Serenity：基于库的 Python 代码分析，用于代码补全和自动化机器学习

Wenting Zhao 计算机科学系 康奈尔大学
wzhao@cs.cornell.edu

Ibrahim Abdelaziz 托马斯·J·沃森研究中心 IBM Research
Ibrahim.abdelaziz1@ibm.com

Julian Dolby 托马斯·J·沃森研究中心 IBM Research
dolby@us.ibm.com

Kavitha Srinivas 托马斯·J·沃森研究中心 IBM Research
Kavitha.Srinivas@ibm.com

Mossad Helali 计算机科学系 康考迪亚大学
mossad.helali@concordia.ca

Essam Mansour 康考迪亚大学计算机科学系
essa
m.mansour@concordia.ca

arXiv:2301.05108v1 [cs.PL] 5 Jan 2023

摘要

Python 等动态类型语言已变得非常流行[1]。除其他优势外，Python 的动态特性及其与本机代码的直接衔接，使其成为许多研究领域（如人工智能）的事实标准语言。然而，这种灵活性也使静态分析变得非常困难。尽管为 Python 创建健全或近似健全的分析仍是一个开放问题，我们在本文中提出了 Serenity，这是一个用于 Python 静态分析的框架，事实证明它足以胜任某些任务。Serenity 框架利用两种基本机制生成代码抽象：(a) 在语言转换的核心依赖动态分派；(b) 对库进行极度抽象。我们通过两个应用展示了 Serenity 分析的效率和实用性：代码补全和自动化机器学习。在这两个应用中，我们证明了此类分析能提供强有力的信号，并可用于实现最先进的性能，分别媲美神经模型和动态分析。

关键词：Python、静态分析、代码补全、自动机器学习

1 引言

Python 的静态分析很困难，部分原因在于一些通常被视为其优势的特征：动态特性，以及与本机代码的直接链接。Python 是动态类型语言，因此无法利用静态类型为 Java 等语言的分析提供的帮助。Python 具有动态对象结构；方法可以自由赋值和修改，使调用解析变得复杂。即使是方法调用和对象创建等基本构造，在基础语法中也可能存在歧义。除了语言本身之外，Python 丰富的库中有许多，尤其是机器学习中使用的计算密集型库，都是用本机代码实现的，这使得分析需要跨语言支持。基于这些原因，据我们所知，尽管此类分析对工具等场景很有价值，但目前仍缺乏广泛使用的 Python 分析框架。

然而，尽管为 Python 创建一种可靠（或 soundy）的分析方法仍然是一个开放问题，我们展示了 Serenity[2]，这一框架，它事实证明足以应对某些任务。除了将 Python 抽象语法树（AST）相对直接地转换为控制流图（CFG）之外，Serenity 还利用了两种基本机制：

1. 在语言转换的核心依赖动态分派。即使要判断某个结构究竟是对象创建还是函数调用，有时也并非总是可能，而这只是一个例子。对于这类情况，我们的方法是将其转换为针对表示组成结构的类型进行的动态分派。我们详细说明了如何以这种方式对 Python 的诸多细微特性进行建模。

2. 对库进行极度抽象。用户代码通常会大量使用 API 来创建领域对象并对其进行操作，例如 NumPy 中的数组，但这些对象对用户代码而言往往相当不透明。因此，我们发现，处理库时通常只需跟踪它们创建的对象以及对这些对象调用的方法即可。这种做法并不可靠（soundy），更谈不上健全；而且它本来也无意做到这一点。不过，我们将展示，这种做法能够对用户代码进行实用的建模。

我们首先讨论如何对 Python 进行建模，以及库抽象如何仍能为用户代码提供有用的分析。随后，我们将说明这类分析的实用价值，并重点介绍两种依赖此类分析输出的应用：

代码补全 是所有集成开发环境都应具备的核心功能，其目标是根据先前的代码，建议可调用的方法和函数。我们展示了数据流分析如何帮助我们聚焦于补全位置的相关代码；将其与

[1]https://www.techrepublic.com/article/programming-languages-pythons-growth-is-absolutely-explosive-says-anaconda-ceo-and-not-slowing- down/

[2]谨以此向 Reinhold Niebuhr 致歉：“赐予我们勇气，去建模必须建模之物；赐予我们宁静，去接受无法建模之物；并赐予我们洞察力，分辨二者。”

函数调用之前的本地程序上下文结合后，代码补全性能远胜于仅使用上下文。

自动化机器学习 它接收以结构化表格形式给出的数据集，并构建有效的机器学习流水线，以根据某些列预测其他列。以往的方法基于动态分析，而我们证明静态分析也同样有效。静态分析更实用，因为实际运行这些流水线既繁琐又耗费成本；此外，还可以通过分析挖掘大型开放存储库，为此类数据库填充数据。

在本文余下部分中，我们首先通过一个运行示例（第 2 节和第 3 节）介绍 Serenity 对 Python 进行建模的技术。随后，我们通过代码补全（第 4 节）和自动化机器学习（AutoML）（第 5 节）这两个应用验证我们的分析。最后，我们在第 6 节综述相关工作，并在第 7 节总结全文。

2 运行示例

图1展示了本文的运行示例，其中包含一段改编自 ETH150K 基准测试中 multi_class_svm.py 脚本的 Python 代码片段。我们所做的修改（第 1 至 10 行）是添加了一些调试选项，以展示 Python 的复杂性。变量 fitit 会根据 debug_level 被赋值为两个函数之一：要么是一个闭包，要么是一个带有附加断言的函数。代码加载数字数据集（第 20 行），读取数据集的特定字段（第 21 行），处理数据集（第 22 行），将数据划分为训练集和测试集（第 23 行），最后创建 X_train_bias（第 26 行）。然后，代码创建机器学习模型 FrankWolfeSSVM（第 42 行）和 LinearSVC（第 72 行）。接着，代码调用它们：对 LinearSVC（第 74 行）的 fitit 调用会接收该模型（第 26 行），而 fitit 则以 X_train_bias 为参数在 FrankWolfeSVC 上调用（第 79 行）。请注意，构造函数 FrankWolfeSSVM 在第 42 行调用，而对象上的 fitit 调用位于第 79 行，这体现了大多数代码的一项特性——调用并非局部发生。

3 分析

3.1 背景：调用图框架

Grove 等人 [18] 提供了一个框架，用于表达面向对象语言的调用图算法。它封装了算法的大部分内容，并通过参数化函数来决定如何添加上下文敏感性。本文不讨论该框架的具体细节，但我们依赖其中两点：

- 首先，后文将用到一种称为过程键选择函数（PKS）的机制，它本质上用于指定何时应以上下文敏感的方式分析被调用的函数。
- 其次，该框架区分函数调用点和对象创建点，正如我们将要

```
1     调试_级别 = 3

2 3 如果 调试_级别 > 5:4 定义 fd(模型, 测试, 训练):5 断言长度(测试) < 500 6 模型.拟合(训练, 测试) 7 fitit =fd 8 9

10 拟合 否则: lambda模型、测试集、训练集：模型。拟合(训练集, 测试集) ↪11 15

来自 sklearn.svm导入 LinearSVC 16 17 从 pystruct.models 导入 MultiClassClf 18 从 pystruct.learners 导入 (NSlackSSVM, OneSlackSSVM, SubgradientSSVM, ↪FrankWolfeSSVM) ↪19 20 手写数字 =加载_手写数字() 21 X, y = 数字数据数据矩阵, 数字数据目标值 22 X =X / 16. 23 X_训练集, X_测试集, y_训练集, y_测试集 =训练_测试_拆分(X, y) ↪24 25 # 我们添加一个值为 1 的常量特征作为偏置项 26 X_训练集_偏置项 = np.hstack([X_训练集, np.ones((X_训练集.shape[0], 1))]) ↪41 42 fw_bc_svm = FrankWolfeSSVM(model, C=.1, max_迭代=50) ↪71 72 libsvm =LinearSVC(multi_类='crammer_singer', ↪C=.1)↪73 开始 =时间() 74 拟合(libsvm, X_训练集, y_训练集) 75 时间_libsvm =时间() - 开始 76 打印("使用 sklearn 和 libsvm 的得分：%f（耗时 %f 秒）" % ↪(libsvm.score(X_测试集, y_测试集), ↪时间_libsvm)) ↪77 78 开始 =时间() 79 拟合(fw_bc_svm, X_训练集_偏置项, y_训练集)
```

