微软技术丛书

跨平台机器学习
ML.NET 架构及应用编程
（Programming ML.NET）

Dino Esposito 和 Francesco Esposito 著

周 靖 译

中文试读版 1-2 章，翻译原稿，仅供参考，

更多精彩内容，请购买正版。

京东购买>> 当当购买>> 淘宝购买>>

配套资源和试读下载： ys168 网盘>> 百度网盘>>

访问中文版博客，提交评论和勘误

清华大学出版社

北 京

ML.Net 编程

Dino Esposito 和 Francesco Esposito 著

致 Silvia、Michela 和新的梦想。

— Dino Esposito

致我所爱的人，我忍不住想把一本书献给他们。

— Francesco Esposito

目录

致谢

来自 Dino：

这是我们父子俩第二次写机器学习的书，与两年前的上一本相比，情况有了很大的变化。在这本书中，我们真正联合起来，我把我的软件经验摆到台面，Francesco 则拿出了他的朝气、活力和数学技能。我们体会到，将机器学习解决方案投入生产会变得多么棘手，而将这些小宝石隐藏在“普通”ASP.NET 应用程序的褶皱中又会变得多么“容易”。

过去两年里，我们还取得了其他成果。例如，我们巩固了对专业网球软件的掌握，并扩展到了医疗、农业和客户关怀领域。共同点始终是那一个：智能软件，最终做智能的事情。它关于的不是要取代人类和扼杀工作机会——恰恰相反。它关于的是用自动化程序取代无聊的、可自动化的任务，将人类从中解放出来，让他们从事更有趣的活动。

与 Giorgio Garcia-Agreda 和 Gaetano Guarino 以及整个 Crionet 团队一起，我们每天都在使我们的网球狂热爱好者的梦想变得更大。我们正在改变游戏规则。与 Vito Lanzotti 和 KBMS

Data Force 团队一起，我们正在默默地创造历史，将医生的手术梦想转化为具体和适用的工件，使病人接受治疗的过程更顺畅。至于 Karma Enterprise 的 Salvo Intilisano 和 Daniel Intilisano 一起工作，其实就是一件技术上的因果关系(karma 本来就是因果的意思)。相同的心态、相同的愿景以及相同的父子商业模式！项目结束后，农业就不一样了，而蜜蜂们会感激不尽!

Youbiquitous 团队正在成长，多名干员挑起了业务的重担，其中主要是 Matteo、Luciano、Martina、Filippo 和 Gabriele。感谢大家在我们享受 ML.NET 带来的乐趣时，还抽出时间来维持业务的运作。

最后，任何一本书都是团队合作的结果，我们很高兴能喊出那些最终使之成为可能的人的名字。最后，我们要向组稿编辑 Loretta Yates 和 Charvi Arora、开发和文案编辑 Rick Kughen 以及技术编辑 Bri Achtman 表示衷心感谢！

来自 Francesco：

我 23 岁了，已经长大到可以独自生活，但还足够年轻。在我众多的爷爷奶奶们面前，我还是一个小孩。难过的是，这个数字比上一本书减少了。向 Salvatore 爷爷表示深深的思念，向 Concetta 奶奶和 Leda 奶奶表示拥抱：我爱你们。更个人地说，这本书是为 Gianfranco（朋友、商业伙伴、第二位父亲和祖父）而写的。他教我如何正确做事，却忘了教我如何做错事。这本书也是为 Michela 写的，她足够坚强，无论如何都要追求自己的道路，而且足够聪明，知道如何选择一条好的道路！

前言

我们需要有人能够梦想从未有过的事物，并问“为什么没有？”。

——约翰·肯尼迪，在爱尔兰议会上的演讲，1963 年 6 月

今天，对数据科学家的追求是持续不断的。似乎有丰富的数据，云计算能力也能用上。这是机器学习最终的完美世界吗？从表面看，似乎已经有了所有必要的食材来烹饪出一道“应用人工智能”（applied AI）。但是，我们仍然缺乏一个明确和有效的方法来组合它们。

数据科学的目的，如同科学的目的一样，是为了证明某些事情是可能的。不过，数据科学并不生产解决方案。那是机器学习领域的另一个分支——数据工程——的目的。

各大企业正在疯狂地寻找数据科学家，但一个好的数据科学团队的成果通常是一个可运行的模型，其软件质量往往是一个原型，而非一个可投入生产的工件。算法与数据紧密结合，而数据必须是完整、干净和平衡的。谁负责这部分的工作往往并不清楚。因此，工作最多也只能算是部分完成。然而，对于其业务会产生大量数据的大型组织(如能源公共事业、金融机构和制造业)来说，一个与应用 AI 流水线的“生产”部分脱钩的数据科学团队仍然是一项应有的投资。对于预算明显有限的小公司来说，将一些应用数据科学的结果作为服务来购买，可能会更便宜。

从数据科学到生产，中间通常有很长的路要走，也有很多数据方面的工作要做。以下是需要考虑的几点：

- 数据如何存储？每天还是每小时？
- 数据是否应该以某种中间格式临时复制？
- 模型要想工作，需要进行什么样的转换？如何使之自动化？
- 一旦部署到生产环境，该模型的性能如何？
- 预计要以什么频率重新训练模型，以保持对实时数据(live data)的适应？
- 如果重新训练很频繁，如何自动化任何相关任务？
- 收集最新数据集，运行训练，并部署最新模型呢？

我们在机器学习模型方面遇到的最大问题，可以追溯到所采用的数据。2021 年 7 月，*MIT Technology Review* 发表了一篇关于人工智能在新冠疫情下所产生的影响的文章。这篇文章的要点在于，通过对适度开发的模型进行大规模审查所发现的许多问题都与研究人员用来开发其工具的数据质量不佳有关。这样，几乎所有工具都失去了有效的用途。这导致人们开始更好地理解数据工程和数据质量的作用。通过 CSV 稀疏文件处理数据对于探查一个想法来说是足够的，但要想建立一个健壮的基础结构，还是需要切换到一些数据库(关系型、NoSQL 或图)和一些严肃的查询语言。而如果这样做，很可能需要超越 Python，进入经典编程的领域。仅仅掌握数据科学还不够，还必须掌握严肃的编程和数据库技能。另一方面，从数据中寻找和特定业务相关的见解，不正是你最终的目的吗？

目前，常规意义上的 AI(更具体地说就是机器学习)只是垂直问题的商品和直接解决方案之间的一种权衡。商品化的云服务提供安全、稳定和可接受的质量。虽然并不能涵盖所有可能的情况，但它们正在扩充，而且在不久的将来会有更多的扩充。

所有这一切都营造了一个环境，使我们能构建同样的旧软件，但使用更强力的工具。我们不只是在谈论编程语言的基元(primitives)和由框架提供的一些类。我们还在谈论由机器学习算法和商品化的云服务支持的智能和预测性工具。

在这个背景下，ML.NET 充当了数据工程和商品化数据科学之间的一座完美桥梁，它和.NET Framework 完全集成。ML.NET 商品有不同的形式：用于浅层学习的内置算法、对 Azure 云服务的便利访问以及与预训练模型(例如 Keras 或 TensorFlow 网络)的整合。

本书面向的读者

在我们看来，如果你已经用了.NET 的那一套东西，那么 ML.NET 就是做机器学习的完美工具，无论你选择的算法和模型的内部机制是什么。

因此，本书是为愿意(或需要)进入机器学习世界的.NET 开发人员而写的。如果你是一名软件开发人员，想把数据科学和机器学习技能添加到自己的技能库中，它就是你理想的选择。如果你是一名数据科学家，愿意学习更多关于 Python 以外的软件知识，那么它也是你理想的选择。不过，这两类人都需要越来越多地了解对方。

本书不面向的读者

本书从 ML.NET 的角度讨论机器学习。ML.NET 是一个特定平台的库。它更多地是为数据工程师和 ML 工程师而不是普通的数据科学家量身而写的。澄清一下，ML 工程师的核心职责是将外部训练好的模型实际地融入到客户的应用程序中，并执行更精致的任务，监督

