务的代价越来越大) 。

   在论文 《Hidden Technical Debt in Machine Learning Systems》 中，作者探讨了在系统设计中需要考虑的、几种特定于机器学习的风险因素，其中包括：边界侵蚀 boundary erosion 、纠缠 entanglement 、隐式反馈环、未声明的消费者、数据依赖、配置问题、外部世界 external world 变化 、以及各种系统级的反模式 anti-patterns 。

2. 在论文中，作者认为机器学习系统有一种招致技术债的特殊能力，因为它们具有传统代码的所有维护问题、以及一组额外的 ML-specific 问题。这种技术债可能难以检测，因为它存在于 system-level 而不是 code-level 。传统的抽象和边界可能会被数据影响机器学习系统行为的事实所破坏或失效。偿还 code-level 技术债的经典方法不足以在 system-level 解决 ML-specifc 技术债。

   本文不提供新颖的 ML 算法，而是试图提高社区对长期实践中必须考虑的困难的 tradeoffs 的认知。我们关注 system-level 交互和接口，这是机器学习技术债可能迅速积累的领域。在 system-level ，机器学习模型可能会悄悄腐蚀抽象边界。输入信号的 re-use 或 chaining 可能会无意中耦合原本不相干的系统。机器学习 packages 可能被视为黑盒子，导致大量的 “胶水代码” 或校准层 calibration layer 并锁死在某些假设中。外部世界的变化可能会以意想不到的方式影响系统行为。如果没有精心的设计，即使监控机器学习系统行为可能也很困难。

3. 机器学习为快速构建有用的、复杂的预测系统提供了一个非常强大的工具包 toolkit 。但是，将这些快速的胜利 quick wins 视为免费是危险的。我们发现在现实世界的机器学习系统中招致大量的持续维护成本 maintenance costs 是很常见的。

1.1 侵蚀的边界

1. 传统的软件工程实践表明：使用封装和模块化设计的强大抽象边界有助于创建可维护 maintainable 的代码。在这些代码中，很容易进行孤立 isolated 的变更 changes 和改进 improvements 。严格的抽象边界有助于表达来自给定组件 component 关于信息输入和输出的不变性 invariants 、以及逻辑一致性 logical consistency 。

   不幸的是，很难通过规定特定的预期行为来为机器学习系统强制执行严格的抽象边界。事实上，机器学习系统的行为依赖于外部数据。在这里，我们研究了导致边界侵蚀 erosion of boundaries 的几种方式，它们可能会显著增加机器学习系统中技术债。

2. 纠缠 Entanglement：机器学习系统将信号混合在一起，将它们相互纠缠，并使得孤立的改进变得不可能。

   例如，考虑一个在模型中使用特征 $x_1, \cdots, x_n$ 的系统。如果我们改变了 x_1 中取值的分布，则剩余 $n-1$ 个特征的重要性 、权重 、以及利用 use 都可能发生变化。无论模型是以 batch 方式完全重新训练 、还是允许模型以在线方式进行调整，都是如此。

   添加一个新的特征 x_{n+1} 会导致类似的变化，就像删除任何特征 x_j 一样。没有任何输入是真正独立 independent 的。我们在此将其称作 CACE 原则：改变任何事情则改变一切 Changing Anything Changes Everything 。 CACE 不仅适用于输入信号，还适用于超参数、学习配置、采样方法、收敛阈值、数据选择、以及基本上所有其它可能的调整。

   有两种缓解策略：

   - 一种可能的缓解策略是隔离 isolate 模型并集成 ensembles 服务 。然而很多情况下集成模型效果很好，是因为各个子模型的误差是不相关的。依赖于这种组合 combination 会产生强烈的纠缠：如果各个子模型的误差是强相关的，那么改进单个子模型实际上可能会使得系统准确性变差。
   - 第二种可能的策略是聚焦于检测在预测行为上发生的变化。 《Ad click prediction: a view from the trenches》 中提出了一种这样的方法，其中使用高维可视化工具让研究人员能够迅速查看跨多个维度和切片 slicings 上的效果。在逐个切片的基础上运行的指标也可能非常有用。

3. 校正级联 Correction Cascades：经常存在这类场景：问题 A 的模型 $\mathcal{M}_a$ 已经存在，但是需要求解一个稍微不同的问题 A' 。在这种情况下，学习以 $\mathcal{M}_a$ 作为输入的 $\mathcal{M}'_a$ 模型、并学习一个小的校正 correction 从而作为解决问题的快速方法，这种思路是很有诱惑力的。