图1。一个运行示例

如图3所示，这在 Python 中通常是不可能的。因此，我们将这两组整合为一组。

框架论文在第694页顶部定义了相关程序特征，我们在此摘录于图2中。我们需要做两处小改动：

InstVariables 被视为可能用作字段名的字符串集合，而非已声明字段名的集合；在原始框架中，它是后者。尽管

可以在 Python 中定义字段名，但这完全是可选的，因此我们忽略此类定义。

新建点 与调用点 集合相同，以实现上面第二项的效果；也就是说，有一个集合汇总了所有可能的调用点和创建点。这体现了每个点都可能涉及类和函数这一事实。

类 程序中的所有类声明**实例变量** 程序中的所有实例变量声明**过程** 程序中的所有过程声明**变量** 程序中使用的所有变量名**调用点** 程序中的所有调用点**新建点** 程序中的所有新建点**加载点** 程序中所有实例变量读取**存储点** 程序中所有实例变量写入

图2。 来自 [18] 的程序特征

```
1  导入系统23 如果 系统.参数向量[1] == "类1" 或系统.参数向量[1] == "实例":
4 类 X: 5 pass 6 7 if sys.argv[1] =="实例": 8 X =X() 9 10 elif sys.argv[1] ==" class2": 11 类 X: 12 def__new__(*args): 13 return0 14 15 elif sys.argv[1] ==" def1": 16 def X(): 17 return 1 18 19 elif sys.argv[1] =="def2": 20 X = lambda: 2 21 22 elif sys.argv[1] =="import": 23 import X 24 25 elif sys.argv[1] =="方法": 26 27 if sys.argv[2] =="static": 28 类 X: 29 def s(): 30 return 5 31 32 X =X.s 33 34 elif sys.argv[2] =="实例": 35 类 X: 36 v = 4 37 def i(self): 38 return self.v 39 40 y = X() 41 X =y.i 42 43 44 print(str(X)) 45 print(str(X()))
```

图3。动态代码示例

3.2 语言建模

图3展示了 Python 分析必须应对的动态性；在本例中，根据 sys.argv[1] 和有时 sys.argv[2] 提供的值，第45行的 X() 有5种不同的含义：

类1 类 X（第 4 行）定义了一个名为 X 的普通类，第 45 行创建了该类的一个实例。**类2** 类 X（第 11 行）定义了一个名为 X 的类，该类重新定义了 new 运算符，因此第 45 行只返回 0。**定义1** def X（第 16 行）定义了一个名为 X 的函数，在第 45 行调用它会返回 1。**定义2** X = lambda. . .（第 20 行）创建一个闭包并将其赋给 X；在第 45 行调用该闭包会返回 2。**导入** 模块 X（第 23 行）覆盖了默认的模块行为，使其可被调用，并在第 45 行返回 3。**静态方法** X（第 32 行）被赋值为类 X（第 28 行）的静态方法 s，该方法在第 45 行返回 5。**实例方法** X（第 41 行）获得了类 X（第 35 行）中一个绑定的 instancemethod（即对 y 的闭包）i，在第 45 行返回 4。

请注意，X 的所有这些定义都可能流向第 45 行的同一处调用，因此，不同类型的分配、调用，甚至在某些情况下的模块之间，实际上在语法上毫无区别。而类名和函数名都是一等对象。因此，分析必须以动态方式处理这些基本操作，这与 Java 等语言不同，在这些语言中，调用、分配和导入都有明确的语法区别。还要注意，即使是基本的方法调用，也需要闭包来处理第 41 行。

第 45 行的 X() 是对 X 的调用，这使我们能够使用标准的动态分派来建模所有这些行为，并在需要时使用合成的“方法”来处理语言语义。我们还会在字段访问时使用类似的“分派”，以处理类字段与实例字段之间的差异；同样，这种差异只能根据被访问的对象确定。我们将在第 3.3 节利用这些间接层来定义框架模型。

我们采用 Grove 等人 [18] 的术语，将我们的工作表述为对标准面向对象调用图构建的扩展。为适应 Python 的动态特性，我们对该工作的核心定义做了一些调整。这些调整反映了 Python 不要求字段必须先声明才能使用，而且在语法上不区分调用和分配。此外，按照在面向对象框架中表示一等实体的惯例，我们为每个一等实体都设置一个类。如图3所示，类、函数、方法和模块都是一等实体，因此我们的分析类集合包括以下内容：

C_{class} 一个表示程序类 C 的类C_{inst} 一个表示类 C 的实例的类M_{inst} 一个表示模块 M 的实例的类D_{inst} 一个表示函数 D 的实例的类S_{inst} 一个表示脚本 S 的实例的类

现在，通过将每个调用视为针对每种接收者类型的 调用点 $*_{inst}$，并将其视为针对每种接收者类型的 新建点，Python 调用和创建中的大多数特殊情况都得到了处理。$*_{class}$。图3 中第 45 行的调用点会有一些类型由每种机制处理。字段也能无缝处理：第 32 行的 X 是一个 C_{class}，而第 41 行的 X 是一个 C_{inst}，因此通过将静态字段设为类对象的实例字段，即可处理静态状态和实例状态。

调用图构建从一个根桩开始，它会创建主脚本的一个实例 S_{inst} 并调用它。

3.3 框架建模

在许多情况下，很难甚至不可能找到 Python 导入对应的实际代码：导入语句中的名称与实际源代码的位置之间没有固定关系。即使存在这种关系，Python 库的结构也意味着其中很大一部分代码是原生代码，因此 Python 分析框架并不适用。即使能够找到 Python 代码，许多库的规模也足以让精确分析变得困难。在我们的情形中，我们关注的是应用程序代码的行为，而非库内部实现，因此基本不分析框架代码，以尽量减少这些问题。

我们的模型称为 Turtles[3], ，它抽象 Python 框架，以捕捉框架如何与用户代码交互，并忽略其所有内部细节。具体而言，我们使用第 3.2 节中的间接层来建模四个方面：

1. 我们将导入语句建模为返回一个新的框架，该框架以导入模块的名称表示。该框架是一个不透明对象，除实现此模型外不具备任何功能。

2. 对框架函数和方法的调用通常会返回某些内容，用户代码随后可能会使用这些内容。我们将每次对框架的调用建模为从框架返回一个新对象；这种模型具有传递性，因此对这些对象的调用会从框架返回更多新对象。我们用访问这些对象的路径为其标记。

3. 对框架对象字段的访问在我们的模型中意义不大，因为我们根本不对框架 -

状态建模。不过，用户代码通常期望字段访问能返回某些内容，因此我们将所有这类字段访问都建模为返回容器对象。4. turtle 方法的参数大多会被忽略，因为我们不对框架如何处理这些参数进行建模；不过，有时函数会作为参数传入， -

我们假设框架可能会调用这些函数。由于我们不对框架内部状态进行建模，因此模型会在回调作为参数传入的位置调用它们。

Grove 等人 [18] 提出的框架提供了实现这一模型所需的自定义支持。我们首先引入一种新的类类型 T_{path}，用于表示 turtle，即不透明模型对象。条目 1 通过将 import M 语句建模为调用合成导入过程来实现，并将 M 作为其参数。该调用被建模为返回一个 T_M。条目 2 通过过程键选择函数（PKS）实现：该函数接收类型为 T_{path} 的接收者和被调用过程的名称 n，并返回一个类型为 $T_{path.n}$的新 turtle。条目 3 通过在读取任何 T_{path} 类型的字段时直接返回 self 来实现。条目 4 通过一个 PKS 实现，该 PKS 会为每个函数类型的参数生成调用（示例中未展示）。

3.4 继承自 Turtle

我们的数据中有一个复杂之处：应用程序类通常继承自 turtle类，这意味着，如果方法 read 从未被赋值，那么对 self 的方法调用在逻辑上就应当是 turtle 方法。也就是说，如果对 self 的读取指向一个从未被赋值的字段或方法，且该类继承自 turtle，那么此次读取应返回一个新的 turtle对象，以捕获未知的超类行为。然而，这样做颇为棘手，因为方法和字段可以在代码的任何位置被赋值，所以通常无法在分析结束之前确定某个方法或字段是否不会被赋值。我们需要记录这类读取，并在分析结束时将其作为 turtle 读取来处理，然后重新启动分析。重新启动分析本身可能需要重复进行，因为读取一个 turtle 可能会使更多代码变得可达。

3.5 对运行示例的分析

将此分析应用于运行示例（图1），得到的结果就是图4所示的数据流图。为说明我们的框架模型，请看第15行对 LinearSVC 的导入调用；作为导入操作，它会返回一个类型为 $LinearSVC_{inst}$ 的对象，也就是该模块的一个实例。当

[3]出自“乌龟一直套着乌龟”。这句话的来源不明，参见 https://en.wikipedia.org/wiki/Turtles_all_the_way_down