基于数据科学规范的模型的建立和训练。本书讨论了具体的工具。

如果你对机器学习解决方案的实际生产没有多大兴趣，这本书可能不是你最理想的。它并不会向你展示前沿的数据科学技术，但它会教你如何开始利用 ML.NET 团队多年来一直在做的事情——在.NET 中整合简单而有效的机器学习解决方案。

本书的组织方式

本书分为三部分：

- 第 1 章~第 3 章对库进行了基础性的概述。
- 第 4 章~第 10 章概述了对常见问题进行数据处理、训练和评估的专门任务。这些问题包括回归、分类、排名、异常检测等等。
- 第 11 章~第 13 章专门讨论在所有浅层学习任务都不合适时可能需要用到的神经网络。此外，我们概念了神经网络，还提供了一个同时使用了商品化 Azure 认知服务和人工打造的定制 Keras 网络的护照识别的例子。

最后，附录 A 讨论了模型的可解释性（explainability）。

系统需求

完成本书的练习需要以下软硬件：

- 一台运行 Windows 10/11，Linux 或 macOS 的计算机
- 任意版本的 Visual Studio 2019/2021 或者 Visual Studio Code
- 互联网连接，用于下载软件和本书示例文件

代码示例

书中所有代码(包括可能的勘误和补充内容)都可以访问以下网址获得：

https://MicrosoftPressStore.com/ProgrammingMLNET/downloads

第 1 章 人工智能软件

让我们计算一下，不用多说，看看谁是对的。

——戈特弗里德·威廉·莱布尼茨，“发现的艺术”，1685 年

软件是 17 世纪以来少数伟大的思想家脑海中闪过的雏形的最终结果。一些数学家、哲学家和最一般意义上的科学家，以不同的方式和不同的抽象程度，对一种能使知识的获取和分享机械化的通用语言产生了憧憬。特别是，戈特弗里德·莱布尼茨(1646~1716)得出结论(或者说只是有点不切实际的梦想)，人类的推理至少有一部分可以机械化。他甚至设计了一个抽象的引擎——推理演算器(calculus ratiocinator)——来作为用某种符号语言和适当的符号书写的语句的通用处理器。

莱布尼茨关于演算(微积分)的富有远见的笔记在两个多世纪里一直没有出版，但符号语言的思路在他于 1684 年引入的无穷小数的符号中得到了直接的应用。几乎在同一时间，艾萨克·牛顿正在发展他的数学方法来解释运动和引力的物理学。

19 世纪末，莱布尼茨在两个世纪前的工作刺激了逻辑学家大胆地超越当时仍占主导地位的亚里士多德的逻辑。特别是，德国科学家戈特洛布·弗雷格毕生致力于设计一个可以用来表示所有数学陈述的综合理论。不过，他的作品存在一个 bug，就在整部作品付印的前几天，我们伟大的 beta 测试员伯特兰·罗素发现了这个 bug，从而引出了著名的罗素悖论。

有趣的是，用罗素悖论去过弗雷格的理论，并不能发现一些错误或虚假陈述。几十年后，哥德尔的不完备定理表明，弗雷格或罗素的推理都没有错。事实上，哥德尔用一个反例证明了由弗雷格设定的、由罗素否认的期望是不现实的。

再之后，阿兰·图灵开始了抽象数学的不朽之旅，最终形成了图灵机——史以来第一个完全定义的符号计算模型。

很好！但是，软件又是一个什么情况？

1.1 软件源起

哥德尔不完备定理划出了数理逻辑不能超越的界限。从本质上讲，不完备性意味着有些事情就是不能通过任何形式推理(formal reasoning)来证明真或假。就是这样。

然而，虽然这是一个令人沮丧的结果，但凡事都要分两面。正是它的另一面，打开了今天我们所说的“软件”的闸门。不完备性扼杀了 17 世纪博学者的梦想，但它同时表明，在一致的形式系统的边界内，任何推理总是可以表示为一组形式转换规则，所以能以某种方式“机械化”。

这一事实是任何基于计算机的推理的理论基础，并标志着今天的“软件”的诞生。

1.1.1 计算机的形式化

哥德尔不完备定理(1931 年)启发了一些独立的研究路线，它们在 20 世纪 30 年代中期获得

了一些成果：

- **一般递归函数** 由哥德尔本人定义，一般递归函数是一种可计算的逻辑函数，它接收一个有限的自然数元组并返回一个自然数。
- **Lambda 演算(λ演算)** 由阿隆佐·邱奇设计的一种形式，用于定义对自然数进行的一些机械计算。
- **图灵机** 能通过写在一条无限长的纸带上的符号进行计算的计算机的理论模型。

接下来，在 1936 年，邱奇－图灵论题统一了三类可计算函数。该论题证明，当且仅当一个函数在图灵机中是可计算的，并且当且仅当它是一个一般递归函数，它就可以在 λ 演算中计算。该论文的最终结果是使建造一个机械装置成为可能，该装置可以通过对符号的操作再现任何可以想象的数学推导过程。

第二次世界大战的爆发加速了能够计算数字和自动化任务的电子机器的发展。现代计算机的著名先祖包括恩尼格玛密码机(Enigma)；相应的解码机 The Bombe(阿兰·图灵做出了重大贡献)；德军使用的洛伦兹密码机(Lorenz)；以及英国的巨型机 Colossus，它最终破解了洛伦兹密码。所有这些模型机都是在欧洲建造的，而美国在战争的最后时日，在现代计算机科学的另一位大人物约翰·冯·诺伊曼的监督下，完成了 ENIAC 计算机的建造。

所有这些机器都基于邱奇－图灵论题所奠定的理论基础。

1.1.2 计算机工程设计

20 世纪 50 年代，远离战争的喧嚣，研究重新开始，科学家们面临着设计计算机架构的紧迫问题。想象一下，站在其中某个伟大人物的立场上，你会怎么做？

想一想：现在是 20 世纪 50 年代，在经历了战争的艰辛之后，世界正在焕然一新。在过去的几年里，你和你的同行基于唯一可行的理论证据建造了专用机器。战争的突发事件迫使你围绕非常具体的任务建造机器，主要是根据数字计算数字，但你知道还有更多可能。在该理论的指导下，已经用机电阀门、电线和转子等设备建造了一种数字运算设备。基于相同的理论，你可以设计一台机器来计算任何可以通过一致的符号语法表达的东西。这是关于创造。 这不是从其他数字中获取数字。

站在任何这样的伟大人物的立场上，你可能会觉得自己是个神。

而且你可能会预见到一台机器能以与人类相同的方式行事。然后，你可能会想知道这个关键问题：机器能思考吗？

对一种计算机进行工程设计时，似乎很自然地把这种机器联想成人脑的代用品。最初的重点是工程部分——如何将物理部分连接到一个整体架构中，以便能灵活地处理数字和表示更复杂的信息。当时没人想到我们今天所说的“软件”这样的东西。

目标是创造一台智能机器，而模型就是人脑。

1.1.3 人工智能的诞生

人工智能(Artificial Intelligence，AI)一词于 1956 年正式提出。当时，约翰·麦卡锡在美国

新罕布什尔州的达特茅斯学院组织了一个为期六周、向一些数学家、工程师、神经学家和心理学家开放的夏季研讨会。

该研讨会被设计为围绕思维机器的想法进行的头脑风暴。当时，思维机器(thinking machines)这个抽象主题正在两个不同的、基本相反的研究背景下进行辩论。自动机(Automata)理论直接来源于丘奇和图灵的工作，控制论(cybernetics)则直接来源于巴贝奇的理论，并由冯·诺伊曼转化为具体的硬件。

为了取悦和吸引两个阵营的研究人员，麦卡锡选择了人工智能这个新名字，因为它具有中立性。另外，麦卡锡想统一两套学术研究，他认为这属于同一个实体。

然而，研讨会的最终目的是为一些共享的方法和实践打下基础，以建立和人脑对应的造物。

许多人拒绝了研讨会的邀请，也没有产生具体的成果。不过，它作为人工智能的诞生而被世人铭记。

1.1.4 作为副作用的软件

计算机的诞生是为了模仿人脑，并以正式和可计算的方式呈现智能。但是，正如近年来许多流行的 Internet“谜因”或“表情包”一样，在某个时间段，一些事情突然就“变歪”了。其结果正是我们目前所说的软件；在某种程度上，它是在追求智能的人工形式时产生的废品。