   然而，这种校正模型 correction models 产生了一个新的系统依赖关系，这使得分析未来对该模型的改进变得更加昂贵：校正级联可能会创建一个 improvement deadlock ，因为提高任何单个子模型的准确性实际上会导致系统级别的损害。

   缓解策略是通过添加特征来区分不同的 cases ，强化 augment $\mathcal{M}_a$ 以直接在同一个模型中学习校正。或者为 A' 创建独立的模型。

4. 未声明的消费者 Undeclared Consumers：通常，来自机器学习模型 $\mathcal{M}_a$ 的预测可以被广泛访问。如果没有访问控制，那么其中一些消费者可能是未声明 undeclared 的，它们悄无声息地将模型的输出作为它们系统的输入。在经典的软件工程中，这些问题被称为可见性债务 visibility debt 。

   未声明的消费者在最好的情况下是昂贵 expensive 的、在最坏的情况下是危险 dangerous 的，因为它们创造了模型 $\mathcal{M}_a$ 到工作流其它部分的、隐式的紧密耦合。$\mathcal{M}_a$ 的变化可能会影响这些部分，可能会以意想不到 的、知之甚少的 、以及有害的方式影响。在实践中，这种紧密耦合会从根本上增加对 $\mathcal{M}_a$ 进行任何变更 changes 的成本 cost 和难度 difficulty ，即使这些变更都是改进 improvements 。此外，未声明的消费者可能会创建隐式反馈环，这在后文有更详细的描述。

   除非系统专门设计用于防范此类情况，例如具有访问限制或严格的服务级别协议，否则可能难以检测到未声明的消费者。

   > 纠缠 Entanglement：模型内部各元素之间的耦合。
   >
   > 校正级联 Correction Cascades：多个模型之间的耦合。
   >
   > 未声明的消费者 Undeclared Consumers：预测系统和消费系统之间的耦合。

1.2 数据依赖

1. 依赖性债务`dependency debt`被认为是经典软件工程环境中代码复杂性和技术债的关键因素。我们发现机器学习系统中的数据依赖性具有类似的建立债务的能力，但是可能更加难以检测。代码依赖性可以通过编译器和链接器的静态分析来识别。如果没有类似的数据依赖检测工具，那么很容易构建难以解开的大型数据依赖链。

2. 不稳定的数据依赖`Unstable Data Dependencies`：为了快速推进，通常可以方便地将其它系统产生的信号作为输入特征来使用。然而，一些输入信号是不稳定的，这意味着它们会随着时间的推移而定性`qualitatively`的或者定量`quantitatively`的改变行为。

   - 当输入信号来自另一个随时间更新的机器学习模型本身，或者来自一个数据依赖的`lookup table`（例如计算的`TF/IDF score`或者语义映射`semantic mappings`）时，这可能会隐式地发生。
   - 当输入信号的工程所有权`engineering ownership`和消费信号的模型的工程所有权分离时，这可能会显式地发生。

   在这些情况下，输入信号可以随时更新`update`。这是危险的，因为即使是对输入信号的`improvements`也可能对消费系统产生任意的有害影响，而这些影响的诊断和解决成本很高。

   例如，考虑输入信号之前被错误校准`mis-calibrations`的情况。消费输入信号的模型可能拟合了这些错误校准，而悄无声息的、纠正信号的更新`update`将对模型产生突然的影响。

   不稳定数据依赖的一种常见缓解策略是创建给定信号的版本化副本`versioned copy`。例如，与其允许单词`words`到主题簇`topic clusters`的语义映射随时间变化，不如创建并使用该映射的冻结版本`frozen version`，直到更新`updated`的版本经过全面审核为止。

   然而，版本控制有其自身的成本，例如潜在的过时性`staleness`以及随着时间的推移需要维护同一个信号的多个版本的成本。

3. 未利用的数据依赖`Underutilized Data Dependencies`：在代码中，未利用的依赖是大多数不需要的`packages`。类似地，未利用的数据依赖是对模型几乎没有收益的输入信号。这些信号会使得机器学习系统变得不必要的脆弱（容易受到变更的影响），有时甚至是灾难性的，即使它们可以被移除而没有损害。