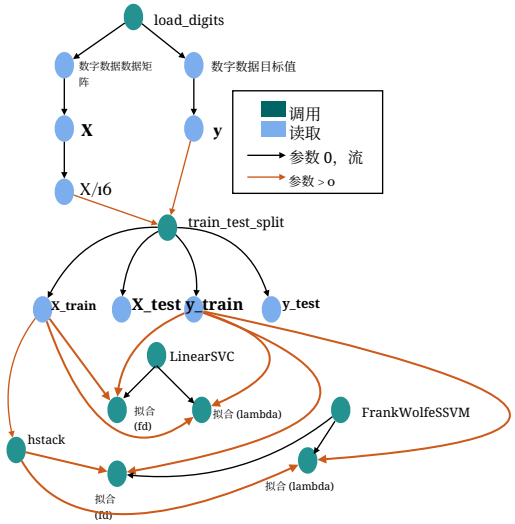

图4。 运行示例的数据流图

调用它时（第72行），会返回一个类型为 $T_{LinearSVC}$ 的 turtle，如绿色节点 LinearSVC 所示。在 fitit 函数中对这个对象调用 fit 时（第74行），第2项表示它会返回一个类型为 $T_{LinearSVC.fit}$ 的派生 turtle，如标记为 fit 的绿色节点所示。由于 fit 是在 LinearSVC 上调用的，因此二者之间由一条黑色数据流边连接。另一方面，传递给 fit 的其他非自身参数以红色箭头表示。其他 turtle 函数也以类似方式展示：load_digits、train_test_split、hstack、FrankWolfeSSVM 请注意，分析并不知道这些函数具体做什么，只知道它们会传递数据。还要注意，fitit 是一个变量，保存着两个一等函数中的其中一个，并且它会被调用来处理创建的两个机器学习模型。要获得图4所示的精确结果，需要能够处理一等函数并执行上下文敏感分析的分析基础设施。具体来说，模型对象以及流向 fit 的普通函数和调试函数的数据流都需要纳入分析，因为这两个函数都有可能流向 fitit。图中通过标注节点所属函数来区分节点。

图4中的其他节点表示局部数据流。最上方的两个蓝色节点表示对 data 和 target 字段的读取，因此它们既有来自 load_digits 调用的边，也有指向各自变量 X 和 y 的边。X 除以 16 进行缩放，这由标记为 X/16 的节点表示。随后，X/16 和 y 通过红色边流向 train_test_split，因为它们是实参。

该图着重展示数据流，捕捉各种 turtle API 在不同程序中的使用模式。这样我们就能学习这些模式，从而为应用提供支持。

3.6 实现

我们的分析是使用 WALA 实现的，并借助 Jython 系统对 Python 2 和 Python 3 的支持。

WALA 旨在支持扩展，我们利用了多项特征来简化实现工作。

主要扩展用于处理 turtle 对象。对于第 1 项，我们重写处理导入的模型函数，使其返回一个合成对象，该对象的 turtle 类型以给定模块命名。对于第 2 项，我们重写 turtle类的被调用方法选择逻辑，使任何调用都指向一个合成方法，该方法会创建并返回一个名称适当扩展的 turtle。对于第 2 项，这个合成方法基本上会忽略其参数，但会为每个参数生成一次调用，以处理回调。对于第 3 项，我们重写处理字段读取的代码，使其在容器属于 turtle类型时直接返回该容器。

另一项配置是为所有 turtle类型增加激进的上下文敏感性。由于合成方法本身很简单，因此确保分别分析每个调用点的成本很低。

4 代码补全应用

我们提出的核心研究问题是：尽管要准确建模 Python 等动态语言面临挑战，Serenity 的分析有多大用处，以及它是否能帮助其他应用。作为首个应用，我们考察了代码补全这一用例，下面将对此进行详细介绍。所谓代码补全，是指给定一段程序代码后，预测一个函数调用的问题，这与集成开发环境提供方法建议的功能类似。我们所说的并非根据代码需求的自然语言描述生成代码（如驱动 GitHub Co-Pilot 的 Codex 模型 [8]），也不是根据函数签名或代码片段以生成式风格生成完整函数的模型（如 CodeT5 [41]）。我们的观察是，对于代码补全，分析的要求是确定哪些方法可由特定类型调用，因此代码补全分析的重点在于检测对象的类型。对于 Python 这样的语言，类型推断很困难，但我们的假设是，代码补全能够显著受益于 Serenity 生成的数据流分析，因为数据流可以为代码补全提供聚焦的上下文。

近年来，出现了大量代码神经模型，例如 [13], [19], [22], [41]，其训练目标是预测代码中随机遮蔽的词元，或预测紧接着的下一个词元；人们可能会认为，后者与代码补全任务相一致。我们的研究问题是，能否利用这些模型在数百万个程序上进行的大规模训练来执行代码补全。具体而言，我们探讨了将 Serenity 提供的数据流分析与这些神经模型结合后，是否能够提升代码补全效果。如果数据流分析确实能从 Serenity 提供任何信号，那么即使这些语言模型已经经过大量训练，它也应能提升代码补全任务的性能。因此我们

```
1 print("使用 pystruct 子梯度 ssvm 得分：%f（耗时 %f 秒）" % ↪(np.mean(y_pred ==y_test), ↪time_subgradient_svm)) ↪2 3 # 标准的一对多多分类 4 # 可能效果也一样好，而且速度更快 5 # 但求解的是不同的模型 6 libsvm =LinearSVC(multi_class='crammer_singer', ↪C=.1)↪7 start =time() 8 libsvm.fit(X_train, y_train) 9 time_libsvm = time() - start 10 print("使用 sklearn 和 libsvm 得分：%f（耗时 %f 秒）" % ↪(libsvm.score(X_test, y_test), ↪time_libsvm)) ↪11 12 13 start =time() 14 fw_bc_svm.?
```

图 5. 用于预测的代码片段

```
1 from sklearn.cross_validation import train_test_split ↪2 from pystruct.models importMultiClassClf 3 from pystruct.learners import (NSlackSSVM, OneSlackSSVM, ↪4 手写数字 =load_digits() 5 数字数据数据矩阵.data 6 数字数据目标值.target 7 X = X / 16. 8 train_test_split(X, y) 9 X_train_bias =np.hstack([X_train, np.ones((X_train.shape[0], 1))]) ↪10 模型 = MultiClassClf(n_features=X_train_bias.shape[1], ↪ n_classes=10) ↪11 fw_bc_svm = FrankWolfeSSVM(model, C=.1, max_iter=50) ↪12 fw_bc_svm.?
```

图 6. 对应于分析图中某个切片的代码片段

将代码补全建模为微调任务，并将微调的训练输入设为以下三种条件之一：

- 函数调用之前的所有代码文本
- 代码切片，仅包含在数据流中与函数调用相关的源表达式
- 代码文本以及切片，两种输入之间用一个词元分隔，以便区分。

对于函数调用之前的所有文本代码，现代语言模型能够拟合的词元数量有限。也就是说，当代码超出这一限制时，就需要进行截断，模型才能运行。一种广泛使用的截断策略是仅保留预测点之前的 n 个词元，其中 n 是最大序列长度；这可能会导致上下文信息相当局部，如图 5 中所示的运行示例（见图1）。图 5 中的关键预测是预测将在 fw_bc_svm 上调用什么方法，但请注意，fw_bc_svm 的构造超出了截断[4]的作用域。

为了获得纯粹受数据流限制的切片，给定一个程序及其对应的数据流图，并要预测函数调用，我们从想要预测的节点开始，反转所有指向该节点的边，并找出所有可达节点。可达集中每个节点都对应于原始程序中的一个源表达式，我们只将那些不是其他任何表达式子表达式的表达式纳入特征。然后，根据这些表达式在源文件中的位置对它们排序，并从分析产物中加入变量名，使代码看起来或多或少像语言模型训练时所用的真实代码。图 6 展示了图 1 中代码对应的数据流切片示例。这里，我们从图 4 中的 fw_bc_svm.fit 调用开始，反转所有指向该节点的边，并执行可达性分析以收集切片，同时加入 digits =load_digits() 等变量名。在此示例中，数据流分析确实提供了有助于预测函数调用的重要信息，因为该切片引入了非局部但相关的代码，例如 fw_bc_svm 的定义，使其进入可馈入神经模型的文本作用域。