什么是软件？

有趣的是，作为后来人，年轻的开发者可能仍然不清楚这样一个事实：软件源于冯·诺伊曼计算机架构的定义，该架构引入了一个关键的概念，即指令必须与硬件分离，而不能硬编码到物理组件中。冯·诺伊曼架构与现代设备使用的架构相同，由一个提供基本计算逻辑的处理单元(CPU)、多级存储器和输入/输出机制组成。

安排机器执行指令的编程语言在 20 世纪 50 年代开始出现，并迅速发展到下一个十年末。在那个时候，软件很明显是使用计算机的必要条件，也是构建智能自动化行为的不可避免的工具。

进入 20 世纪 60 年代后，人们意识到，为了编写软件以实现任何最低限度的可接受行为，都需要专注、纪律和方法。并非巧合的是，软件工程一词在 1968 年左右开始流行。

软件在让宇航员登月方面发挥了关键作用，但它和人工智能完全不同；软件需要人脑来指挥行动。然而，业界认为软件已经够用了，人工智能的梦想可以再等等。然后，20 世纪 70 年代初给我们带来了关系数据库和明确的方向：软件是为商业服务的，人工智能则转移到了学术领域。

1.2 软件在今天的作用

现在已经进入二十一世纪，软件在我们的日常生活中无所不在。但刚开始的时候，情况并不是这样的。几十年来，软件被设计成围绕核心和原始数据的一个相对薄的抽象层。储存和读取业务数据是主要目标，而软件只是平滑业务流程齿轮的必要油脂。软件被设计为正

确和(理想情况下)快速地执行任务。

没有对用户的关怀。在某种程度上，人们还要仅仅因为能用上计算机，就不得不感到由衷的高兴和感激。20 年前，Internet——然后是 10 年后的 iPhone——改变了这个局面。一堆新的方法论被引入了，而且最重要的是，工程师和管理者终于意识到了用户的重要性。

为了善待用户，使信息在他们的指尖上就能得到(这是比尔·盖茨的一句老口号)，软件必须以不同的方式，并围绕不同的——完全不同的——用户故事进行设计。总而言之，在现代社会，任何软件都有三个主要目标：

- 使人们免于重复和无聊的任务
- 反映现实世界中发生的过程
- 帮助并赋予人们能力

自 ENIAC、Fortran、IBM 大型机和工作站出现的早期岁月以来，这些目标并没有发生显著变化。云时代(以及，脑洞一下，未来可能的量子计算时代)亦是如此。它们构成了软件的本质，并且在某种程度上是通用的。

过去几十年发生的变化，以及可能进一步发生变化的，是这些目标与一般人、公司、组织和企业的相关性。软件的目标不会变，其本质也预计不会有变。相反，预计会发生变化的是为这些目标赋予越来越多的相关性。预计软件会越来越接近现实世界，并为越来越多的人赋予能力。与此同时，为其他人类特有的行动节省时间，或者只是变得更有趣。

出于这些目标，软件需要变得更加智能。

目前还没有能作为“魔法棒”使用的人工智能；想要有，但它还没有到来。不过，至少我们应该编写更智能的软件。为此，一种方式是使用当前在研究人工智能的大环境下创建的工具。

1.2.1 自动化任务

在软件行业的初期，像存储和检索数据这样简单的事情被视为一种成功的自动化形式。但这么多年过去了，对自动化的需求和对“可自动化任务”的定义都发生了变化。今天，许多 20 年前会被人类用户愉快接受的任务被认为是枯燥的、重复的，适合用更智能的软件应用来接手。

简单形式的人工智能可能有助于此类任务的一些很好的例子是时间线分析、文档扫描和文档的标准处理，例如调整照片大小或录入护照信息。在这种情况下，我们可以将人工智能与条码阅读器进行比较。多年前，条码阅读器令人难以置信地理顺并加速了数据录入过程。同样，相对标准的神经网络——甚至是专用算法的实现——也能为今天类似的业务领域带来类似的助力。

1.2.2 反映现实世界

你现在买的新车子让一切变得简单：转动钥匙时，嵌入式软件会欢迎你，如果是周六，它会立即问你要不要设置通往购物中心的同一条导航路线，因为你几乎每个周末都会这样走。哇，好聪明！当你开车时碰巧离另一辆车太近，相同的嵌入式软件会根据你的车型和当时

的情况向你报警或帮你刹车。哇，这更聪明！

但这无关于聪明。它更多的是关于理解和模仿现实世界中发生的事情。Dino(本书作者)在软件行业摸爬滚打了 30 多年，非常清楚地记得用户必须适应软件(而不是相反)的那些早期岁月。毫无疑问，这种情况仍会发生——即使在他今天编写的代码中，尽管频率有所降低。

请注意，受影响的不仅仅是用户体验，也不仅仅是前端和界面。为了能提供有价值的用户体验，通常还必须提供一个严肃、灵活和可扩展的后端。

在使软件以更接近我们人类的习惯进行交互的过程中，你肯定应该了解一下当前的所谓认知服务(聊天机器人和基于语音的数据输入)。 但是，如果只是进入云服务商店，订阅这样的服务，然后就满足了，那么你只是看到了问题的表面。认知服务的“智能”是一个不错的“加分项”，但不应该以牺牲对用户友好的表单/菜单以及实现业务流程/统一语言(ubiquitous languages)为代价。

假设你是一名软件工程师，在“认知服务”的框架下设计了一个软件系统。如果该软件因为没有正确命名关键业务实体而无法识别它们，就意味着你没有正确反映现实世界，也没有为客户提供优质的服务。重点在于，这里的一个简单的错误拼写，或者一个存储结构的不太合理的设计，再加上需要多年的日常使用，就可能被评判为重大缺陷。

在这些日子，我们太容易落入把 AI 套用到任何地方的圈套，并偏向于使用某种“时髦”的技术，而不是关心什么才能真正解决问题。这个问题至少短期内会一直存在。我们已经看到，一些项目因为设计和产品选择不当而陷入困境。而且，不幸的是，这种现象很普遍。

1.2.3 赋予人们能力

软件越能反映现实世界，最终就越能增强人们在日常的个人、社交和商业活动中的能力。反过来也是如此：经常使用软件的人越多，软件的发展和改进就越多。

今天，为了真正赋予用户能力，软件必须是智能和主动的。一个好的用户体验似乎越来越像一种商品。它被认为是一个好东西，只是现在它产生的影响越来越少。AI 是一种有效的工具，可以研究如何为用户提供更强大的功能，并对用户使用的数据产生更多的见解。

对于几乎任何用户活动，及时的建议、推荐以及准确的估计和预测都非常有用。如今，AI 不再是一个晦涩难懂的研究领域，而是完全集成在 .NET 6 等用途广泛的框架中，因此没有更多不使用它的借口。

挑战在于，如何在尽可能反映真实世界的真实应用程序的上下文中使用它。

1.3 人工智能不过是软件

人工智能与公司已经在使用的软件没有什么不同。事实上，AI 只是软件的另一面。 它是软件的一种新的推动力，可以增强用户的利益。尽管如此，围绕 AI 还是存在大量炒作。

为了真正有效和广泛地使用，AI 解决方案必须易于编码，甚至更容易集成到新的和现有的应用程序中。没有任何 AI 解决方案是独立的，挑战在于如何将智能特性无缝念成到常规软件中。如果没有很好地隐藏在友好的界面后面，即使是最高级的神经网络也不是很容易被

用户使用。

专家系统(expert systems)是人工智能的第一种具体形式，应用于巡航控制系统、法律、税务、金融和医疗保健等多个业务领域。这些智能软件系统可以完成人类专家的工作，并且可以对固定数量的问题给出同样富有洞察力的答案。专家系统的更新成本很高，但在某些时候，它们会面临淘汰。机器学习是下一步。

