基于Kubeflow的机器学习服务组件化开发方法

# 引言

机器学习是一个近年来迅速崛起的领域，得到了广大开发人员的关注。然而，开发出一个新的机器学习模型并不容易，首先需要深厚的机器学习理论知识，其次需要大量的训练数据（对于深度学习，尤其如此），最后，训练一个模型往往需要消耗大量的运算力，特别是对GPU的要求非常高。

相对于专业级的数据科学家，非专业级数据科学家一般不具有数据科学类学科的专业背景，编写代码的能力较弱，不参与数据科学的全流水线活动，而是利用自己在某一领域的知识和经验优势，主要借助数据科学工具完成数据科学流水线的某一或少数活动。近年来随着数据科学进入各个领域，非专业级数据科学家的数量逐步递增，非专业级机器学习可视化流程编排工具的需求随之而来。因此需要对机器学习流程进行组件化的开发，并以组件的形式向外提供服务以简化机器学习流程。

当然，机器学习也属于软件设计领域，所以采用组件化模式也是自然而然的方法。机器学习可以在三个层次上进行模块化：

* + 1. 应用层：应用层可以提供高度封装好的功能接口，使用者甚至无需任何机器学习背景知识，对外接口也可以看不出采用了机器学习的方法。
    2. 中间层：在中间层可以提供训练好的模型，使用者可以在项目中直接加载模型，也可以在自己的数据集上对模型进行再训练，实现个性化需求。这种方法不再需要海量数据，就可以获得不错的性能。
    3. 框架层：框架层可以提供梯度递减算法、softmax、卷积运算等等，这些是构建深度学习必要的基础组件，但开发者必须设计模型，需要有深厚的机器学习背景知识，以及大量的数据。

组件所在的层级越高，接口越简单，使用越方便，但灵活度越低，只适合非常通用的场景。层级越低，灵活性越高，但编写和训练模型就越复杂。所以处在中间层级的模型组件化能够在复杂性和灵活性之间取得很好的平衡。

基于Kubeflow的机器学习服务组件化开发方法主要是在机器学习服务组件化开发的基础之上，使用Kubeflow作为所开发组件的载体，向外提供服务，同时使用Kubeflow作为组件流程编排工具和组件调度工具，在机器学习中，将组件以pipeline的方式进行编排，编排完成之后输入Kubeflow对编排好的流程进行调度和执行组件并返回最终的结果。在流程调度过程中，数据的传输以及元数据的管理也是使用Kubeflow作为载体进行管理。

# 机器学习服务组件化开发方法

## 机器学习服务

服务性导向的机器学习（Machine Learning as a Service, MLaaS）是自动化和半自动化云平台的一个概括性定义，这个云平台范围很广，包含数据预处理、模型训练、模型评估以及预测。而且预测结果可以通过REST API与其他IT系统进行交互。其旨在以接口的方式向外提供功能调用，简化机器学习流程，降低机器学习使用门槛。亚马逊机器学习服务，微软Azure机器学习和谷歌云AI是三个领先的云MLaaS服务，在只有少量甚至没有数据科学专业知识的用户也能够利用这些服务进行快速模型训练和部署。

## 组件化与模块化

组件：组件的最初的目的是代码重用，功能相对单一或者独立。在整个系统的代码层次上位于最底层，被其他代码所依赖，所以说组件化是纵向分层。

模块：最初的目的是将同一类型的代码整合在一起，所以模块的功能相对复杂，但都同属于一个业务。不同模块之间也会存在依赖关系，但大部分都是业务性的互相跳转，从地位上来说它们都是平级的。

总体而言，组件化开发和模块化开发实际上有各自的适用领域。组件化开发更多被应用于技术底层的实际实现，而模块化开发则需要结合实际的业务功能。组件化和模块化的具体区别如表1。在项目的实际开发过程中，两种开发模式往往是并存的，结合使用的。

表1 组件化和模块化的具体区别

| **类别** | **目的** | **特点** | **接口** | **成果** | **架构定位** |
| --- | --- | --- | --- | --- | --- |
| 组件 | 重用、解耦 | 高重用、松耦合 | 无统一接口 | 基础库、基础组件 | 纵向分层 |
| 模块 | 隔离/封装 | 高内聚、松耦合 | 统一接口 | 业务框架、业务模块 | 横向分块 |

## 服务的组件化

当机器学习平台需要向外提供服务的时候，为了划分不同的功能对外部提供服务，所以需要采用组件化的方式对不同的功能进行低耦合的开发。组件化的开发可以使得机器学习流程中的服务都是以单独的方式向其上下游提供服务，不会依赖于其他组件，组件之间的运行互不影响。