4.1 数据集

我们使用了广受欢迎的 ETH150K 基准测试 [35], ，其中包含 10 万个用于训练的程序，其余 5 万个用作测试集。我们使用 Serenity 分析了 ETH150K，成功分析了 150,000 个文件中的 147,288 个。对于已分析的文件，我们使用 Python AST 解析器逐一解析，并收集所有函数调用。对于 AST 识别出的每个函数调用，我们检查能否在分析输出中找到对应函数；如果能找到，则检查该调用的源位置是否与 AST 中的匹配。我们发现，Jython 的源映射有时会出错，因此使用两项指标衡量分析的完整性。分析发现，AST 中有 58.77% 的函数调用具有匹配的源位置；放宽源位置匹配要求后，这一比例为 67.36%。对少数源位置不匹配的情况进行人工检查后发现，问题确实出在映射上

[4]在此示例中，根据其中一个 CuBERT 模型的要求，截断长度设置为 1024 个词元 [22]

```
def f_Hp(self, pars, p,inpt, target):eps = 1E-6导数 =self.fprime(pars, inpt,target)偏移后 =self.fprime(pars + p*eps,inpt, target) 返回(offseted - 导数) / eps
```

图 7. 叶节点为表达式的代码示例

在 Jython 中则会出错。进一步调查发现，许多缺失的调用实际上是 Serenity 不建模而是视为无操作的 Python 原语实例，例如 repr 和 FutureWarning。另有一小部分确实是死代码，尤其是在 Python 文件作为大型应用不可或缺的一部分时——ETH150K 中经常如此。

为了生成切片，我们从叶节点开始，并将范围限制在反转边后节点深度至少为 1 的情况。我们注意到，在大多数情况下，叶节点实际上是表达式，如图 7 中的代码示例所示。由于我们关注的是无法通过纯粹的代码词法分析解决的问题，因此在创建数据集时忽略了这些表达式。将范围限制为可能是函数调用而非表达式的节点后，我们从 65.35% 的程序中生成了切片；这些程序中至少存在一个叶节点可能是函数调用的切片。通过遍历数据流图中的所有叶节点，我们分别为训练集和测试集中的程序生成了 334,415 个和 162,847 个切片。将叶节点视为用于预测的节点后，我们在训练集和测试集中为代码补全总共生成了超过 65K 个标签。图 8 绘出了标签数量对应的标签累积分布。如图所示，标签分布遵循常见的幂律；不过，最常见的标签在训练集和测试集中出现的比例仅为 1.7%，排名前 10 的标签累计出现比例也只有 11.0%。换言之，这是一个困难的分类问题[5]。我们认为，与其他代码补全方式（例如衡量下一个词元预测）相比，这种定义代码补全的方法更为贴合实际，因为集成开发环境通常关注的正是这种情况。

4.2 语言模型选择

为了确定哪种神经模型最适合作为代码补全实验的基础，我们测试了多种与代码相关的语言模型，包括 CodeBERT [13], Graph- CodeBERT [19],CuBERT [22]和 CodeT5 [41]。CodeBERT[13]

[5]我们将以开源形式提供本节所报告的所有工作的数据集和代码。

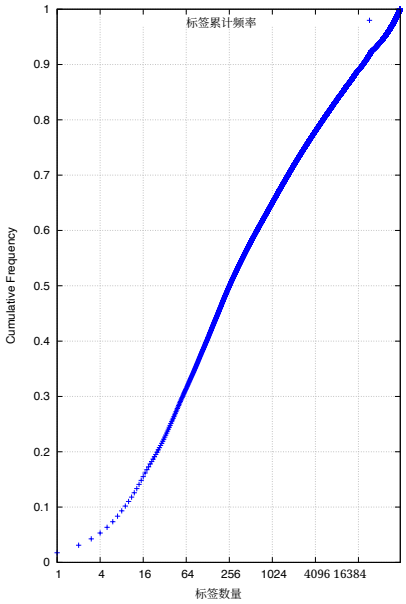

图 8. 分类任务的标签分布

是一种双模态模型，使用涵盖六种编程语言（Python、Java、JavaScript、PHP、Ruby 和 Go）的自然语言（NL）-编程语言（PL）对数据集（例如文档/代码对）进行训练。类似地，GraphCodeBERT 使用 NL-PL 对预训练代码语言模型，但其基础是从抽象语法树中提取的局部数据流图。CuBERT [22] 是另一种基于 BERT 的模型，经过微调以执行多项分类任务，例如检查是否存在特定错误以及预测异常类型。CuBERT 仅使用 Python 代码进行训练，此外还将语言级别词元作为模型输入。CodeT5 [41]是一种基于 T5 架构的编码器-解码器模型 [33] ，融入了代码专属知识，经过训练后能够区分哪些词元是标识符，并在它们被遮蔽时将其恢复。CodeT5 使用多项 CodeXGLUE 基准测试进行微调，其中包括代码缺陷检测和克隆检测等理解任务，以及代码摘要和翻译等生成任务。

图9展示了这些不同模型在代码补全任务上的表现，分别对应未进行微调时的 Top-1 和 Top-5 情况。我们将代码补全建模为掩码预测任务，其中待预测的函数调用就是被掩码的词元。如图9所示，CuBERT 在这项任务中的表现最佳，这并不令人意外，因为它是唯一一个专门使用 Python 训练、并采用语言级别词元的模型，其他模型则没有。值得注意的是，CodeT5 的表现出乎意料地差，但我们认为这在一定程度上可能是因为它是在自然语言与编程语言配对数据上训练的，而且它严格来说是一种生成式模型，

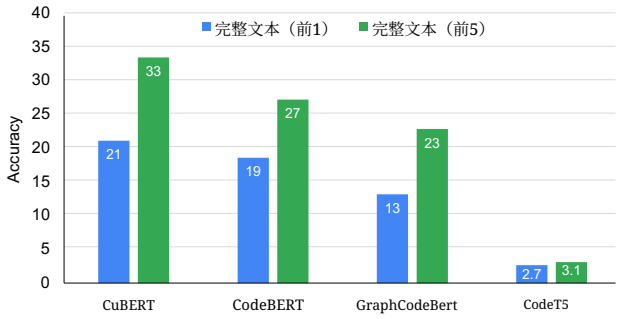

图9。 基础语言模型在 Top-5 和 Top-1 测试数据上的准确率。由于分词问题，CuBERT 在切片上的表现分别基于完整文本和切片的 150,739 个和 137,987 个示例。对于所有其他系统，测试示例数量为 167,816。

因此我们需要指定生成长度。它在最终词元预测方面也需要最多的调优，包括指定不同的搜索策略，所以我们可能没有为它选择最优的搜索策略。不过，就我们的用途而言，我们选择了 CuBERT 作为基础模型，主要是因为我们预计它能从微调中获益最多。需要指出的是，鉴于我们的标签分布，语言模型在完整文本上的 Top-1 表现达到 21%、Top-5 表现达到 33%，已经相当不错。

现在我们转向使用训练输入对 CuBERT 进行微调这一问题，以了解分析是否确实能像我们定义的那样提升代码补全。请注意，CuBERT 的预训练是通过向模型输入 500 万个程序的逻辑行来完成的，因此，至少需要针对要预测函数调用的代码上下文进行一些微调。如前所述，我们对比了三种不同的训练条件：

- **完整文本**：我们向模型提供从调用处开始、向前回溯的文本，如图 5 所示
- **切片**：其中我们使用了如图 6 所示的反向切片
- **组合**：将**完整文本**和**切片**中的文本连接起来，并使用分隔符词元作为模型的输入。

测试针对的是**完整文本**，或**组合**。我们选择这些条件，是因为从示例中观察到，对于代码生成问题，数据流本身并不足够。图10展示了这样一个例子。在这段代码中，第5行和第15行包含了预测 id 所需的线索，但它们与第22行进行调用的接收者无关。然而，代码的局部模式具有相同的变量名，并且相同的一组调用在各个函数中重复出现，这表明 id 可能是一个不错的候选标签。相比之下，相应的切片包含的信息极少，如图11所示，因为

```
1 response.json.return_value =
dict(response, total_count=3, limit=0, offset=0) ↪2
个项目 =self.redmine.project.all() ↪3
self.assertEqual(projects.limit,0) 4
self.assertEqual(projects.offset, 0)↪5
self.assertEqual(projects[0].id,1) 6
self.assertEqual(projects[1].id,2) 7
self.assertEqual(projects[2].id,3) 8 9 def test_
offset_limit(self): 10 response_with_limit_offset =
{'total_count': 2, 'limit': 3,
↪                                    'offset': 1,
'projects': ↪response['projects'][1:3]} ↪11
self.response.json.return_value =response_with_
limit_offset ↪12 projects =
self.redmine.project.all()
↪                                    [1 3]: 13
self.assertEqual(projects.limit,3) 14
self.assertEqual(projects.offset, 1)↪15
self.assertEqual(projects[0].id,2) 16
self.assertEqual(projects[1].id,3) 17 18 def test_
offset_limit_mimic(self): 19 projects =
self.redmine.project.all()
↪                                    [1 3]: 20
self.assertEqual(projects.limit,3) 21
self.assertEqual(projects.offset, 1)↪22
self.assertEqual(projects[0].?
```