   例如，假设为了简化从旧的产品编号方案到新的产品编号方案之间的转换，两种方案都作为特征保留在系统中。新的产品仅有一个新编号，但是旧的产品可能新编号、旧编号两者都有，并且模型继续依赖于某些产品的旧编号。一年以后，所有的产品就只有新编号。这对于机器学习系统的维护者来说，情况不妙。

   > 例如某公司两个部门之间的产品线拥有各自独立的类目编码体系，后来部门融合，两套类目编码体系需要迁移为统一的、新的类目编码体系。

   未利用的数据依赖可以通过几种方式蔓延到模型中。

   - 遗留的特征`Legacy Features`：最常见的情况是，特征`F`在模型开发的早期就包含在模型中。随着时间的推移，新特征使得`F`变得冗余，但是这一情况未被发现。
   - 捆绑的特征`Bundled Features`：有时，我们评估一组特征并发现它们是有益的。由于`deadline`的压力或者类似的影响，这一个捆绑`bundle`中的所有特征一起被添加到模型中，可能包括没有任何价值的特征。

     > 由于工期进度有限，我们没有精力逐一区分这组特征里面哪些是有价值的、哪些是没有价值的。

   - ϵ-Features：作为机器学习研究人员，即使在准确性增益`accuracy gain`非常小、或者复杂性开销可能很高的情况下，我们也会去尝试提升模型准确性。这类特征我们称作 ϵ 特征。
   - 相关的特征`Correlated Features`：两个特征是强相关的，但是其中一个是另一个更直接的因果关系`causal`。很多机器学习方法很难检测到这一点，并且对这两个特征同等地看待，或者甚至可能选择其中的非因果`non-causal`特征。如果全局行为`world behavior`后来改变了这种相关性`correlations`，这会导致模型的脆弱性`brittleness`。

可以通过全面的 `leave-one-feature-out` 评估来检测未利用的数据依赖。这应该定期运行，从而识别和删除不必要的特征。

4. 数据依赖的静态分析 `Static Analysis of Data Dependencies`：在传统代码中，编译器和构建系统 `build systems` 对依赖图 `dependency graphs` 进行静态分析。静态分析数据依赖的工具并不常见，但是这些工具对于错误检查、追踪消费者、以及强制迁移和更新是必不可少的。其中一种工具是 `《Ad click prediction: a view from the trenches》` 中描述的自动化特征管理系统。该系统可以对数据源和特征进行注解 `annotated`。然后可以运行自动检查从而确保所有依赖都具有适当的注解，并且可以完全解析依赖树 `dependency trees` 。这种工具可以使迁移 `migration` 和删除 `deletion` 在实践中更加安全。

1.3 反馈环

1. 实时机器学习系统的一个关键特征是：如果系统随着时间的推移而更新，它们通常最终会影响自己的行为。这导致了一种分析债务的形式 `form`，其中很难在给定模型发布之前预测模型的行为。

   这些反馈环 `feedback loop` 可以有不同的形式，但是如果它们随着时间缓慢发生（例如在模型更新不是很频繁时），那么反馈环很难检测和解决。

2. 直接反馈环 `Direct Feedback Loops`：模型可能会直接影响它自己未来训练数据的选择（例如推荐算法）。通常在实践中使用标准的监督学习算法，但是理论上正确的解决方案是使用老虎机算法 `bandit algorithms` 。这里的问题是老虎机算法（例如上下文老虎机 `contextual bandits`）不一定能够很好地适应 `scale well` 现实世界问题通常所需的动作空间 `action spaces` 的大小。

   可以通过使用一定量的随机化、或者通过隔离数据的某些部分不受给定模型的影响，从而缓解这些影响。

3. 隐式反馈环 `Hidden Feedback Loops`：直接反馈环的分析成本很高，但是至少机器学习研究人员可能会自然发现并进行研究。一种更困难的情况是隐式反馈环，其中两个系统通过真实世界间接地相互影响。