机器学习是 AI 的一个子集，它通过创建一个模型来回答从未明确编程回答的问题。事实上，专家系统大部分是由复杂但固定的分支网络组成的。专家系统能给出的任何答案都来自硬编码的学习路径。而在机器学习系统中，情况发生了变化。实际上，机器学习模型使用事先确定的数学函数从给定的输入计算输入。在使用之前，该模型先就一个大的样本数据集进行训练，目的就是为了发现捕捉输入和输出之间隐藏关系的数学函数。

关键不在于你应该使用哪种机器学习，或者 Python 是否比 ML.NET 更可取。关键是找到最合适的技术解决方案，为你的应用程序添加新的智能功能，将现有软件转变为更智能的软件。

说到底，人工智能不过是一种软件。

本章几乎没有触及现代软件的任何“智能”因素。鉴于当今大多数软件的复杂程度，我们面临的挑战是使其更智能，更贴近用户的需求。机器学习技术——无论是浅层学习算法还是神经网络——是一种有效的方法。本书其余部分将深入研究 ML.NET(.NET 6 平台的原生机器学习框架)的特性和功能。

第 2 章 透视 ML.NET 架构

“在数学中，提问题的艺术比解决问题更有价值。”

——乔治·康托尔(Georg Cantor)的博士论文，1867

作为消费者，我们从认知 AI(cognitive AI)不断获得令人愉悦的体验(例如，亚马逊、谷歌、苹果、微软和网飞等)。 作为普通人，我们自然希望看到医疗保健等传统行业也提供类似的体验。传统行业的一些公司的规模虽然比网络巨头们小，但比它们富有得多。在这些行业中，AI 的应用是缓慢而稳定的。用惯了智能软件，你会发现这是一种“降级”。虽然很少有传统行业的公司需要和 Alexa 或 Cortana 等同一水平的认知 AI，但所有正在运营的企业都能从更智能化的功能中受益。

那么，什么是智能软件呢？

任何软件都会静态设计为了解其运行时的上下文。但是，只有智能软件的设计是为了在运行时*动态*了解业务上下文。另一方面，这不正是智能的本意吗，即获取知识并将其转化为专长的一种能力？简单地说，智能结合了认知能力，包括感知、记忆、语言和推理，并使用特定的学习方法来提取、转换和存储信息。将所有这些转化为代码需要用到特别的工具，这些工具有别于在任何编程语言或核心框架中都有的基本逻辑功能。

营销部门喜欢将这些工具泛泛地称为人工智能，特别是在机器学习(ML)这个细分领域。那么，我们如何进行机器学习？

今天，大多数 ML 解决方案是利用 Python 生态系统所提供的工具建立的。但这只一个方便与否的问题，而不是技术优秀与否的问题。本章将介绍 ML.NET 平台，即机器学习的.NET 方式，这也是本书的核心话题。但不只如此，我们还要从架构的角度探讨常规的 ML 解决方案，并说明在我们看来，是什么原因使 ML.NET 在正确的时间成为了正确的事物。

2.1 Python 以外的生活

在群体想象中，ML 与 Python 编程语言是紧密耦合的。粗略地扫视一下众多招聘职位的描述，就能清楚地看出这一点。像 Python 和 C++这样的语言处于 ML 的最前沿，这既有历史方面的原因，也有用起来方便的原因。但是，并不存在严格的商业或技术方面的原因阻止.NET 和相关语言(C#和 F#)被有效地应用于构建 ML 模型。

2.1.1 为什么 Python 在机器学习中如此受欢迎？

Python 是一种解释型和面向对象的编程语言，由吉多·范罗苏姆(Guido Van Rossum)于 20 世纪 80 年代末创建，其宣称的目标是语法最小化和可读性。作为一种编程语言，Python 的愿景是作为一个小型的核心语言引擎，拥有一个大型标准库和一个易于扩展的解释器。

诞生于科研环境中的 Python 已经成为科学家实践、探索和实验数字的一种事实上的标准编程语言。在某种程度上，它取代了 Fortran 在 20 世纪 60 年代和 70 年代的地位。最开始的时候，在一个热门的新科学领域(例如机器学习)中使用 Python 是一种自然的选择。而且随

着时间的推移，鉴于语言自然的可扩展性，它导致了一个庞大的、包含各种专用库和工具的生态系统的建立。反过来，这也加强了人们的信念，即通过 Python 来构建计算模型是最好的选择。

今天，大多数数据科学家发现 Python 在机器学习项目中使用起来很舒服，这可能是由于该语言的简单性，再加上它有丰富的工具和示例。作为开发者，我们也发现 Python 很适合用来快速重塑数据以找到最合适的格式，快速测试算法，以及探索不同的方向。

一旦勾勒出清晰的路径，就必须训练 ML 模型并将其集成到运行环境。然后，必须监测它对于实时数据的表现，而且必须及时应用更改并重新部署。这就是 ML 的生命周期，也被称为 MLOps[①]。但是，当你从工具和库的实验中走出来，只寻找企业需要的东西——能正常工作并且可维护的代码时——Python 的结构性局限就暴露出来了。最起码，它是另一个需要集成到.NET 或 Java 体系中的栈，而这正是大多数商业应用的编写方式。

注意 从 ML 实验(通常用 Python 和笔记本完成)到部署是很困难的。事实上，根据一份 2020 年企业机器学习状况的报告，只有 22%的使用机器学习的公司成功将机器学习模型部署到生产中。详情参见 https://bit.ly/3y8BxOH。
这就是 ML.NET 的一大优势——.NET 使项目投入生产变得超级容易！

2.1.2 Python 机器学习库的分类

在 Python 中，工具和库的生态系统可以分为五个主要领域：数据处理、数据可视化、数值计算、模型训练和神经网络。这可能不是一个详尽的清单，因为还其他许多库的存在，它们负责其他任务，并专注于机器学习的一些特定领域，如自然语言处理和图像识别。

使用 Python 时，构建机器学习管道的步骤通常在笔记本的范围内进行。所谓“笔记本”，是在一个特定的 Web 或本地交互环境中创建的文档，称为 Jupyter Notebook(参见 https://jupyter.org)。每个笔记本都包含可执行的 Python 代码、富格式文本、数据网格、图表和图片的组合。通过它们，你可以建立并分享你的开发故事。在某种程度上，一个“笔记本”相当于 Visual Studio 中的一个“解决方案”。

在一个笔记本中，你可以执行诸如数据操作、绘图和训练等任务，而且可以使用一些预定义的、经过实战检验的库。

数据处理和分析

Pandas(https://pandas.pydata.org)是一个以 `DataFrame`(数据帧)对象为中心的库，开发者可以通过它加载和操作内存中的表格(扁平)数据。对象可以从 CSV 文件、文本文件和 SQL 数据库中导入内容，并支持一些核心功能，例如条件查找、过滤、索引和排序、数据切片、分组和列操作(包括添加、删除和重命名等)。`DataFrame` 对象内置了灵活重塑和透视数据以

[①] 译注：可理解为机器学习时代的 DevOps。

及合并多个数据帧的能力。它也能很好地处理时间序列数据。

Pandas 库是数据准备步骤的理想选择。它与交互式笔记本的集成，使你能对不同的配置和数据分组进行即时测试。

数据可视化

Matplotlib(https://matplotlib.org)是一个辅助库，它与机器学习管道的任何常见任务都没有直接关系，但它在数据可视化方面特别好用，特别适合可视化地表示数据准备步骤各阶段的数据，或者表示模型训练好后获得的指标。

简单地说，它只是一个为 Python 代码建立的数据可视化库。它包括一个 2D/3D 渲染引擎，支持常见的图表类型，如直方图、饼图和柱状图(条形图)。图表在线条样式、字体属性、坐标轴、图例等方面是完全可定制的。

数值计算

由于 Python 是一种主要用于科学环境的语言，所以不可能没有一堆专门为数值计算而设计的扩展。在这个领域，最流行的库是 NumPy 和 SciPy，虽然再者的作用略有区别。