## 组件化的开发方法

组件化的开发主要是将一定细粒度的机器学习方法封装为一个组件，封装好的组件能够直接的向外提供服务。将机器学习中需要执行和调用的方法封装为各个组件，每一个组件有其特定的功能。组件是最基本的能够重用的功能封装，不需要统一的接口，因此在开发上没有过多的要求。

# 基于Kubeflow的机器学习服务组件化平台架构

Kubeflow旨在支持多种机器学习框架运行在 Kubernetes 之上，比如 Tensorflow, Pytorch, Caffe 等常见框架。它包含了 operator、pipeline、超参数调优、serving 等诸多模块。它通过提供对应的 operator，基于 Kubernetes 的 Pod/headless Service 等基础资源为框架提供与之相配的更高层次的资源。比如 tf-operator 为 Tensorflow 提供了 job 维度的生命周期管理能力，以满足 Tensorflow 分布式训练的资源和拓扑需求，达到了一键式部署 Tensorflow 训练任务的效果。

机器学习服务组件化平台主要是使用Kubeflow的pipeline功能，Pipeline是一个基于Argo实现了面向机器学习场景的工作流项目，提供机器学习流程的创建、编排调度和管理，还提供了一个Web UI。

pipeline是由多个组件(Component)组成，每一个组件都可以是以往标准机器学习代码中的一个方法。该平台采用组件堆叠的方式动态的组成一条pipeline，一条pipeline就是一个机器学习从数据处理到模型训练的全流程。全流程中不同的功能由不同的组件来实现，pipeline的构建者可以动态的选择不同的组件来以不同的方式完成同一个步骤。基于Kubeflow的 机器学习服务组件化开发平台架构图如下图所示。



基于Kubeflow的机器学习服务组件化开发平台架构

在整个平台的架构中，从外而内可以分为pipeline可视化编排层、pipeline解析层和Kubeflow执行调度层。在调度层之后还有pipeline结合的adapter适配器层，组件开发和生成的组件生成层。每一层的具体功能和设计如下所示：

* 1. Pipeline 编排层：可视化图形的方式或者是脚本编写的方式对机器学习流程进行编排。通过拖拉组件的方式进行机器学习流程进行编排，并生成pipeline。生成pipeline之后传递到后端进行pipeline的解析工作。
  2. Pipeline解析层：在接收到Pipeline可视化编排层编排好的Pipeline脚本后，解析工作便Pipeline解析层就开始对脚本进行解析，其解析工作是将Pipeline脚本转换为Pipeline开发SDK标准的python脚本代码。最最后运行Python代码生成Pipeline的yaml文件，并将文件上传到Pipeline仓库中进行存储。
  3. pipeline调度中心：在pipeline调度中心中，需要对pipeline执行时，使用pipeline的管理界面从pipeline仓库中拉取需要运行的pipeline并创建一次运行，运行起来之后pipeline的所有操作数据和返回数据都存储在元数据管理中心。
  4. pipeline仓库：pipeline包含组件仓库和pipeline仓库两种仓库。组件仓库是对组件进行统一集中管理的地方，所有生成的组件都上传到组件仓库。pipeline编排层会在组件仓库拉取组件生成pipeline，Kubeflow调度中心会在组件仓库拉取组件进行调度工作。pipeline仓库会保存已经生成好的pipeline，需要运行pipeline时可直接从组件仓库中拉取运行。
  5. 组件生成层：组件的生成主要是有三种方式，其最终目标都是生成一个功能组件的yaml描述文件，三种组件生成的方式如下所示：
     1. 人工组件开发：人工使用SDK对机器学习功能组件进行开发，开发完成之后手动编译得到组件的yaml文件并上传至组件管理中心。使用人工开发的方式速度较慢，但人工直接按照标准的组件开发方法进行开发，组件出错的概率小，组件的质量能够得到保证。
     2. 对第三方库进行转换：对第三方的库代码进行转换有两种方式，第一种方式是直接对第三方库代码的源码进行转换，获取到源码之后直接扫描所有源码将源码的每一个方法按照组件开发标准进行转换。这种方法需要将当前转换方法所调用的方法都递归的添加到当前方法的方法体中，此时会出现递归深度过大的问题。第二种转换方式是在第三方库的API方法外使用一层方法进行调用包裹，之后再对包裹方法进行转换，这种转换方式只需要引入第三方库，不需要递归的将API实现方法内所调用的方法递归的引入进去，因此转换速度快，不会出现递归深度过大的问题，但API方法的识别便是第二种转换方法的一个难点。两种转换方法转换完成之后对转换之后的代码进行编译得到组件的yaml文件，将yaml文件上传到组件管理中心进行集中管理。
     3. 对已有工程进行转：在对第三方工程进行转换主要是对第三方工程的源码中所有的方法进行转换，转换方法与对第三方库转换的第一种方法相同，转换后组件描述文件提交到组件管理中心。