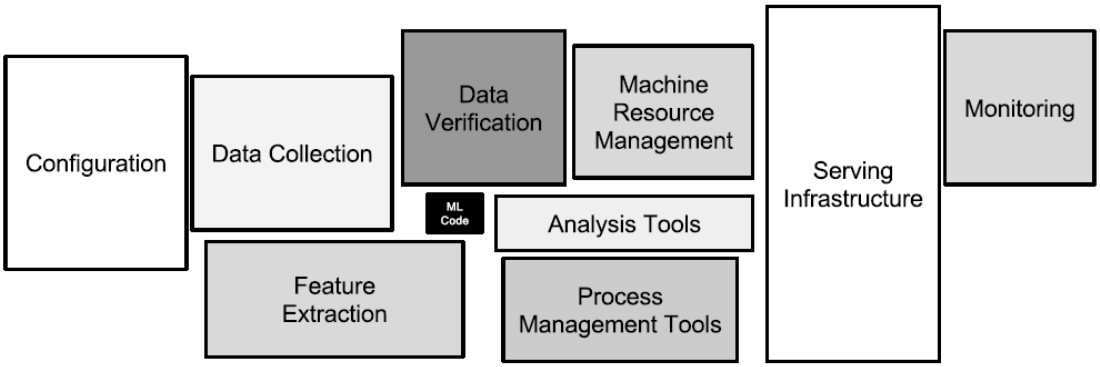
   这方面的一个例子是，如果两个系统独立地确定网页的各个方面 `facets`，例如一个系统选择要展示的产品、另一个系统选择相关的评论。改进一个系统可能会导致另一个系统的行为发生变化，因为用户开始或多或少地点击另一个部分从而响应这些变化。

   注意，这些隐式反馈可能存在于完全不相交 `disjoint` 的系统之间。考虑来自两家不同投资公司的两个股市预测模型的案例。一方的改进 `improvements` （或者更可怕的 `bugs` ）可能会影响另一方的出价 `bidding` 和购买 `buying` 行为。

1.4 反模式

1. 学术界可能会惊讶地发现：很多机器学习系统只有一小部分代码实际上专门用于学习或预测，如下图中间部分的黑框所示。其余的周边基础设施庞大而复杂，这部分可以描述为管道 `plumbing` 。

   不幸的是，结合了机器学习方法的系统经常以高负债 `high-debt` 的设计模式而告终。在这里，我们研究了几种可能出现在机器学习系统中的系统设计反模式 `system-design anti-patterns`，应该尽可能避免、或者重构它们。



2. 胶水代码`Glue Code`：机器学习研究人员倾向于将通用解决方案开发为独立包`self-contained packages`。在像`mloss.org`这样的地方，或者内部代码、专有包、云平台这些地方，有各种各样的开源包。

   使用通用包通常会导致胶水代码的系统设计模式，其中编写了大量的支持代码来将数据导入和导出通用包。从长远来看，胶水代码的成本很高，因为它倾向于将系统冻结到一个特定包的特性上，测试替代品可能会变得非常昂贵。通过这种方式，使用一个通用包可能会抑制改进`improvements`，因为它使得利用领域特定的属性、或者调整目标函数来实现领域特定的目标变得更加困难。

   因为一个成熟的系统最终可能是（最多）`5%`的机器学习代码和（至少）`95%`的胶水代码，所以创建一个干净的原生解决方案可能比重用一个通用包的成本更低。

   对抗胶水代码的一个重要策略是将黑盒`black-box`封装到通用`API`中。这使得支撑的基础设施的可重用性`reusable`更高，并降低了包变更的成本。

3. 管道丛林`Pipeline Jungles`：作为胶水代码的一个特例，管道丛林经常出现在数据准备阶段。随着新信号的识别和新信息源的逐渐加入，这些可以有机地发展`evolve organically`。

   如果不小心地维护，以机器学习友好格式而准备数据的系统，最终会变为一个由抓取、连接、以及采样等步骤组成的、带中间文件输出的丛林。管理这些管道、错误检测、以及从故障中恢复都是困难和昂贵的。测试这类管道通常需要昂贵的端到端的集成测试。所有这些都增加了系统的技术债，并使得进一步创新的代价更高。

   只能通过全面考虑数据收集和特征提取来避免管道丛林。废弃管道丛林并从头开始重新设计的全新方法确实是一项重大的工程投资，但是它可以显著降低系统前进的成本并加快进一步的创新。

   胶水代码和管道丛林是集成问题`integration issues`的征兆，其根本原因可能是过度分离的“研究”角色和“工程”角色。当机器学习`packages`在象牙塔环境中研发时，最终结果对于在实践中使用它们的团队而言可能看起来像是黑盒。