NumPy(https://www.numpy.org)专注于数组操作，提供创建、操作和重塑一维和多维数组的功能。此外，这个库还支持线性代数、傅里叶变换和随机数运算。

SciPy (https://scipy.org)用多项式、文件 I/O、图像和信号处理以及更多的高级功能(如积分、插值、优化和统计)扩展了 NumPy。

在科学计算领域，Theano(https://github.com/Theano/Theano)这个 Python 库也值得一提。Theano 基于多维数组对数学表达式进行求值，非常高效地利用了 GPU 的算力。它还能对具有一个或多个输入的函数进行符号微分。

模型训练

虽然最初是为数据挖掘而设计的，但今天的 scikit-learn(https://scikit-learn.org)已经是一个主要侧重于模型训练的库。它提供了回归、分类和聚类(clustering)等流行算法的实现。此外，scikit-learn 还提供了数据预处理的方法，如降维(dimensionality reduction)、特征提取(feature extraction)和规范化(normalization)。

简而言之，scikit-learn 是浅层学习(相对深度学习)的 Python 基础。

神经网络

浅层学习是机器学习的一个领域，涵盖了一系列广泛的基本问题，如回归和分类。在浅层学习的领域之外，还有深度学习和神经网络。还有更多专门的库用于在 Python 中构建神经网络。

TensorFlow(https://www.tensorflow.org)可能是训练深度神经网络的最流行的库。它是一个综

合框架的一部分，可以在不同层次上进行编程。例如，你可以使用高层次的 Keras API 来构建神经网络，或者手动构建所需的拓扑结构，并通过代码指定前进和激活步骤、自定义层和训练循环。总的来说，TensorFlow 是一个端到端的机器学习平台，也提供了用于训练和部署的机制。

Keras(https://keras.io)可能是切入令人眼花缭乱的深度学习世界的最简单方法。它提供了一个非常直接的编程界面，至少在快速原型设计时很方便。要注意的是，可以在 TensorFlow 中使用 Keras。

还有一个选择是 PyTorch，可从 https://pytorch.org 获取。PyTorch 是一个基于现有的 C 语言库的 Python 改编，专门用于自然语言处理和计算机视觉。在这三个神经网络选项中，Keras 目前是最理想的切入点。只要它能满足你的诉求，就是一个首选的工具。需要构建复杂的神经网络时，PyTorch 和 TensorFlow 则是首选，但它们使用不同的方法来完成任务。TensorFlow 要求在训练网络之前定义好整个网络的拓扑结构。相比之下，PyTorch 采用了一种更敏捷的方法，并提供了一种更动态的方式来对图进行修改。在某些方面，它们的差异可以被概括为“瀑布式与敏捷式”。PyTorch 比较年轻，还不像 TensorFlow 那样已经建立了一个庞大的社区。

2.1.3 Python 模型顶部的端到端方案

使用 Python，你可以很容易地找到建立和训练一个机器学习模型的方法。模型最终是一个二进制文件，它必须加载到一个客户端应用程序并被调用。通常，是一个 Java 或.NET 应用程序作为 ML 模型的客户端应用程序。

有三种主要的方式来使用训练好的模型：

- 在 Web 服务中托管训练好的模型，并通过 REST 或 gRPC API 访问。
- 在应用程序中将训练好的模型作为一个序列化的文件导入，并通过它基于的基础结构(例如 TensorFlow 或 scikit-learn)所提供的编程接口(API)与之进行交互。只有在基础结构为客户端应用程序的语言提供了绑定时，这才有可能。
- 训练好的模型通过新的通用 ONNX 格式对外公开，客户端应用程序要集成一个用于使用 ONNX 二进制文件的包装器(wrapper)。

虽然 Web 服务选项最常用，但如果只是为了使用训练好的模型，客户端语言所特有的一个直接的 API 似乎才是最快的方式。不过，有几个方面需要注意：

- 使用直接的 API 会使你无法利用硬件加速和网络分布的优势。事实上，如果 API 托管在本地，任何专用硬件(如 GPU)都能随你取用。出于这个原因，如果想以非常高的速度实时调用一个图，那么应该考虑使用一个特制的、硬件加速的云主机。
- 你所选择的语言可能不存在和特定训练模型的绑定。例如，TensorFlow 原生支持 Python、C、C++、Go、Swift 和 Java。
- 从.NET 代码中调用 Python(或 C++)库并不是一个不可逾越的技术问题。然而，调用一个特定的库，例如一个训练好的机器学习模型，通常会比调用一个普通的 Python 或 C++类更难。

总之，一个 ML 解决方案不是独立存在的，它必须在一个端到端业务解决方案的框架内。

由于许多商业解决方案都是基于.NET 栈的，所以现在是时候推出一个在.NET 中原生训练 ML 模型的平台了。使用 ML.NET，你可以一直留在.NET 生态系统中，而不必面对将 Python 集成到.NET 应用程序中的麻烦。

2.2 介绍 ML.NET

ML.NET 于 2019 年春季首次发布，是一个免费的跨平台和开源的.NET 框架，旨在建立和训练机器学习模型，并在.NET 应用程序中托管它们。详情可以参考 https://dotnet.microsoft.com/zh-cn/apps/machinelearning-ai/ml-dotnet。

ML.NET 旨在为数据科学家和开发人员提供在 Python 生态系统中能找到的同一系列功能(参考之前的描述)。ML.NET 是专门为.NET 开发者建立的(例如，API 反映了.NET 框架和相关开发实践的通用模式)，它围绕经典的 ML 管道(ML pipeline，也称为 ML 流水线)概念建立：收集数据、设置算法、训练和部署。此外，任何所需的编程步骤对于任何使用.NET 框架和 C#及 F#编程语言的人来说都是非常熟悉的。

ML.NET 最有趣的地方在于，它提供了一个相当务实的编程平台，围绕预定义学习任务的思路进行布置。即使是机器学习的新手，它配备的库也使其相对容易处理常见的机器学习场景，如情感分析、欺诈检测或价格预测，整个过程和平时的编程别无二致。

与前面介绍的 Python 生态系统的支柱相比，ML.NET 基本上可被视为 scikit-learn 模型构建库的对应物。然而，该框架还包括一些可以在 Pandas 或 NumPy 中找到的、用于数据准备和分析的基本设施。ML.NET 还允许使用深度学习模型(尤其是 TensorFlow 和 ONNX)。另外，开发者可以通过模型生成器(Model Builder)训练图像分类和物体检测[①]模型。最值得注意的是，整个 ML.NET 库是建立在整个.NET Core 框架的强大功能之上的。

ML.NET 框架以一组 NuGet 包的形式提供。要开始构建模型，你不需要其他更多的东西。然而，从 16.6.1 版本开始，Visual Studio 还提供了模型生成器向导，它可以分析你的输入数据并选择最佳的可用算法。我们将在第 3 章“ML.NET 基础”中再次讨论模型生成器。

2.2.1 ML.NET 中的学习管道

典型的 ML.NET 解决方案通常用三个不同的项目来组织：

- 一个对任何机器学习管道步骤进行协调的应用程序，这些步骤包括数据收集、特征工程(feature engineering)、模型选择/训练/评估以及存储训练好的模型。
- 一个类库，其中包含了必要的数据类型，使最终模型在客户应用程序中托管后能进行预测。但要注意的是，输入和输出模式并不严格要求建立它们自己的项目，因为这些类可在负责训练或使用模型的同一个项目中定义。
- 一个客户端应用程序(网站、移动或桌面应用)。

协调器(orchestrator)可以是任何类型的.NET 应用程序，但最自然的选择是作为控制台应用

[①] object detection，目前在 Visual Studio 中翻译为“对象检测”。

程序创建。

但值得注意的是，这种特殊的代码并不是一次性的。不是说一旦生成了模型，它就没用了。更多的时候，无论在在生产之前，还是在生产中运行之后(尤其是后者)，模型必须被重新创建很多次。出于这个原因，训练器(trainer)必须设计成可重复使用，而且易于配置和维护。