在整个平台运行的过程中，在pipeline流程编排上可以分为以下几个步骤；



pipeline流程编排步骤图

* + - * 1. Pipeline编排：在前端页面进行pipeline脚本的编排或者是可视化pipeline编排，编排好之后将编排好的pipeline描述文件发送到后段脚本解析层进行pipeline脚本解析；
        2. pipeline脚本解析：pipeline解析器的主要工作是解析pipeline描述文件为kubeflow pipeline SDK语法规则的pipeline python描述代码。同时将此代码进行运行并生成pipeline的yaml描述文件，之后再将描述文件上传到pipeline仓库中。
        3. pipeline调度：当需要对pipeline执行的之后，调度中心会从元数据管理中心拉取和保存元数据，并对pipeline中的组件进行调度运行。

# 机器学习组件开发规则

机器学习组件在开发时需要使用Kubeflow pipeline组件化开发的SDK进行开发。因此，在开发时需要满足SDK中一些指定的规则，开发出来的组件才可以正常的被调用。其具体的规则主要表现在以下几个方面。

* + 1. 组件的输入和输出都是数据而不是方法或者引用，在组件封装的过程当中，每个组件都会被单独出来，组件调用执行的时候是一个单独的容器，和其原来的上下文环境完全是隔离开来的，因此如果存在上下文环境的关系，会导致组件在运行过程中捕获不到上下文环境而报错。
    2. 组件内部形成闭包环境，组件内代码对第三方包的引入和使用都必须在组件方法定义内声明，代码不能够写在组件方法定义外。
    3. 方法内不能存在中文，方法内不能存在任何中文。如果代码内存在了任何中文，在pipeline执行时Kubeflow会出现异常导致组件无法运行。
    4. 需要安装的第三方包都需要添加到装饰器内的packages\_to\_install内。

满足以上规则之后便可以开发出一些基本的机器学习组件，如果需要满足本平台组件开发要求还应该添加以下编码规范。

1. 组件方法定义代码内参数必须使用输入数据元数据的形式。使用元数据的形式有助于降低组件之间的耦合度，并且能够更加容易的结合组件。组件之间采用元数据的形式进行参数的传递也是pipeline的设计思想之一。
2. 组件方法不能存在返回值，返回参数均使用元数据的形式返回。组件方法内不使用return语句进行返回，需要return的数据都存到输出元数据指定的位置。

如果是使用人工开发组件的方式进行组件开发，则只需要满足以上规则即可，如果是使用代码转换的方式进行组件的开发，便还需要额外的满足代码转换的相关规则，在第三方工程满足代码转换的规则之后才可以转换出正确的组件代码。具体转换规则如下：