图10。 局部文本可以帮助预测的代码片段

```
1 从 tests 导入 unittest,mock, Redmine,URL↪2
Redmine(self.url)3 项目 = self.redmine. 项目.
全部()[1: 3]4          self.断言相等(projects
[0].?
```

图11。数据流缺乏充分上下文的代码

接收者 projects[0] 仅在函数 test_offset_limit_mimic 中定义。

不过，我们注意到，即使截断没有截掉预测所需的关键信息，切片有时也能发挥作用。图12展示了这样一个例子：第3行导入了预测的函数 partial，但实际调用出现在第28行。另一方面，在图13中，该行之前唯一的调用就是导入语句，因此切片可以让相关信息彼此靠近，使神经模型能够更多地关注代码中邻近的元素。

```
1 导入 sys 2 导入 logging 3 从 functools 导入 partial 4 从 datetime 导入 datetime 5 从 abc 导入 ABCMeta, abstractmethod 6 导入 json 7 从_config 导入 AttrDict 8⁹ __all__ = ['multikey_getter_gen', 'unescape_json', 'LogParser', ↪'JSONParser', 'LogLine', ↪10 'AccessLog', 'CommonLogFormat', 'uWSGIParser'↪ ]11 12 定义 multikey_getter_gen(parser, keys, is_indices= False, delimiter="\t"): ↪13 """生成器元函数，用于返回一个函数 ↪14 解析日志行并返回多个键（以制表符分隔）""" ↪15 如果 is_indices: 16 keys =map(int, keys) 17 18 定义 multikey_getter(line, parser, keyset): ↪19 data =parser(line.strip()) 20 返回 delimiter.join((unicode(data[k]) ↪for k in keyset)) ↪21 22 定义 multiindex_getter(line, parser, keyset): ↪23 data =parser(line.strip()) 24 返回 delimiter.join((unicode( data.by_index( idx-1,↪ raw=True)) for idx inkeys)) ↪25 26 如果 is_indices is True: 27 # 字段索引 28 返回 ?
```

图12. 完整文本中可能包含与预测相关的文本，但这些文本距离调用点较远的示例

```
1 部分 = #!/usr/bin/envpython #2 来自 functools 导入partial3 键 = 映射(整数, 键)4 返回?
```

图13. 数据流非常集中的示例

4.3 模型详情

我们使用由 [22][6], 发布的 CuBERT 模型，该模型有 24 层、16 个注意力头和 1024 个隐藏单元，并且在 400 万个不重复的 Python 文件上进行了预训练，这些文件来自 GitHub。微调时，我们将批量大小设为 10，并使用 8 块配备 32GB 内存的 Tesla V100 训练模型。学习率为 5e-5，并在最初的 300 次梯度更新中逐步预热学习率；这些是 Hugging Face 库 [43] 提供的默认值。训练在 20 个训练轮次后停止，或者在评估准确率连续三个训练轮次没有提升后结束。对于**完整文本** 和**切片**模型，我们使用 512 个词元的模型；使用**组合**模型时，则使用 1024 个词元的模型，以便将**完整文本** 和**切片** 中完全相同的词元与分隔符一起使用。

我们将 CuBERT 的分词应用于 ETH150K 中的 Python 程序。首先，使用标准 Python tokenize 模块对这些程序进行分词[7,8]。然后，我们按照 CuBERT tokenize 器所采用的子词分词方法 [39], , 进一步将程序词元拆分为 49,558 个子词。

4.4 微调结果

图14展示了在不同训练和测试条件下，准确预测函数调用的准确率。如图14所示，使用**切片** 进行训练，在完整文本（**切片-完整**）上测试时，准确率为 47%，显著高于 CuBERT 的 Top-1 基线准确率 21%。然而，使用**完整文本** 进行训练，在相同文本（**完整-完整**）上的表现要好得多，达到 62%。这并不令人意外，因为检查示例（例如图10）可以发现，当数据流局限于局部时，完整文本通常包含切片中的表达式；此外，编码模式中的重复也能带来益处，即使不存在真正的关联，这些模式也可能暗示标签。关键问题在于，切片是否能在完整文本所带来的益处之外提供额外帮助。使用组合数据进行训练的结果表明，切片确实提供了强有力的信号：在完整文本（**组合-完整**）上的准确率为 65%，在组合文本（**组合-组合**）上的准确率为 69%。我们还比较了不同条件下的 Top-5 性能，以便与基线语言模型进行比较。不出所料，这一结果提高了所有条件下的准确率，其中**组合-组合** 条件表现最佳，达到 78%。结果表明，数据流分析可以显著提升代码补全性能。

4.5 切片如何帮助代码补全?

我们分析了切片可能如何提升代码补全性能；也就是说，了解切片是否能帮助模型更好地补全稀有标签对应的代码，

[6]CuBERT 模型可在 github.com/google-research/google-research/tree/master/cubert 获取

[7]github.com/python/cpython/blob/main/Lib/tokenize.py[8]我们注意到，tokenize 只会为不存在任何语法错误的代码片段输出词元；否则，它会返回 IndentationError 或 TokenError。为了预测函数调用，我们经常会输入不完整、因而语法不正确的代码；因此，我们必须修改原始模块，使其始终返回截至目前已完成 tokenize 的内容。

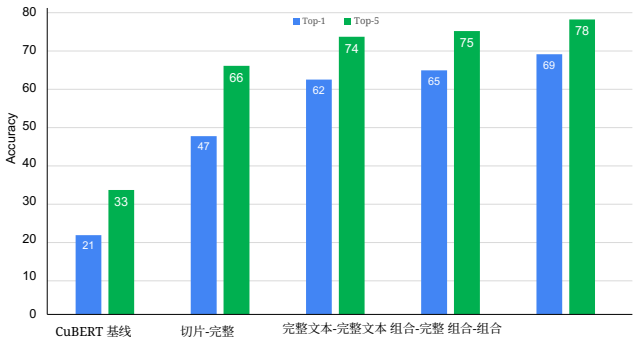

图14. 在不同“训练—测试”条件下微调的结果；即在{切片、完整文本或组合}上训练，并在{完整文本或组合}上测试

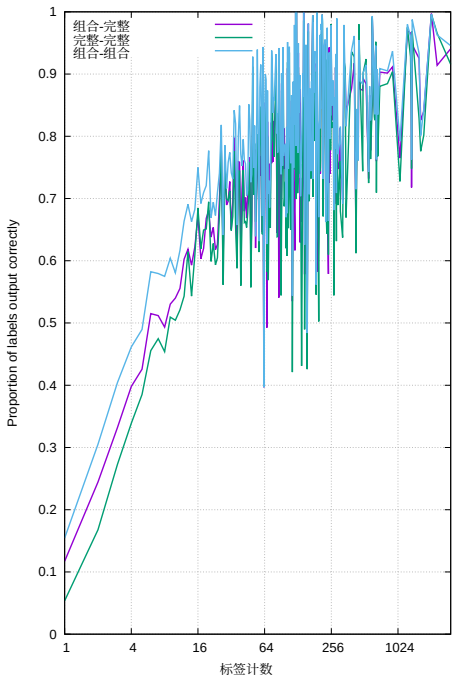

图15. 不同数量标签上的准确率

相较于更常见的标签，因为统计方法可能对常见标签效果更好，对稀有标签则不然。图15展示了不同模型的性能；在训练和测试时使用切片，能提升稀有标签的代码补全性能，且提升幅度大于常见标签，尽管常见标签似乎也能从中受益。**组合-组合** 模型在计数为1的标签上准确率为15%，这类标签约有18,000个；当标签计数达到3时，准确率便迅速接近40%。随着标签越来越常见，这些模型之间的差异