上手实作

听起来很简单，但你确实可以像图 2.1 那样先在 Visual Studio 中手动创建三个这样的项目。该图展示了三个雏形的项目，其中缺少许多文件和引用，但细节已经足够丰富，可以帮助你体会最终的效果。

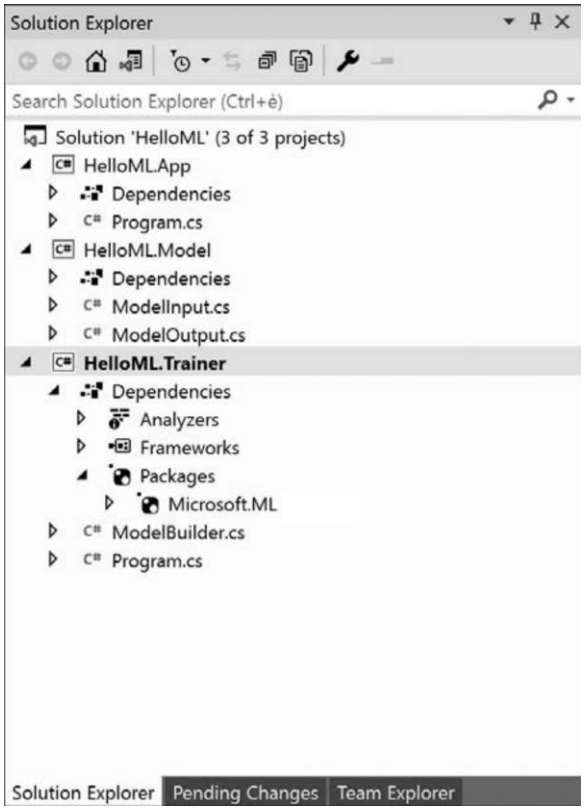

图 2.1 Visual Studio 中的 ML.NET 项目框架

除了对所选.NET 框架(无论是 3.x 还是 5)的核心引用外，你唯一需要额外加载的是 Microsoft.ML NuGet 包。

这个包并不全面，这意味着根据你打算做什么，可能需要安装更多的包。然而，这个包足以让你开始使用，并开始对库进行实验。让我们将重点放在训练器上，看看它需要与 ML.NET 库进行什么样的交互。

管道入口点

ML.NET 管道的入口点(entry point)是 `MLContext` 对象。它的使用方式与 Entity Framework `DBContext` 对象或数据库连接对象的使用方式差不多。需要有该类的一个实例，它在参与模型构建的各个对象之间共享。在大多数教程中，包括由 Model Builder 向导生成的示例代码，都采用了一种常见的做法，就是将模型构建工作流包装在一个专门的类中，通常把它直接命名为 `ModelBuilder`。

```
public static class ModelBuilder
{
    private static MLContext Context = new MLContext();

    // 主方法
    public static void CreateModel(string inputDataFileName, string outputModelFileName)
    {
        // 加载数据

        // 构建训练管道

        // 训练模型

        // 快速评做模型

        // 保存输出模型

    }
}
```

`MLContext` 类的实例对于类的方法来说是全局的，包含训练数据的文件和最终输出文件的名称被作为参数传递。`CreateModel` 方法(或者你为它选择的任何名称)的主体围绕着几个步骤展开，这些步骤涉及到 ML.NET 库中更多的具体类，例如数据转换、特征工程、模型选择、训练、评估和持久化(persistence)等活动。

数据准备

ML.NET 框架可以从各种数据源(例如，CSV 样式的文本文件、二进制文件或任何基于 `IEnumerable` 的对象)读取数据，并通过围绕特定接口而构建的几个专门的加载器进行。这个特定的接口就是 `IDataView`，它是描述表格(扁平)数据的一种灵活而有效的方式。

基于 `IDataView` 的加载器作为一个数据库游标(database cursor)工作，并提供了以任何可接受的步调(pace)浏览数据集的方法。它还提供了一个驻留在内存中的高速缓存，以及将修改后的内容写回磁盘的多种方法。下面是一个简单的例子：

```
// 为管道创建上下文
Context = new MLContext();

// 通过 DataView 对象将数据加载到管道
var dataView = Context.Data.LoadFromTextFile<ModelInput>(INPUT_DATA_FILE);
```

示例代码从指定文件中加载训练数据，并将其作为 `ModelInput` 类型的集合进行管理。不

消说，ModelInput 类型肯定是一个自定义类，反映了从文本文件中加载的数据行。以下代码展示了一个示例 ModelInput 类。LoadColumn 特性(attribute)指定当前属性(例如 Month)绑定到 CSV 的哪一列(例如列 0)。

```
public class ModelInput
{
    [LoadColumn(0)]
    public string Month { get; set; }

    [LoadColumn(1)]
    public float Sales { get; set; }
}
```

上述代码有一点需要强调。代码其实做了一个关键的假设，即加载的数据已经是机器学习操作可以接受的格式。但更多的时候，需要对现有数据进行一些转换。其中最重要的是，所有数据必须是数字，因为算法只能读取数字。下面描述了一个现实的场景。

你的客户有来自多种数据源的大量数据。这些数据源包括时间轴系列、稀疏 Office 文档，或者由在线 Web 前端填充的数据库表。如果使用其原始格式，这些数据无论数量多少，可能都不太好使。数据的适当格式取决于所需的结果和所选择的训练算法。因此，在挂载(mounting)最终的管道之前，可能需要进行大量数据转换操作，例如以列的形式呈现数据，添加特殊的特征列，删除某些列，对值进行汇总和规范化，并尽可能添大密度。取决于具体情况，这些步骤可能只需完成一次，也可能每次训练好模型后都需要完成。

注意 从表面看，似乎在管道中集成数据处理是浪费时间，而且每次构建模型时都这样做似乎没什么价值。但这是一个利弊权衡的问题。我们通常谈论的是大量的数据，将其处理成某种中间格式可能很昂贵。但另一方面，如果原始数据和整理过的数据差别没有那么大，在每次建立模型时都转换数据可以带来巨大的灵活性，因为你可以在方便的时候更改转换参数。这是纯粹就是速度与灵活性之间的一个权衡。

训练器及其分类

训练是机器学习管道的关键阶段。训练包括挑选一种算法，以某种方式设置其配置参数，并针对给定的(训练)数据集反复运行。训练阶段的输出是导致算法产生最佳结果的参数集。用 ML.NET 的行话来说，该算法称为训练器(trainer)。更准确地说，ML.NET 中的训练器是一个算法加一个任务。同样的算法(例如 L-BFGS)可以用于不同的任务，例如回归或多分类(multiclass classification，将实例分为三个或更多类别)。

表 2.1 列出了一些支持的训练器，这些训练器被归为几种不同的任务。本书以后会更深入地介绍 ML.NET 任务，并研究其编程接口。

表 2.1 与训练有关的 ML.NET 任务

任务	说明
AnomalyDetection(异常检测)	检测与接受的训练相比，意外或不寻常的

	事件或行为
BinaryClassification(二分类)	将数据分为两类中的一类
Clustering(聚类)	在将数据分为若干可能相关的组，同时不知道哪些方面可能会使数据项发生关联
Forecasting(预测)	解决预测问题
MulticlassClassification(多分类)	将数据分为三类或更多类别
Ranking(排名)	解决排名问题
Regression(回归)	预测一个数据项的值