1. 方法内没有对方法外部变量的引用
2. 方法参数均是以值传递的方式
3. 方法返回代码只能够返回变量，不能够返回表达式或者是方法定义

如图4.1代码便是满足上述规则的代码，在使用代码转换方式对代码转换之后便可得到图4.2所示的组件代码。

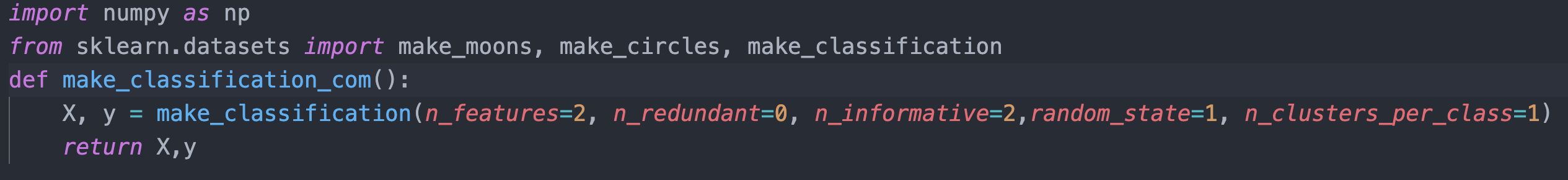


图4.1 满足组件转换代码样例

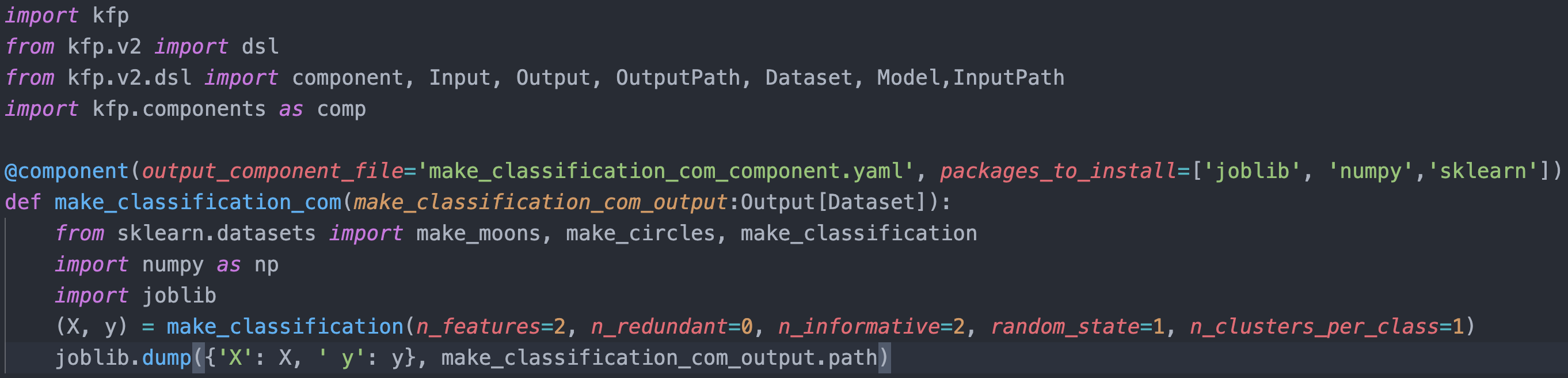


图4.2 使用组件转换脚本转换之后的代码样例

# 机器学习组件代码转换脚本

在组件代码转换脚本的第三方工程转换工作中，可以把第三方工程分为标准化的代码和非标准化的代码，若工程代码满足上一章节中所列出的所有规则便将代码定义为标准化的代码，如果存在不满足的条款则将其称为非标准化代码。

## 标准化代码处理

对于标准化的代码我们可以使用**ast**的将代码进行规则进行转换，将标准化的代码批量转换成为标准的kubeflow pipeline的组件代码。

AST (Abstract Syntax Tree(抽象语法树)) 是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构。它由一堆节点（Node）组成，每个节点都表示源代码中的一种结构。

在python中，内置了ast包，能够将传入的代码生成AST，并且能够使用ast的内置api对AST进行遍历和处理，在python的ast包中提供了两个类对ast进行处理，分别是class ast.NodeVisitor 和 class ast.NodeTransformer这两个的区别就是visit是修改原来的node,transformer可以替换一个新的node。这两种类采用的对AST遍历的方式都是递归遍历，结合当前代码处理的需求，我最终选择了后序遍历，在遍历过程中对节点进行处理和替换，并选择使用ast.NodeTransformer类进行处理。

在使用ast之后，对代码种类的判断可以直接交给ast包内置的api来实现，我们只需要关注对不同种类的代码进行不同的处理即可，当实现了对应节点的visit方法以后便可以对指定的节点进行处理，每一个种类的节点都是一个实体类，对节点内的代码处理就是对实体类进行处理，并返回修改或者是新生成的实体类便可以便可以完成一个节点的替换。对抽象语法树上的每一种节点进行处理对应的是一个遍历类中不同的处理方法，其优点是能够将不同节点的处理逻辑分开来，提高代码的可维护性，提高程序的健壮性。代码可复用性强，粘连性低。并且不用特定的去匹配指定的语句，不同的语法结构有不同的节点来表示，对一种节点下的其他节点能够使用递归的方式遍历到，因此采用合适的递归方式就能够找出当前节点的所有子节点进行处理。

在ast包使用的过程中同时也发现了使用ast仍然不能很好解决的问题，在AST中，return返回代码的定义是一个Return节点，这个节点的value是return内表达式的节点，但是return可以返回很多种种类的值，不仅仅是变量，还可以是表达式或者是方法的调用等等，因此没办法使用ast对返回值进行处理。也因此采用了字符串的方式处理了这部分代码。

在使用ast对代码进行处理之后，我们能够更加快捷方便的对整个代码进行处理，能够使用一定的结构方法去操作代码块，同时能够处理整个项目中的代码，而非局限于一个.py文件内的代码，对代码的处理消除了想不到的书写模式。能够精确的定位到方法代码中对其他方法的调用等。使得代码更加的规范，兼容性更强。

## 非标准化的代码

非标准化的代码就是不满足标准化的条件，一般不满足最多的便是**代码返回值**问题。在处理这一类问题时我们可以将return中的表达式、复杂数据类型初始化等代码转换成变量再将变量进行返回的方式将非标准化的代码转换成标准化的代码。

# 总结

基于Kubeflow的机器学习服务组件化开发方法，能够使用可视化的方式对机器学习流程进行编排，并全程通过可视化的方式完成整个机器学习流程。同时可以整合第三方的工程，批量的添加功能组件到平台中去。但在整个组件集成的流程当中还存在着一些问题需要去解决，比如代码自动转换时如何将非标准代码转换为标准代码、解决递归深度过高的问题等。