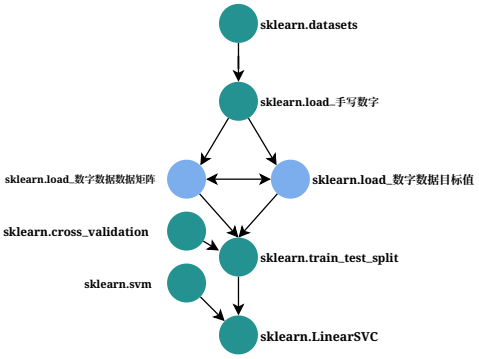

图16. 过滤掉作为自动化机器学习系统输入的内容后，KGpip 在运行示例中的训练图。

模型变得更加嘈杂，但**组合-组合**的情况仍然占优。

4.6 与现有工作的比较

在这一领域，比较颇为棘手，因为没有通用或标准的基准测试，而且具体问题也各不相同。我们选择了 ETH150K，它至少是一个知名的代码仓库；但例如，依赖自身 GitHub 样本的研究，其结果就无法进行比较。具体问题也有所不同：有些工具（包括我们的方法）预测函数调用，有些只预测方法调用，还有一些则预测所有词元的下一个词元。这个领域确实需要一个基准测试；为帮助建立这样的基准测试，我们将向社区发布自己的切片和完整数据集。[25] 报告的总体准确率约为 0.7，与我们的结果几乎相同，但该论文预测的是所有词元类型的下一个词元，因此可能受益于某些预测（例如在 def 中，‘)’后面跟着‘:’）可由语法推导这一事实。Pythia [37] 使用神经网络而非语言模型，但报告的 Top-1 准确率与我们的相当。其 Top-5 数值更高，但他们似乎预测的是方法调用，而非像我们这样预测所有函数；在这种情况下，接收者可能会提供有助于预测的上下文。

这些方法也可能互为补充。我们的工作表明，将切片数据加入局部上下文，能显著提高模型的准确性。其他方法也主要依赖局部信息，因此也有可能从切片中获益。我们计划在未来的工作中进一步研究这一点。

5 自动化 ML 流水线的应用

自动化机器学习流水线（Au-toML）旨在通过搜索有效的数据转换和学习器，并为每个学习器进行超参数优化，从而自动构建流水线。我们的研究问题是：能否基于对大型代码库的挖掘来进行学习器和转换选择，

这些代码库包含由 Serenity 通过静态方式提取的抽象化 ML Python 脚本。与动态分析不同，Serenity 对机器学习流水线进行静态分析的优势在于成本低，因此能够扩展到数百万个脚本。具体而言，我们的问题是：将 Serenity 提取的一组流水线的语义与神经图模型结合，能否帮助预测新的流水线?

我们开发了一个名为 KGpip 的自动机器学习系统，并在另一项工作中对其进行了详细介绍 [20], 。该系统利用 Serenity 分析构建一个数据库，其中包含数据集及其对应的历史使用过的机器学习流水线。KGpip 将机器学习流水线的自动化建模为图生成问题。它基于两项假设：神经图生成器将能够：i）更简洁地捕捉实践中针对给定数据集使用的多条流水线；ii）更有效地捕捉不同流水线之间的统计相似性。在 KGpip 中，我们会对 Serenity 的分析结果进行过滤，移除与机器学习无关的组件，例如对目标机器学习库（Sklearn、XGBoost 和 LGBM）以外的库的调用，以及指示流水线脚本中调用位置的节点等。图16展示了运行示例的过滤后图。图17还概述了 KGpip 在训练和推理阶段的工作方式。KGpip 使用从 Serenity 获得的代码分析图和数据集嵌入，训练图生成模型，使其生成的机器学习流水线尽可能接近训练数据中的目标流水线。在推理时，KGpip 会识别与输入数据集最接近的数据集，并将其嵌入作为图生成模型的输入；模型随后会输出一组可能的机器学习流水线。接着，这些流水线会经过验证，并输入超参数优化器，以获得在输入数据集上性能最高的最佳流水线。

KGpip 旨在与自动机器学习系统（例如 AutoSklearn [15] 和 FLAML [40], ）协同工作，以利用它们的超参数优化器。我们使用 2000 个机器学习 Python 脚本训练了一个图生成神经网络，使其能够针对给定数据集生成机器学习流水线图。我们使用来自不同基准测试的 77 个数据集进行了全面评估，例如自动化机器学习和 Penn Machine Learning Benchmark (PMLB) 中的数据集，以及来自 Kaggle 和 OpenML 等不同机器学习门户的数据集。表 1 展示了 KGpip 的整体性能：它显著提升了当前最先进的自动机器学习系统 Auto-Sklearn 和 FLAML 在数据转换和学习算法选择方面的表现。需要指出的是，AutoSklearn 会查询流水线和数据集数据库，并根据与未见数据集最接近的邻居来选择流水线以启动搜索；不过，AutoSklearn 数据集中的流水线是基于实际执行效果筛选出的有效流水线。我们还将 KGpip 与 AL [6], 进行了比较。AL 使用动态代码分析来分析现有机器学习流水线，以选择优化后的流水线。由于搜索流水线耗尽了时间，AL 无法处理 60 个数据集。在 AL 能够处理的 77 个数据集中较小的子集（17 个）上，

	平均性能	T检验
AutoSklearn	0.71 (0.24)	-
KGpip + AutoSklearn	0.77 (0.22)	0.0002
FLAML	0.71 (0.27)	-
KGpip + FLAML	0.77 (0.20)	0.0132

表 1. KGpip 与 FLAML 和 AutoSklearn 的性能（平均值和标准差）对比。KGpip 的两种变体均显著优于现有系统，双尾T检验 $p < 0.05$

AL 的平均性能为 0.36，而 FLAML、Auto-Sklearn、KGpip + FLAML 和 KGpip +AutoSklearn 的平均性能分别为 0.745、0.705、0.79 和 0.765。与 AL 和 AutoSklearn 的这一比较清楚地展现了 Serenity 的价值，即使与依赖动态运行时分析的方法相比也是如此。

6 相关工作

Python 静态分析：近年来，Python 静态分析受到了广泛关注，也出现了多种方法。类型推断一直是许多研究的重点，有些研究采用了抽象解释等技术；近来，也有研究使用机器学习。最知名的工作可能是 MyPy [1] 和 Pytype [2]。MyPy 专注于检查和推断符合 PEP 484 [38],所定义的 Python 类型语法的类型。由于 PEP 484 在函数边界处表达类型，MyPy 专注于单个函数内的推断。Pytype 进行类型推断，并能处理变量在不同位置具有不同类型的情况。它所进行的过程间分析也相对有限。

Moat 等人 [28] 提出了一种用于 Python 类型推断的抽象解释方法，该方法对多个域进行建模，以计算更准确的信息，并利用新近性抽象处理别名。不过，目前该方法对过程间分析的支持有限，而内联可以实现过程间分析。Fritz 和 Hage [16] 提出了一种用于类型推断的数据流分析方法，在成本和精度之间提供了多种权衡。该方法能够处理一等函数等特征。

机器学习也用于 Python 的类型推断。TypeWriter [31] 使用代码语料库训练神经模型，标签数据则来自用户注释。与基于程序分析的类型推断不同，TypeWriter 将代码中的注释作为神经模型的输入。尽管这类系统确实在进行分析，但其方法和机制与我们大不相同。

也有其他针对 Python 的分析，通常用于特定目的。Ariadne [11] 与我们一样使用 WALA，但专注于推断机器

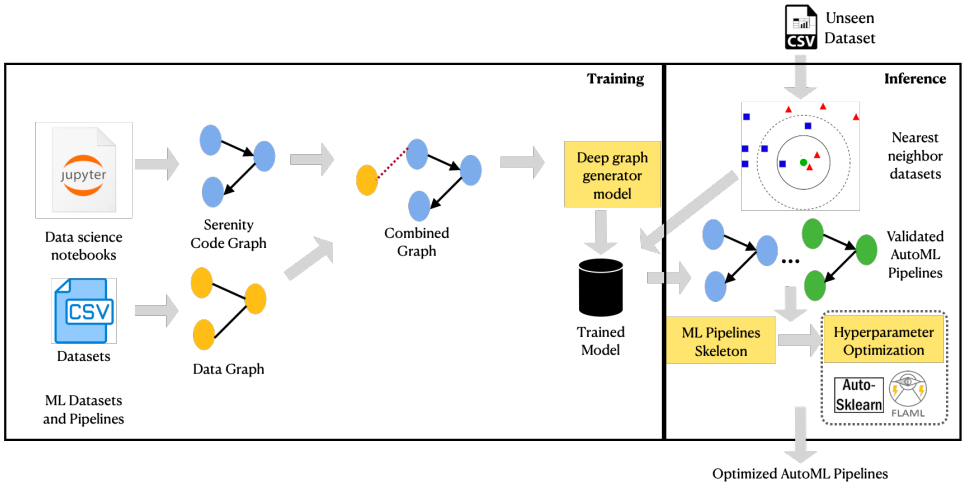

图17. KGpip 训练与推理工作流概览

学习程序中张量的形状。与我们处理库的方法不同，Ariadne 会根据需要对机器学习库进行建模，以支持张量相关操作。**代码补全**：代码补全一直是提升在集成开发环境中工作效率的重要研究领域。该领域的主要挑战之一是如何表示代码；绝大多数研究使用词元或抽象语法树来表示代码（有关该领域的详细综述，请参阅 [3]）。[25] 将 Python 和 JavaScript 代码表示为抽象语法树，并使用指针网络更好地预测代码补全中的词汇表外词。Pythia [37] 是另一种代码补全方法，它使用抽象语法树和长短期记忆网络模型。