图 2-2 展示了 ML.NET 任务对象的列表，它们通过“智能感知”从 MLContext 管道入口点显示出来。

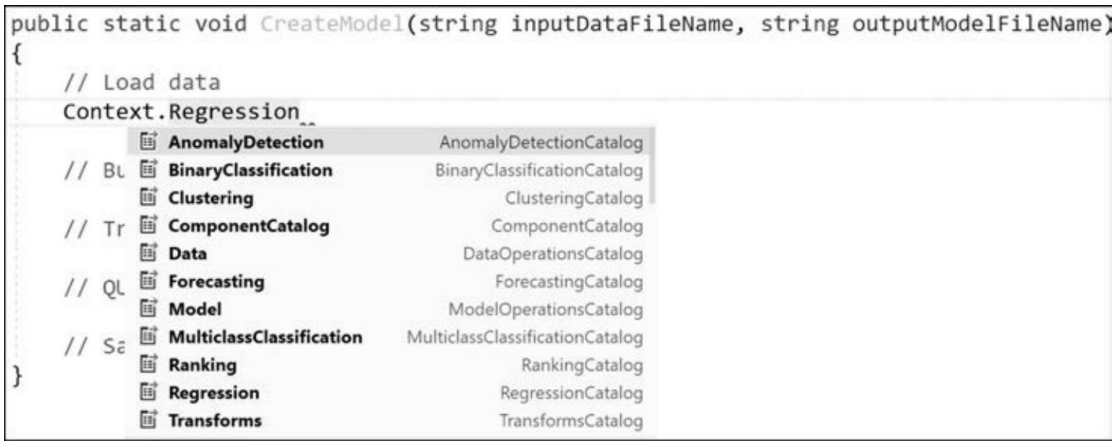

图 2.2 ML.NET 任务对象列表

表 2.1 中列出的每个任务对象都有一个 Trainers 属性，列出了框架支持的预定义算法。例如，对于一个预测任务，一个好的算法是在线梯度下降(Online Gradient Descent)算法。

```
var dataProcessPipeline = mlContext.Transforms.Text.FeaturizeText(...);
var trainer = mlContext.Regression.Trainers.OnlineGradientDescent(...);
var trainingPipeline = dataProcessPipeline.Append(trainer);
```

这段代码选择了算法的一个实例，然后将其附加到数据处理管道，管道末尾会输出编译好的模型。这些简短的代码包含了整个 ML.NET 编程模型的精华。这就是整个管道一步步构建，然后运行的方式。

还要注意的是，ML.NET 为每个预定义的任务支持几种特定的算法。特别是，对于回归任务，ML.NET 框架还支持“泊松回归”(Poisson Regression)和“随机双坐标上升”

(Stochastic Dual Coordinate Ascent)算法，还有许多算法可以通过新的 NuGet 软件包随时添加到项目中。

一旦管道建成并完全配置好，它就可以在提供的数据上运行。从这个方面说，管道是一种抽象的工作流，它处理数据的方式类似于.NET 中的 LINQ 可查询对象对“集合”和“数据集”的处理方式。

一旦训练完成，模型就只不过是计算图的序列化，它代表了某种数学表达式，或者在某些情况下，代表了一棵决策树。表达式的确切细节取决于算法，在某种程度上，还要取决于问题类别的性质。

2.2.2 模型训练执行摘要

解释模型训练的机制远远超出了本书的范围，我们的重点是 ML.NET 框架本身。然而，至少要对它的含义和工作原理做一个简要的回顾。为了进行更深入的分析，你可以很容易地找到网上资源以及书籍。特别是，你可以参考我们的《Introducing Machine Learning》一书(Microsoft Press，2020 年)。在那本书中，我们主要关注大多数问题背后的数学知识以及迄今为止为每一类问题发现的算法解决方案。

训练阶段的目的

从抽象角度说，算法是导致问题获得解决的一系列步骤。 在人工智能中，主要有两类问题：实体分类和预测。每一类问题都有几个子类，例如排名、预测、回归、异常检测、图像和文本分析等。

实际上，机器学习管道的输出是由算法(或算法链)构成的一种软件工件(software artifact)，其参数部分(设置和可配置的元素)已根据提供的训练数据进行了调整。换言之，机器学习管道的输出是算法的实例，和面向对象语言中的“类”的实例一样，它已经进行了初始化，其中容纳了一个给定的配置。用于算法实例的配置是在训练阶段发现的。图 2.3 概述了这一模式。

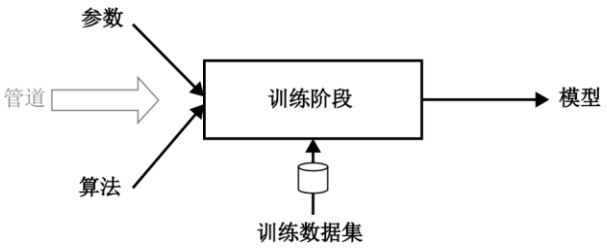

图 2.3 机器学习过程的训练阶段的总体模式

计算图

如前所述，模型是一个数学黑箱(一个计算图，即 computation graph)，它接受输入并计算输出。输入和输出是数字列表。在面向对象的环境中(例如在.NET 中)，它们是用类来建模的。

图 2.4 展示了一个抽象的和具体的视图，描述了客户端应用程序如何最终使用一个训练好

的模型。输入数据流入，图中的工具处理数字并生成一个响应，供应用程序处理。

图 2.4　总体模式：使用训练好的模型

例如，如果你有一个训练好的模型来检测金融应用中可能存在的欺诈性交易，就会调用模型中的图来处理这一笔交易的数值形式，并生成一些值，这些值可被解释为该笔交易是应该批准，拒绝，还是标记为需要进一步调查。

模型的性能

机器学习这个词听起来很吸引人，但它并不总是能完全代表在 ML.NET 或 Python 的 scikit-learn 等低级别 ML 框架中真正发生的事情。在这个层面上，训练阶段只是反复处理训练数据集中的记录，以最小化一个误差函数。

- 生成数值的函数——计算图——由选定的算法定义。
- 误差函数是添加到处理管道中的另一个元素，在某种程度上也取决于所选的算法。
- 误差函数测量图为测试数据生成的值与嵌入训练数据集中的预期数据之间的距离。
- 图以默认配置进入训练阶段，如果测得的误差对预期目标来说太大，就更新该配置。
- 当速度和准确度之间达到良好的折衷时，训练结束，图的当前配置被固定并序列化以用于生产。

整个过程在 ML.NET 框架内的训练阶段反复进行。图 2.5 总结了这些步骤。

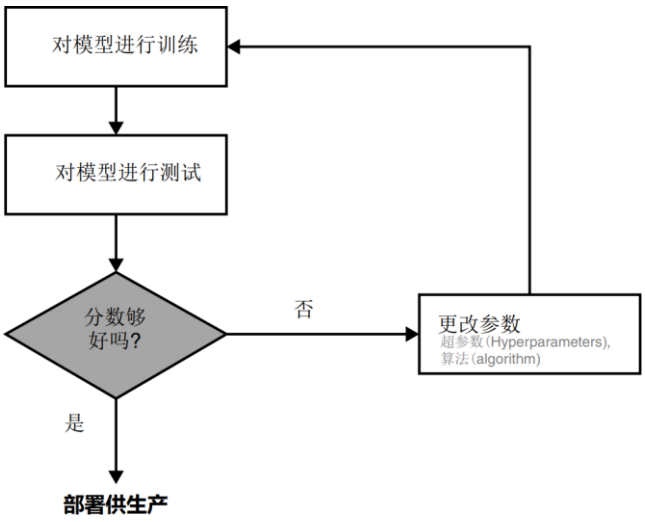

图 2.5　ML 模型的生成

应该注意的是，这里描述的评估阶段是在 ML.NET 框架内发生的，更常规地说，是在所选

择的 ML 框架的范围内发生的。评估的实际性能是基于预先给定的算法参数(称为超参数)和内部计算出来的系数的一个集合，在训练数据上获得的。

这和在生产中测量模型的性能不一样。在训练阶段，你测量的只是模型在样本数据上的表现。但是，样本数据只是模型部署到生产环境后所面对的数据一个现实快照。更关键的评估阶段要在以后进行，甚至可能要基于不同的超参数和不同的算法来重建模型。

在 ML.NET 中，训练阶段的模型质量是通过被为评估程序(evaluators)的特殊组件来测量的。

了解评估程序

评估程序是一个实现给定指标的组件。评估指标(evaluation metrics)特定于具体的算法；在 ML.NET 中，它特定于正在进行的 ML 任务。请访问以下网址，了解每个 ML.NET 任务最适合的评估程序是什么：

https://docs.microsoft.com/en-us/dotnet/machine-learning/resources/metrics

关于使每个指标符合特定任务的数学原因的更深入讨论，可以从之前提到的《Introducing Machine Learning》一书中找到。

例如，对于一个预测问题，例如估计出租车的费用(以及一般的回归和排名/推荐问题)，需要考虑的一个关键指标是平方损失(Squared Loss)或者均方差损失(MSE)。这个指标的作用是测量回归线与测试预计值的接近程度。对于每一个输入的测试值，评估程序取计算的实际值和预期值之间的距离，求平方，再计算均值。之所以要平方，是为了增加较大差值的相关性。