   将工程师和研究人员嵌入到同一个团队（实际上，通常是一个人）的混合研究方法`hybrid research approach`可以帮助显著减少这种摩擦来源`friction source`。

4. 死亡的实验代码路径`Dead Experimental Codepaths`：胶水代码和管道丛林的一个常见结果是：通过将实验代码路径实现为主力生产代码中的条件分支，在短期内内使用替代方法进行实验变得越来越有吸引力。

   对于任何单个变更，以这种方式进行实验的成本相对较低，因为周围的基础设施都不需要重新开发。然而，随着时间的推移，这些累计的代码路径会产生越来越多的债务，因为维护向后兼容性`compatibility`的难度越来越大，复杂度也呈指数型增长。对代码路径之间所有可能的交互进行测试，这变得困难或者不可能。这里一个著名的危险例子是：`Knight Capital`的系统在`45`分钟内损失了`4.65`亿美金。这显然是由于过时的实验代码路径的意外行为。

   与传统软件中的`dead flags`情况一样，定期重新检查每个实验分支以查看是否可删除通常是有益的。通常只有一小部分可能的分支被实际使用，很多实验分支可能已经被测试过并且被放弃了。

5. 抽象债务`Abstraction Debt`：上述问题强调了这样一个事实，即明显缺乏支持机器学习系统的强大抽象。`Zheng`最近做了一个令人信服的比较，把机器学习状态的抽象和数据库技术的状态进行比较，指出在机器学习文献中没有任何东西能够像关系数据库这个基础抽象一样成功。描述数据流、模型、或预测的正确接口是什么？

   特别是对于分布式学习，仍然缺乏广泛接受的抽象。可以说`Map-Reduce`在机器学习中的广泛使用是由于缺乏强大的分布式学习抽象。事实上，近年来为数不多的几个广泛认同的领域之一似乎是：`Map-Reduce`对迭代式的机器学习算法而言是一个糟糕的抽象。

   参数服务器`parameter-server`抽象看起来更加健壮，但是这个基本思想有多个相互竞争的规范`specifications`。缺乏标准的抽象使得组件之间的界限很容易变得模糊。

6. 常见气味`Common Smells`：在软件工程中，设计气味`design smell`可能表明组件或系统中存在潜在的问题。我们确定了一些机器学习系统中的常见气味，不是硬性规定而是作为主观指标。

   - 普通旧数据类型气味`Plain-Old-Data Type Smell`：机器学习系统使用和产生的丰富信息通常都是使用普通的数据类型（例如原始浮点数、原始整数）来编码的。在一个健壮的系统中，一个模型参数应该知道它是一个对数几率乘子`log-odds multiplier`、还是一个决策阈值，预测值应该知道有关生成它的模型的各种信息以及它如何被消费。
   - 多语言气味`Multiple-Language Smell`：用特定的语言编写系统的特定部分通常很有吸引力，尤其是当该语言有一个方便的库或语法来完成手头的任务时。然而，使用多种语言通常会增加有效测试的成本，并且增加将所有权`ownership`转移给其他人的难度。
   - 原型气味`Prototype Smell`：通过原型在小范围`small scale`内测试新想法是很方便的。然而，经常依赖原型环境可能是一个指标，表明大范围`full scale`的系统是脆弱的、难以变更的，或者可以从改进`improved`的抽象和接口中受益。

     维护原型环境需要付出代价，而且时间压力可能会鼓励将原型系统用作生产解决方案，这是非常危险。此外，在小范围内发现的结果很少可以反映大范围的现实。

     > 例如，`DNN`模型在小样本下的表现和大样本下的表现可能截然不同。模型在小样本下可能很快训练完毕，但是在大样本下可能无法训练（资源需求是天文数字）。

1.5 配置的债务

1. 积累债务的另一个潜在的、令人惊讶的领域是机器学习系统的配置`configuration`。任何大型系统都有广泛的可配置项`configurable options`，包括：使用哪些特征、如何选择数据、各种特定于算法学习的设置、潜在的预处理或后处理 、验证方法等等。

   我们观察到，研究人员和工程师可能会将配置（以及配置的扩展`extension`）视为事后想法`afterthought`（指的是未经周密考虑的、在事后添加的事物）。 事实上，配置的验证或测试甚至可能被认为并不重要。在一个高度开发的成熟系统中，配置的行数可能远远超过传统代码`traditional code`的行数。每个配置项都有可能出错。