一些方法也尝试利用基于数据流和控制流的表示 [4, 7, 14]。在 JavaScript 中，[21] 利用程序依赖图检测代码重复。 [4] 使用基于 AST 的表示，并结合局部数据流和控制流，以预测变量名和变量误用。 [14] 将基于词元的代码表示与基于对象使用的边以及抽象语法树节点相结合，以预测方法的文档。为了针对 Java API 调用执行代码补全， [29, 30] 主要采用过程内分析，挖掘通过控制流和数据流增强的图。

随着 BERT[10] 和 GPT [5, 32], 等预训练语言模型的兴起，许多近期方法 [13, 19, 22,26, 41] 开始利用这些模型中已有的丰富语言理解能力，并针对代码摘要、翻译、完整文本、错误检测等各种代码理解任务进行微调。CodeBERT[13] 是一种基于 BERT 的模型，使用六种编程语言的自然语言与编程语言样本对进行训练。GraphCodeBERT [19] 也使用 BERT，但通过基于抽象语法树的数据流图来表示代码。GraphCodeBERT 中的数据流完全局部化，不像 Serenity 那样进行过程间分析。例如，它会为表达式中使用的所有变量与其定义之间添加边。CuBERT [22] 和 CodeT5 [41] 分别是另外两种基于 BERT 和 T5 [33]架构的模型。

与我们的方法不同，所有这些方法都将代码表示为词元序列 [13, 22], 抽象语法树 [25, 27, 37, 41],，或表示为从抽象语法树派生的数据流 [19]。

自动化机器学习方法：近年来，已有多种自动化机器学习框架被提出 [6, 12, 15, 40]。在大多数自动化机器学习系统中，学习器和预处理选择由流水线及数据实际执行数据库驱动。例如， [15, 36] 会计算数据集元特征数据库，例如行数和列数；而 [6]则通过对 Kaggle 等平台上公开可用的机器学习流水线进行动态代码分析，挖掘学习器和预处理器输入的运行时信息库。对学习器和预处理器的预测基于目标数据集与存储特征之间的相似度度量。在 KGpip 中，我们利用从数据集原始内容中提取的稠密向量嵌入来度量这种相似度，并利用图神经网络选择学习器/预处理器并生成流水线。

一些现有系统（如 TPOT [24] 或 RECIPE [9]）使用进化算法进行流水线生成。另一些系统则将其视为概率矩阵分解问题 [17], 、与用户指定语法结合的人工智能规划问题 [23, 42],，或与蒙特卡洛树搜索相结合的贝叶斯优化问题 [34]。然而，这些方法都没有利用分析来构建其数据库。

7 结论

本文介绍了 Serenity：一个用于 Python 代码静态分析的框架。Serenity 依赖两种机制：（a）在语言转换的核心进行动态分派；（b）对库进行极度抽象。为了展示

Serenity 分析的实用性，我们将其用于两个重要的代码相关应用：代码生成和自动化机器学习。Serenity 的分析在这两种应用中都展现出非常有前景的性能，使我们在某些情况下优于基于动态分析的方法，并在代码补全方面具有竞争力。我们还基于 WALA（一个流行的程序分析框架）将 Serenity 实现为开源实现。

参考文献

[1] [无日期]. MyPy. http://mypy-lang.org. 访问日期：2021-11-18。[2] [无日期]. Pytype. https://github.com/google/pytype. 访问日期：2021-11- 18。[3] Miltiadis Allamanis、Earl T Barr、Premkumar Devanbu 和 Charles Sutton。2018。大型代码与自然性机器学习综述。*ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37。[4] Miltiadis Allamanis、Marc Brockschmidt 和 Mahmoud Khademi。2018。学习用图表示程序。载于*ICLR*。OpenReview.net。[5] Tom B Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell 等。2020。语言模型是少样本学习器。*arXiv* 预印本 *arXiv:2005.14165* (2020)。[6] José P. Cambronero 和 Martin C. Rinard。2019。AL：自动生成监督学习程序。载于*ACM* 编程语言论文集，第 3 卷。10 1145 3360601 。[7] https://doi.org/ . / Kwonsoo Chae、Hakjoo Oh、Kihong Heo 和 Hongseok Yang。2017。为类 C 语言自动生成用于学习程序分析启发式方法的特征。*ACM* 编程语言论文集1, OOPSLA，文章 101（2017 年 10 月），25 页。[8] Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray、Nick Ryder、Mikhail Pavlov、Alethea Power、Lukasz Kaiser、Mohammad Bavarian、Clemens Winter、Philippe Tillet、Fe- lipe Petroski Such、Dave Cummings、Matthias Plappert、Fotios Chantzis、Elizabeth Barnes、Ariel Herbert-Voss、William Hebgen Guss、Alex Nichol、Alex Paino、Nikolas Tezak、Jie Tang、Igor Babuschkin、Suchir Balaji、Shantanu Jain、William Saunders、Christopher Hesse、An- drew N. Carr、Jan Leike、Josh Achiam、Vedant Misra、Evan Morikawa、Alec Radford、Matthew Knight、Miles Brundage、Mira Murati、Katie Mayer、Peter Welinder、Bob McGrew、Dario Amodei、Sam McCandlish、Ilya Sutskever 和 Wojciech Zaremba。2021。评估基于代码训练的大型语言模型。（ 2021 2107 03374 [ ]）。arXiv: . cs.LG [9] Alex G. C. de Sá、Walter José G. S. Pinto、Luiz Otavio V. B. Oliveira 和 Gisele L. Pappa。2017。RECIPE：用于自动演化分类流水线的基于语法的框架。载于遗传编程，James McDermott、Mauro Castelli、Lukas Sekanina、Evert Haasdijk 和 Pablo García-Sánchez（编）。246–261。https://doi.org/10.1007/978- 3-319-55696-3 16_[10] Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。2018。BERT：用于语言理解的深度双向 Transformer 预训练。*arXiv* 预印本 *arXiv:1810.04805* (2018)。[11] Julian Dolby、Avraham Shinnar、Allison Allain 和 Jenna Reinen。2018。Ariadne：机器学习程序分析。载于第 *2* 届 *ACM SIGPLAN* 机器学习与编程语言国际研讨会论文集（美国宾夕法尼亚州费城）*(MAPL 2018)*。计算机协会，纽约，美国，1–10。https://doi.org/ 10 1145 3211346 3211349 . / . Iddo Drori、Lu Liu、Yi Nian、[12] Sharath C Koorathota、Jung-Shian Li、Antonio Khalil Moretti、Juliana Freire 和 Madeleine Udell。2019。使用元数据语言嵌入的自动化机器学习。*ArXiv*（2019）。https: // arxiv.org/abs/1910.03698[13] Zhangyin Feng、Daya Guo、Duyu Tang、Nan Duan、Xiaocheng Feng、Ming Gong、Linjun Shou、Bing Qin、Ting Liu、Daxin Jiang 和 Ming Zhou。2020。CodeBERT：面向编程语言和自然语言的预训练模型。*CoRR* abs/2002.08155（2020）。arXiv:2002.08155 https://arxiv.org/abs/ 2002 08155 。[14] Patrick Fernandes、Miltiadis Allamanis 和 Marc Brockschmidt。2018。结构化神经摘要。*CoRR* abs/1811.01824（2018）。[15] Matthias Feurer、Aaron Klein、Katharina Eggensperger、Jost Springenberg、Manuel Blum 和 Frank Hutter。2015。高效且稳健的自动化机器学习。在神经信息处理系统国际会议论文集（*NeurIPS*）中。2962–2970。https://dl.acm.org/doi/ 10 5555 2969442 2969547 。/ 。[16] Levin Fritz 和 Jurriaan Hage。 。2017。Python 近似类型的成本与精度。在 *2017* 年美国计算机协会 *SIGPLAN* 部分求值与程序操作研讨会论文集（法国巴黎）（*PEPM 2017*）中。计算机协会，纽约州纽约，美国，89 98 10 1145 3018882 3018888 – 。https://doi.org/ 。/ 。[17] Nicolo Fusi 、 Rishit Sheth 和 Melih Elibol。 。2018。用于自动化机器学习的概率矩阵分解。在神经信息处理系统国际会议论文集（*NeurIPS*）中。3352 3361 10 5555 3327144 3327254 – 。https://dl.acm.org/doi/ 。/ 。[18] David Grove 和 Craig Chambers。 。2001。调用图构造算法框架。*ACM* 编程语言与系统汇刊 23，6（2001 年 11 月），685–746。https://doi.org/10.1145/506315.506316[19] Daya Guo、Shuo Ren、Shuai Lu、Zhangyin Feng、Duyu Tang、Shujie Liu、Long Zhou、Nan Duan、Alexey Svyatkovskiy、Shengyu Fu、Michele Tufano、Shao Kun Deng、Colin B. Clement、Dawn Drain、Neel Sundaresan、Jian Yin、Daxin Jiang 和 Ming Zhou。2021。GraphCodeBERT：利用数据流预训练代码表示。在第九届国际学习表征会议（*ICLR 2021*），*2021* 年 *5* 月 *3–7* 日，奥地利虚拟会议中。OpenReview.net。https://openreview.net/forum? id=jLoC4ez43PZ[20] Mossad Helali、Essam Mansour、Ibrahim Abdelaziz、Julian Dolby 和 Kavitha Srinivas。2022。一种基于图神经网络的可扩展自动化机器学习方法。*VLDB* 基金会学报 15，11（2022 年 9 月），2428–2436。https://doi.org/ 10 14778 3551793 3551804 。/ 。Chun-Hung Hsiao、Michael J. Cafarella 和 Satish Narayanasamy。2014。降低面向文本的应用中的 MapReduce 抽象开销。在第 *43* 届国际并行处理会议，*ICPP 2014*，美国明尼苏达州明尼阿波利斯，*2014* 年 *9* 月 *9–12* 日中。40–49。https: // doi.org/ 10 1109 2014 13 。/ICPP。 。[21] Aditya Kanade、Petros Maniatis、Gogul Balakrishnan 和 Kensen Shi。2020。学习和评估源代码的上下文嵌入。在第 *37* 届国际机器学习会议，*ICML 2020*，*2020* 年 *7* 月 *12–18* 日（机器学习研究论文集）中。PMLR。[22] Michael Katz、Parikshit Ram、Shirin Sohrabi 和 Octavian Udrea。2020。通过规划探索上下文无关语言：自动化机器学习的理由。在国际自动规划与调度会议论文集（*ICAPS*）中。403–411。https://ojs.aaai.org//index.php/ICAPS/article/view/6686 [24] 2020 Trang T Le、Weixuan Fu 和 Jason H Moore。 。使用特征集选择器将基于树的自动化机器学习扩展到生物医学大数据。生物信息学 36，1（2020），250–256。https://doi.org/10.1093/ bioinformatics/btz470[25] Jian Li、Yue Wang、Michael R Lyu 和 Irwin King。2017。利用神经注意力和指针网络进行代码补全。*arXiv* 预印本 *arXiv:1711.09573*（2017）。[26] Shuai Lu、Daya Guo、Shuo Ren、Junjie Huang、Alexey Svyatkovskiy、Ambrosio Blanco、Colin Clement、Dawn Drain、Daxin Jiang、Duyu Tang 等。2021。CodeXGLUE：用于代码理解与生成的机器学习基准数据集。*arXiv* 预印本