有趣的是，嵌入 Visual Studio 的 Model Builder 为你做了一些工作。它首先让你选择问题的类别(ML 任务)并指定训练数据集。在此基础上，它自动选择一些匹配的算法，对其进行训练，并根据自动选择的指标测量其性能。然后，它做出最终决定，并建议你应该如何开始编码你的机器学习解决方案。

一般来说，机器学习项目中有几件事情可能出问题：

- 在探索给定数据集时，所选的算法(或算法的超参数)可能不是最适合的。
- 原始数据集需要更多(或更少)的列转换。
- 原始数据集对于预期的目的来说太小了。

作为一个例子，表 2.2 总结了 Model Builder 为预测(回归)任务选择的各种算法的分数。

表 2.2 一个示例回归任务的多种算法(和分数)

算法	均方差损失	绝对损失	RSquared	RMS 损失
`LightGbmRegression`	4.49	0.38	0.9513	2.12
`FastTreeTweedieRegression`	4.70	0.44	0.9491	2.17

FastTreeRegression	4.83	0.41	0.9486	2.19
SdcaRegression	10.52	0.87	0.8845	3.27

经过 Model Builder 的一次测试运行后，显示所有特征算法最终都获得了良好的分数，但 Model Builder 按如表格所示的顺序排名，所以我们应根据指标所提供的证据选用 LightGbmRegression 算法。要特别留意“平方损失”这一列。对于排名前三的算法来说，得分是可以接受的，而对于 SdcaRegression 来说，得分则明显较差。但另一方面，SdcaRegression 的训练速度要快得多。机器学习的黄金法则是，一切都需要权衡。

 **注意** 要考虑的另一个方面是，一旦模型投入生产，即使有最好的指标，仍有可能出错，预测不符合业务预期。当这种情况发生时，有可能是用于训练的数据行不充分。至少，与生产环境中要求模型管理的真实数据相比，是不充分的。

2.3 使用训练好的模型

训练阶段结束后，你会得到一个模型，其中包含运行哪种算法和使用哪种配置的指示。该模型文件是某种序列化格式的压缩文件。请注意，存在一种通用的、可互操作的格式，即 ONNX 格式。ML.NET 也支持这种格式。

但现在的模型只是一个死物。为了让它活起来，需要在运行时环境中加载它，这样就可以公开一个 API，以便从外部调用计算。

2.3.1 使模型可从外部调用

一旦保存到文件(通常是一个 ZIP 文件)，模型就是对一些输入数据进行计算的直接描述。第一步是把它包装到一个框架引擎中，后者知道如何反序列化图形并在一些输入数据上执行它。

ML.NET 有一套量身定做的方法可供使用。为了在 ML.NET 中调用先前训练好的一个模型，需要使用以下基本代码。

```
public ModelOutput RunModel(string modelFileName, ModelInput input)
{
    var ml = new MLContext();
    var model = ml.Model.Load(modelFileName, out var schema);
    var engine = ml.Model.CreatePredictionEngine<ModelInput, ModelOutput>(model);
    return engine.Predict(input);
}
```

示例函数获取的参数包括序列化好的模型文件的路径，以及对其进行预测的输入数据。例如，如果模型估计的是打车费用，那么 ModelInput 类就要描述此次出行的具体情况。模型使用的细节通常包括距离、乘车时间(白天和晚上的费率不一样)、要求的服务类型、路况、涉及的城市区域以及其他任何确定的内容。ModelOutput 类描述了用于训练的算法的输出。它通常是一个简单的 C#类，只有几个数值属性。下面是一个例子。

```
public class ModelOutput
{
    public double Prediction { get; set; }
}
```

ML.NET 外壳代码创建了一个预测引擎的实例，它将执行反序列化、执行图以及返回计算值的任务。从软件开发者的角度来看，调用一个 ML 模型与调用一个类库方法并没有什么区别。

2.3.2 其他部署场景

将训练好的模型直接嵌入到客户端应用程序目前最简单的一种部署方案。但有两个潜在的痛点需要强调。

一个是针对目标“运行时”环境(这种情况下是.NET 框架)，对模型进行反序列化，并将其转化为可执行的计算图所产生的成本。另一个是建立预测引擎的(相关)成本。如果客户端应用程序是一个每秒会发生数千次调用的 Web 应用，那么这两种操作的执行成本都会相当高。这正是像 `PredictionEnginePool` 这样的类能派上用场的地方。

因此，前面展示的代码片段对于理解整个过程是很好的，但不一定适合生产。更现实的是，企业需要训练一个模型，将对业务来说十分关键的过程作为一种服务向各种运行中的软件应用程序公开。这意味着该模型应该被集成到某个 Web 服务中，而且应该使用适当的缓存和负载平衡层以确保适当的性能。

简而言之，一个训练好的模型可被看成是一个业务黑盒，它可以作为本地类库使用，作为 Web 服务使用，甚至作为一个有自己的存储和微前端的微服务。没有哪个选项比其他选项更优秀。相反，所有选项都是可行的，供架构师选择。

2.3.3 从数据科学到编程

如果把训练好的模型看成是一个自主的、集成在特定类型的软件应用程序中的黑盒工件，那么应该也能看到数据科学和编程之间的边界。数据科学提供了模型；编程使其可用。这两个方面都是严格需要的，也是不可避免的。

任何训练好的模型，如果没有一个像样的编程接口(无论是以类库还是 Web 服务的形式)，都会是一无是处的。需要掌握特定的技能才能构建一个有效的模型。首先，你需要领域的专业知识。其次，要会统计学和数学，要会辨别算法和指标，并会对数字进行解释。在极端情况下，还需要有开发新算法(包括神经网络)或定制现有算法的能力。单纯的开发人员很少会这些技能。

同样地，为了公开一个能发挥作用的模型，需要充分注意主机应用程序的整体性能和可伸缩性，还要照顾到用户体验。一个打车费用预测模型最终需要用数值来表示任何类型的信息。但是，你很难指望在路上使用该应用程序的人通过输入数字来指定他们的目的地。这就是编程工作的意义所在。

在这种情况下，ML.NET 接受了一个有趣的挑战：使开发人员能自主地编码他们的机器学习任务——至少对相对简单的问题实例，以及不要求相当高的精度的情况而言。这正是

ML 任务和 AutoML(Model Builder 背后的引擎)的最终目的。在这本书中，我们将深入介绍 ML 任务，但也用最后几章来给问题一个更真实的视角。高精度，如果有必要的话，是有代价的！

2.4 小结

ML.NET 目前被定位于在.NET 领域进行机器学习的一个参考平台。它主要用于浅层学习，不提供构建神经网络和深度学习的直接支持(仅支持使用现有的网络)。而另一方面，Python 领域同时有浅层学习的库(scikit-learn)和构建神经网络的库。

但是，ML.Net 最有趣和有前途的一个方面是，它提供了一种整体方法，能使机器学习易于使用，开发人员也很容易设计。没有开发者能在一夜之间变成专家级的数据科学家——即使在消化了本书的内容之后也不会。但是，任何对新技术和人工智能感兴趣的开发者都能通过 ML.NET 快速进入令人眼花缭乱的机器学习世界，这一点是最让人感到舒适的。

如前所述，虽然 Python 在数据科学家中相当流行，但没有严格的理由说为什么机器学习模型不能使用.NET(或其他语言，包括 Java 和 Go)来开发和测试。一切都有生态和易用性有关。ML.NET 依靠的是.NET Core 基础结构以及 Visual Studio。

下一章将研究一个简单而完整的例子：打车费用预测。我们会讨论特征工程和特征选择。更重要的是，会演示一个客户端 Web 应用。