   考虑下面一个配置的例子：

   - 特征`A`在`9-14 ~ 9-17`日期间被错误的记录（因此需要剔除）。
   - 特征`B`在`10-7`日之前的数据上不可用。
   - 由于日志格式的更改，用于计算特征`C`的代码必须在`11-1`日之前和之后针对数据处理进行调整。
   - 特征`D`在生产环境中不可用，因此在在线配置中，`query`模型时必须使用替代特征D'和D''。
   - 如果使用特征`Z`，那么用于训练的作业`job`必须由于`lookup tables`而被给予额外的内存，否则它们的训练效率将很低。
   - 由于延迟限制`latency constraints`，特征`Q`排除了特征`R`的使用。

所有这些混乱`messiness`使得配置难以正确地修改，也很难理解。然而，配置中的错误可能代价昂贵，导致严重的时间损失、计算资源浪费、或者生产事故。这促使我们阐明了良好配置系统的以下原则：

- 应该很容易地将配置指定为对先前配置`previous configuration`的一个微小的变更。
- 应该很难出现人工错误、遗漏或疏忽。
- 应该很容易从视觉上`visually`看出两个配置之间的差异。
- 应该很容易自动化关于配置的断言`assert`和基本事实验证：使用的特征数量、数据依赖的传递闭包`transitive closure`等等。
- 应该可以检测到未使用的或者冗余的设置`settings`。
- 配置应该经过完整的代码审查`code review`并`check into`代码仓库`repository`。

1.6 处理外部世界的变更

1. 机器学习系统如此迷人的原因之一是它们经常直接与外部世界`external world`交互。经验表明：外部世界很少是稳定`stable`的。这种背景变化率`background rate`会产生持续的维护成本`ongoing maintenance cost`。

2. 动态系统中的固定阈值`Fixed Thresholds in Dynamic Systems`：通常需要为给定的模型选择一个决策阈值来执行某些操作：预测真假、将电子邮件标记为垃圾邮件`spam`或非垃圾邮件`not spam`、展示或不展示给定广告。

   机器学习中的一种经典方法是从一组可能的阈值中选择一个阈值，以便在某些指标（如`precision`和`recall`）上取得良好的`tradeoff`。然而，这样的阈值通常是手动设置的。因此，如果模型更新了新数据，旧的、手动设置的阈值可能无效。

   跨多个模型手动更新多个阈值既费时又费力。针对这类问题的一种缓解策略是：对`hold-out`验证数据进行简单评估来自动学习阈值。

3. 监控和测试`Monitoring and Testing`：单个组件的单元测试和整个运行系统的端到端测试很有价值，但是面对不断变化的世界，此类测试不足以提供系统按预期工作的证据。实时地对系统行为进行全面的实时监控，并结合自动响应（如告警），这对于系统的长期可靠性`reliability`至关重要。

   关键问题是：监控什么？可测试`testable`的不变量`invariants`并不总是显而易见的，因为许多机器学习系统旨在随着时间的推移而适应`adapt`。我们提供以下建议：

   - 预估偏差`Prediction Bias`：在按预期工作的系统中，通常情况下，预估`label`的分布等于真实`label`的分布。这绝不是一个全面的测试`comprehensive test`，因为这可以通过一个空模型`null model`来满足，该模型简单地不考虑输入特征而预估`label`出现的均值。然而，这是一个非常有用的诊断方法，并且诸如此类指标的变化通常表明存在需要注意的问题。例如，这种方法可以帮助检测世界行为`world behavior`突然改变的情况，使得从历史数据中提取的训练数据分布不再反映当前的现实。

     按照各种维度切片`slicing`的预估偏差（即在某些维度样本上的`label`分布差异）可以快速隔离问题`isolate issues`，也可用于自动告警。

   - 行动限制`Action Limits`：在用于对现实世界采取行动的系统中，例如对`item`出价`bidding`或者将消息标记为垃圾邮件，设置`set`和执行`enforce`行动限制作为健全性检查`sanity check`可能很有用。这些限制应该足够宽，不会误触发。如果系统达到给定行动的限制，则应该出发自动告警并触发人工干预或人工调查。