arXiv:2102.04664（2021）。[27] Chris Maddison 和 Daniel Tarlow。2014。自然源代码的结构化生成模型。在国际机器学习会议中。PMLR，649–657。[28] R. Monat、A. Ouadjaout 和 A. Miné。2020。通过对 Python 程序进行抽象解释的静态类型分析。在第 *34* 届欧洲面向对象编程会议论文集（*ECOOP'20*）（虚拟会议）（莱布尼茨国际信息学会议论文集（*LIPIcs*），第 *166* 卷）中。Dagstuhl 出版社，17:1–17:29。https://doi.org/10.4230/ LIPIcs.ECOOP.2020.17 http://www-apr.lip6.fr/~mine/publi/article- monat-al-ecoop20.pdf。[29] Anh Tuan Nguyen 和 Tien N. Nguyen。2015。基于图的代码统计语言模型。在第 *37* 届国际软件工程会议论文集——第 *1* 卷（意大利佛罗伦萨）（*ICSE '15*）中。IEEE 出版社，皮斯卡塔韦，新泽西州，美国，858–868。http://dl.acm.org/citation.cfm?id=2818754.2818858[30] Tung Thanh Nguyen、Hoan Anh Nguyen、Nam H. Pham、Jafar M. Al-Kofahi 和 Tien N. Nguyen。2009。基于图的多对象使用模式挖掘。在欧洲软件工程会议与美国计算机协会 *SIGSOFT* 软件工程基础研讨会第七届联合会议论文集（荷兰阿姆斯特丹）（*ESEC/FSE '09*）中。美国计算机协会，纽约州纽约，美国，383–392。https://doi.org/ 10 1145 1595696 1595767 。/ 。Michael [31] Pradel、Georgios Gousios、Jason Liu 和 Satish Chandra。2020。TypeWriter：带有基于搜索的验证的神经类型预测。在第 *28* 届美国计算机协会欧洲软件工程会议与软件工程基础研讨会联合会议论文集中。209–220。[32] Alec Radford、Jeffrey Wu、Rewon Child、David Luan、Dario Amodei、Ilya Sutskever 等。2019。语言模型是无监督的多任务学习器。*OpenAI* 博客 1，8（2019），9。[33] Colin Raffel、Noam Shazeer、Adam Roberts、Katherine Lee、Sharan Narang、Michael Matena、Yanqi Zhou、Wei Li 和 Peter J Liu。2019。利用统一的文本到文本转换器探索迁移学习的极限。*arXiv* 预印本 *arXiv:1910.10683*（2019）。[34] Herilalaina Rakotoarison、Marc Schoenauer 和 Michèle Sebag。2019。利用蒙特卡洛树搜索进行自动化机器学习。在国际人工智能联合会议论文集（*IJCAI*）中。 3296 3303 10 24963 2019 457 – 。https://doi.org/ [35] 。/ijcai。/ Veselin Raychev、Pavol Bielik 和 Martin 2016 Vechev。 。利用决策树构建代码概率模型。*SIGPLAN* 通知 51，10（2016 年 10 月）， 731 747 10 1145 3022671 2984041 – 。https://doi.org/ [36] 。/ 。Matthias Reif、Faisal Shafait 和 Andreas Dengel。2012。分类器进化参数优化的元学习。机器学习 87, 3 2012 357 380 10 1007 10994 012 5286 7 （），– 。https://doi.org/ [37] 。/s - - - Alexey Svyatkovskiy 、 Ying Zhao 、 Shengyu Fu 和 Neel Sundaresan。2019。Pythia：人工智能辅助代码补全系统。在第 *25* 届美国计算机协会 *SIGKDD* 国际知识发现与数据挖掘会议论文集中。2727–2735。[38] Guido van Rossum、Jukka Lehtosalo 和 Lukasz Langa。2014。PEP 484——类型提示。https://www.python.org/dev/peps/pep-0484/[39] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。2017。注意力就是你所需要的一切。在神经信息处理系统进展中。5998–6008。[40] Chi Wang、Qingyun Wu、Markus Weimer 和 Erkang Zhu。2021。FLAML：快速且轻量级的自动化机器学习库。在机器学习与系统论文集（*MLSys*）第 3 卷中。434–447。https://proceedings.mlsys.org/paper/2021/file/92cc227532d17e56e07902b254dfad10-Paper.pdf [41] Yue Wang、Weishi Wang、Shafiq Joty 和 Steven C. H. Hoi。2021。CodeT5：面向代码理解与生成的标识符感知统一预训练编码器-解码器模型。在*2021* 年自然语言处理实证方法会议论文集中，*EMNLP 2021*.[42] Marcel Wever、Felix Mohr 和 Eyke Hüllermeier。2018。适用于无限长度机器学习流水线的 ML-Plan。载于 国际机器学习会议（*ICML*）自动化机器学习研讨会。https: // ris.uni-paderborn.de/download/3852/3853/38.pdf[43]Thomas Wolf、Lysandre Debut、Victor Sanh、Julien Chaumond、Clement Delangue、Anthony Moi、Pierric Cistac、Tim Rault、Rémi Louf、Morgan Funtowicz、Joe Davison、Sam Shleifer、Patrick von Platen、Clara Ma、Yacine Jernite、Julien Plu、Canwen Xu、Teven Le Scao、Syl- vain Gugger、Mariama Drame、Quentin Lhoest 和 Alexander M. Rush。2020。Transformer：最先进的自然语言处理。载于 *2020*年自然语言处理实证方法会议论文集：系统演示。计算语言学协会，线上，38–45。https://www.aclweb.org/anthology/ 2020.emnlp-demos.6