     > 例如：系统连续`1`个小时对所有邮件标记为垃圾邮件；系统对`item`出价达到一个非常大的值。

   - 上游生产者`Up-Stream Producers`：数据通常从各个上游生产者输入到学习系统。应该对这些上游过程进行彻底的监控、测试，并定期满足服务水平目标`service level objective`（该目标考虑了下游机器学习系统需求）。

此外，任何上游告警都必须传播到机器学习系统的控制面板，从而确保机器学习系统的准确性`accuracy`。类似地，机器学习系统在满足既定服务水平目标方面的任何失败也会在传播到下游的所有消费者（如果可行的话，直接传播到这些消费者的控制面板）。

由于外部变化是实时发生的，因此响应也必须实时发生。依靠人工干预来响应告警是一种策略，但是对于时间敏感的问题可能很脆弱`brittle`。创建无需直接人工干预即可自动响应的系统通常是非常值得投资的。

1.7 其它领域

1. 我们现在简单地强调一些与机器学习相关的技术债的其它领域。
   - 数据测试债`Data Testing Debt`：如果在机器学习系统中数据代替了代码，并且代码应该被测试，那么很显然，对于输入数据的一些测试对于一个运行良好的系统而言是至关重要的。

     基本的健全性检查`sanity checks`是很有用的，因为更复杂的测试可以监控输入分布的变化。
   - 复现性债`Reproducibility Debt`：作为科学家，重要的是我们可以重复实验并获得类似的结果。但是设计真实世界的系统以允许严格的可复现性`reproducibility`是一项艰巨的任务，因为随机算法、并行学习中固有的不确定性`non-determinism`、依赖的初始条件、以及外部世界的交互所造成的。
   - 流程管理债`Process Management Debt`：本文中描述的大多数用例`use cases`都谈到了维护单个模型的成本，但是成熟的系统可能同时运行几十个或几百个模型。这引发了一系列重要问题，包括安全地自动地更新许多相似模型的许多配置`configurations`的问题、如何在具有不同业务优先级的模型之间管理和分配资源、如何可视化和检测生产管道`production pipeline`中数据流的阻塞。

     开发工具来帮助从生产事故中恢复`recovery`也很重要。需要避免的一个重要的`system-level smell`是：具有很多人工步骤的通用流程`common processes`。
   - 文化债`Cultural Debt`：机器学习研究和工程之间有时存在很强的边界，但是这对于长期的系统健康可能适得其反。重要的是要创建团队文化，鼓励删除特征、降低复杂性、提升可复现性`reproducibility`/稳定性`stability`、以及监控，其重视程度和重视提高准确性`accuracy`的程度相同。

     根据我们的经验，这最有可能发生在机器学习研究和工程方面都有优势的异质`heterogeneous`团队（两拨人分别处理机器学习研究和工程）中。

1.8 结论

1. 技术债是一个有用的比喻，但不幸的是，它没有提供可以随时间跟踪的严格指标。我们如何衡量系统中的技术债，或者评估这种债务的全部成本？仅仅注意到“一个团队仍然能够快速行动`move quickly`”，这件事本身并不是低债务或良好实践的证据，因为债务的全部成本只有在随着时间的推移才会变得明显。

   事实上，快速行动通常都会带来技术债务。一些需要考虑的有用问题是：
   - 一种全新的算法方法可以多容易地全面`full scale`测试？
   - 所有数据依赖的传递闭包`transitive closure`是什么？
   - 新变更`new change`对系统的影响可以精确到什么程度？
   - 改进一个模型或信号会降低其它模型或信号的质量吗？
   - 团队的新成员能多快上手？

   我们希望本文有助于鼓励可维护机器学习领域的进一步发展，包括更好的抽象、更好的测试方法、和更好的设计模式`design patterns`。

也许获得的最重要的洞察`insight`是：技术债是工程师和研究人员都需要注意的问题。以大幅度增加系统复杂性为代价，从而提供微弱`tiny`的准确性收益的解决方案不是明智的做法。即使添加一两个看似无害的数据依赖也会减缓进一步的进度。

偿还机器学习相关的技术债需要做出具体的承诺，这通常只能依靠团队文化的转变来实现。认识到`recognizing`、优先考虑`prioritizing`、以及奖励`rewarding`这方面的努力对于成功的机器学习团队的长期健康是非常重要的。